CUBRID 2008 R3.1 User Manual

Table of Contents

Introduction to Manual

Manual Contents

The contents of the CUBRID Database Management System (CUBRID DBMS) product manual are as follows:

- Getting Started with CUBRID : The "Getting Started with CUBRID" provides users with a brief explanation on what to do when first starting CUBRID. The chapter contains information on new features added to CUBRID, on how to install and execute the system, and provides a simple guide on how to use the CSQL Interpreter and CUBRID Manager. The chapter also includes examples of how to write application programs using JDBC, PHP, ODBC, CCI, etc.
- Introduction to CUBRID : This chapter provides a description of the structure and characteristics of the CUBRID DBMS.
- CSQL Interpreter : CSQL is an application that allows you to use SQL statements through a command-driven interface. This chapter explains how to use the CSQL Interpreter and associated commands.
- Administrator's Guide : This chapter provides instructions on how to create, drop, back up, restore, migrate, and replicate a database. Also it includes instructions on how to use CUBRID utilities, which starts and stops the Server, Broker and CUBRID Manager servers, etc.
- Performance Tuning : The "Performance Tuning" chapter provides instructions on setting system parameters that may influence the performance. This chapter provides information on how to use the configuration file for the Server, Broker and CUBRID Manager and describes the meaning of each parameter.
- CUBRID SQL Guide : This chapter describes SQL syntaxes such as data types, functions and operators, data retrieval or table manipulation. The chapter also provides SQL syntaxes used for indexes, triggers, partitioning, serial and user information changes, etc.
- CUBRID Manager : The chapter provides instructions on how to use the CUBRID Manager, which is a GUI (Graphic User Interface) mode database management and query tool. The CUBRID Manager makes the easy handling of numerous management tasks possible and also provides a "query editor" function, which can execute the SQL syntax in the connected database.
- API Reference : This chapter provides information on JDBC API, ODBC API, OLE DB API, PHP API, and CCI API.

Manual Conventions

The following table provides conventions on definitions used in the CUBRID Database Management System product manual to identify "statements," "commands" and "reference within texts."

Convention	Description	Example
Italics	Italics type is used to show the variable names.	*persistent*: stringVariableName
Boldface	**Boldface** type is used for names such as the member function name, class name, constants, CUBRID keyword or names such as other required characters.	**fetch** ( ) member function class **odb_User**
Constant Width	Constant Width type is used to show segments of code example or describes a command's execution and results.	csql database_name
UPPER-CASE	UPPER-CASE is used to show the CUBRID keyword (see **Boldface**).	**SELECT**
Single Quotes (' ')	Single quotes (' ') are used with braces and brackets,	{'{'const_list'}'}

	and shows the necessary sections of a syntax. Single quotes are also used to enclose strings.	
Brackets ([ ])	Brackets ([ ]) indicate optional parameters or keywords.	[ONLY]
Underline(_)	Underline (_) indicates a default keyword if no keyword is specified.	[DISTINCT|UNIQUE|ALL]
Vertical bar(|)	Vertical bar (|) indicates that one or another option can be specified.	[COLUMN|ATTRIBUTE]
Braces around parameters({ })	Braces around parameters indicate that one of those parameters must be specified in a statement syntax.	CREATE {TABLE|CLASS}
Braces around values({ })	Braces around values indicate that every value is a member of the same set.	{2. 4, 6}
Braces with ellipsis({ }...)	Braces before an ellipsis indicate that a parameter can be repeated.	{, class_name}...
Angle brackets(< >)	Angle brackets indicate a single key or a series of key strokes.	<Ctrl+n>

Introduction to CUBRID

This chapter explains the architecture and features of CUBRID. CUBRID is an object-relational database management system (DBMS) consisting of the Database Server, the Broker, and the CUBRID Manager. It is optimized for Internet data services, and provides various user-friendly features.

This chapter covers the following topics:

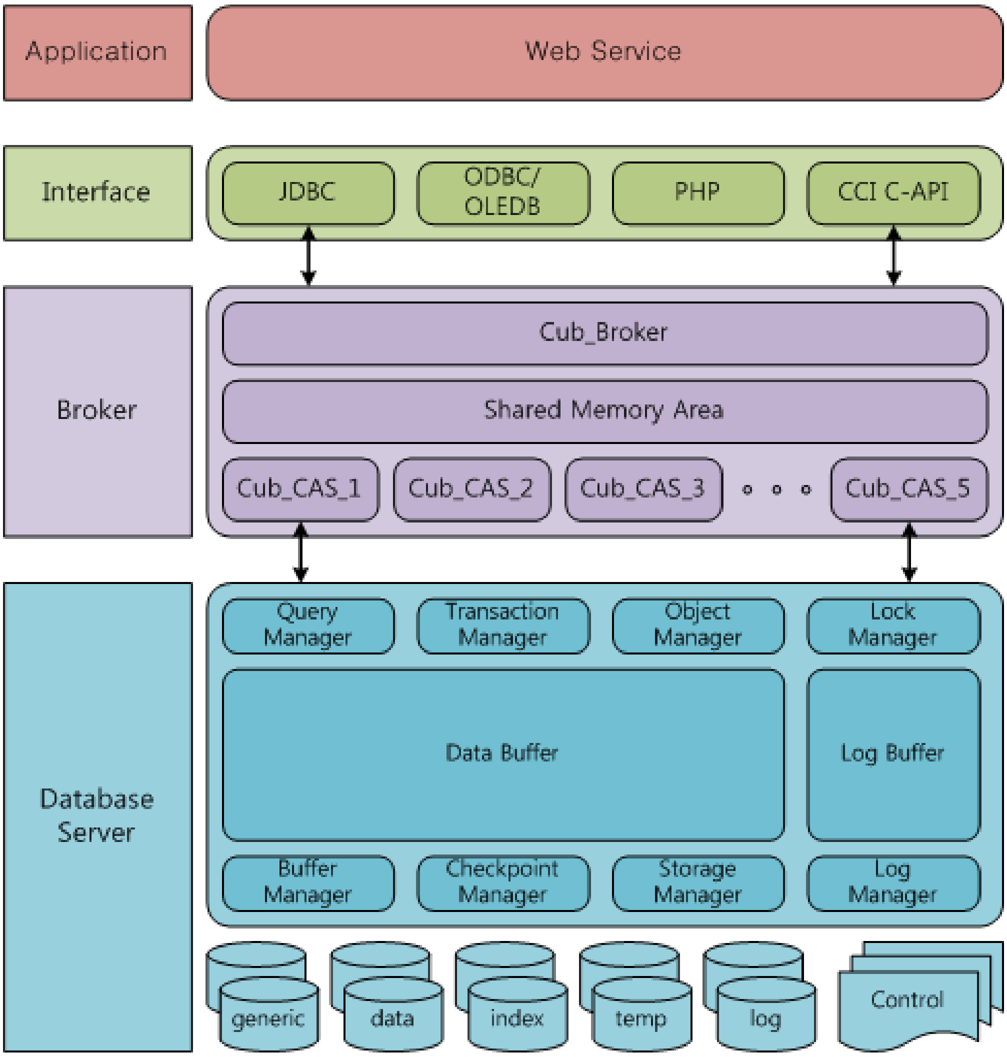
- System Architecture
- Features of CUBRID

System Architecture

System Architecture

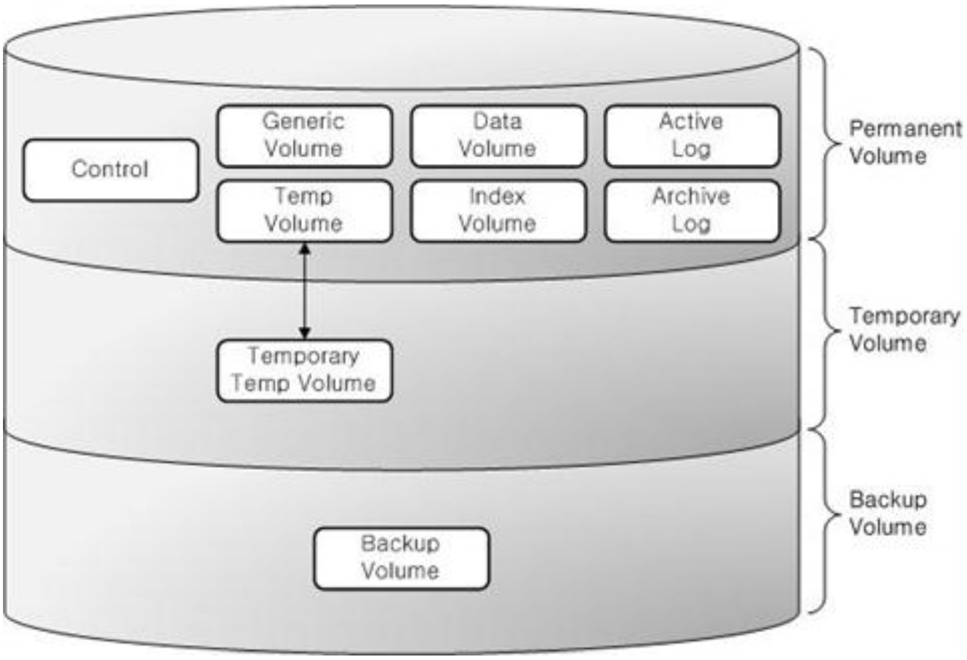
CUBRID is an object-relational database management system (DBMS) consisting of the Database Server, the Broker, and the CUBRID Manager.

- As the core component of the CUBRID Database Management System, the Database Server stores and manages data in multi-threaded client/server architecture. The Database Server processes the queries requested by users and manages objects in the database. The CUBRID Database Server provides seamless transactions using locking and logging methods even when multiple users use the database at the same time. It also supports database backup and restore for the operation.
- The Broker is a CUBRID-specific middleware that relays the communication between the Database Server and external applications. It provides functions including connection pooling, monitoring, and log tracing and analysis.
- The CUBRID Manager is a GUI tool that allows users to remotely manage the database and the Broker. It also provides the Query Editor, a convenient tool that allows users to execute SQL queries on the Database Server. See CUBRID Manager for more information on the CUBRID Manager.



Database Volume Structure

The following diagram illustrates the CUBRID database volume structure. As you can see, the database is divided into three volumes: permanent, temporary and backup. This chapter will examine each volume and its characteristics.



Permanent Volume

Permanent volume is a database volume that exists permanently once it is created. Its types include generic, data, temp, index, control, active log and archive log.

Generic Volume

For efficient management, the volume type to be added to the database can be specified as one of the following: data, temp or index. If data usage is not specified, it is specified as a generic volume.

Data Volume

Data volume is a volume for storing data such as instances, tables and multimedia data.

Temp Volume

Temporary volume is a volume used temporarily for query processing and sorting. However, the temporary volume is not a volume where the storage is created and destroyed temporarily, but one of the permanent volumes with permanent spaces where the data is stored and destroyed temporarily. Therefore, the data in the temporary volume space gets initialized when CUBRID restarts without leaving any log info.

Index Volume

Index volume is a volume that holds the index information for fast query processing or integrity constraint checks.

Control File

The control file contains the volume, backup and log information in the database.

- **Volume Information** : The information that includes names, locations and internal volume identifiers of all the volumes in the database. When the database restarts, the CUBRID reads the volume information control file. It records a new entry to that file when a new database volume is added.
- **Backup Information** : Locations of all the backups for data, index, and generic volumes are recorded to a backup information control file. This control file is maintained where the log files are managed.

- **Log Information** : This information contains names of all active and archive logs. With the log information control file, you can verify the archive log information. The log information control file is created and managed at the same location as the log files.

Control files include the information about locations of database volumes, backups and logs. Since these files will be read when the database restarts, users must not modify them arbitrarily.

Active Log

Active log is a log that contains recent changes to the database. If a problem occurs, you can use active and archive logs to restore the database completely up to the point of the last commit before the occurrence of the fault.

Archive Log

Archive log is a volume to store logs continuously created after exhausting available active log space that contains recent changes. The archive log volume will be generated only after exhausting available active log volume space, just as the temporary temp volume will be generated after exhausting available permanent temp volume space. However, unlike the temporary temp volume, the archive log volume is not destroyed automatically when the server process terminates. Therefore, a **DBA** needs to manually delete necessary archive logs. The archive log volume can be deleted anytime by **DBA**.

Temporary Volume

Temporary volume has the opposite meaning to the permanent volume. That is, the temporary volume is a storage created only when the accumulated data exceeds the space specified by the user as the permanent volume. The temporary volume is destroyed when the server process terminates. One of such volumes created or destroyed temporarily is the temporary temp volume.

Temporary Temp Volume

Temporary temp volume is a temporary volume created temporarily by the system after exhausting the space specified as the permanent temp volume, whereas the temporary volume belongs to the permanent volume with the permanent space specified. Therefore, the **DBA** should consider the database operations first to free up the permanent temp volume with an appropriate size.

The temporary temp volume is created to free up disk space needed for joining/sorting or index creation. Examples of such large-scale queries of creating temporary volumn are: 1) SQL statements with a **GROUP BY** or **ORDER BY**, 2) SQL statements that contain coordinated subqueries, 3) join queries that perform sort-merge joins, and 4) a **CREATE INDEX** statement.

- **File name of the temporary temp volume** : The file name of the temporary temp volume of CUBRID has the format of *db_name*_**t***num*, where *db_name* is the database name and *num* is the volume identifier. The volume identifier is decremented by 1 from 32766.
- **Configuring the temporary temp volume size** : The number of temporary temp volumes to be created is determined by the system depending on the space size needed for processing transactions. However, users can limit the temporary temp volume size by configuring the **temp_file_max_size_in_pages** parameter value in the database parameter configuration file (**cubrid.conf**). If the **temp_file_max_size_in_pages** parameter value is configured to 0, the temporary temp volume will not be created even after exhausting the permanent temp volume.
- **Configuring save location of the temporary temp volume** : By default, the temporary temp volume is created where the first database volume was created. However, you can specify a different directory to save the temporary temp volume by configuring the **temp_volume_path** parameter value.
- **Deleting the temporary temp volume** : The temporary temp volume exists temporarily only when the database is running. You must not delete the temporary temp volume while the server is running. The temporary temp volume is deleted when the client connection with the server is terminated while the database is running in a standalone mode. On the other hand, the temporary temp volume is deleted when the server process is normally terminated by the **cubrid** utility while the database is running in a client/server mode. If the database server is abnormally terminated, the temporary temp volume will be deleted when the server restarts.

Backup Volume

Backup volume is a database snapshot; based on such backup and log volumes, you can restore transactions to a certain point of time.

You can use the **cubrid backupdb** utility to copy all the data needed for database restore, or configure the **backup_volume_max_size_bytes** parameter value in the database configuration file (**cubrid.conf**) to adjust the backup volume size.

Database Server

Database Server Process

Each database has a server process. The server process is the core component of the CUBRID Database Server, and handles a user's requests by directly accessing database and log files. The client process connects to the server process via TCP/IP communication. Each server process creates threads to handle requests by multiple client processes. System parameters can be configured for each database, that is, for each server process. The server process can connect to as many client processes as specified by the **max_clients** parameter value.

Master Process

The master process is a broker process that allows the client process to connect to and communicate with the server process. One master process runs for each host. (To be exact, one master process exists for each connection port number specified in the **cubrid.conf** system parameter file.) While the master process listens on the TCP/IP port specified, the client process connects to the master process through that port. The master process changes a socket to server port so that the server process can handle connection.

Execution Mode

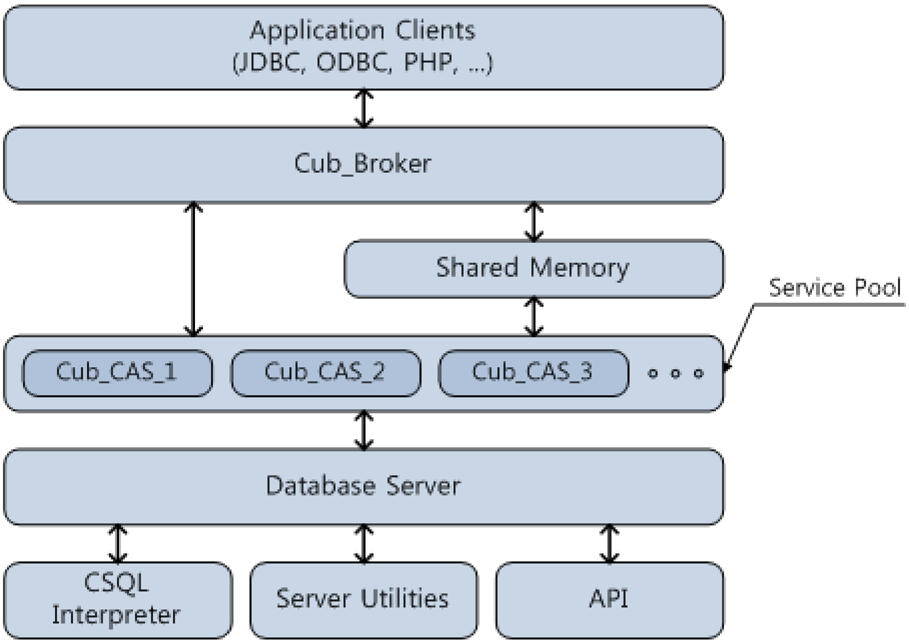
All CUBRID utilities except the server process have two execution modes: client/server mode and standalone mode.

- In client/server mode, the utilities operate as a client process and connect to the server process.
- In the standalone mode, a process is shared between a client and a server, wherein a master process is not required and a database can be directly accessed.

For example, a database creation or a restore utility runs in the standalone mode so it can use the database exclusively by denying the access by multiple users. Another example is that the CSQL Interpreter can either connect to the server process in client/server mode or execute SQL statements by accessing the database in the standalone mode. Note that one database cannot be accessed simultaneously by a server process and a standalone program.

Broker

The Broker is a middleware that allows various application clients to connect to the Database Server. As shown below, the CUBRID system, which includes the Broker, has multi-layered architecture consisting of application clients, **cub_broker**, **cub_cas** and the Database Server.



Application Client

The interfaces that can be used in application clients include C-API, ODBC, JDBC, PHP, Tcl/Tk, Python, and Ruby, OLEDB, and ADO.NET.

cub_cas

cub_cas (CUBRID Common Application Server) acts as a common application server used by all the application clients that request connections. cub_cas also acts as the Database Server's client and provides the connection to the Database Server upon the client's request. The number of cub_cas(s) running in the service pool can be specified in the configuration file, and this number is dynamically adjusted by cub_broker.

cub_cas is a program linked to the CUBRID Database Server's client library and functions as a client module in the server process. In the client module, tasks such as query parsing, optimization, execution plan creation are performed.

cub_broker

cub_broker relays the connection between the application client and the cub_cas. That is, when an application client requests access, the **cub_broker** checks the status of the **cub_cas** through the shared memory, and then delivers the request to an accessible **cub_cas**. It then returns the processing results of the request from the **cub_cas** to the application client.

The **cub_broker** also manages the server load by adjusting the number of **cub_cas**(s) in the service pool and monitors and manages the status of the **cub_cas**. If the **cub_broker** delivers the request to **cub_cas** but the connection to **cub_cas** 1 fails because of an abnormal termination, it sends an error message about the connection failure to the application client and restarts **cub_cas** 1. Restarted **cub_cas** 1 is now in a normal stand-by mode, and will be reconnected by a new request from a new application client.

Shared Memory

The status information of the **cub_cas** is saved in the shared memory, and the **cub_broker** refers to this information to relay the connection to the application client. With the status information saved in the shared memory, the system manager can identify which task the **cub_cas** is currently performing or which application client's request is currently being processed.

Interface Module

CUBRID provides various Application Programming Interfaces (APIs). The following APIs are supported by CUBRID. CUBRID also provides interfaces modules for each interface.

- JDBC : A standard API used to create database applications in Java. CUBRID provides the JDBC driver as an interface module.
- ODBC : A standard API used to create database applications in Windows. CUBRID provides the ODBC driver as an interface module.
- OLE DB : An API used to create COM-based database applications in Windows. CUBRID provides the OLE DB provider as an interface module.
- PHP : CUBIRD provides a PHP interface module to create database applications in the PHP environment. The PHP module is based on the CCI library.
- CCI : CCI is a C language interface provided by CUBRID. The interface module is provided as a C library.

All interface modules access the Database Server through the Broker. The Broker is a middleware that allows various application clients to connect to the Database Server. When it receives a request from an interface module, it calls a native C API provided by the Database Server's client library.

CUBRID Features

Transaction Support

CUBRID supports the following features to completely ensure the atomicity, consistency, isolation and durability in transactions.

- Supporting commit, rollback, savepoint per transaction
- Ensuring transaction consistency in the event of system or database failure
- Ensuring transaction consistency between replications
- Supporting multiple granularity locking of databases, tables and records
- Resolving deadlocks automatically
- Supporting distributed transactions (two-phase commit)

Database Backup and Restore

A database backup is the process of copying CUBRID database volumes, control files and log files; a database restore is the process of restoring the database to a certain point in time using backup files, active logs and archive logs copied by the backup process. For a restore, there must be the same operating system and the same version of CUBRID installed as in the backup environment.
The backup methods which CUBRID supports include online, offline and incremental backups; the restore methods include restore using incremental backups as well as partial and full restore.

Table Partitioning

Partitioning is a method by which a table is divided into multiple independent logical units. Each logical unit is called a partition, and each partition is divided into a different physical space. This will lead performance improvement by only allowing access to the partition when retrieving records. CUBRID provides three partitioning methods:

- Range partitioning: Divides a table based on the range of a column value
- Hash partitioning: Divides a table based on the hash value of a column
- List partitioning: Divides a table based on the column value list

HA Functionalities

High Availability (HA) refers to ability to minimize system down time while continuing normal operation of server in the event of hareware, software, or network failure; that is, the CUBRID HA is functionality that is applied to CUBRID. The CUBRID HA feature has a shared-nothing architecture. The CUBRID performs realtime monitoring for system and CUBRID state with the CUBRID Heartbeat. Then in case of system failure, it automatically performs failover. It follows the two steps below to synchronize data from the master to the slave database servers.

- A transaction log multiplication step where the transaction log created in the database server is replicated in real time to another node
- A transaction log reflection step where data is applied to the slave database server through the analysis of the transaction log being replicated in real time

Replication

Replication is a technique that duplicates data from one database to other databases to improve performance and increase server availability by distributing requests from applications that use the same data into multiple databases. Currently, CUBRID supports replication only on Linux and UNIX. The CUBRID replication system runs based on transaction logs, and it provides real-time replication and ensures transaction consistency/schema independence of the slave database. Additionally, it offers a feature for a master database to be minimally affected by replication. The replication feature consists of the following components:

- Master database: The source database that becomes the target to be replicated. All operations including a read and write operations are performed in this database. Since the replication is performed asynchronously, there will be no effect on the master database administration. Replication logs are created in the master server, which are sent to the slave server via the replication server and the replication agent.

- Slave database: The database replicated from the source database. It allows a client a read operation only in the salve database. If a write operation occurs in the master database, the transaction is automatically replicated to multiple slave databases, so read operations can be distributed on multiple databases.
- Distribution database: Saves the information about the master and the slave databases. It ensures transaction consistency and effects replication to be distributed.
- Replication server: The replication server runs on the master system and transfers a transaction log in the master database to the replication agent.
- Replication agent: The replication agent is a process that runs on the slave system and performs the actual replication tasks by analyzing and applying the transferred replication log to the slave database server.

Java stored procedure

A stored procedure is a method to decrease the complexity of applications and to improve the reusability, security and performance through the separation of database logic and middleware logic. A stored procedure is written in Java (generic language), and provides Java stored procedures running on the Java Virtual Machine (JVM). To execute Java stored procedures in CUBRID, the following steps should be performed:

- Install and configure the Java Virtual Machine
- Create Java source files
- Compile the files and load Java resources
- Publish the loaded Java classes so they can be called from the database
- Call the Java stored procedures

Click Counter

In the Web, it is a common scenario to count and keep the number of clicks to the database in order to record retrieval history.

The above scenario is generally implemented by using the **SELECT** and **UPDATE** statements; SELECT retrieves the data and UPDATE increases the number of clicks for the retrieved queries.

This approach can cause significant performance degradation due to increased lock contention for **UPDATE** when a number of **SELECT** statements are executed against the same data.

To address this issue, CUBRID introduces the new concept of the click counter that will support optimized features in the Web in terms of usability and performance, and provides the **INCR** function and the **WITH INCREMENT FOR** statement.

Extending the Relational Data Model

Collection

For the relational data model, it is not allowed that a single column has multiple values. In CUBRID, however, you can create a column with several values. For this purpose, collection data types are provided in CUBRID. The collection data type is mainly divided into **SET**, **MULTISET** and **LIST**; the types are distinguished by duplicated availability and order.

- **SET** : A collection type that does not allow the duplication of elements. Elements are stored without duplication after being sorted regardless of their order of entry.
- **MULTISET** : A collection type that allows the duplication of elements. The order of entry is not considered.
- **LIST** : A collection type that allows the duplication of elements. Unlike with **SET** and **MULTISET**, the order of entry is maintained.

Inheritance

Inheritance is a concept to reuse columns and methods of a parent table in those of child tables. CUBRID supports reusability through inheritance. By using inheritance provided by CUBRID, you can create a parent table with some common columns and then create child tables inherited from the parent table with some unique columns added. In this way, you can create a database model which can minimize the number of columns.

Composition

In a relational database, the reference relationship between tables is defined as a foreign key. If the foreign key consists of multiple columns or the size of the key is significantly large, the performance of join operations between tables will be degraded. However, CUBRID allows the direct use of the physical address (OID) where the records of the referred table are located, so you can define the reference relationship between tables without using join operations.

That is, in an object-oriented database, you can create a composition relation where one record has a reference value to another by using the column displayed in the referred table as a domain (type), instead of referring to the primary key column from the referred table.

Getting Started with CUBRID

This chapter contains useful information on starting CUBRID such as how to install and run CUBRID; also it provides instructions on how to use the CSQL Interpreter and CUBRID Manager. This chapter also includes examples on how to write application programs using JDBC, PHP, ODBC and CCI, etc.

This chapter covers the following topics :

- Installing and Running CUBRID
- Before You Start CUBRID
- Using the CSQL Interpreter
- Using the CUBRID Manager
- Writing Programs using JDBC
- Writing Programs using PHP
- Writing Programs using ODBC and ASP
- Writing CCI Programs

Installing and Running

Installing and Running on Linux

Details to Check when Installing

Check the following before installing CUBRID for Linux.

Category	Description
Operating System	Only supports glibc 2.3.4 or later. The glibc version can be checked as follows: rpm -q glibc
64-bit	Since version 2008 R2.0, CUBRID supports both 32-bit and 64-bit Linux. You can check the version as follows: % uname -a Linux host_name 2.6.18-53.1.14.el5xen #1 SMP Wed Mar 5 12:08:17 EST 2008 x86_64 x86_64 x86_64 GNU/Linux Make sure to install the CUBRID 32-bit version on 32-bit Linux and the CUBRID 64-bit version on 64-bit Linux. The followings are the libraries that should be added. Curses Library (rpm -q ncurses) gcrypt Library (rpm -q libgcrypt stdc++ Library (rpm -q libstdc++)

Installing CUBRID

The installation program consists of shell scripts that contain binary; thus it can be installed automatically. The following example shows how to install CUBRID with the "CUBRID-8.3.0.0312-linux.x86_64.sh" file on the Linux.

```
[cub_user@cubrid ~]$ sh CUBRID-8.3.1.0168-linux.x86_64.sh
Do you agree to the above license terms? (yes or no) : yes
Do you want to install this software(CUBRID) to the default(/home1/cub_user/CUBRID)
directory? (yes or no) [Default: yes] : yes
Install CUBRID to '/home1/cub_user/CUBRID' ...
In case a different version of the CUBRID product is being used in other machines, please
note that the CUBRID 2008 R3.0 servers are only compatible with the CUBRID 2008 R3.0
clients and vice versa.
Do you want to continue? (yes or no) [Default: yes] : yes
Copying old .cubrid.sh to .cubrid.sh.bak ...

CUBRID has been successfully installed.

demodb has been successfully created.

If you want to use CUBRID, run the following commands
  % . /home1/cub_user/.cubrid.sh
  % cubrid service start
```

As shown in the example above, after installing the downloaded file (CUBRID-8.3.0.0312-linux.x86_64.sh), the CUBRID related environment variables must be set in order to use the CUBRID database. Such setting has been made automatically when logging in the concerned terminal. Therefore there is no need to re-set after the first installation.

```
[cub_user@cubrid ~]$ . /home1/cub_user/.cubrid.sh
```

After the CUBRID Manager is installed, you can start the CUBRID Manager server and Broker as follows:

```
[cub_user@cubrid ~]$ cubrid service start
```

After starting the CUBRID service, if you wish to check whether the service was properly started, then check whether the cub_* processes have been started with grep (as shown below).

```
[cub_user@cubrid ~]$ ps -ef | grep cub_
cub_user 15200 1 0 18:57 ? 00:00:00 cub_master
cub_user 15205 1 0 18:57 pts/17 00:00:00 cub_broker
cub_user 15210 1 0 18:57 pts/17 00:00:00 query_editor_cub_cas_1
cub_user 15211 1 0 18:57 pts/17 00:00:00 query_editor_cub_cas_2
cub_user 15212 1 0 18:57 pts/17 00:00:00 query_editor_cub_cas_3
cub_user 15213 1 0 18:57 pts/17 00:00:00 query_editor_cub_cas_4
cub_user 15214 1 0 18:57 pts/17 00:00:00 query_editor_cub_cas_5
cub_user 15217 1 0 18:57 pts/17 00:00:00 cub_broker
cub_user 15222 1 0 18:57 pts/17 00:00:00 broker1_cub_cas_1
cub_user 15223 1 0 18:57 pts/17 00:00:00 broker1_cub_cas_2
cub_user 15224 1 0 18:57 pts/17 00:00:00 broker1_cub_cas_3
cub_user 15225 1 0 18:57 pts/17 00:00:00 broker1_cub_cas_4
cub_user 15226 1 0 18:57 pts/17 00:00:00 broker1_cub_cas_5
cub_user 15229 1 0 18:57 ? 00:00:00 cub_auto start
cub_user 15232 1 0 18:57 ? 00:00:00 cub_js start
```

Installing CUBRID (rpm File)

You can install CUBRID by using rpm file that is created on CentOS5. The way of installing and uninstalling CUBRID is the same as that of using general rpm utility. While CUBRID is being installed, a new system group (cubrid) and a user account (cubrid) are created. After installation is complete, you should log in with a cubrid user account to start a CUBRID service.

```
$ rpm -Uvh CUBRID-8.3.1.0618-el5.x86_64.rpm
```

When rmp is executed, CUBRID is installed in the cubrid home directory (/opt/cubrid) and related configuration file (cubrid.[c]sh) is installed in the /etc/profile.d directory. Note that demodb is not automatically installed. Therefore, you must executed /opt/cubrid/demo/make_cubrid_demo.sh. When installation is complete, enter the code below to start CUBRID.

```
[cubrid@cubrid ~]$ cubrid service start
```

Note You must check RPM dependency when installing with RPM. If you ignore (--nodeps) dependency, it may not be executed.

Note Even if you remove RPM, user accounts and databases that are created after installing, you must remove it manually, if needed.

CUBRID Upgrade

When you specify an installation directory where the previous version of CUBRID is already installed, a message which asks to overwrite files in the directory will appear. Entering **no** will stop the installation.

```
Directory '/home1/cub_user/CUBRID' exist!
If a CUBRID service is running on this directory, it may be terminated abnormally.
And if you don't have right access permission on this directory(subdirectories or files),
install operation will be failed.
Overwrite anyway? (yes or no) [Default: no] : yes
```

Choose whether to overwrite the existing configuration files during the CUBRID installation. Entering **yes** will overwrite and back up them as extension .bak files.

```
The configuration file (.conf or .pass) already exists. Do you want to overwrite it? (yes
or no) : yes
```

Environment Configuration

To modify the environment such as service ports etc, edit the parameters of a configuration file located in the **$CUBRID/conf** directory. See Environment Configuration for more information.

Note You must check the dependency when you attempt to install using RPM. Installation may not succeed if the dependency is ignored (--nodeps).

Installing and Running on Windows

Details to Check when Install

CUBRID 2008 R2.0 supports both 32-bit and 64-bit Windows. You can check the version by selecting [My Computer] > [System Properties]. Make sure to install the CUBRID 32-bit version on 32-bit Windows and the CUBRID 64-bit version on 64-bit Windows.

The CUBRID Manager and Java stored procedures require the Java Runtime Environment (JRE) version 1.6 or later.

You must install the Microsoft Visual C++ 2008 Redistributable Package and the .NET Framework version 2.0 or later on Windows as for Windows. If they are not first installed, you will get error messages that "CUBRIDService cannot be registered into Windows service" and "CUBRID_Service_Tray.exe is not started".

Selecting Install Type

- **Full Installation** : If you select [Full Installation] in the CUBRID Manager installation wizard, CUBRID Server, CSQL (a command line tool), CUBRID Manager (a GUI management tool), CUBRID manual and interface drivers (OLEDB Provider, ODBC, JDBC, C API) are all installed.
- **Management Tool and Driver Installation** : If you select [Management Tool and Driver Installation] in the CUBRID Manager installation wizard, only CUBRID Manager, CUBRID manual and interface drivers (OLEDB Provider, ODBC, JDBC, C API) are installed. You can select this type of installation if development or operation is performed by remote connection to the computer in which the CUBRID database server is installed.

CUBRID Upgrade

To install a new version of CUBRID in an environment in which a previous version has already been installed, select [CUBRID Service Tray] > [Exit] from the menu to stop currently running services, and then remove the previous version of CUBRID. Note that when you are prompted with "Do you want to delete all the existing version of databases and the configuration files?" you must select "No" to protect the existing databases.

For more information on migrating a database from a previous version to a new version, see Migrating Database.

Environment Configuration

To change configuration such as service ports to meet the user environment, the parameter values of the files stated below should be changed in the **%CUBRID%\conf** directory.

File	Description
cm.conf	CUBRID Manager's configuration file; the port number 8001 is configured by default. Two port numbers are required to use CUBRID; a configured number and the number added by 1 are used. For example, 8001 is configured for connection, the port number 8001 and 8002 are reserved.
cubrid.conf	Server configuration file is used to set the following: database memory, the number of threads due to the number of concurrent users, connection port between the Broker and Server, etc. See cubrid_broker.conf Configuration File and Default Parameters for details.
cubrid_broker.conf	Broker configuration file; the port is used by the broker that is operated. The file is used to set the number of CAS, SQL LOGs, etc. The ports shown in drivers such as JDBCs are the concerned Broker's ports. See Parameter by Broker for details.

Configuring Environment Variable and Starting CUBRID

Configuring the Environment Variable

The following environment variables need to be set in order to use the CUBRID. The necessary environment variables are automatically set when the CUBRID system is installed or can be changed, as needed, by the user.

CUBRID Environment Variables

- CUBRID : The default environment variable that designates the location where the CUBRID is installed. This variable must be set accurately since all programs included in the CUBRID system uses this environment variable as reference.
- CUBRID_DATABASES : The environment variable that designates the location of the database location information file. The CUBRID system saves and manages the absolute path of database volumes that are used in the **$CUBRID_DATABASES/databases.txt** file. See databases.txt file.
- CUBRID_LANG : The environment variable that designates the language that will be used in the CUBRID system. Currently, CUBRID provides English (en_US) and Korean (ko_KR.euckr and ko_KR.utf8). it is not a mandatory setting. Therefore, if the variable has not been set, then refer to the LANG environment variable or use en_US, which is the default value. See Language Setting.

The above mentioned environment variables are set when the CUBRID is installed. However, the following commands can be used to verify the setting.

For Linux :

```
% printenv CUBRID
% printenv CUBRID_DATABASES
% printenv CUBRID_LANG
```

In Windows :

```
C:\> set CUBRID
```

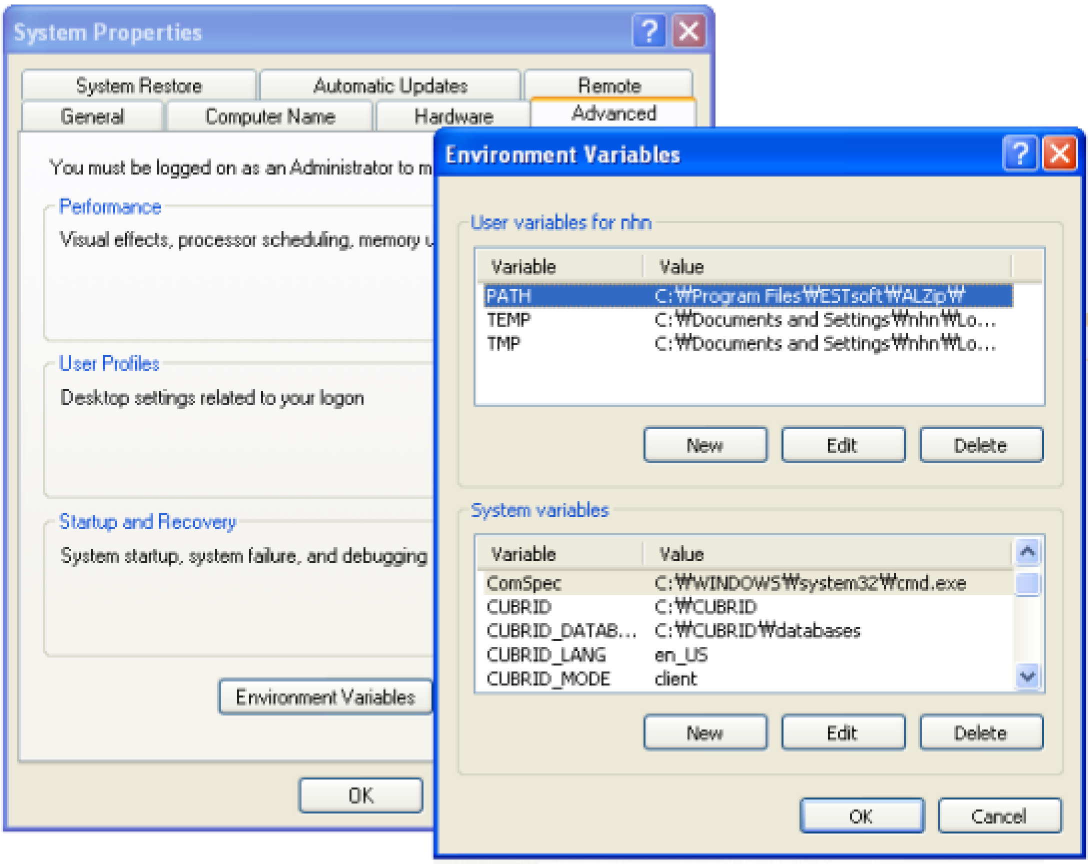
OS Environment and Java Environment Variables

- PATH : In the Linux environment, the directory **$CUBRID/bin**, which includes a CUBRID system executable file, must be included in the PATH environment variable.
- LD_LIBRARY_PATH : In the Linux environment, **$CUBRID/lib**, which is the CUBRID system's dynamic library file (libjvm.so), must be included in the **LD_LIBRARY_PATH** (or **SHLIB_PATH** or **LIBPATH**) environment variable.
- Path : In the Windows environment, the **$CUBRID/bin**, which is a directory that contains CUBRID system's execution file, must be included in the **Path** environment variable.
- JAVA_HOME : To use the Java stored procedure in the CUBRID system, the Java Virtual Machine (JVM) version 1.6 or later must be installed, and the **JAVA_HOME** environment variable must designate the concerned directory. See the Environment Configuration for Java Stored Functions/Procedures.

Configuring the Environment Variable

For Windows

If the CUBRID system has been installed in the Windows environment, then the installation program automatically sets the necessary environment variable. Select [Systems Properties] in [My Computer] and select the [Advanced] tab. Click the [Environment Variable] button and check the setting in the [System Variable]. The settings can be changed by clicking on the [Edit] button. See the Windows help for more information on how to change the environment variable in the Windows environment.



You must be logged on as an Administrator to m

Visual effects, processor scheduling, memory u

Desktop settings related to your logon

System startup, system failure, and debugging

Variable	Value
PATH	C:₩Program Files₩ESTsoft₩ALZip₩
TEMP	C:₩Documents and Settings₩nhn₩Lo...
TMP	C:₩Documents and Settings₩nhn₩Lo...

Variable	Value
ComSpec	C:₩WINDOWS₩system32₩cmd.exe
CUBRID	C:₩CUBRID
CUBRID_DATAB...	C:₩CUBRID₩databases
CUBRID_LANG	en_US
CUBRID_MODE	client

For Linux

If the CUBRID system has been installed in the Linux environment, the installation program automatically creates the **.cubrid.sh** or **.cubrid.csh** file and makes configurations so that the files are automatically called from the installation account's shell log-in script. The following is the .cubrid.sh environment variable setting file that was created in an environment that uses **sh**, **bash**, etc.

```
CUBRID=/home1/cub_user/CUBRID
CUBRID_DATABASES=/home1/cub_user/CUBRID/databases
CUBRID_LANG=en_US
ld_lib_path=`printenv LD_LIBRARY_PATH`
if [ "$ld_lib_path" = "" ]
then
LD_LIBRARY_PATH=$CUBRID/lib
else
LD_LIBRARY_PATH=$CUBRID/lib:$LD_LIBRARY_PATH
fi
SHLIB_PATH=$LD_LIBRARY_PATH
LIBPATH=$LD_LIBRARY_PATH
PATH=$CUBRID/bin:$CUBRID/cubridmanager/:$PATH
export CUBRID
export CUBRID_DATABASES
export CUBRID_LANG
export LD_LIBRARY_PATH
export SHLIB_PATH
export LIBPATH
export PATH
```

Language Setting

The language that will be used in the CUBRID DBMS can be designated with the **CUBRID_LANG** environment variable. The following are values that can currently be set in the **CUBRID_LANG** environment variable.

- **en_US** : English (Default value)
- **ko_KR.euckr** : Korean EUC-KR encoding
- **ko_KR.utf8** : Korean utf-8 encoding

The language setting in the CUBRID system does not represent the character sets of data that is saved. In other words, even though the **CUBRID_LANG** is set to ko_KR.utf8, the data may not be changed to the concerned encoding. CUBRID's language setting will have an influence on the message printed from the program and will impact the date/time data type constant displayed throughout the use of the program.

If the **CUBRID_LANG** is not set, then the value of the LANG environment variable will be used. If the set value does not support the **CUBRID_LANG** or **LANG** value, then the action will be made as if the setting has been made to en_US, the default value.

Starting the CUBRID Service

Configure environment variables and language, and then start the CUBRID service. For more information on configuring environment variables and language, see Registering Services or Starting and Stopping Services.

Shell Command

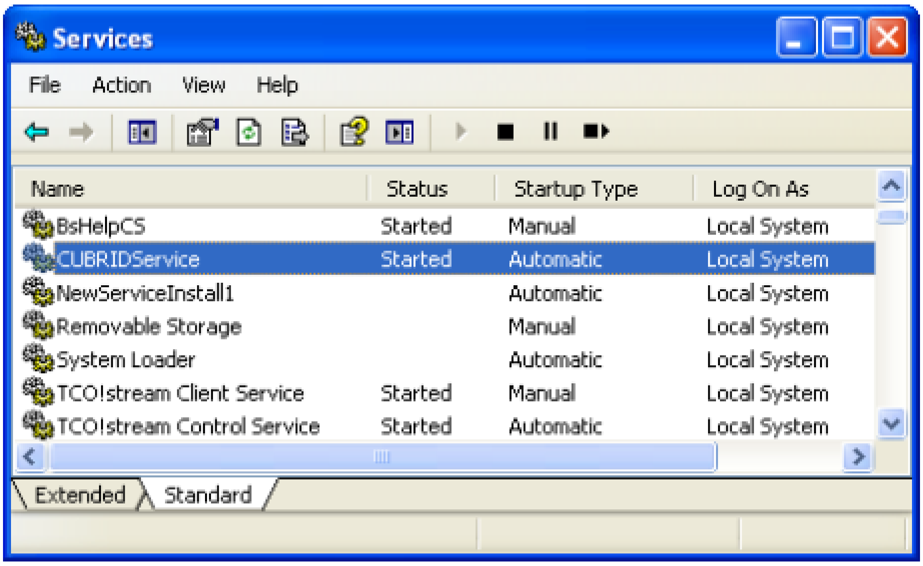
The following shell command can be used to start the CUBRID service and the demodb included in the installation package.

```
% cubrid service start

@ cubrid master start
++ cubrid master start: success
@ cubrid broker start
++ cubrid broker start: success
@ cubrid manager server start
++ cubrid manager server start: success

% cubrid server start demodb
@ cubrid server start: demodb

This may take a long time depending on the amount of recovery works to do.

CUBRID 2008 R3.0

++ cubrid server start: success

@ cubrid server status
Server demodb (rel 8.3, pid 31322)
```

CUBRIDService or CUBRID Service Tray

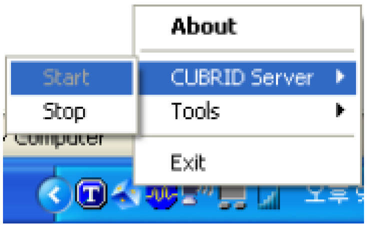
On the Windows environment, you can start or stop a service as follows:

- Go to [Control Panel] > [Performance and Maintenance] > [Administrator Tools] > [Services] and select the CUBRIDService to start or stop the service.



Name	Status	Startup Type	Log On As
BsHelpCS	Started	Manual	Local System
CUBRIDService	Started	Automatic	Local System
NewServiceInstall1		Automatic	Local System
Removable Storage		Manual	Local System
System Loader		Automatic	Local System
TCO!stream Client Service	Started	Manual	Local System
TCO!stream Control Service	Started	Automatic	Local System

- Go to [CUBRID Server] > [Start] to start CUBRID or go to [CUBRID Server] > [Stop] to stop CUBRID after you select CUBRID Service Tray on the system tray. If you click [Exit] while CUBRID is running, all the services and process in the server are stopped.



Note An administrator level (SYSTEM) authorization is required to start/stop CUBRID processes through the CUBRID Service tray; a login level user authorization is required to start/stop them with shell commands. If you cannot control the CUBRID processes on the Windows environment, log in with an administrator or start/stop them through CUBRID Service tray. When all processes of CUBRID Server stops, an icon on the CUBRID Service tray turns out red.

CSQL Interpreter

Starting the CSQL Interpreter

The CSQL Interpreter is a program used in CUBRID. The entered SQL statements and results can be saved in the file for later use. For more information Introduction to the CSQL Interpreter and CSQL Execution Mode.

CUBRID offers the "CUBRID Manager" program, a convenient GUI program. All SQL can be executed and the results can be viewed from the CUBRID Manager's query editor. For more information, see Query Editor Structure.

We recommend that users use the CUBRID Manager rather than the CSQL Interpreter, which is in command mode, in the Windows environment. Therefore, in this section, we will provide information on using the CSQL Interpreter in the Linux environment.

Starting the CSQL Interpreter

The CSQL program can be started in the shell as shown below.

```
% csql demodb
          CUBRID SQL Interpreter
Type ';help' for help messages.
csql> ;help
=== <Help: Session Command Summary> ===
    All session commands should be prefixed by ';' and only blanks/tabs
    can precede the prefix. Capitalized characters represent the minimum
    abbreviation that should be entered to execute the specified command.

    ;REAd   [<file-name>]       - read a file into command buffer.
    ;Write  [<file-name>]       - (over)write command buffer into a file.
    ;APpend [<file-name>]       - append command buffer into a file.
    ;PRINT                      - print command buffer.
    ;SHELL                      - invoke shell.
    ;CD                         - change current working directory.
    ;EXit                       - exit program.

    ;CLear                      - clear command buffer.
    ;EDIT                       - invoke system editor with command buffer.
    ;List                       - display the content of command buffer.

    ;RUn                        - execute sql in command buffer.
    ;Xrun                       - execute sql in command buffer,
                                  and clears the command buffer.
    ;COmmit                     - commit the current transaction.
    ;ROllback                   - roll back the current transaction.
    ;AUtocommit [ON|OFF]        - enable/disable auto commit mode.
    ;REStart                    - restart database.

    ;SHELL_Cmd [shell-cmd]      - set default shell, editor, print and pager
    ;EDITOR_Cmd [editor-cmd]      command to new one, or display the current
    ;PRINT_Cmd [print-cmd]        one, respectively.

    ;DATE                       - display the local time, date.
    ;DATAbase                   - display the name of database being accessed.
    ;SChema class-name          - display schema information of a class.
    ;SYntax [sql-cmd-name]      - display syntax of a command.
    ;TRigger [`*'|trigger-name] - display trigger definition.
    ;Get system_parameter       - get the value of a system parameter.
    ;SEt system_parameter=value - set the value of a system parameter.
    ;PLan [simple|detail|off]   - show query execution plan.
    ;Info <command>             - display internal information.
    ;TIme [ON/OFF]              - enable/disable to display the query
                                  execution time.
    ;HISTORYList                - display list of the executed queries.
    ;HISTORYRead <history_num>  - read entry on the history number into command buffer.
    ;HElp                       - display this help message.
csql>
```

Executing the SQL with CSQL

After the CSQL has been executed, you can enter the SQL into the CSQL prompt. Each SQL statement must end with a semicolon (;). Multiple SQL statements can be entered in a single line. To execute the SQL statements entered, use the ;x session command. You can find the simple usage of the session commands with the ;help command. For more information, see Session Commands.

```
% csql demodb
CUBRID SQL Interpreter
Type `;help' for help messages.
csql> select * from olympic;
csql> ;x
=== <Result of SELECT Command in Line 1> ===

    host_year  host_nation           host_city             opening_date  closing
_date  mascot                slogan                introduction
================================================================================
=======================================================================
         2004  'Greece'              'Athens'              08/13/2004    08/29/2
004    'Athena  Phevos'      'Welcome Home'        'In 2004 the Olympic Games re
turned to Greece, the home of both the ancient Olympics and the first modern Olympics.

<omitted>
25 rows selected.

Current transaction has been committed.

1 command(s) successfully processed.
csql> SELECT SUM(n) FROM (SELECT gold FROM participant WHERE nation_code='KOR'
csql> UNION ALL SELECT silver FROM participant WHERE nation_code='JPN') AS t(n);
csql> ;x

=== <Result of SELECT Command in Line 1> ===

       sum(n)
=============
           82


1 rows selected.

Current transaction has been committed.

1 command(s) successfully processed.
csql> ;exit
```

Getting Started with the CUBRID Manager

Running CUBRID Manager

The CUBRID Manager is a GUI-based database management and query tool. It facilitates various management tasks and provides the Query Editor, allowing users to execute SQL statements against the connected database.

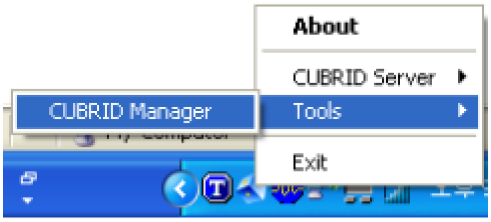
Note that the CUBRID Manager runs in the JAVA Runtime Environment (JRE) version 1.6 or later. You can download Java from http://java.sun.com.

For more information on the CUBRID Manager, see CUBRID Manager Architecture and Running CUBRID Manager Server.

Starting the CUBRID Manager

For Windows

- To start the CUBRID Manager, select [Tools] > [CUBRID Manager] in the CUBRID Service Tray.



For Linux

Enter the following command in the shell to start the CUBRID Manager.

- % cubridmanager
- % $CUBRID/cubridmanager/cubridmanager

Details to Check When an Error Occurs

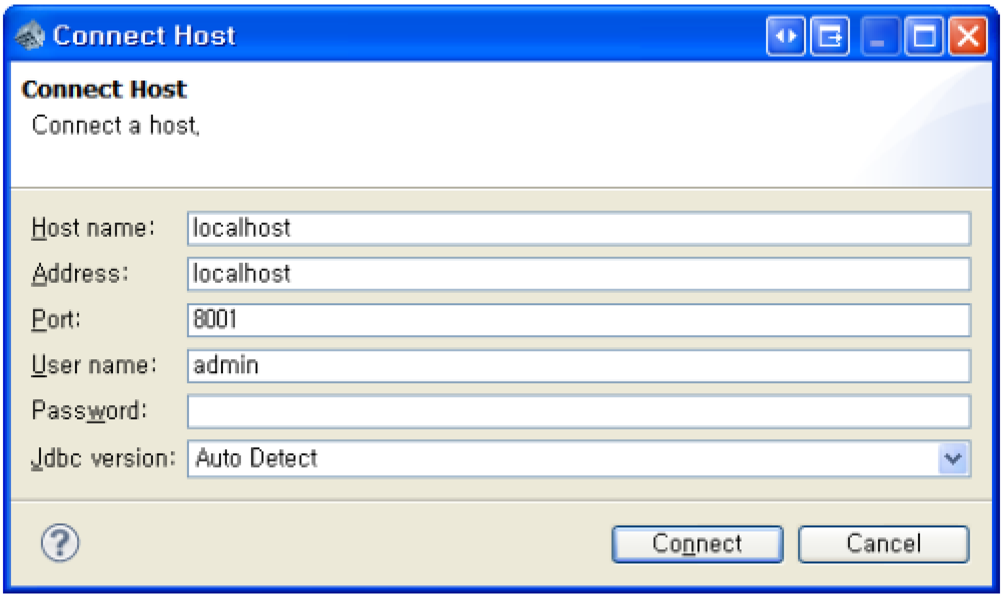
In case the CUBRID Manager is not running normally, an error message which means "Cannot connect to a server. Please check the configuration environment of the CUBRID Manager server and other connection." will be displayed. To resolve this problem, check the following list.

- Check whether the CUBRID Manager server is running.
- Open the configuration file of the CUBRID Manager server, and make sure that the value of the **cm_port** parameter is identical to the registered connection port. See Configuring CUBRID Manager Server .
- If a firewall is installed on the system where the Manager client is running, allow all connection ports to be accessed to the Manager client connection (**cm_port**, **cm_port** + 1). For example, if **cm_port** is 8001, the port 8002 must also be open.
- When the same operation is already being executed by the server, the message "Cannot execute the current operation because the previous operation is already running." is displayed. Then retry the operation.

Registering a Host

Register a host site to be connected to by the CUBRID Manager. That is, you are required to enter the information of the host where the CUBRID Manager server is located.

Right-click the mouse and then choose [Add host] to add a host in the host tab. The default value is localhost, the connection port is set to 8001, and the user ID and password are both set to "admin" by default. You must change the password of the **admin** ID to access the database; you cannot set it to "admin" which is the same as the value provided by default. See Default Host Information for more information.



Connect Host

Connect Host

Connect a host.

Host name: localhost

Address: localhost

Port: 8001

User name: admin

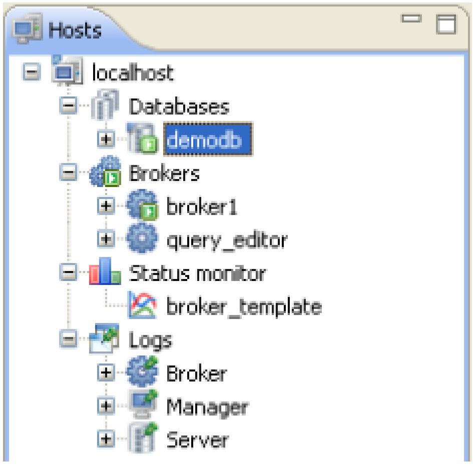
Password:

Jdbc version: Auto Detect

Connecting to a Host

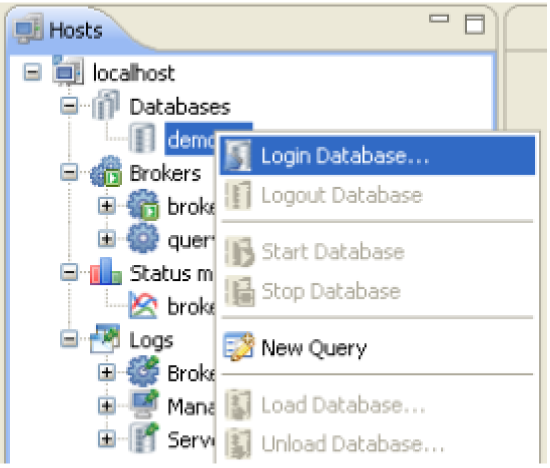
Double-click the registered host to connect to the host in the Host navigation tree; you can right-click the mouse and then choose [Connection]. The CUBRID Manager will be connected to the host after a user name is verified.

If succeed, the host icon is changed from to . The following figure shows a tree menu while the database (demodb) is running after host connection.

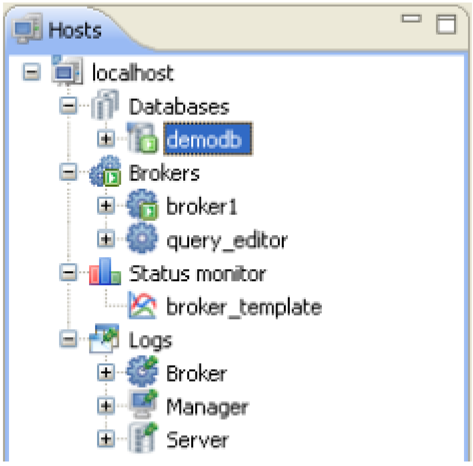


Connecting to a Database

You must log in to a database to run it in the CUBRID Manager. Right-click the mouse item to select [Login] or double-click it in the host navigation tree. In the [Login Database] dialog, check the account and password of the database to be connected. The default user name is **dba**; no password is required, so simply press the <ENTER> key.

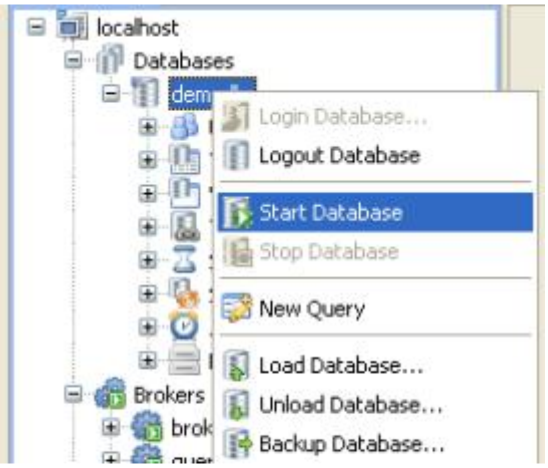


Once logged into the database, you will see database information such as users and tables as shown below. To start the database, right-click the database in the left navigation tree and then select [Start Database]. While the database is running, you can add a user or change the current password by right-clicking the [Users] node.



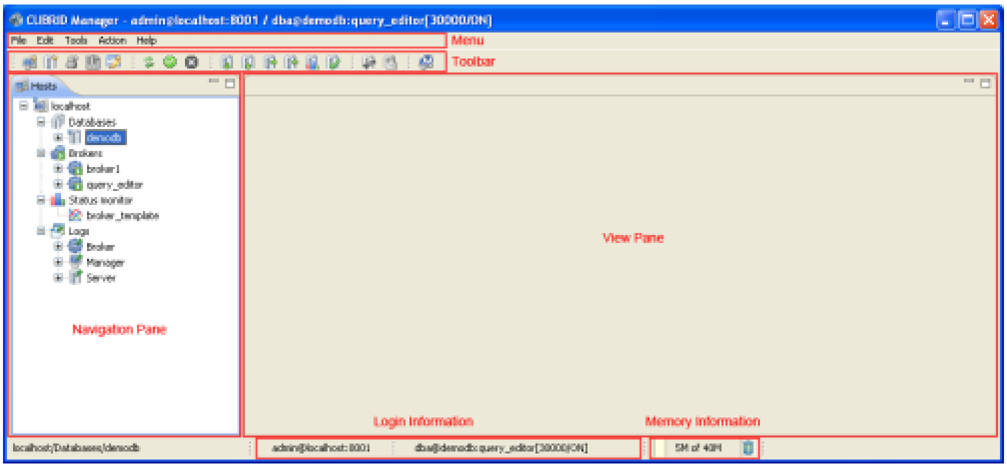
Starting a Database

You should start a database to execute queries in the CUBRID Manager. Select a database and then click [Start ▶] in the toolbar.



CUBRID Manager Layout

CUBRID Manager is a tool that allows users to manage the database with more convenience and efficiency. When you start the CUBRID Manager, you will see an interface window that consists of the menu bar, the toolbar, the navigation tree, View window, login information, and memory information. For more information, see RCP Application.



Using the Query Editor

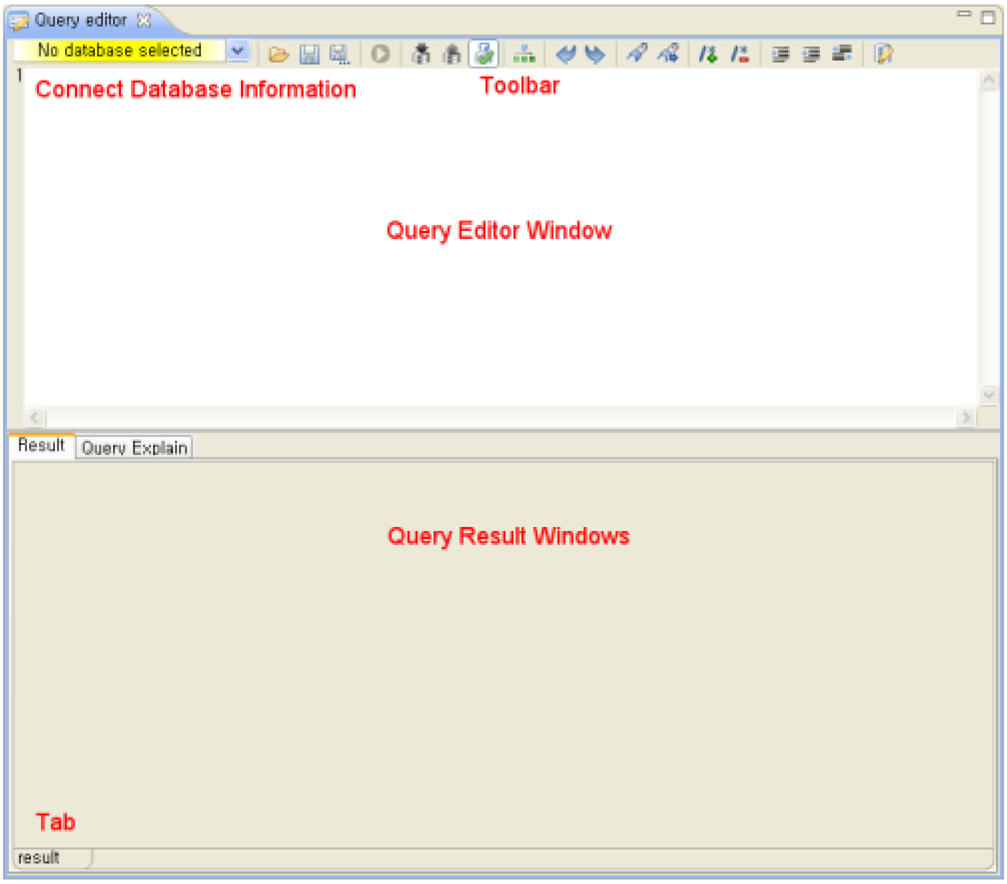
Starting the Query Editor

CUBRID Manager's Query Editor is a query tool that supports all **DML**, **DDL** and **DCL** statements, allowing users to edit and execute queries more easily.

To start the Editor, click [File] > [New query] in the menu, or click [New query editor] in the toolbar; right-click the mouse to choose [New query editor].

Query Editor Layout

The CUBRID Query Editor is composed of a tree pane on the upper, a query editor pane on the bottom. In the query editor pane, you can enter or edit queries; the query editor contains a toolbar which shows the most-frequently used menus. In the query result pane, you can view query results in a tab, and you can check the query execution time. For more information, see Query Editor Structure.


Query editor
No database selected
1
Connect Database Information
Toolbar
Query Editor Window
Result
Query Explain
Query Result Windows
Tab
result

Programming with JDBC

Setting up the JDBC Environment

System Requirements

- JDK 1.6 or later
- CUBRID 2008 R1.0 or later
- CUBRID JDBC Driver 2008 R1.0 or later

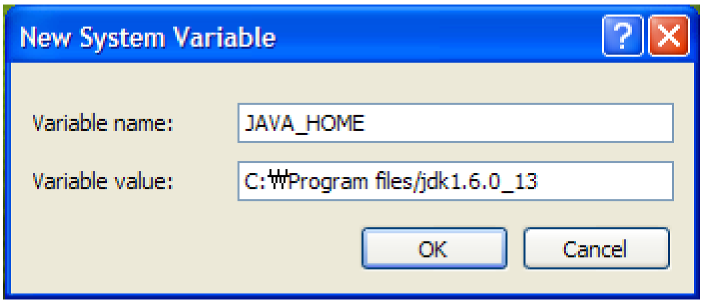
Installing and Configuring Java Environment

You must already have Java installed and the JAVA_HOME environment variable set on your system. To install Java, download it from the Java homepage (http://java.sun.com). For more information, see Environment Settings for Java Stored Functions/Procedures.

Configuring Envrionment Variables for Windows

After installing JAVA, double click [My Computer] and click [System Properties]. In the [Advanced] tab, click [Envrionment Variables]. The [Environment Variables] dialog will appear.

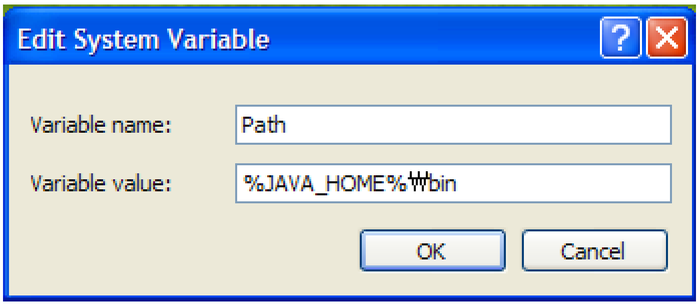
In the [System Variables], click [New]. Enter **JAVA_HOME** and Java installation path such as C:\Program Files\Java\jdk1.6.0_16 and then press [Enter].



Variable name: JAVA_HOME

Variable value: C:₩Program files/jdk1.6.0_13

Select "Path" and then click [Edit]. Add **%JAVA_HOME%\bin** to the variable and then click [OK].



Variable name: Path

Variable value: %JAVA_HOME%₩bin

You can configure **JAVA_HOME** and **PATH** in the shell.

```
set JAVA_HOME= C:\Program Files\Java\jdk1.6.0_16
set PATH=%PATH%;%JAVA_HOME%\bin
```

Configuring the Environment Variables for Linux

Specify the directory path where Java is installed (example : /usr/java/jdk1.6.0_16) in the **JAVA_HOME** environment variable, and add **$JAVA_HOME/bin** to the **PATH** environment variable.

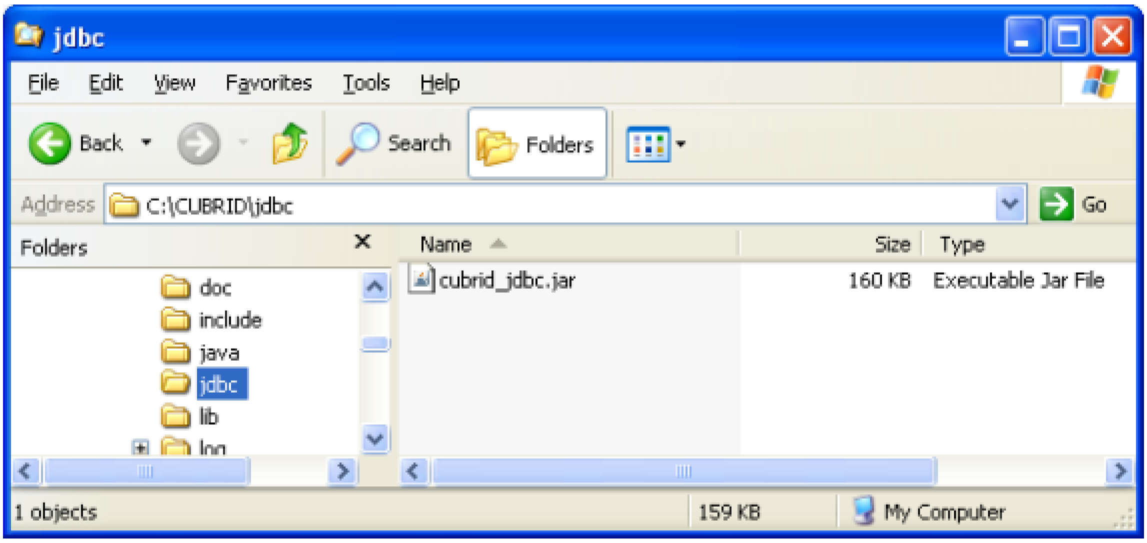
```
export JAVA_HOME=/usr/java/jdk1.6.0_16      //bash
export PATH=$JAVA_HOME/bin:$PATH            //bash

setenv JAVA_HOME /usr/java/jdk1.6.0_16      //csh
```

```
set path = ($JAVA_HOME/bin $path)            //csh
```

JDBC Driver Setting

To use the JDBC, set your **CLASSPATH** environment variable to the path where the CUBRID JDBC driver is located.

The CUBRID JDBC driver (**cubrid_jdbc.jar**) is located in jdbc directory which is subdirectory where CUBRID is installed.



Configuring the CLASSPATH Environment Variables for Windows

```
set CLASSPATH=%CUBRID%\jdbc\cubrid_jdbc.jar:.
```

Configuring the CLASSPATH Environment Variables for Linux

```
export CLASSPATH=$HOME/CUBRID/jdbc/cubrid_jdbc.jar:.
```

Note If a CUBRID JDBC driver has been installed in the same library directory (**$JAVA_HOME/jre/lib/ext**) where the JRE is located, it may be loaded ahead of the server-side JDBC driver used by the Java stored procedure, causing it to malfunction. In a Java stored procedure environment, make sure not to install the generic CUBRID JDBC driver in the directory where the JRE is installed (**$JAVA_HOME/jre/lib/ext**).

JDBC Sample

The following is a simple example that connects to CUBRID by using the JDBC driver and retrieves and inserts data. To run the sample program, make sure that the database you are trying to connect to and the CUBRID Broker are running. In the sample, you will use the **demodb** database that is created automatically during the installation.

JDBC Driver Load

To connect to CUBRID, load the JDBC driver using the for Name() method provided in the class. For more information, see the CUBRID JDBC Driver.

```
Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
```

How to Make the Connection to Database

When the JDBC driver is loaded, use the getConnection() method provided in the DriverManager to connect to the database. To create a Connection object, you must specify the url for describing the location of the database, database user name, password, etc. For more information, see the Connection Configuration.

```
String url = "jdbc:cubrid:localhost:30000:demodb:::";
String userid = "dba";
String password = "";
```

```
Connection conn = DriverManager.getConnection(url,userid,password);
```

Manipulating database (executing queries and processing the ResultSet)

To send a query statement to the connected database and execute it, create the **Statement**, **PrepardStatement**, and **CallableStatement** objects. When a statement object has been created, execute the query using the **executeQuery()** method or the **executeUpdate()** method for the statement object. The **next()** method can process the following row from the ResultSet that is returned from the **executeQuery()** method. For more information, see the BRID JDBC Driver.

Disconnecting from the database

Each method can be disconnected from the database by executing the **close()** method.

JDBC Sample 1

The sample code shown below creates a table, executes a query with a prepared statement, and then rolls back the query. Modify the parameter value of the **getConnection()** method for practice.

```
import java.util.*;
import java.sql.*;

public class Basic {
   public static Connection connect() {
      Connection conn = null;
      try {
           Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
           conn = DriverManager.getConnection("jdbc:cubrid:localhost:30000:::","dba","");
           conn.setAutoCommit (false) ;
      } catch ( Exception e ) {
           System.err.println("SQLException : " + e.getMessage());
      }
      return conn;
   }

   public static void printdata(ResultSet rs) {
      try {
          ResultSetMetaData rsmd = null;

          rsmd = rs.getMetaData();
          int numberofColumn = rsmd.getColumnCount();

          while (rs.next ()) {
              for(int j=1; j<=numberofColumn; j++ )
                  System.out.print(rs.getString(j) + "  " );
              System.out.println("");
          }
      } catch ( Exception e ) {
           System.err.println("SQLException : " + e.getMessage());
      }
   }

   public static void main(String[] args) throws Exception {
      Connection conn = null;
      Statement stmt = null;
      ResultSet rs = null;
      PreparedStatement preStmt = null;

      try {
           conn = connect();

           stmt = conn.createStatement();
           stmt.executeUpdate("create class xoo ( a int, b int, c char(10))");

           preStmt = conn.prepareStatement("insert into xoo values(?,?,''''100'''')");
           preStmt.setInt (1, 1) ;
           preStmt.setInt (2, 1*10) ;
           int rst = preStmt.executeUpdate () ;

           rs = stmt.executeQuery("select a,b,c from xoo" );
           printdata(rs);

           conn.rollback();
           stmt.close();
           conn.close();
      } catch ( Exception e ) {
           conn.rollback();
           System.err.println("SQLException : " + e.getMessage());
      } finally {
           if ( conn != null ) conn.close();
      }
   }
}
```

JDBC Sample 2

The following is an example of executing **SELECT** statement by connecting to demodb that is provided by CUBRID during installation.

```
import java.sql.*;
public class SelectData {
   public static void main(String[] args) throws Exception {
       Connection conn = null;
       Statement stmt = null;
       ResultSet rs = null;
       try {
       // Connect to CUBRID
       Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
       conn =
DriverManager.getConnection("jdbc:CUBRID:localhost:30000:demodb:::","dba","");
       String sql = "select name, players from event";
       stmt = conn.createStatement();
       rs = stmt.executeQuery(sql);
       while(rs.next()) {
           String name = rs.getString("name");
           String players = rs.getString("players");
           System.out.println("name ==> " + name);
           System.out.println("Number of players==> " + players);
           System.out.println("\n=========================================\n");
       }
       rs.close();
       stmt.close();
       conn.close();
       } catch ( SQLException e ) {
           System.err.println(e.getMessage());
       } catch ( Exception e ) {
           System.err.println(e.getMessage());
       } finally {
           if ( conn != null ) conn.close();
       }
   }
}
```

JDBC Example 3

The following is an example of executing **INSERT** statement by connecting to demodb that is provided by CUBRID during installation. You can delete or modify data the same way as you insert data. This means that you can reuse the code below by simply changing the query statements.

```
import java.sql.*;
public class insertData {
   public static void main(String[] args) throws Exception {
       Connection conn = null;
       Statement stmt = null;
       try {
           // CUBRID Connect
           Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
           conn =
DriverManager.getConnection("jdbc:cubrid:localhost:30000:demodb:::","dba","");
```

```
           String sql = "insert into olympic(host_year, host_nation, host_city,
opening_date,            closing_date) values (2008, 'China', 'Beijing', to_date('08-08-
2008','mm-dd-yyyy'),            to_date('08-24-2008','mm-dd-yyyy'))";
           stmt = conn.createStatement();
           stmt.executeUpdate(sql);
           System.out.println("Data is inserted.");
           stmt.close();
       } catch ( SQLException e ) {
           System.err.println(e.getMessage());
       } catch ( Exception e ) {
           System.err.println(e.getMessage());
       } finally {
           if ( conn != null ) conn.close();
       }
   }
}
```

Programming with PHP

Installing the PHP Module

Go to the CUBRID website and the see how to install the PHP module.

Installing PHP for Windows

After compiling and building cubrid_php_[version].dll from php_cubrid.sln in the win directory, create a directory named CUBRID in the directory where PHP is installed, and then copy the the cubrid_php_[version].dll file. For more information, refer to the INSTALL file
Add required settings as shown in the example below by editing the php.ini file.

```
extension_dir=C:\PHP\CUBRID
extension=cubrid_php5.1.4.dll
```

Once the configuration is complete, restart the web server. If you create test.php using using the phpinfo() function of PHP and enter a url as http://localhost/test.php on your Web browser, you will see the CUBRID information if the installation was successful.

Installing PHP for Linux

After compiling and building cubrid.so file by running phpize in the src directory, create a directory named php/extensions in the directory where PHP is installed, and then copy the the cubrid.so file from the module directory. For more information, refer to the INSTALL file
Add required settings as shown in the example below by editing the php.ini file.

```
extension_dir=/usr/lib/php5/lib/php/extentions
extension=cubrid.so
```

After restarting Web server, check the configuration using phpinfo() function.

As with the Windows version of PHP, if you can see the CUBRID information on the web browser, it means that the installation was successful.

PHP Sample

The following is a simple example that establishes a connection between PHP and CUBRID. This section will cover the most basic and notable features. Before running the sample program, a database and the Broker you are trying to connect must be running. This example uses the **demodb** database created during the installation.

Example of Data Retrieval

```
<html>
<head>
<meta http-equiv='content-type' content='text/html; charset=euc-kr'
</head>
<body>
<center>
<table border=2>
<?
   // Set server information for CUBRID connection. host_ip is the IP address where the
CUBRID Broker is installed (localhost in this example), and host_port is the port number
of the CUBRID Broker. The port number is the default given during the installation. For
details, see "Administrator's Guide."
   $host_ip = "localhost";
   $host_port = 30000;
   $db_name = "demodb";
   // Connect to CUBRID Server. Do not make the actual connection, but only retain the
connection information. The reason for not making the actual connection is to handle
transaction more efficiently in the 3-tier architecture.
   $cubrid_con = @cubrid_connect($host_ip, $host_port, $db_name);
   if (!$cubrid_con) {
       echo "Database Connection Error";
       exit;
   }
?>
<?
   $sql = "select sports, count(players) as players from event group by sports";
   // Request the CUBRID Server for the results of the SQL statement. Now make the actual
connection to the CUBRID Server.
   $result = cubrid_execute($cubrid_con, $sql);
   if ($result) {
       // Get the column names from the result set created by the SQL query.
       $columns = cubrid_column_names($result);
       // Get the number of columns in the result set created by the SQL query.
       $num_fields = cubrid_num_cols($result);
       // List the column names of the result set on the screen.
       echo("<tr>");
       while (list($key, $colname) = each($columns)) {
       echo("<td align=center>$colname</td>");
       }
       echo("</tr>");
       // Get the results from the result set.
       while ($row = cubrid_fetch($result)) {
           echo("<tr>");
           for ($i = 0; $i < $num_fields; $i++) {
               echo("<td align=center>");
               echo($row[$i]);
               echo("</td>");
           }
           echo("</tr>");
       }
   }
   // The PHP module in the CUBRID runs in a 3-tier architecture. Even when calling SELECT
for transaction processing, it is processed as a part of the transaction. Therefore, the
transaction needs to be rolled back by calling commit or rollback even though SELECT was
called for smooth performance.
   cubrid_commit($cubrid_con);
   cubrid_disconnect($cubrid_con);
?>
</body></html>
```

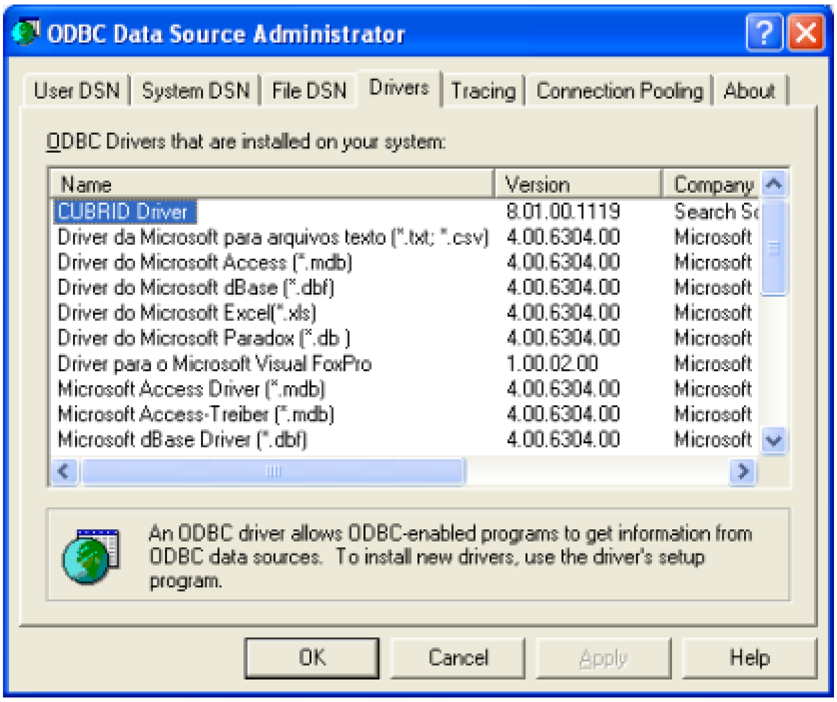
Example of Data Insertion

```
<html>
<head>
<meta http-equiv='content-type' content='text/html; charset=euc- kr'>
</head>
<body>
<center>
<table border=2>
<?
   $host_ip = "localhost";
   $host_port = 30000;
    $db_name = "demodb";
   $cubrid_con = @cubrid_connect($host_ip, $host_port, $db_name);
   if (!$cubrid_con) {
        echo "Database Connection Error";
       exit;
   }
?>
<?
   $sql = "insert into olympic (host_year,host_nation,host_city,opening_date,closing_date)
values     (2008, 'China', 'Beijing', to_date('08-08-2008','mm-dd- yyyy'),to_date('08-24-
2008','mm-dd-yyyy'))    ;"
   $result = cubrid_execute($cubrid_con, $sql);
   if ($result) {
       // Handled successfully, so commit.
       cubrid_commit($cubrid_con);
       echo("Inserted successfully ");
   } else {
       // Error occurred, so the error message is output and rollback is called.
       echo(cubrid_error_msg());
       cubrid_commit($cubrid_con);
   }
   cubrid_disconnect($cubrid_con);
?>
</body></html>
```

Programming with ODBC and ASP

Configuring the Environment of ODBC and ASP

CUBRID ODBC is compatible for version 3.52 ODBC and LEVEL2. Note that backward compatibility is not guaranteed for applications that are written with ODBC Spec 2.x. The CUBRID ODBC driver is automatically installed while CUBRID is installed. You can verify it from [Control Panel] > [Administrative Tools] > [Data Source (ODBC)] > [Drivers] tab.

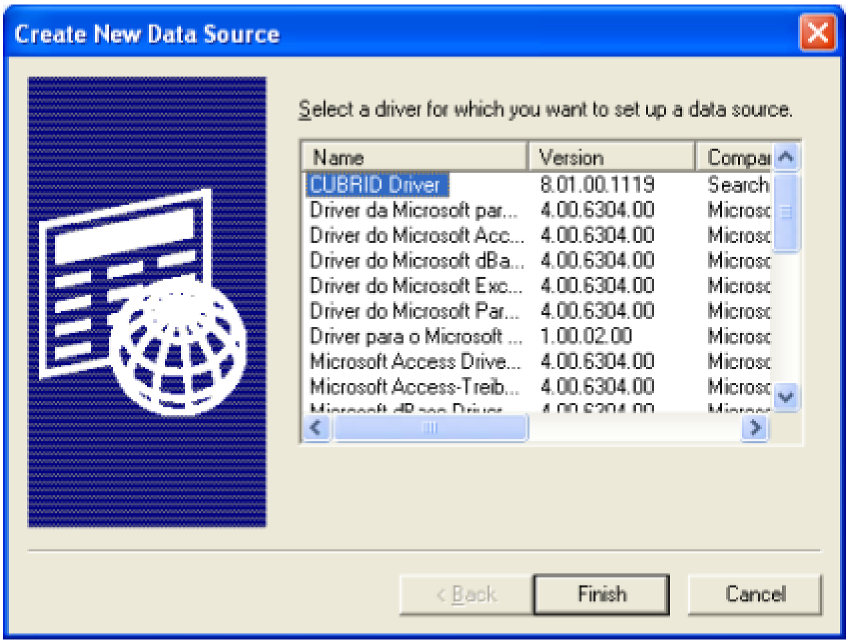


ODBC Drivers that are installed on your system:

Name	Version	Company
CUBRID Driver	8.01.00.1119	Search Sc
Driver da Microsoft para arquivos texto (*.txt; *.csv)	4.00.6304.00	Microsoft
Driver do Microsoft Access (*.mdb)	4.00.6304.00	Microsoft
Driver do Microsoft dBase (*.dbf)	4.00.6304.00	Microsoft
Driver do Microsoft Excel(*.xls)	4.00.6304.00	Microsoft
Driver do Microsoft Paradox (*.db)	4.00.6304.00	Microsoft
Driver para o Microsoft Visual FoxPro	1.00.02.00	Microsoft
Microsoft Access Driver (*.mdb)	4.00.6304.00	Microsoft
Microsoft Access-Treiber (*.mdb)	4.00.6304.00	Microsoft
Microsoft dBase Driver (*.dbf)	4.00.6304.00	Microsoft

An ODBC driver allows ODBC-enabled programs to get information from ODBC data sources. To install new drivers, use the driver's setup program.

If the CUBRID ODBC driver is detected, set a DSN as a database where the application is trying to connect. To set up a DSN, click the [Add] button in the ODBC Data Source Administrator dialog box. Then, the following dialog box appears. Select "CUBRID Driver," and then click the [Finish] button.



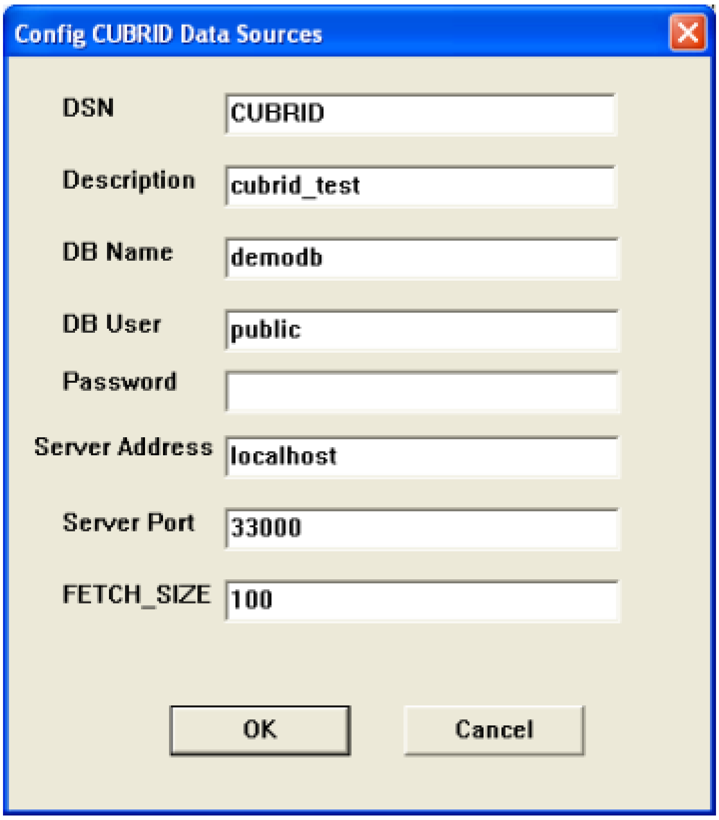
Create New Data Source

Select a driver for which you want to set up a data source.

Name	Version	Compai
CUBRID Driver	8.01.00.1119	Search
Driver da Microsoft par...	4.00.6304.00	Microsc
Driver do Microsoft Acc...	4.00.6304.00	Microsc
Driver do Microsoft dBa...	4.00.6304.00	Microsc
Driver do Microsoft Exc...	4.00.6304.00	Microsc
Driver do Microsoft Par...	4.00.6304.00	Microsc
Driver para o Microsoft ...	1.00.02.00	Microsc
Microsoft Access Drive...	4.00.6304.00	Microsc
Microsoft Access-Treib...	4.00.6304.00	Microsc

When the following [Config CUBRID Data Sources] dialog box appears, enter the database name that you try to connect to in the [DB Name] field, the port number of the CUBRID Broker in the [Server Port] field, and then click [OK] button. You can verify the number in the **cubrid.broker.conf** file.

FETCH_SIZE refers to the number of records fetched from server whenever **cci_fetch**() function of CCI library is called; the CCI library is internally used by ODBC driver.



Config CUBRID Data Sources

DSN	CUBRID
Description	cubrid_test
DB Name	demodb
DB User	public
Password	
Server Address	localhost
Server Port	33000
FETCH_SIZE	100

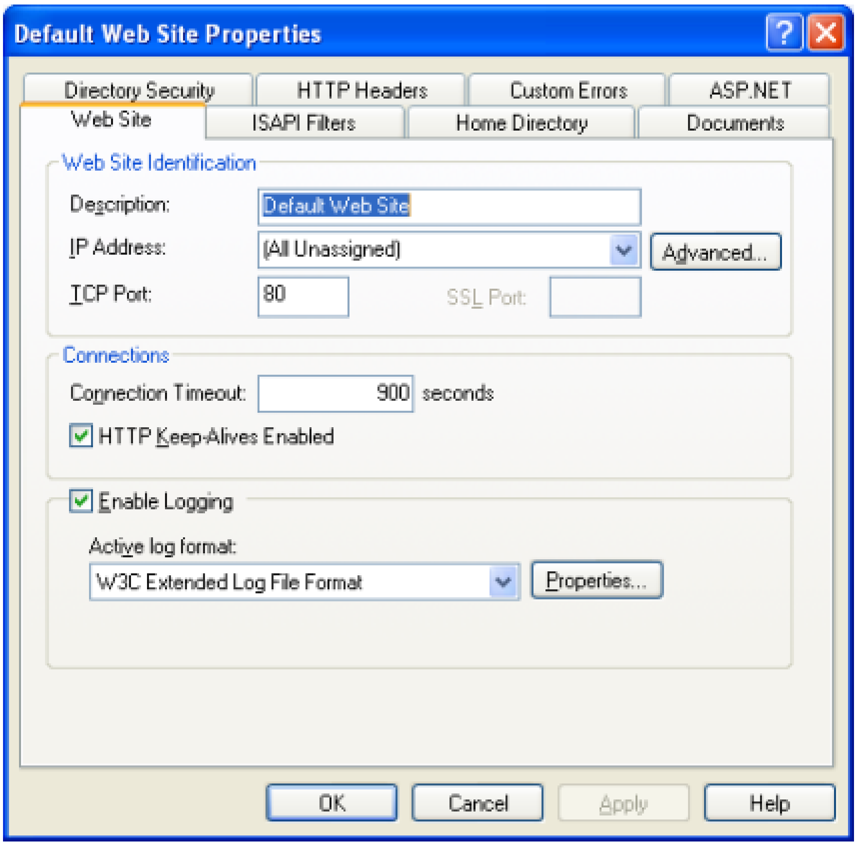
For more information on CUBRID ODBC driver, see "ODBC API Reference."

- CUBRID ODBC Driver

- Using OIDs and Collections
- Supported Functions and Backward Compatibility

ASP Sample

In the virtual directory where the ASP sample program runs, right-click "Default Web Site" and click [Properties].



Description: Default Web Site

IP Address: (All Unassigned)

TCP Port: 80

SSL Port:

Connection Timeout: 900 seconds

☑ HTTP Keep-Alives Enabled

☑ Enable Logging

Active log format: W3C Extended Log File Format

The dialog box shown above will appear. Under the **Web Site Identification**, in the **IP Address** drop-down box, select "(All Unassigned)." This sets the IP address to localhost. If you want to run the sample program using a specific IP address, configure the directory with the IP address as a virtual directory and register the IP address in Properties.

The following is an example in which the IP address is set to localhost.

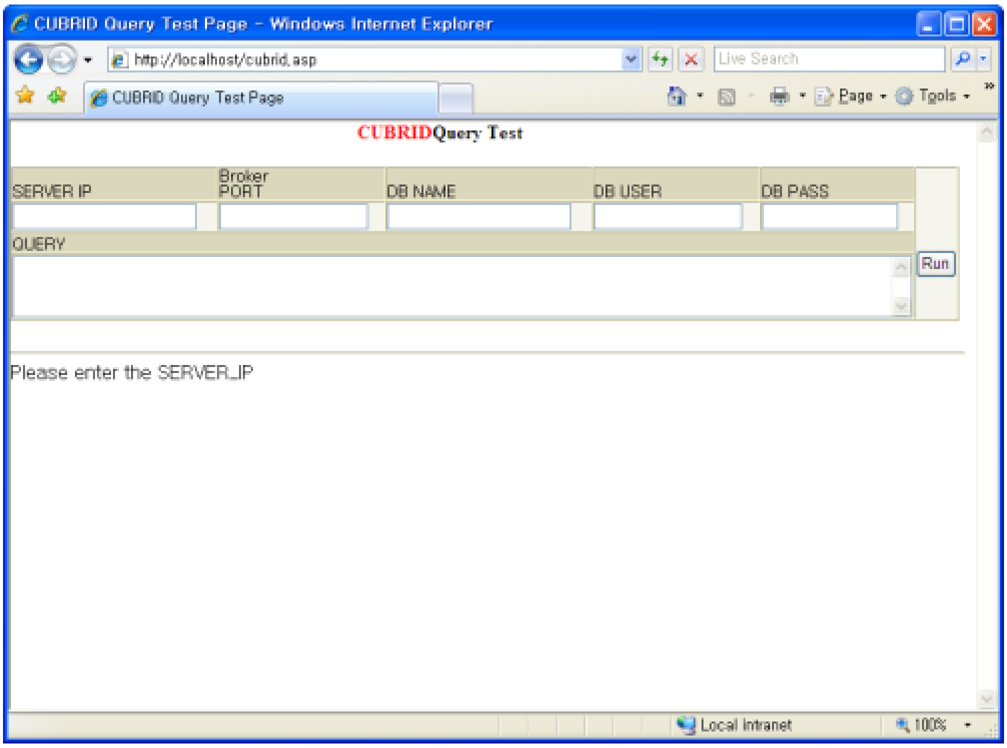
Example

Save the following sample code as cubrid.asp in the virtual directory.

```
<HTML>
    <HEAD>
     <meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">
       <title>CUBRID Query Test Page</title>
  </HEAD>
 <BODY topmargin="0" leftmargin="0">

 <table border="0" width="748" cellspacing="0" cellpadding="0">
   <tr>
     <td width="200"></td>
     <td width="287">
       <p align="center"><font size="3" face="Times New Roman"><b><font 
color="#FF0000">CUBRID</font>Query Test</b></font></td>
     <td width="200"></td>
   </tr>
 </table>
 <form action="cubrid.asp" method="post" >
 <table border="1" width="700" cellspacing="0" cellpadding="0" height="45">
```

```
   <tr>
     <td width="113" valign="bottom" height="16" bgcolor="#DBD7BD"
bordercolorlight="#FFFFCC"><font size="2">SERVER IP</font></td>
     <td width="78"  valign="bottom" height="16" bgcolor="#DBD7BD"
bordercolorlight="#FFFFCC"><font size="2">Broker PORT</font></td>
     <td width="148" valign="bottom" height="16" bgcolor="#DBD7BD"
bordercolorlight="#FFFFCC"><font size="2">DB NAME</font></td>
     <td width="113" valign="bottom" height="16" bgcolor="#DBD7BD"
bordercolorlight="#FFFFCC"><font size="2">DB USER</font></td>
     <td width="113" valign="bottom" height="16" bgcolor="#DBD7BD"
bordercolorlight="#FFFFCC"><font size="2">DB PASS</font></td>
     <td width="80" height="37" rowspan="4" bordercolorlight="#FFFFCC" bgcolor="#F5F5ED">
       <p><input type="submit" value="Execute" name="B1" tabindex="7"></p></td>
   </tr>
   <tr>
     <td width="113" height="1" bordercolorlight="#FFFFCC" bgcolor="#F5F5ED"><font
size="2"><input type="text" name="server_ip" size="20" tabindex="1" maxlength="15"
value="<%=Request("server_ip")%>"></font></td>
     <td width="78"  height="1" bordercolorlight="#FFFFCC" bgcolor="#F5F5ED"><font
size="2"><input type="text" name="cas_port" size="15" tabindex="2" maxlength="6"
value="<%=Request("cas_port")%>"></font></td>
     <td width="148" height="1" bordercolorlight="#FFFFCC" bgcolor="#F5F5ED"><font
size="2"><input type="text" name="db_name" size="20" tabindex="3" maxlength="20"
value="<%=Request("db_name")%>"></font></td>
     <td width="113" height="1" bordercolorlight="#FFFFCC" bgcolor="#F5F5ED"><font
size="2"><input type="text" name="db_user" size="15" tabindex="4"
value="<%=Request("db_user")%>"></font></td>
     <td width="113" height="1" bordercolorlight="#FFFFCC" bgcolor="#F5F5ED"><font
size="2"><input type="password" name="db_pass" size="15" tabindex="5"
value="<%=Request("db_pass")%>"></font></td>
   </tr>
   <tr>
     <td width="573" colspan="5" valign="bottom" height="18" bordercolorlight="#FFFFCC"
bgcolor="#DBD7BD"><font size="2">QUERY</font></td>
   </tr>
   <tr>
     <td width="573" colspan="5" height="25" bordercolorlight="#FFFFCC"
bgcolor="#F5F5ED"><textarea rows="3" name="query" cols="92"
tabindex="6"><%=Request("query")%></textarea></td>
   </tr>
 </table>
 </form>
 <hr>
</BODY>
</HTML>
<%
    ' Fetch the DSN and SQL statement.
    strIP = Request( "server_ip" )
    strPort = Request( "cas_port" )
    strUser = Request( "db_user" )
    strPass = Request( "db_pass" )
    strName = Request( "db_name" )
    strQuery = Request( "query" )

if strIP = "" then
   Response.Write "Please enter the SERVER_IP"
        Response.End 'If no IP entered, end the page
    end if
    if strPort = "" then
       Response.Write "Please enter the port number"
        Response.End ' If no port entered, end the page
    end if
    if strUser = "" then
       Response.Write "Please enter the DB_USER"
        Response.End ' If no DB_User entered, end the page
    end if
    if strName = "" then
       Response.Write "Please enter the DB_NAME "
        Response.End ' If no DB_NAME entered, end the page
    end if
    if strQuery = "" then
       Response.Write "Please enter the query you want to check"
        Response.End ' If no Query entered, end the page
    end if
 ' Create the connection object
  strDsn = "driver={CUBRID Driver};server=" & strIP & ";port=" & strPort & ";uid=" &
strUser & ";pwd=" & strPass & ";db_name=" & strName & ";"
' Connect to DB
Set DBConn = Server.CreateObject("ADODB.Connection")
       DBConn.Open strDsn
    ' Execute SQL
    Set rs = DBConn.Execute( strQuery )
    ' Show message depending on the SQL statement
    if InStr(Ucase(strQuery),"INSERT")>0 then
        Response.Write "The record has been added."
        Response.End
    end if

    if InStr(Ucase(strQuery),"DELETE")>0  then
        Response.Write "The record has been deleted."
        Response.End
    end if

    if InStr(Ucase(strQuery),"UPDATE")>0  then
        Response.Write "The record has been modified."
        Response.End
    end if
%>
<table>
<%
    ' Show the field name
    Response.Write "<tr bgColor=#f3f3f3>"
    For index =0 to ( rs.fields.count-1 )
        Response.Write "<td><b>" & rs.fields(index).name & "</b></td>"
    Next
    Response.Write "</tr>"
    ' Show the field value
    Do While Not rs.EOF
        Response.Write "<tr bgColor=#f3f3f3>"
        For index =0 to ( rs.fields.count-1 )
            Response.Write "<td>" & rs(index) & "</td>"
        Next
        Response.Write "</tr>"

        rs.MoveNext
    Loop
%>
<%
    set  rs = nothing
%>
</table>
```

You can check the result of the sample program at http://localhost/ASP/cubrid.asp. When you execute the sample code above, you will get the following output. Enter appropriate values in each field, and then enter the query statement in the Query field. When you click [Run], the query result will be displayed at the lower portion of the page.

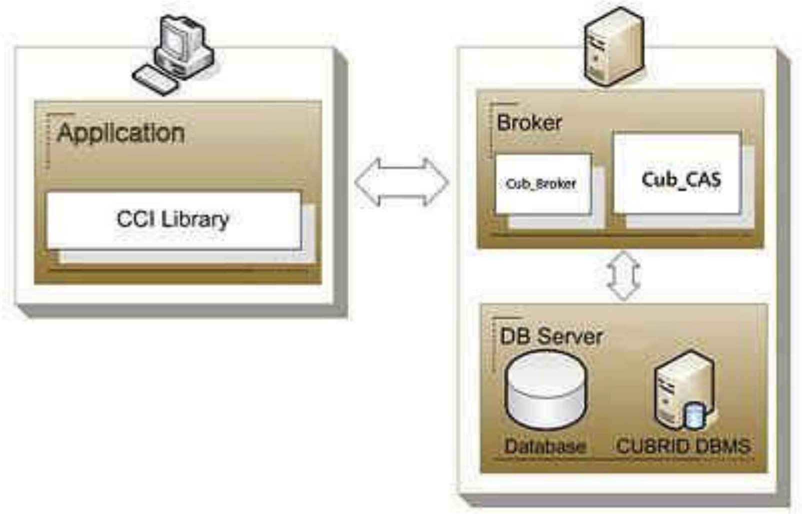


Please enter the SERVER_IP

Programming with CCI

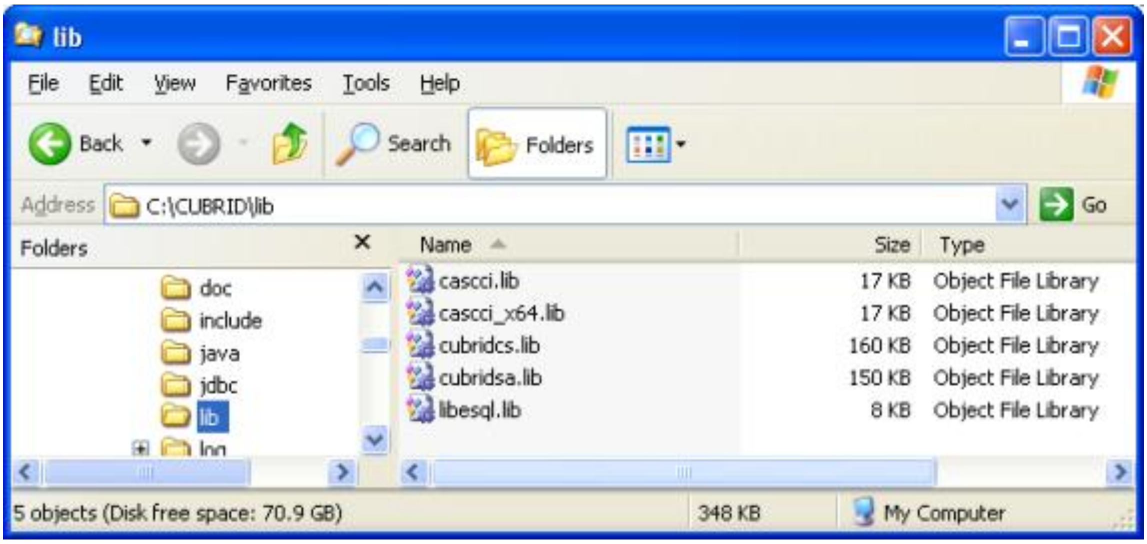
CCI Library

The CCI Library is a C language interface provided by CUBRID. CCI is connected to the application through the Broker, so you can manage it the same way as other interfaces such as JDBC, PHP and ODBC. In fact, CCI provides a foundation to implement PHP, ODBC, Python and, Ruby interfaces.



CCI Installation and Configuration

The CCI library is contained in the CUBRID installation package. The following figure shows where the files are located.



Name	Size	Type
cascci.lib	17 KB	Object File Library
cascci_x64.lib	17 KB	Object File Library
cubridcs.lib	160 KB	Object File Library
cubridsa.lib	150 KB	Object File Library
libesql.lib	8 KB	Object File Library

5 objects (Disk free space: 70.9 GB) 348 KB My Computer

Operating System	Windows	UNIX/Linux
C header file	include/cas_cci.h	include/cas_cci.h
Static library	lib/cascci.lib	lib/libcascci.a
Dynamic library	lib/cascci.lib bin/cascci.dll	lib/libcascci.so

Using CCI

Basic Flow Diagram of the Application Using CCI

To use CUBRID, the following procedures are required for applications using the CCI libraries to execute queries: connection to CAS, query preparation, query execution, response handling, and disconnection. In each process, CCI communicates with the application using connection, query and response handles.

The following flowchart shows the process of the application using CCI and the functions used in each step. See CCI API in the API Reference for more information.

- Opening a database connection handle (related function : cci_connect)
- Getting the request handle for a prepared statement (related function : cci_prepare)
- Binding data to the prepared statement (related function : cci_bind_param)
- Executing the prepared statement (related function : cci_execute)
- Processing the execution result (related function : cci_cursor, cci_fetch, cci_get data, cci_get_result_info)
- Closing the request handle (related function : cci_close_req_handle)
- Closing a database connection handle (related function : cci_disconnect)

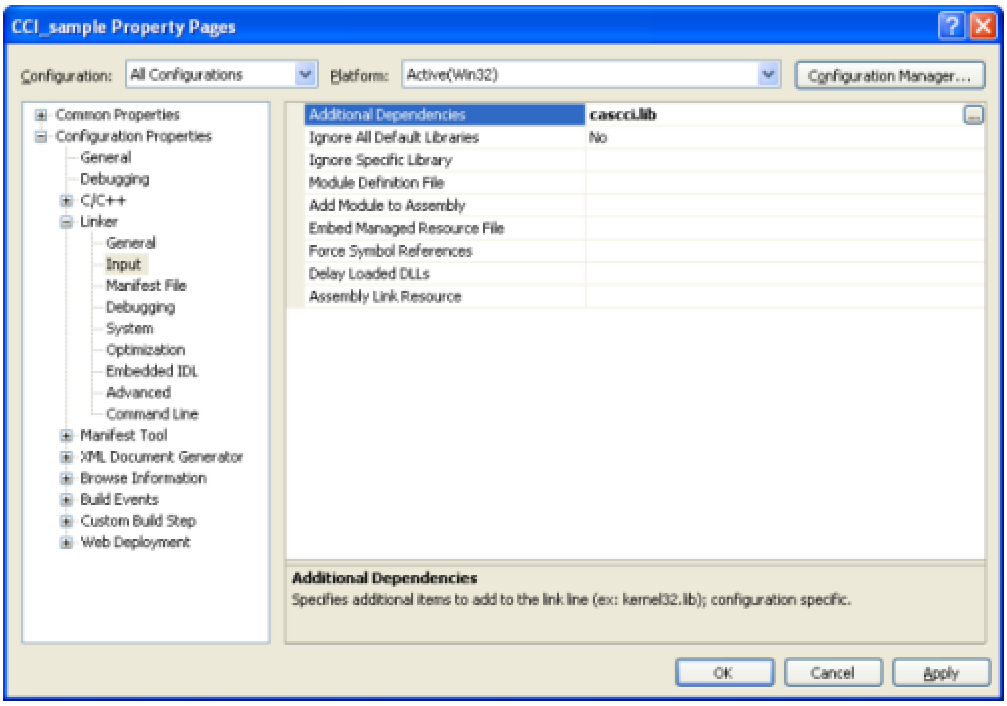
How to use

Once you have created the application using CCI, you should decide, according to its features, whether to execute CCI as a static link or dynamic link before you build it. Determine the library to use by referring to the table in the CCI Installation and Configuration.

The following is an example Makefile to use the dynamic link library on UNIX/Linux:

```
CC=gcc
CFLAGS = -g -Wall -I. -I$CUBRID/include
LDFLAGS = -L$CUBRID/lib -lcascci -lnsl
TEST_OBJS = test.o
EXES = test
all: $(EXES)
test: $(TEST_OBJS)
 $(CC) -o $@ $(TEST_OBJS) $(LDFLAGS)
```

The following is the settings for using the static library on Windows:



Configuration: All Configurations

Platform: Active(Win32)

Additional Dependencies	cascci.lib
Ignore All Default Libraries	No
Ignore Specific Library	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded DLLs	
Assembly Link Resource	

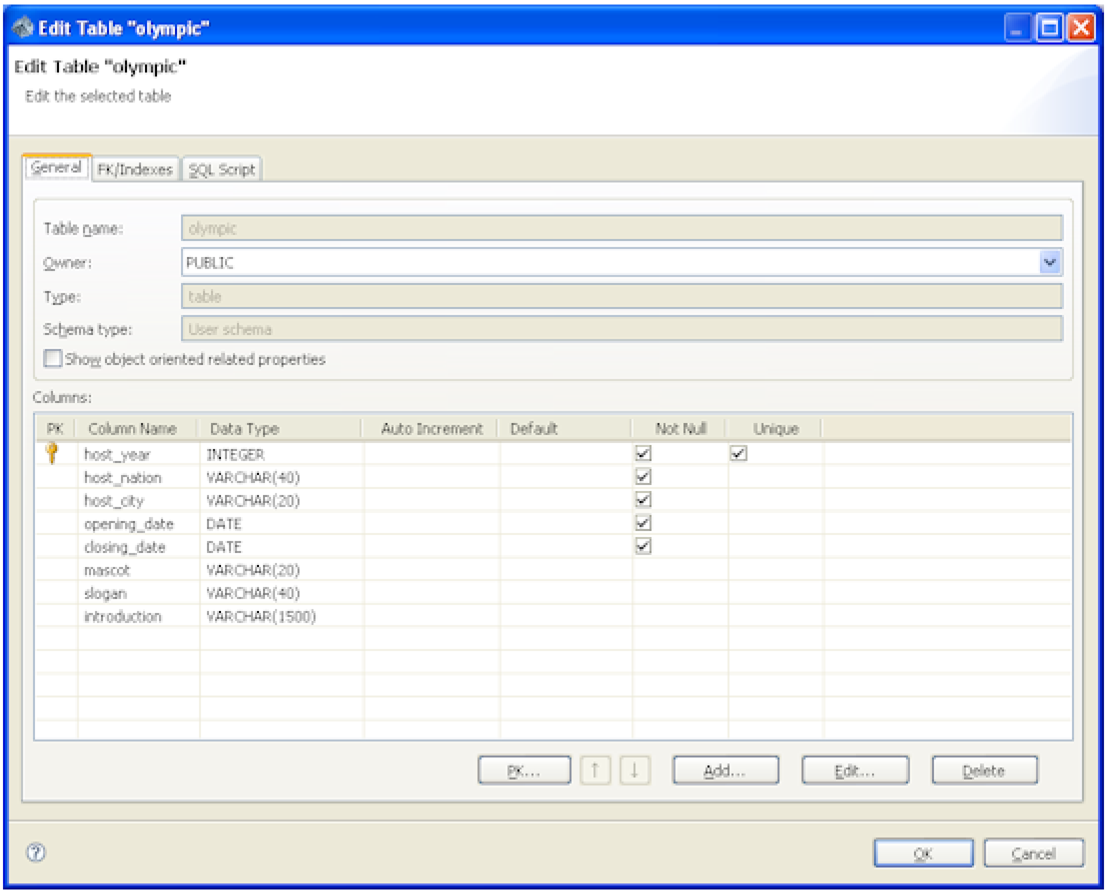
Additional Dependencies
Specifies additional items to add to the link line (ex: kernel32.lib); configuration specific.

CCI Sample

Introduction

The sample program is to create a simple application using CCI through the connection to the **demodb** database deployed by default during the CUBRID installation. Follow the processes of connection to CAS, query preparation, query execution, response handling and disconnection in the sample. The sample is created in a way that uses dynamic links on Linux.

The following is schema information of the **olympic** table in the **demodb** database used in the sample.



Edit Table "olympic"

Edit the selected table

Table name: olympic

Owner: PUBLIC

Type: table

Schema type: User schema

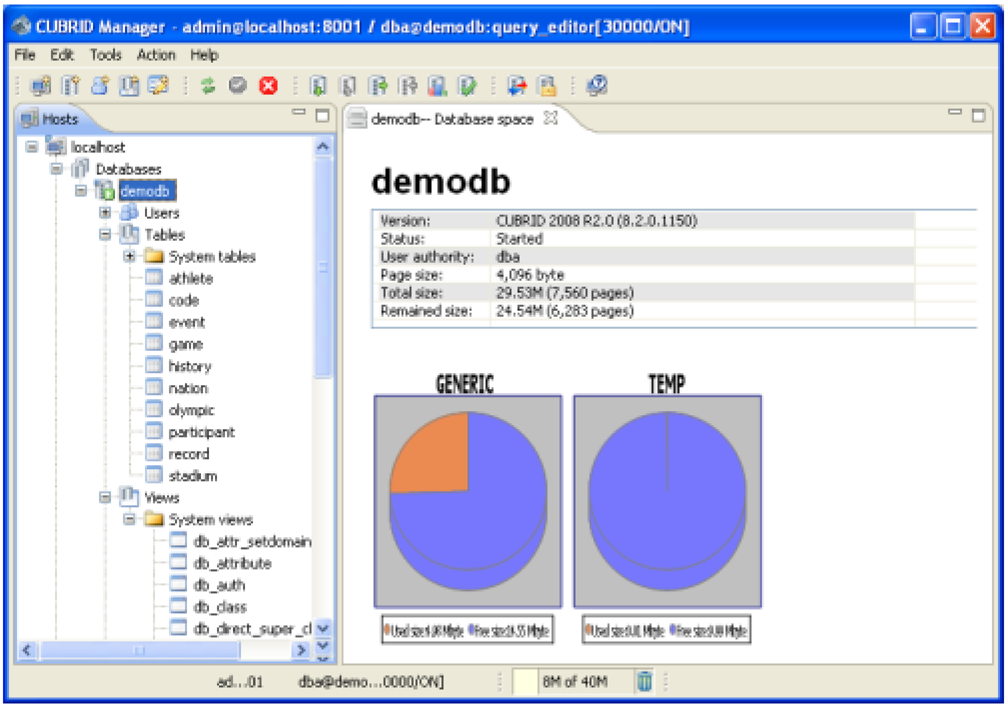
□ Show object oriented related properties

Columns:

PK	Column Name	Data Type	Auto Increment	Default	Not Null	Unique
🔑	host_year	INTEGER			☑	☑
	host_nation	VARCHAR(40)			☑	
	host_city	VARCHAR(20)			☑	
	opening_date	DATE			☑	
	closing_date	DATE			☑	
	mascot	VARCHAR(20)				
	slogan	VARCHAR(40)				
	introduction	VARCHAR(1500)				

Preparation

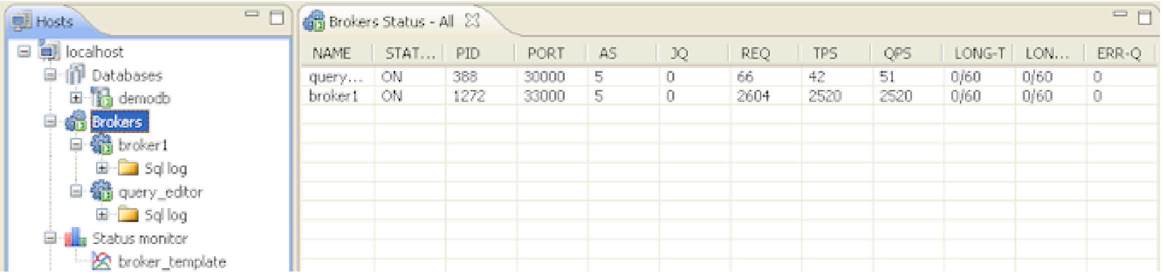
Make sure that the **demodb** database and the Broker are running before you execute the sample program. You can start the **demodb** database and the Broker through the CUBRID Manager. The following figure shows that the **demodb** database is running through the CUBRID Manager.



demodb

Version:	CUBRID 2008 R2.0 (8.2.0.1150)
Status:	Started
User authority:	dba
Page size:	4,096 byte
Total size:	29.53M (7,560 pages)
Remained size:	24.54M (6,283 pages)

The following shows that the Broker is running with the CUBRID Manager.



NAME	STAT...	PID	PORT	AS	JQ	REQ	TPS	QPS	LONG-T	LON...	ERR-Q
query...	ON	388	30000	5	0	66	42	51	0/60	0/60	0
broker1	ON	1272	33000	5	0	2604	2520	2520	0/60	0/60	0

Build

With the program source and the Makefile ready, executing "make" will create an executable file called "test." If you use a static library, there is no need to deploy additional files and the execution will be faster. However, it increases the program size and memory usage. If you use a dynamic library, there will be some performance overhead, but the program size and memory usage can be optimized.

The following is a command line example. It builds the test program using the dynamic library instead of "make" on Linux.

```
cc -o test test.c -I$CUBRID/include -L$CUBRID/lib -lnsl -lcascci
```

Sample Code

```
#include <stdio.h>
#include <cas_cci.h>
char *cci_client_name = "test";
int main (int argc, char *argv[])
{
    int con = 0, req = 0, col_count = 0, res, ind, i;
    T_CCI_ERROR error;
    T_CCI_COL_INFO *res_col_info;
    T_CCI_SQLX_CMD cmd_type;
    char *buffer, db_ver[16];
    printf("Program started!\n");
    if ((con=cci_connect("localhost", 30000, "demodb", "PUBLIC", ""))<0) {
        printf( "%s(%d): cci_connect fail\n", __FILE__, __LINE__);
        return -1;
    }

    if ((res=cci_get_db_version(con, db_ver, sizeof(db_ver)))<0) {
        printf( "%s(%d): cci_get_db_version fail\n", __FILE__, __LINE__);
        goto handle_error;
    }
    printf("DB Version is %s\n",db_ver);
    if ((req=cci_prepare(con, "select * from event", 0,&error))<0) {
        printf( "%s(%d): cci_prepare fail(%d)\n", __FILE__, __LINE__,error.err_code);
        goto handle_error;
    }
    printf("Prepare ok!(%d)\n",req);
    res_col_info = cci_get_result_info(req, &cmd_type, &col_count);
    if (!res_col_info) {
        printf( "%s(%d): cci_get_result_info fail\n", __FILE__, __LINE__);
        goto handle_error;
    }

    printf("Result column information\n"
           "========================================\n");
    for (i=1; i<=col_count; i++) {
        printf("name:%s  type:%d(precision:%d scale:%d)\n",
            CCI_GET_RESULT_INFO_NAME(res_col_info, i),
            CCI_GET_RESULT_INFO_TYPE(res_col_info, i),
            CCI_GET_RESULT_INFO_PRECISION(res_col_info, i),
            CCI_GET_RESULT_INFO_SCALE(res_col_info, i));
    }
    printf("========================================\n");
    if ((res=cci_execute(req, 0, 0, &error))<0) {
        printf( "%s(%d): cci_execute fail(%d)\n", __FILE__, __LINE__,error.err_code);
        goto handle_error;
    }
    if ((res=cci_fetch_size(req, 100))<0) {
        printf( "%s(%d): cci_fetch_size fail\n", __FILE__, __LINE__);
        goto handle_error;
    }

    while (1) {
        res = cci_cursor(req, 1, CCI_CURSOR_CURRENT, &error);
        if (res == CCI_ER_NO_MORE_DATA) {
            printf("Query END!\n");
            break;
        }
        if (res<0) {
            printf( "%s(%d): cci_cursor fail(%d)\n", __FILE__, __LINE__,error.err_code);
            goto handle_error;
        }

        if ((res=cci_fetch(req, &error))<0) {
            printf( "%s(%d): cci_fetch fail(%d)\n", __FILE__, __LINE__,error.err_code);
            goto handle_error;
        }

        for (i=1; i<=col_count; i++) {
            if ((res=cci_get_data(req, i, CCI_A_TYPE_STR, &buffer, &ind))<0) {
                printf( "%s(%d): cci_get_data fail\n", __FILE__, __LINE__);
                goto handle_error;
            }
            printf("%s \t|", buffer);
        }
        printf("\n");
    }
    if ((res=cci_close_req_handle(req))<0) {
           printf( "%s(%d): cci_close_req_handle fail", __FILE__, __LINE__);
           goto handle_error;
    }
    if ((res=cci_disconnect(con, &error))<0) {
          printf( "%s(%d): cci_disconnect fail(%d)", __FILE__, __LINE__,error.err_code);
          goto handle_error;
    }
    printf("Program ended!\n");
    return 0;
```

```
handle_error:
    if (req > 0)
        cci_close_req_handle(req);
    if (con > 0)
        cci_disconnect(con, &error);
    printf("Program failed!\n");
    return -1;
}
```

CSQL Interpreter

To execute SQL statements in CUBRID, you need to use either a Graphical User Interface (GUI)-based CUBRID Manager or a console-based CSQL Interpreter.

CSQL is an application that allows users to use SQL statements through a command-driven interface. This section briefly explains how to use the CSQL Interpreter and associated commands.

- Introduction to the CSQL Interpreter
- Running CSQL
- Session Commands

Introduction to the CSQL Interpreter

A Tool for SQL

The CSQL Interpreter is an application installed with CUBRID that allows you to execute in an interactive or batch mode and viewing query results. The CSQL Interpreter has a command-line interface. With this, you can save SQL statements together with their results to a file for a later use.

The CSQL Interpreter provides the best and easiest way to use CUBRID. You can develop database applications with various APIs (e.g. JDBC, ODBC, PHP, CCI, etc.; you can use the CUBRID Manager, which is a management and query tool provided by CUBRID. With the CSQL Interpreter, users can create and retrieve data in a terminal-based environment.

The CSQL Interpreter directly connects to a CUBRID database and executes various tasks using SQL statements. Using the CSQL Interpreter, you can:

- Retrieve, update and delete data in a database by using SQL statements
- Execute external shell commands
- Save or print query results
- Create and execute SQL script files
- Select table schema
- Retrieve or modify parameters of the database server system
- Retrieve database information (e.g. schema, triggers, queued triggers, workspaces, locks, and statistics)

A Tool for DBA

A database administrator (**DBA**) performs administrative tasks by using various administrative utilities provided by CUBRID; a terminal-based interface of CSQL Interpreter is an environment where **DBA** executes administrative tasks.

It is also possible to run the CSQL Interpreter in a standalone mode. In this mode, the CSQL Interpreter directly accesses database files and executes commands including server process properties. That is, SQL statements can be executed to a database without running a separate database server process. The CSQL Interpreter is a powerful tool that allows you to use the database only with a **csql** utility, without any other applications such as the Database Server or the Brokers.

Executing CSQL

CSQL Execution Mode

Interactive Mode

With CSQL Interpreter, you can enter and execute SQL statements to handle schema and data in the database. Enter statements in a prompt that appears when running the **csql** utility. After executing the statements, the results are listed in the next line. This is called the interactive mode.

Batch Mode

You can save SQL statements in a file and execute them later to have the **csql** utility read the file. This is called the batch mode. For more information on the batch mode, see CSQL Startup Options.

Standalone Mode

In the standalone mode, CSQL Interpreter directly accesses database files and executes commands including server process functions. That is, SQL statements can be sent and executed to a database without a separate database server process running for the task. Since the standalone mode allows only one user access at a given time, it is suitable for management tasks by Database Administrators (**DBAs**).

Client/Server Mode

CSQL Interpreter usually operates as a client process and accesses the server process.

Using CSQL (Syntax)

Connecting to Local Host

Description

Execute the CSQL Interpreter using the **csql** utility. You can set options as needed. To set the options, specify the name of the database to connect to as a parameter. The following is a **csql** utility statement to access the database on a local server:

Syntax

```
csql [ options ] database_name
```

Connecting to Remote Host

Descripton

The following is a **csql** utility statement to access the database on a remote host:

Syntax

```
csql [ options ] database_name@remote_host_name
```

Make sure that the following conditions are met before you run the CSQL Interpreter on a remote host.

- The CUBRID installed on the remote host must be the same version as the one on the local host.
- The port number used by the master process on the remote host must be identical to the one on the local host.
- You must access the remote host in a client/server mode using the **-C** option.

Example

The following is an example statement that accesses the **demodb** database on the remote host with the IP address 192.168.1.3 and calls the **csql** utility.

```
csql -C demodb@192.168.1.3
```

CSQL Startup Options

To display the option list in the prompt, execute the **csql** utility without specifying the database name as follows:

```
 % csql
interactive SQL utility, version R3.0
usage: csql [OPTION] database-name valid options:
  -S, --SA-mode                standalone mode execution
  -C, --CS-mode                client-server mode execution
  -u, --user=ARG               alternate user name
  -p, --pasword=ARG            password string, give "" for none
  -e, --error-continue         don't exit on statement error
  -i, --input-file=ARG         input-file-name
  -o, --output-file=ARG        output-file-name
  -s, --single-line            single line oriented execution
  -c, --command=ARG            CSQL-commands
  -l, --line-output            display each value in a line
  -r, --read-only              read-only mode
      --no-auto-commit         disable auto commit mode execution
      --no-pager               do not use pager
      --sysadm                system admin mode
```

Options

The following table lists the options that can be issued with the **csql** utility.

Options	Description
-S	Executes the csql utility in a standalone mode.
-C	Executes the csql utility in a client/server mode.
-u *user_name*	Specifies the user that tries to access the database. The default value is **PUBLIC**.
-p *password*	Specifies the password of the user that tries to access the database (if any).
-e	Continues the session even when an error occurs.
-i *input_file*	Executes the csql utility in a batch mode. The *input_file* parameter is the file name where SQL statements are saved.
-o *output_file*	Saves a result of the statement execution in the specified *output_file* without displaying it on the screen.
-s	Executes multiple SQL statements one by one in the file where they are saved consecutively. Multiple SQL statements are separated by semicolons (;).
-c "CSQL commands"	Executes SQL statements directly from the prompt. To use this option, enclose the SQL statement to execute in double quotes.
-l	Displays the query results in a line format instead of a column. By default, the results will be displayed in a column format.
-r	Connects to a database in read-only mode.
--no-auto-commit	Configures the auto-commit mode of the CSQL Interpreter to OFF.
--no-pager	Displays the results of the query performed by the CSQL Interpreter at once instead of page-by-page.

Executing in a standalone mode (-S)

The following is an example to connect to the **demodb** database in a standalone mode and execute the **csql** utility with the **-S** option. When you want to use the **demodb** database exclusively, use the **-S** option.

```
csql -S demodb
```

Executing in a client/server mode (-C)

The following is an example to connect to the **demodb** database in a client/server mode and execute the **csql** utility with the **-C** option. In an environment where multiple clients connect to the **demodb** database, use the **-C** option. Even when you connect to a database on a remote host in a client/server mode, the error log created during the **csql** utility execution will be saved in the **cub_client.err** file on the local host.

```
csql -C demodb
```

Specifying the name of the input file to use in a batch mode (-i)

The following is an example to specify the name of the input file that will be used in a batch mode with the **-i** option. In the **infile** file, more than one SQL statement are saved. Without the **-i** option specified, the CSQL Interpreter will run in an interactive mode.

```
csql -i infile demodb
```

Specifying the output file to save the execution results (-o)

The following is an example to save the execution results to the specified file instead of displaying on the screen with the **-o** option. This option is useful when you want to retrieve the results of the query performed by the CSQL Interpreter at a later time.

```
csql -o outfile demodb
```

Specifying the user name (-u)

The following is an example to specify the name of the user that will connect to the specified database with the **-u** option. If the **-u** option is not specified, **PUBLIC** that has the lowest level of authorization will be specified as a user. If the user name is not valid, an error message is displayed and the **csql** utility is terminated. If there is a password for the user name you specify, you will be prompted to enter the password.

```
csql -u DBA demodb
```

Specifying the user password (-p)

The following is an example to enter the password of the user specified with the **-p** option. Especially since there is no prompt to enter a password for the user you specify in a batch mode, you must enter the password using the **-p** option. When you enter an incorrect password, an error message is displayed and the **csql** utility is terminated.

```
csql -u DBA -p *** demodb
```

Executing SQL statements one by one (-s)

The following is an example to execute SQL statements one by one with the **-s** option. Use this option when you want to allocate less memory for the query execution. Multiple SQL statements are separated by semicolons (;).

```
csql -s -i infile demodb
```

Executing SQL statements directly from the shell (-c)

The following is an example to execute more than one SQL statement from the shell with the **-c** option. Multiple statements are separated by semicolons (;).

```
csql -c "select * from olympic;select * from stadium" demodb
```

Displaying the results in a line format (-l)

The following is an example to display the execution results of the SQL statement in a line format with the **-l** option. The execution results will be output in a column format if the **-l** option is not specified.

```
csql -l demodb
```

Continuing the execution even with an error (-e)

The following is an example to continue to execute subsequent SQL statements even when a syntax error or a runtime error occurs in a previous SQL statement by using the **-e** option. When there is an error in the SQL statement, the database will be terminated even though the **-e** option is specified.

```
csql -e demodb
```

Connecting to a database in read-only mode (-r)

The following is an example to connect to a database in read-only mode by using the **-r** option. Creating a table or manipulating data is not allowed; only retrieving data is allowed.

```
csql -r demodb
```

No auto-commit mode (--no-auto-commit)

The following is an example to stop the auto-commit mode with the **--no-auto-commit** option. If you don't configure **--no-auto-commit** option, the CSQL Interpreter runs in an auto-commit mode by default, and the SQL statement is committed automatically at every execution. Executing the **;AUtocommit** session command after starting the CSQL Interpreter will also have the same result.

```
csql --no-auto-commit demodb
```

Displaying all the execution results at once (--no-pager)

The following is an example to display the execution results by the CSQL Interpreter at once instead of page-by-page with the **--no-pager** option. The results will be output page-by-page if **--no-pager** option is not specified.

```
csql --no-pager demodb
```

Session Commands

In addition to SQL statements, CSQL Interpreter provides special commands allowing you to control the Interpreter. These commands are called session commands. All the session commands must start with a semicolon (;).

Session Commands

Enter the **;help** command to display a list of the session commands available in the CSQL Interpreter. Note that only the uppercase letters of each session command are required to make the CSQL Interpreter to recognize it. Session commands are not case-sensitive.

```
CUBRID SQL Interpreter
Type `;help' for help messages.
csql> ;help
=== <Help: Session Command Summary> ===
   All session commands should be prefixed by `;' and only blanks/tabs
   can precede the prefix. Capitalized characters represent the minimum
   abbreviation that you need to enter to execute the specified command.
   ;REAd   [<file-name>]       - read a file into command buffer.
   ;Write  [<file-name>]       - (over)write command buffer into a file.
   ;APpend [<file-name>]       - append command buffer into a file.
   ;PRINT                      - print command buffer.
   ;SHELL                      - invoke shell.
   ;CD                         - change current working directory.
   ;EXit                       - exit program.
   ;CLear                      - clear command buffer.
   ;EDIT                       - invoke system editor with command buffer.
   ;List                       - display the content of command buffer.
   ;RUn                        - execute sql in command buffer.
   ;Xrun                       - execute sql in command buffer, and clear the command
buffer.
   ;COmmit                     - commit the current transaction.
   ;ROllback                   - roll back the current transaction.
   ;AUtocommit [ON|OFF]        - enable/disable auto commit mode.
   ;CHeckpoint                 - issue checkpoint.
   ;Killtran                   - kill transaction.
   ;REStart                    - restart database.
   ;SHELL_Cmd  [shell-cmd]     - set default shell, editor, print and pager
   ;EDITOR_Cmd [editor-cmd]      command to new one, or display the current
   ;PRINT_Cmd  [print-cmd]       one, respectively.
   ;PAger_cmd  [pager-cmd]
   ;DATE                       - display the local time, date.
   ;DATAbase                   - display the name of database being accessed.
   ;SChema class-name          - display schema information of a class.
   ;SYntax [sql-cmd-name]      - display syntax of a command.
   ;TRigger [`*'|trigger-name] - display trigger definition.
   ;Get system_parameter       - get the value of a system parameter.
   ;SEt system_parameter=value - set the value of a system parameter.
   ;PLan [simple/detail/off]   - show query execution plan.
   ;Info <command>             - display internal information.
   ;TIme [ON/OFF]              - enable/disable to display the query execution time.
   ;HISTORYList                - display list of the executed queries.
   ;HISTORYRead <history_num>  - read entry on the history number into command buffer.
   ;HElp                       - display this help message.
csql>
```

Options

Reading SQL statements from a file (;REAd)

The **;REAd** command reads the contents of a file into the command buffer. This command is used to execute SQL commands saved in the specified file. To view the contents of the file loaded into the buffer, use the **;List** command.

```
csql> ;rea nation.sql
The file has been read into the command buffer.
csql> ;list
insert into "sport_event" ("event_code", "event_name", "gender_type", "num_player") values
```

```
(20001, 'Archery Individual', 'M', 1);
insert into "sport_event" ("event_code", "event_name", "gender_type", "num_player") values
20002, 'Archery Individual', 'W', 1);
....
```

Saving SQL statements into a file (;Write)

The **;Write** command saves the contents of the command buffer into a file. This command is used to save SQL commands that you entered or modified in the CSQL Interpreter.

```
csql> ;w outfile
Command buffer has been saved.
```

Appending to a file (;APpend)

This command appends the contents of the current command buffer to an **outfile** file.

```
csql> ;ap outfile
Command buffer has been saved.
```

Executing a shell command (;SHELL)

The **;SHELL** session command calls an external shell. Starts a new shell in the environment where the CSQL Interpreter is running. It returns to the CSQL Interpreter when the shell terminates. If the shell command to execute with the **;SHELL_Cmd** command has been specified, it starts the shell, executes the specified command, and returns to the CSQL Interpreter.

```
csql> ;shell
% Is -al
total 2088
drwxr-xr-x 16 DBA cubrid   4096 Jul 29 16:51 .
drwxr-xr-x  6 DBA cubrid   4096 Jul 29 16:17 ..
drwxr-xr-x  2 DBA cubrid   4096 Jul 29 02:49 audit
drwxr-xr-x  2 DBA cubrid   4096 Jul 29 16:17 bin
drwxr-xr-x  2 DBA cubrid   4096 Jul 29 16:17 conf
drwxr-xr-x  4 DBA cubrid   4096 Jul 29 16:14 cubridmanager
% exit
csql>
```

Registering a shell command (;SHELL_Cmd)

The **;SHELL_Cmd** command registers a shell command to execute with the **SHELL** session command. As shown in the example below, enter the **;shell** command to execute the registered command.

```
csql> ;shell_c ls -la
csql> ;shell
total 2088
drwxr-xr-x 16 DBA cubrid   4096 Jul 29 16:51 .
drwxr-xr-x  6 DBA cubrid   4096 Jul 29 16:17 ..
drwxr-xr-x  2 DBA cubrid   4096 Jul 29 02:49 audit
drwxr-xr-x  2 DBA cubrid   4096 Jul 29 16:17 bin
drwxr-xr-x  2 DBA cubrid   4096 Jul 29 16:17 conf
drwxr-xr-x  4 DBA cubrid   4096 Jul 29 16:14 cubridmanager
csql>
```

Changing the current working directory (;CD)

This command changes the current working directory where the CSQL Interpreter is running to the specified directory. If you don't specify the path, the directory will be changed to the home directory.

```
csql> ;cd /home1/DBA/CUBRID
Current directory changed to  /home1/DBA/CUBRID.
```

Exiting the CSQL Interpreter (;EXit)

This command exits the CSQL Interpreter.

```
csql> ;ex
```

Clearing the command buffer (;CLear)

The **;CLear** session command clears the contents of the command buffer.

```
csql> ;cl
csql> ;list
```

Displaying the contents of the command buffer (;List)

The **;List** session command lists the contents of the command buffer that have been entered or modified. The command buffer can be modified by **;READ** or **;Edit** command.

```
csql> ;l
```

Executing SQL statements (;RUn)

This command executes SQL statements in the command buffer. Unlike the **;Xrun** session command described below, the buffer will not be cleared even after the query execution.

```
csql> ;ru
```

Clearing the command buffer after executing the SQL statement (;Xrun)

This command executes SQL statements in the command buffer. The buffer will be cleared after the query execution.

```
csql> ;x
```

Committing transaction (;COmmit)

This command commits the current transaction. You must enter a commit command explicitly if it is not in auto-commit mode. In auto-commit mode, transactions are automatically committed whenever the session commands **;RUn** or **;Xrun** is executed.

```
csql> ;co
Current transaction has been committed.
```

Rolling back transaction (;ROllback)

This command rolls back the current transaction. Like a commit command (**;COmmit**), it must enter a rollback command explicitly if it is not in auto-commit mode (**OFF**).

```
csql> ;ro
Current transaction has been rolled back.
```

Setting the auto-commit mode (;AUtocommit)

This command sets auto-commit mode to **ON** or **OFF**. If any value is not specified, current configured value is applied by default. The default value is **ON**.

```
csql> ;au off
AUTOCOMMIT IS OFF
```

CHeckpoint Execution (;CHeckpoint)

This command executes the checkpoint within the CSQL session. This command can only be executed when a DBA group member, who is specified for the custom option (**-u** *user_name*), connects to the CSQL interpreter in system administrator mode (**--sysadm**).

Checkpoint is an operation of flushing log files (dirty pages) from the current data buffer to disks. You can also change the checkpoint interval using a command (**;set** *parameter_name* value) to set the parameter values in the CSQL session.

You can see the examples of the parameter related to the checkpoint execution interval (**checkpoint_interval_in_mins** and **checkpoint_every_npages**). For more information, see Logging-related Parameters.

```
csql> ;ch
Checkpoint has been issued.
```

Transaction Monitoring Or Termination (;Killtran)

This command checks the transaction status information or terminates a specific transaction in the CSQL session. This command prints out the status information of all transactions on the screen if a parameter is omitted it terminates the transaction if a specific transaction ID is specified for the parameter. It can only be executed when a DBA group member, who is specified for the custom option (**-u** *user_name*), connects to the CSQL interpreter in system administrator mode (**--sysadm**).

```
csql> ;k
Tran index      User name      Host name      Process id      Program name
-------------------------------------------------------------------------------
      1(+)            dba      myhost              664            cub_cas
      2(+)            dba      myhost             6700               csql
      3(+)            dba      myhost             2188            cub_cas
      4(+)            dba      myhost              696               csql
      5(+)         public      myhost             6944               csql

csql> ;k 3
The specified transaction has been killed.
```

Restarting database (;REStart)

A command that tries to reconnect to the target database in a CSQL session. Note that when you execute the CSQL interpreter in CS (client/server) mode, it will be disconnected from the server. When the connection to the server is lost due to a HA failure and failover to another server occurs, this command is particularly useful in connecting to the switched server while maintaining the current session.

```
csql> ;res
The database has been restarted.
```

Displaying the current date (;DATE)

The **;DATE** command displays the current date and time in the CSQL Interpreter.

```
csql> ;date
     Tue July 29 18:58:12 KST 2008
```

Displaying the database informatio (;DATAbase)

This command displays the database name and host name where the CSQL Interpreter is working. If the database is running, the HA mode (one of those followings: active, standby, or maintenance) will be displayed as well.

```
csql> ;data
     demodb@localhost (active)
```

Displaying schema information of a class (;SChema)

The **;SChema** session command displays schema information of the specified table. The information includes the table name, its column name and constraints.

```
csql> ;sc event
=== <Help: Schema of a Class> ===
 <Class Name>
     event
 <Attributes>
     code           INTEGER NOT NULL
     sports         CHARACTER VARYING(50)
     name           CHARACTER VARYING(50)
     gender         CHARACTER(1)
     players        INTEGER
 <Constraints>
     PRIMARY KEY pk_event_event_code ON event (code)
```

Displaying syntax (;SYntax)

This command displays the syntax of the SQL statement specified. If there is no specific syntax specified, all the syntaxes defined and their rules will be displayed.

```
csql> ;sy alter
=== <Help: Command Syntax> ===
 <Name>
   ALTER
 <Description>
Change the definition of a class or virtual class.
 <Syntax>
<alter> ::= ALTER [ <class_type> ] <class_name> <alter_clause> ;
<class_type> ::= CLASS | TABLE | VCLASS | VIEW
<alter_clause> ::= ADD <alter_add> [ INHERIT <resolution_comma_list> ] |
                   DROP <alter_drop> [ INHERIT <resolution_comma_list> ] |
                   RENAME <alter_rename> [ INHERIT <resolution_comma_list> ] |
>                   CHANGE <alter_change> |
                   INHERIT <resolution_comma_list>
<alter_add> ::= [ ATTRIBUTE | COLUMN ] <class_element_comma_list> |
                CLASS ATTRIBUTE <attribute_definition_comma_list> |
                FILE <file_name_comma_list> |
                METHOD <method_definition_comma_list> |
                QUERY <select_statement> |
                SUPERCLASS <class_name_comma_list>
......
```

Displaying the trigger (;TRriger)

This command searches and displays the trigger specified. If there is no trigger name specified, all the triggers defined will be displayed.

```
csql> ;tr
=== <Help: All Triggers> ===
     trig_delete_contents
```

Checking the parameter value(;Get)

You can check the parameter value currently set in the CSQL Interpreter using the **;Get** session command. An error occurs if the parameter name specified is incorrect.

```
csql> ;g isolation_level
=== Get Param Input ===
isolation_level=4
```

Setting the parameter value (;SEt)

You can use the **;Set** session command to set a specific parameter value. Note that only client parameter values can be changed. Server parameter values cannot be changed.

```
csql> ;se block_ddl_statement=1
=== Set Param Input ===
block_ddl_statement=1
```

Setting the view level of executing query plan (;PLan)

You can use the **;PLan** session command to set the view level of executing query plan the level is composed of **simple**, **detail**, and **off**. Each command refers to the following:

- **off** : Not displaying the query execution plan
- **simple** : Displaying the query execution plan in simple version (OPT LEVEL=257)
- **detail** : Displaying the query execution plan in detailed version (OPT LEVEL=513)

Displaying information (;Info)

The **;Info** session command allows you to check information such as schema, triggers, the working environment, locks and statistics.

```
csql> ;i lock
*** Lock Table Dump ***
```

```
 Lock Escalation at = 100000, Run Deadlock interval = 1
Transaction (index  0, unknown, unknown@unknown|-1)
Isolation REPEATABLE CLASSES AND READ UNCOMMITTED INSTANCES
State TRAN_ACTIVE
Timeout_period -1
......
```

Outputting statistics information of server processing (;.Hist)

This command shows the statistics information of server processing. The information is collected after this command is entered. Therefore, the execution commands such as **.dump_hist** or **.x** must be entered to output the statistics information while a parameter, **communication_histogram**, is set to **yes.**
You can also view this information by using the **cubrid statdump** utility. Following options are provided for this session command.

- **on** : Outputs statistics information for the current transaction.
- **all** : Outputs statistics information for all server processing.
- **off** : Does not collect statistics information of server.

This example shows the statistics information for all server processing. For information on specific items, see Outputting Statistics Information of Server.

```
csql> ;.hist all
…
--Executing queries
…
csql> ;.x

 *** SERVER EXECUTION GLOBAL STATISTICS ***
Num_data_page_fetches  = 938215
Num_data_page_dirties  = 851792
Num_data_page_ioreads  = 15050
Num_data_page_iowrites = 0
Num_iosynches          = 15187
Num_log_page_ioreads   = 0
Num_log_page_iowrites  = 15187
Num_log_append_records = 625838
Num_log_archives       = 0
Num_log_checkpoints    = 0
Num_tran_commits       = 40080
Num_tran_rollbacks     = 40
Hit ratio of Page Buffer = 98.40%

csql> ;.h off
```

Displaying query execution time (;TIme)

The **;TIme** session command can be set to display the elapsed time to execute the query. It can be set to **ON** or **OFF**. The current setting is displayed if there is no value specified. csql> ;ti ON

```
csql> ;ti
TIME IS ON
```

Displaying query history (;HISTORYList)

This command displays the list that contains previously executed commands (input) and their history numbers.

```
csql> ;historyl
----< 1 >----
select * from nation;
----< 2 >----
select * from athlete;
```

Reading input with the specified history number into the buffer (;HISTORYRead)

You can use **;HISTORYRead** session command to read input with history number in the **;HISTORYList** list into the command buffer. You can enter **;ru** or **;x** directly because it has the same effect as when you enter SQL statements directly.

```
csql> ;historyr 1
```

Calling the default editor (;EDIT)

This command calls the specified editor. The default editor is **vi** in Linux environment **Notepad** in Windows environment. Use **;EDITOR_Cmd** command to specify a different editor.

```
csql> ;edit
```

Specifying the editor (;EDITOR_Cmd)

This command specifies the editor to be used with **;EDIT** session command. As shown in the example below, you can specify other editor (ex: **emacs**) which is installed in the system.

```
csql> ;editor_c emacs
csql> ;edit
```

CUBRID SQL Guide

This chapter describes SQL syntax such as data types, functions and operators, data retrieval or table manipulation. You can also find SQL statements used for index, trigger, partition, serial and changing user information.

The main topics covered in this chapter are as follows:

- Data types
- Operators and functions
- Data retrieval
- Query optimization
- Table manipulation
- Data manipulation
- Virtual tables (VIEW)
- Indexes (INDEX)
- Transaction management
- Triggers (TRIGGER)
- Methods
- Partitions
- Serials
- Grant access statements
- Java stored functions/procedures
- CUBRID SQL statements

Glossary

CUBRID is an object-relational database management system (ORDBMS), which supports object-oriented concepts such as inheritance. In this manual, relational database terminology is also used along with object-oriented terminology for better understanding. Object-oriented terminology such as class, instance and attribute is used to describe concepts including inheritance, and relational database terminology is mainly used to describe common SQL syntax.

The following table provides the summary:

Relational Database	CUBRID
table	class, table
column	attribute, column
record	instance, record
data type	domain, data type

Comment

The CSQL Interpreter is a SQL-style method; the SQL-style comment starts with the double dashes (--) and the comment line after the double dashes is regarded as comment. Additionally, it supports C++ style, which start with double slashes (//), and C-style, which starts and ends with '/*' and '*/' respectively.

The following are examples of comments supported in the CSQL Interpreter.

Example

- --

```
-- This is a SQL-style comment.
```

- //

```
This is a C++ style comment.
```

- /* */

```
/* This is a C-style comment.*/
/* This is an example to use two lines
as comment by using the C-style. */
```

Identifier

Guidelines for Creating Identifiers

The guidelines for creating identifiers in the CSQL Interpreter are as follows:

- An identifier must begin with a letter it must not begin with a number or a symbol.
- It is not case-sensitive.
- CUBRID keywords are not allowed.

< identifier>

```
:: = < identifier_letter> [ { < other_identifier>   } ]
```

< identifier_letter>

```
:: = < upper_case_letter>
  | < lower_case_letter>
```

< other_identifier>

```
:: = < identifier_letter>
  | < digit>
  | _
  | #

```

< digit>

```
:: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

< upper_case_letter>

```
:: = A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P| Q | R | S | T | U | V
| W | X | Y | Z
```

< lower_case_letter>

```
:: = a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p| q | r | s | t | u | v
| w | x | y | z
```

Legal Identifiers

Beginning with a Letter

An identifier must begin with a letter. All other special characters except operator characters are allowed. The following are examples of legal identifiers.

```
a
a_b
ssn#
fg%
this_is_an_example_#%%#
```

Enclosing in Double Quotes, Square Brackets, or Backtick Symbol

Identifiers or a reserved keywords shown as below are not allowed' however, if they are enclosed in in double quotes, square brackets, or backtick symbol, they are allowed as an exception. Especially, the double quotations can be used as a symbol enclosing identifiers when the **ansi_quote** parameter is set to **yes**. If this value is set to **no**, double quotations are used as a symbol enclosing character strings. The followings are examples of legal identifiers.

```
" select"
" @lowcost"
" low cost"
" abc" " def"
[position]
```

Illegal Identifiers

Beginning with special characters or numbers

An identifier starting with a special character or a number is not allowed. As an exception, a underline (_) and a sharp symbol (#) are allowed for the first character.

```
_a
#ack
%nums
2fer
88abs
```

An identifier containing a space

An identifier that a space within characters is not allowed.

```
coll tl
```

An identifier containing operator special characters

An identifier which contains operator special characters (+, -, *, /, %, ||, !, < , > , =, |, ^, & , ~) is not allowed.

```
col+
col~
col& &
```

Reserved Words

The following keywords are previously reserved as a command, a function name or a type name in CUBRID. You are restricted to use these words for a class name, an attribute name, a variable name. Note than these reserved keywords can be used an identifier when they are enclosed in double quotes, square brackets, or backtick symbol (`).

ABSOLUTE	ACTION	ADD
ADD_MONTHS	AFTER	ALIAS
ALL	ALLOCATE	ALTER
AND	ANY	ARE
AS	ASC	ASSERTION
ASYNC	AT	ATTACH
ATTRIBUTE	AVG	
BEFORE	BETWEEN	BIGINT
BIT	BIT_LENGTH	BLOB
BOOLEAN	BOTH	BREADTH
BY		
CALL	CASCADE	CASCADED
CASE	CAST	CATALOG
CHANGE	CHAR	CHARACTER
CHECK	CLASS	CLASSES
CLOB	CLOSE	CLUSTER
COALESCE	COLLATE	COLLATION
COLUMN	COMMIT	COMPLETION
CONNECT	CONNECT_BY_ISCYCLE	CONNECT_BY_ISLEAF
CONNECT_BY_ROOT	CONNECTION	CONSTRAINT
CONSTRAINTS	CONTINUE	CONVERT
CORRESPONDING	COUNT	CREATE
CROSS	CURRENT	CURRENT_DATE
CURRENT_DATETIME	CURRENT_TIME	CURRENT_TIMESTAMP
CURRENT_USER	CURSOR	CYCLE
DATA	DATA_TYPE	DATABASE
DATE	DATETIME	DAY
DAY_HOUR	DAY_MILLISECOND	DAY_MINUTE
DAY_SECOND	DEALLOCATE	DEC
DECIMAL	DECLARE	DEFAULT
DEFERRABLE	DEFERRED	DELETE
DEPTH	DESC	DESCRIBE
DESCRIPTOR	DIAGNOSTICS	DICTIONARY
DIFFERENCE	DISCONNECT	DISTINCT

DISTINCTROW	DIV	DO
DOMAIN	DOUBLE	DUPLICATE
DROP		
EACH	ELSE	ELSEIF
END	EQUALS	ESCAPE
EVALUATE	EXCEPT	EXCEPTION
EXCLUDE	EXEC	EXECUTE
EXISTS	EXTERNAL	EXTRACT
FALSE	FETCH	FILE
FIRST	FLOAT	FOR
FOREIGN	FOUND	FROM
FULL	FUNCTION	
GENERAL	GET	GLOBAL
GO	GOTO	GRANT
GROUP	HAVING	HOUR
HOUR_MILLISECOND	HOUR_MINUTE	HOUR_SECOND
IDENTITY	IF	IGNORE
IMMEDIATE	IN	INDEX
INDICATOR	INHERIT	INITIALLY
INNER	INOUT	INPUT
INSERT	INT	INTEGER
INTERSECT	INTERSECTION	INTERVAL
INTO	IS	ISOLATION
JOIN		
KEY		
LANGUAGE	LAST	LDB
LEADING	LEAVE	LEFT
LESS	LEVEL	LIKE
LIMIT	LIST	LOCAL
LOCAL_TRANSACTION_ID	LOCALTIME	LOCALTIMESTAMP
LOOP	LOWER	
MATCH	MAX	METHOD
MILLISECOND	MIN	MINUTE
MINUTE_MILLISECOND	MINUTE_SECOND	MOD
MODIFY	MODULE	MONETARY
MONTH	MULTISET	MULTISET_OF
NA	NAMES	NATIONAL
NATURAL	NCHAR	NEXT
NO	NONE	NOT
NULL	NULLIF	NUMERIC

OBJECT	OCTET_LENGTH	OF
OFF	OID	ON
ONLY	OPEN	OPERATION
OPERATORS	OPTIMIZATION	OPTION
OR	ORDER	OTHERS
OUT	OUTER	OUTPUT
OVERLAPS		
PARAMETERS	PARTIAL	PENDANT
POSITION	PRECISION	PREORDER
PREPARE	PRESERVE	PRIMARY
PRIOR	PRIVATE	PRIVILEGES
PROCEDURE	PROTECTED	PROXY
QUERY		
READ	REAL	RECURSIVE
REF	REFERENCES	REFERENCING
REGISTER	RELATIVE	RENAME
REPLACE	RESIGNAL	RESTRICT
RETURN	RETURNS	REVOKE
RIGHT	ROLE	ROLLBACK
ROLLUP	ROUTINE	ROW
ROWNUM	ROWS	
SAVEPOINT	SCHEMA	SCOPE
SCROLL	SEARCH	SECOND
SECOND_MILLISECOND	SECTION	SELECT
SENSITIVE	SEQUENCE	SEQUENCE_OF
SERIALIZABLE	SESSION	SESSION_USER
SET	SET_OF	SETEQ
SHARED	SIBLINGS	SIGNAL
SIMILAR	SIZE	SMALLINT
SOME	SQL	SQLCODE
SQLERROR	SQLEXCEPTION	SQLSTATE
SQLWARNING	STATISTICS	STRING
STRUCTURE	SUBCLASS	SUBSET
SUBSETEQ	SUBSTRING	SUM
SUPERCLASS	SUPERSET	SUPERSETEQ
SYS_CONNECT_BY_PATH	SYS_DATE	SYS_DATETIME
SYS_TIME	SYS_TIMESTAMP	SYS_USER
SYSDATE	SYSDATETIME	SYSTEM_USER
SYSTIME		
TABLE	TEMPORARY	TEST

THEN	THERE	TIME
TIMESTAMP	TIMEZONE_HOUR	TIMEZONE_MINUTE
TO	TRAILING	TRANSACTION
TRANSLATE	TRANSLATION	TRIGGER
TRIM	TRUE	TRUNCATE
TYPE		
UNDER	UNION	UNIQUE
UNKNOWN	UPDATE	UPPER
USAGE	USE	USER
USING	UTIME	
VALUE	VALUES	VARCHAR
VARIABLE	VARYING	VCLASS
VIEW	VIRTUAL	VISIBLE
WAIT	WHEN	WHENEVER
WHERE	WHILE	WITH
WITHOUT	WORK	WRITE
XOR		
YEAR	YEAR_MONTH	
ZONE		

Data Types

Numeric Types

Definition and Characteristics

Definition

CUBRID supports the following numeric data types to store integers or real numbers.

Numeric Types Supported by CUBRID

Type	Bytes	Mix	Max	Exact/Approx.
SHORT SMALLINT	2	-32,768	+32,767	exact numeric
INT INTEGER	4	-2,147,483,648	+2,147,483,647	exact numeric
BIGINT	8	-9,223,372,036,854,775,808	+9,223,372,036,854,775,807	exact numeric
NUMERIC DECIMAL	16	precision p : 1 scale s : 0	precision p : 38 scale s : 38	exact numeric
FLOAT REAL	4	-3.402823466E+38 (ANSI/IEEE 754-1985 standard)	+3.402823466E+38 (ANSI/IEEE 754-1985 standard)	approximate numeric floating point : 7
DOUBLE DOUBLE PRECISION	8	-1.7976931348623157E+308 ANSI/IEEE 754-1985 standard)	+1.7976931348623157E+308(ANSI/IEEE 754-1985 standard)	approximate numeric floating point : 15
MONETARY	12	-3.402823466E+38	+3.402823466E+38	approximate numeric

Numeric data types are divided into exact and approximate types. Exact numeric data types (**SMALLINT**, **INT**, **BIGINT**, **NUMERIC**) are used for numbers whose values must be precise and consistent, such as the numbers used in financial accounting. Note that even when the literal values are equal, approximate numeric data types (**FLOAT**, **DOUBLE**, **MONETARY**) can be interpreted differently depending on the system.

CUBRID does not support the UNSIGNED type for numeric data types.

Characteristics

Precision and Scale

The precision of numeric data types is defined as the number of significant digits. This applies to both exact and approximate numeric data types.

The scale represents the number of digits following the decimal point. It is significant only in exact numeric data types. Attributes declared as exact numeric data types always have fixed precision and scale. **NUMERIC** (or **DECIMAL**) data type always has at least one-digit precision, and the scale should be between 0 and the precision declared. Scale cannot be greater than precision. For **INTEGER**, **SMALLINT**, or **BIGINT** data types, the scale is 0 (i.e. no digits following the decimal point), and the precision is fixed by the system.

Numeric Literals

Special signs can be used to input numeric values. The plus sign (+) and minus sign (-) are used to represent positive and negative numbers respectively. You can also use scientific notations. In addition, you can use currency signs specified in the system to represent currency values. The maximum precision that can be expressed by a numeric literal is 255.

Numeric Coercions

All numeric data type values can be compared with each other. To do this, automatic coercion to the common numeric data type is performed. For explicit coercion, use the **CAST** operator. When different data types are sorted or calculated in a numerical expression, the system performs automatic coercion. For example, when adding a **FLOAT** attribute value to an **INTEGER** attribute value, the system automatically coerces the **INTEGER** value to the most approximate **FLOAT** value before it performs the addition operation.

Caution Earlier version than CUBRID 2008 R2.0, the input constant value exceeds **INTEGER**, it is handled as **NUMERIC**. However, 2008 R2.0 or later versions, it is handled as **BIGINT**.

INT/INTEGER

Description

The **INTEGER** data type is used to represent integers. The value range is available is from -2,147,483,648 to +2,147,483,647. **SMALLINT** is used for small integers, and **BIGINT** is used for big integers.

```
INTEGER
```

Remark

- If a real number is entered for an **INT** type, the number is rounded to zero decimal place and the integer value is stored.
- **INTEGER** and **INT** are used interchangeably.

Example

```
If you specify 8934 as INTEGER, 8934 is stored.
If you specify 7823467 as INTEGER, 7823467 is stored.
If you specify 89.8 to an INTEGER, 90 is stored (all digits after the decimal point are
rounded).
If you specify 3458901122 as INTEGER, an error occurs (if the allowable limit is exceeded).
```

SHORT/SMALLINT

Description

The **SMALLINT** data type is used to represent a small integer type. The value range is available is from -32,768 to +32,767.

```
SMALLINT
```

Remark

If a real number is entered for an **SMALLINT** type, the number is rounded to zero decimal place and the integer value is stored.

Example

```
If you specify 8934 as SMALLINT, 8934 is stored.
If you specify 34.5 as SMALLINT, 35 is stored (all digits after the decimal point are
rounded).
If you specify 23467 as SMALLINT, 23467 is stored.
If you specify 89354 as SMALLINT, an error occurs (if the allowable limit is exceeded).
```

BIGINT

Description

The **BIGINT** data type is used to represent big integers. The value range is available from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

```
BIGINT
```

Remark

- If a real number is entered for a **BIG** type, the number is rounded to zero decimal place and the integer value is stored.
- Based on the precision and the range of representation, the following order applies.

SMALLINT ⊂ INTEGER ⊂ BIGINT ⊂ NUMERIC

Example

```
If you specify 8934 as BIGINT, 8934 is stored.
If you specify 89.1 as BIGINT, 89 is stored.
If you specify 89.8 as BIGINT, 90 is stored (all digits after the decimal point are
rounded).
If you specify 3458901122 as BIGINT, 3458901122 is stored.
```

NUMERIC/DECIMAL

Description

NUMERIC or **DECIMAL** data types are used to represent fixed-point numbers. As an option, the total number of digits (precision) and the number of digits after the decimal point (scale) can be specified for definition. The minimum value for the precision *p* is 1. When the precision *p* is omitted, you cannot enter data whose integer part exceeds 15 digits because the default value is 15. If the scale s is omitted, an integer rounded to the first digit after the decimal point is returned because the default value is 0.

```
NUMERIC [(p [, s])]
```

Remark

- Precision must be equal to or greater than scale.
- Precision must be equal to or greater than the number of integer digits + scale.
- **NUMERIC**, **DECIMAL**, and **DEC** are used interchangeably.

Example

```
If you specify 12345.6789 as NUMERIC, 12346 is stored (it rounds to the first place after
the decimal point since 0 is the default value of scale).
If you specify 12345.6789 as NUMERIC(4), an error occurs (precision must be equal to or
greater than the number of integer digits).
If you declare NUMERIC(3,4), an error occurs (precision must be equal to or greater than
the scale).
If you specify 0.12345678 as NUMERIC(4,4), .1235 is stored (it rounds to the fifth place
after the decimal point).
If you specify -0.123456789 as NUMERIC(4,4), -.1235 is stored (it rounds to the fifth
place after decimal point and then prefixes a minus (-) sign).
```

FLOAT/REAL

Description

The **FLOAT** (or **REAL**) data type is used to represent floating point numbers. The value range is available from -3.402823466E+38 to -1.175494351E-38, 0, and from +1.175494351E-38 to +3.402823466E+38. It conforms to the ANSI/IEEE 754-1985 standard.

The minimum value for the precision p is 1 and the maximum value is 38. When the precision *p* is omitted or it is specified as seven or less, the data is represented as single precision (in seven significant figures) and it is converted into the **DOUBLE** data type.

```
FLOAT[(p)]
```

Remark

- **FLOAT** is in seven significant figures.
- Representable range is different based on system where CUBRID is running.
- Extra cautions are required when comparing data because the **FLOAT** type stores approximate numeric.
- **FLOAT** and **REAL** are used interchangeably.

Example

```
If you specify -1234.56789 as FLOAT, -1.234568e+003 is stored (if precision is omitted,
8th digit is rounded because it is represented as seven significant figures).
If you specify 1234.56789 as FLOAT(5), 1.234568e+003 is stored (if precision is in seven
or less, 8th digit is rounded because it is represented as seven significant figures).
If you specify 12345678.9 as FLOAT(5), 1.234568e+007 is stored (if precision is in seven
or less, 8th digit is rounded because it is represented as seven significant figures).
If you specify 12345678.9 as FLOAT(10), 1.234567890000000e+007 is stored (if precision is
in seven or greater and 38 or less, 0s are filled because it is represented as 15
significant figures).
```

DOUBLE/DOUBLE PRECISION

Description

The **DOUBLE** data type is used to represent floating point numbers. The value range is available from -1.7976931348623157E+308 to 2.2250738585072014E-308, 0, and from 2.2250738585072014E-308 to 1.7976931348623157E+308. It conforms to the ANSI/IEEE 754-1985 standard.

The precision *p* is not specified. The data specified as this data type is represented as double precision (in 15 significant figures).

```
DOUBLE
```

Remark

- **DOUBLE** is in 15 significant figures.
- Representable range is different based on system where CUBRID is running.
- Extra caution is required when comparing data because the DOUBLE type stores approximate numeric.
- **DOUBLE** and **DOUBLE PRECISION** are used interchangeably.

Example

```
If you specify 1234.56789 as DOUBLE, 1.234567890000000e+003 is stored.
```

MONETARY

Description

MONETARY data type is an approximate numeric data type. Representable range is the same as FLOAT, which is represented to two decimal places; the representable range can be different based on system. A comma is appended to every 1000th place.

```
MONETARY
```

Remark

You can use a dollar sign or a decimal point, but a comma is not allowed.

Example

```
If you specify 12345.67898934 as MONETARY, $12,345.68 is stored (it is rounded to third
decimal place).
If you specify 123456789 as MONETARY, $123,456.789.00 is stored.
```

Date/Time Types

Definition and Characteristics

Definition

DATE-TIME data types are used to represent the date or time (or both together). CUBRID supports the following data types:

Date-Time Types Supported by CUBRID

Type	Mim	Max	Note
DATE	0001-01-01	9999-12-31	0-0-0 is not allowed.
TIME	00:00:00	23:59:59	0:0:0 is not allowed.
TIMESTAMP	1970-01-01 00:00:00(GMT) 1970-01-01 09:00:00(KST)	2038-01-10 03:14:07 (GMT) 2038-01-19 12:14:07 (KST)	Note that TIMESTAMP at the point of entering data is not stored even though data is inserted into or updated in the TIMESTAMP column.
DATETIME	0001-01-01 00:00:000	9999-12-31 23:59:599	

Characteristics

Range and Resolution

- By default, the range of a time value is represented by the 24-hour system. Dates follow the Gregorian calendar. An error occurs if a value that does not meet these two constraints is entered as a date or time.
- The range of year in **DATE** is 0001 - 9999 AD.
- From the CUBRID 2008 R3.0 version,if time value is represented with two-digit numbers, a number from 00 to 69 is converted into a number from 2000 to 2069; a number from 70 to 99 is converted into a number from 1970 to 1999. In earlier than CUBRID 2008 R3.0 version, if time value is represented with two-digit numbers, a number from 01 to 99 is converted into a number from 0001 to 0099.
- The range of **TIMESTAMP** is from January 1, 1970 00:00:00 GMT to January 19, 2038 03:14:07. For KST (GMT+9), values from January 1, 1970 00:00:00 to January 19, 2038 12:14:07 can be stored.
- The results of date, time and timestamp operations may differ depending on the rounding mode. In these cases, for Time and Timestamp, the most approximate second is used as the minimum resolution; for Date, the most approximate date is used as the minimum resolution.

Coercions

The **Date-Time** types can be cast explicitly using the **CAST** operator only when they have the same field and implicit coercion is not performed. The following table shows types that allows explicit coercions. For implicit coercion, see Arithmetic Operation and Type Casting of DATE/TIME Data Types.

Explicit Coercions

FROM TO	DATE	TIME	DATETIME	TIMESTAMP
DATE	--	X	O	O
TIME	X	--	X	X
TIMESTAMP	O	O	--	O
DATETIME	O	O	O	--

DATE

Description

The **DATE** data type is used to represent the year (yyyy), month (mm) and day (dd). Supported range is "01/01/0001" to "12/31/9999." The year can be omitted. If it is, the year value of the current system is specified automatically.

Output and input formats are as follows:

```
'mm/dd[/yyyy]'
'[yyyy-]mm-dd'
```

Remark

- All fields must be entered as integer.
- The date value is outputted in the format of 'MM/DD/YYYY' in CSQL, and it is outputted in the format of 'YYYY-MM-DD' in JDBC application programs and the CUBRID Manager.
- The **TO_DATE**() function is used to convert a character string type into a **DATE** type.

Example

```
DATE '2008-10-31' is stored as '10/31/2008'.
DATE '10/31' is stored as '10/31/2010'(if a value for year is omitted, the current year is
automatically specified).
DATE '00-10-31' is stored as '10/31/2000'.
DATE '0000-10-31' is handled as an error (a year value should be at least 1).
DATE '70-10-31' is stored as '10/31/1970'.
DATE '0070-10-31' is stored as '10/31/0070'.
```

TIME

Description

The **TIME** data type is used to represent the hour (hh), minute (mm) and second (ss). Supported range is "00:00:00" to "23:59:59." Second can be omitted; if it is, 0 seconds is specified. Both 12-hour and 24-hour notations are allowed as an input format.

The input format of **TIME** is as follows:

```
'hh:mi [:ss] [am | pm]'
```

Remark

- All items must be entered as integer.
- AM/PM time notation is used to display time in the CSQL; while the 24-hour notation is used in the CUBRID Manager.
- AM/PM can be specified in the 24-hour notation. An error occurs if the time specified does not follow the AM/PM format.
- Every time value is stored in the 24-hour notation. **db_time_decode**, one of C API functions, is used to return a value in the 24-hour notation.
- The **TO_TIME**() function is used to return a character string type into a TIME type.

Example

```
TIME '00:00:00’ is outputted as '12:00:00 AM'.
TIME '1:15' is regarded as '01:15:00 AM'.
TIME '13:15:45' is regarded as '01:15:45 PM'.
TIME '13:15:45 pm' is stored normally.
TIME '13:15:45 am' is an error (an input value does not match the AM/PM format).
```

TIMESTAMP

Description

The **TIMESTAMP** data type is used to represent a data value in which the date (year, month, date) and time (hour, minute, second) are combined. Representable range is from GMT 1970-01-01 00:00:00 to 2038-01-19 03:14:07. The **DATETIME** type can be used if the value exceeds the range or the time data in milliseconds is stored.

The input format of **TIMESTAMP** is as follows:

```
'hh:mi [:ss] [am|pm] mm/dd [/yyyy]'
'hh:mi [:ss] [am|pm] mm/dd [/yyyy]'
'hh:mi [:ss] [am|pm] [yyyy-]mm-dd'

'mm/dd [/yyyy] hh:mi [:ss] [am|pm]'
'[yyyy-]mm-dd hh:mi [:ss] [am|pm]'
```

Remark

- All fields must be entered in integer format.
- If the year is omitted, the current year is specified by default. If the time value (hour/minute/second) is omitted, 12:00:00 AM is specified.
- You can store the timestamp value of the system in the **TIMESTAMP** type by using the **SYS_TIMESTAMP** (or **SYSTIMESTAMP**, **CURRENT_TIMESTAMP**) function. Note that the timestamp value is specified as a default value at the time of creating the table, not at the time of **INSERT** the data, if **SYS_TIMESTAMP** is specified as a **DEFAULT** value for a **TIMESTAMP** column when creating a table.
- The **TIMESTAMP()** or **TO_TIMESTAMP**() function is used to cast a character string type into a **TIMESTAMP** type.

Example

```
TIMESTAMP '10/31' is outputted as '12:00:00 AM 10/31/2010' (if year/time is omitted, a
default value is outputted).
TIMESTAMP '10/31/2008' is outputted as '12:00:00 AM 10/31/2008' (if time is omitted, a
default value is outputted).
TIMESTAMP '13:15:45 10/31/2008' is outputted as '01:15:45 PM 10/31/2008'.
TIMESTAMP '01:15:45 PM 2008-10-31' is outputted '01:15:45 PM 10/31/2008'.
TIMESTAMP '13:15:45 2008-10-31' is outputted as '01:15:45 PM 10/31/2008'.
TIMESTAMP '10/31/2008 01:15:45 PM' is outputted as '01:15:45 PM 10/31/2008'.
TIMESTAMP '10/31/2008 13:15:45' is outputted as '01:15:45 PM 10/31/2008'.
TIMESTAMP '2008-10-31 01:15:45 PM' is outputted as '01:15:45 PM 10/31/2008'.
TIMESTAMP '2008-10-31 13:15:45' is outputted as '01:15:45 PM 10/31/2008'.
TIMESTAMP '2099-10-31 01:15:45 PM' is an error (un-representable TIMESTAMP)
```

DATETIME

Description

The **DATETIME** data type is used to represent a data value in which the data (year, month, date) and time (hour, minute, second) are combined. Representable range is from GMT 0001-01-01 00:00:00.000 to 9999-12-31 23:59:59.999.

```
'hh:mi [:ss[.ff]] [am|pm] mm/dd [/yyyy]'
'hh:mi [:ss[.ff]] [am|pm] [yyyy-]mm-dd'
'mm/dd[/yyyy] hh:mi[:ss[.ff]] [am|pm]'
'[yyyy-]mm-dd hh:mi[:ss[.ff]] [am|pm]'
```

Remark

- All fields must be entered as integer.
- If you year is omitted, the current year is specified by default. If the value (hour, minute/second) is omitted, 12:00:00.000 AM is specified.
- You can store the timestamp value of the system in the **DATETIME** type by using the **SYS_DATETIME** (or **SYSDATETIME**, **CURRENT_DATETIME, CURRENT_DATETIME()**, **NOW()**) function. Note that the timestamp value is specified as a default value at the time of creating the table, not at the time of **INSERT** the data, if **SYS_DATETIME** is specified as a **DEFAULT** value for a **DATETIME** column when creating a table.

- The **TO_DATETIME**() function is used to cast a character string type into a **DATETIME** type.

Example

```
DATETIME '10/31' is outputted as '12:00:00.000 AM 10/31/2010' (if year/time is omitted, a
default value is outputted).
DATETIME '10/31/2008' is outputted as '12:00:00.000 AM 10/31/2008'.
DATETIME '13:15:45 10/31/2008' is outputted as '01:15:45.000 PM 10/31/2008'.
DATETIME '01:15:45 PM 2008-10-31' is outputted as '01:15:45.000 PM 10/31/2008'.
DATETIME '13:15:45 2008-10-31' is outputted as '01:15:45.000 PM 10/31/2008'.
DATETIME '10/31/2008 01:15:45 PM' is outputted as '01:15:45.000 PM 10/31/2008'.
DATETIME '10/31/2008 13:15:45' is outputted as '01:15:45.000 PM 10/31/2008'.
DATETIME '2008-10-31 01:15:45 PM' is outputted as '01:15:45.000 PM 10/31/2008'.
DATETIME '2008-10-31 13:15:45' is outputted as '01:15:45.000 PM 10/31/2008'.
DATETIME '2099-10-31 01:15:45 PM' is outputted as '01:15:45.000 PM 10/31/2099'.
```

Bit Strings

Definition and Characteristics

Definition

A bit string is a sequence of bits (1's and 0's). Images (bitmaps) displayed on the computer screen can be stored as bit strings. CUBRID supports the following two types of bit strings:

- Fixed-length bit string (**BIT**)
- Variable-length bit string (**BIT VARYING**)

A bit string can be used as a method argument or an attribute domain. Bit string literals are represented in a binary or hexadecimal format. For binary format, append the string consisting of 0's and 1's to the letter **B** or append a value to the **0b** as shown example below.

```
B'1010'
0b1010
```

For hexadecimal format, append the string consisting of the numbers 0 - 9 and the letters A - F to the uppercase letter **X** or append a value to the **0x**. The following is hexadecimal representation of the same number that was represented above in binary format.

```
X'a'
0xA
```

The letters used in hexadecimal numbers are not case-sensitive. That is, X'4f' and X'4F' are considered as the same value.

Characteristics

Length

If a bit string is used in table attributes or method declarations, you must specify the maximum length. The maximum length for a bit string is 1,073,741,823 bits.

Bit String Coercion

Automatic coercion is performed between a fixed-length and a variable-length bit string for comparison. For explicit coercion, use the **CAST** operator.

BIT(n)

Description

Fixed-length binary or hexadecimal bit strings are represented as **BIT**(*n*), where *n* is the maximum number of bits. If *n* is not specified, the length is set to 1.

Remark

- *n* must be a number greater than 0.
- If the length of the string exceeds *n*, it will be processed as an error.
- If a bit string smaller than *n* is stored, the remainder of the string is filled with 0s.

Example

```
CREATE TABLE bit_tbl(a1 BIT, a2 BIT(1), a3 BIT(8), a4 BIT VARYING);
INSERT INTO bit_tbl VALUES (B'1', B'1', B'1', B'1');
INSERT INTO bit_tbl VALUES (0b1, 0b1, 0b1, 0b1);
INSERT INTO bit_tbl(a3,a4) VALUES (B'1010', B'1010');
INSERT INTO bit_tbl(a3,a4) VALUES (0xaa, 0xaa);
SELECT * FROM bit_tbl;
;xr

=== <Result of SELECT Command in Line 6> ===

  a1                    a2                    a3                    a4

=========================================================================
  X'8'                  X'8'                  X'80'                 X'8'
  X'8'                  X'8'                  X'80'                 X'8'
  NULL                  NULL                  X'a0'                 X'a'
  NULL                  NULL                  X'aa'                 X'aa'
```

BIT VARYING(n)

Description

A variable-length bit string is represented as **BIT VARYING**(*n*), where *n* is the maximum number of bits. If *n* is not specified, the length is set to 1,073,741,823 (maximum value).

Remark

- If the length of the string exceeds *n*, it will be processed as an error.
- The remainder of the string is not filled with 0s even if a bit string smaller than *n* is stored.
- *n* must be a number greater than 0.

Example

```
CREATE TABLE bitvar_tbl(a1 BIT VARYING, a2 BIT VARYING(8));
INSERT INTO bitvar_tbl VALUES (B'1', B'1');
INSERT INTO bitvar_tbl VALUES (0b1010, 0b1010);
INSERT INTO bitvar_tbl VALUES (0xaa, 0xaa);
INSERT INTO bitvar_tbl(a1) VALUES (0xaaa);
SELECT * FROM bitvar_tbl;
;xr

=== <Result of SELECT Command in Line 6> ===

  a1                    a2
============================================
  X'8'                  X'8'
  X'a'                  X'a'
  X'aa'                 X'aa'
  X'aaa'                NULL

INSERT INTO bitvar_tbl(a2) VALUES (0xaaa);
;xr

ERROR: Data overflow coercing X'aaa' to type bit varying.
```

Character Strings

Definition and Characteristics

Definition

CUBRID supports the following four types of character strings:

- Fixed-length character string: **CHAR**(*n*)
- Variable-length character string: **VARCHAR**(*n*)
- Fixed-length national character string: **NCHAR**(*n*)
- Variable-length national character string: **NCHAR VARYING**(*n*)

The followings are the rules that are applied when using the character string types.

- In general, single quotations are used to enclose character string. Double quotations may be used as well depending on the value of **ansi_quotes**, which is a parameter related to SQL statement. If the **ansi_quotes** value is set to **no**, character string enclosed by double quotations is handled as character string, not as an identifier. The default value is **yes**. For more information, Statement/Type-Related Parameters.
- If there are characters that can be considered to be blank (e.g. spaces, tabs, or line breaks) between two character strings, these two character strings are treated as one according to ANSI standard. For example, the following example shows that a line break exists between two character string.

```
'abc'
'def'
```

- The two strings above are considered identical to one string below.

```
'abcedf'
```

- If you want to include a single quote as part of a character string, enter two single quotes in a row. For example, the character string on the left is stored as the one on the right.

```
''abcde''fghij'             'abcde'fghij
```

- The maximum size of the token for all the character strings is 16KB.
- National character strings are used to store national (except for English alphabet) character strings in a multilingual environment. Note that **N** (uppercase) should be followed by a single quote which encloses character strings.

```
N'Ha rder'
```

Characteristics

Length

For a **CHAR** or **VARVAHR** type, specify the length (bytes) of a character string for a **NCHAR** or **NCHAR VARYING** type, specify the number of character strings (number of characters).

When the length of the character string entered exceeds the length specified, the characters in excess of the specified length are truncated if they are space characters (ASCII 32), or an error occurs if they are non-space characters. Note that the data is not truncated according to the length specified.

For a fixed-length character string type such as **CHAR** or **NCHAR**, the length is fixed at the declared length. Therefore, the right part (trailing space) of the character string is filled with space characters when the string is stored. For a variable-length character string type such as **VARCHAR** or **NCHAR VARYING**, only the entered character string is stored, and the space is not filled with space characters.

The maximum length of a **CHAR** or **VARCHAR** type to be specified is 1,073,741,823 the maximum length of a **NCHAR** or **NCHAR VARYING** type to be specified is 536,870,911. The maximum length that can be input or output in a CSQL statement is 8,192 KB.

Character Set, charset

A character set (charset) is a set in which rules are defined that relate to what kind of codes can be used for encoding when specified characters (symbols) are stored in the computer.

CUBRID supports the following character sets and you can specify them as the **CUBRID_LANG** environment variable. You can store data in other character sets (e.g. utf-8), but string function or **LIKE** search are not supported.

Character Set	CUBRID_LANG
8 bits ISO 8859-1 Latin	en_US
KSC 5601-1992 (EUC_KR)	ko_KR.euckr

Any characters from the above character sets can be included in a character string (the **NULL** character is represented as '\0').

Collating Character Sets

A collation is a set of rules used for comparing characters to search or sort values stored in the database when a certain character set is specified. Therefore, such rules are applied only to character string data types such as **CHAR()** or **VARCHAR()**. For a national character string type such as **NCAHR()** or **NCHAR VARYING()**, the sorting rules are determined according to the encoding algorithm of the specified character set.

Character String Coercion

Automatic coercion takes place between a fixed-length and a variable-length character string for the comparison of two characters, applicable only to characters that belong to the same character set. For example, when you extract a column value from a CHAR(5) data type and insert it into a column with a CHAR(10) data type, the data type is automatically coerced to CHAR(10). If you want to coerce a character string explicitly, use the **CAST** operator (See CAST Operator).

CHAR(n)

Description

A fixed-length character string is represented as **CHAR***(n)*, in which n is the number of bytes in an ASCII character string. For the English alphabet, each character takes up one byte. However, for Korean characters, note that the number of bytes taken up by each character differs depending on the character set of the data input environment (e.g. EUC-KR: 2 bytes, utf-8: 3 bytes). If *n* is not specified, the length is set to the default value 1.

When the length of a character string exceeds *n* and the characters in excess of *n* are whitespace characters, they are truncated; an error occurs when they are non-whitespace characters. When character string which is shorter than *n* is stored, whitespace characters are used to fill up the trailing space.

CHAR(*n*) and **CHARACTER**(*n*) are used interchangeably.

Remark

- The **CHAR** data type is always based on the ISO 8859-1 (Latin-1) character set.
- *n* is an integer between 1 and 1,073,741,823 (1G).
- Empty quotes (' ') are used to represent a blank string. In this case, the return value of the **LENGTH** function is not 0, but is the fixed length defined in **CHAR**(*n*). That is, if you enter a blank string into a column with **CHAR**(10), the **LENGTH** is 10; if you enter a blank value into a **CHAR** with no length specified, the **LENGTH** is the default value 1.
- Space characters used as filling characters are considered to be smaller than any other characters, including special characters.

Example 1

```
If you specify 'pacesetter' as CHAR(12), 'pacesetter ' is stored (a 10-character string
plus two whitespace characters).
If you specify 'pacesetter ' as CHAR(10), 'pacesetter' is stored (a 10-character string;
two whitespace characters are truncated).
If you specify 'pacesetter' as CHAR(4), an error occurs (the length of the character
string is greater than 4).
If you specify 'p ' as CHAR, 'p' is stored (if n is not specified, the length is set to
the default value 1).
```

Example 2

```
If you specify '큐브리드' as CHAR(10) in the EUC-KR encoding, it is processed normally.
If you specify '큐브리드' as CHAR(10) and the use the CHAR_LENGTH() function in the EUC-KR
encoding, 10 is stored.
If you specify '큐브리드, as CHAR(10) in the utf-8 encoding, an error occurs (because one
Korean character takes up three bytes in the utf-8 encoding).
If you specify '큐브리드' as CHAR(12) in the utf-8 encoding, it is processed normally.
```

VARCHAR(n)/CHAR VARYING(n)

Description

Variable-length character strings are represented as **VARCHAR**(*n*), where *n* is the maximum number of ASCII character strings. Each English character takes up one byte. For Korean characters, note that the number of bytes taken up by each character differs depending on the character set of the data input environment (e.g. EUC-KR: 2 bytes, utf-8: 3 bytes). If *n* is not specified, the length is set to the maximum length of 1,073,741,823.

When the length of a character string exceeds *n* and the characters in excess of *n* are whitespace characters, they are truncated; an error occurs when they are non-whitespace characters. When character string which is shorter than *n* is stored, whitespace characters are used to fill up the trailing space; for **VARCHAR**(*n*), the length of string used are stored.

VARCHAR(*n*), **CHARACTER, VARYING**(*n*), and **CHAR VARYING**(*n*) are used interchangeably.

Remark

- **STRING** is the same as the **VARCHAR** (maximum length).
- *n* is an integer between 1 and 1,073,741,823 (1G).
- Empty quotes (' ') are used to represent a blank string. In this case, the return value of the **LENGTH** function is not 0.

Example 1

```
If you specify 'pacesetter' as CHAR(4), an error occurs (the length of the character
string is greater than 4).
If you specify 'pacesetter' as VARCHAR(4), an error occurs (the length of the character
string is greater than 4. Note that the character string which exceeds the specified size
is not truncated).
If you specify 'pacesetter' as VARCHAR(12), 'pacesetter' is stored (a 10-character string).
If you specify 'pacesetter ' as VARCHAR(12), 'pacesetter ' is stored (a 10-character
string plus two whitespace characters).
If you specify 'pacesetter ' as VARCHAR(10), 'pacesetter' is stored (a 10-character string;
two whitespace characters are truncated).
If you specify 'p ' as VARCHAR, 'p' is stored (if n is not specified, the default value
1,073,741,823 is used, and the trailing space is not filled with whitespace characters).
```

Example 2

```
If you specify '큐브리드' as VARCHAR(10) in the EUC-KR encoding, it is processed normally.
If you specify '큐브리드' as CHAR(10) and then use CHAR_LENGTH() function in the EUC-KR
encoding, 8 is stored.
If you specify '큐브리드' as VARCHAR(10) in the utf-8 encoding, an error occurs (one Korean
character takes up three bytes in the utf-8 encoding).
If you specify '큐브리드' as VARCHAR(12) in the utf-8 encoding, it is processed normally.
```

STRING

Description

STRING is a variable-length character string data type. **STRING** is the same as the VARCHAR with the length specified to the maximum value. That is, **STRING** and **VARCHAR**(1,073,741,823) have the same value.

NCHAR(n)

Description

NCHAR(*n*) is used to store non-English character strings. It can be used only for character sets supported by CUBRID described above. n is the number of characters. If *n* is omitted, the length is specified as the default value 1. When the length of a character string exceeds *n* and the characters in excess of *n* are whitespace characters, they are truncated; an error occurs when they are non-whitespace characters. When character string which is shorter than *n* is stored, whitespace characters are used to fill up the space.

To store a Korean character string as a national character string type, you must set the locale of the operating system to Korean, or set the value of the **CUBRID_LANG** environment variable to **ko_KR.euckr** before creating the table.

Remark

- *n* is an integer between 1 and 5,368,709,111.
- The number of national character sets that can be used in a single database is set to be one. For example, 8-bit ISO 8889-1 (Latin-1) and EUC code sets cannot be used simultaneously in the same database.
- An error occurs if a non-national character string (whether it is fixed-length or variable-length) is specified for an attribute declared as a national character string.
- Using two different character code sets at the same time also causes an error.

Example

```
If you specify ‘큐브리드’ as NCHAR(5) in the EUC-KR encoding, it is processed normally.
If you specify '큐브리드' as NCHAR(5) and then use the CHAR_LENGTH() function in the EUC-KR
encoding, 5 is stored.
If you specify '큐브리드' as NCHAR(5) in the utf-8 encoding, an error occurs (utf-8
character set is not supported).
```

NCHAR VARYING(n)

Description

NCHAR VARYING(*n*) is a variable-length character string type. For details, see description and note of NCHAR(n). The difference is that the right part (trailing space) of the character string is not filled with whitespace characters, even when the number of strings is smaller than n.

NCHAR VARYING(*n*), **NATIONAL CHAR VARYING**(*n*), and **NATIONAL CHARACTER VARYING(n)** are used interchangeably.

Example

```
If you specify '큐브리드' as NCHAR VARYING(5) in the EUC-KR encoding, it is processed
normally.
If you specify '큐브리드' as NCHAR VARYING(5) and then use CHAR_LENGTH() function in the
EUC-KR encoding, 4 is stored.
If you specify '큐브리드' as HCHAR VARYING(5) in the utf-8 encoding, an error occurs (utf-8
character set is not supported).
```

BLOB/CLOB Data Types

Definition and Characteristics

Definition

An External **LOB** type is data to process Large Object, such as text or images. When LOB-type data is created and inserted, it will be stored in a file to an external storage, and the location information of the relevant file (**LOB** Locator) will be stored in the CUBRID database. If the **LOB** Locator is deleted from the database, the relevant file that was stored in the external storage will be deleted as well. CUBRID supports the following two types of **LOB**:

- Binary Large Object (**BLOB**)
- Character Large Object (**CLOB**)

Related Terms

- **LOB** (Large Object) : Large-sized objects such as binaries or text.
- **FBO** (File Based Object) : An object that stores data of the database in an external file.
- **External LOB** : An object better known as FBO, which stores **LOB** data in a file into an external DB. It is supported by CUBRID. Internal **LOB** is an object that stores **LOB** data inside the DB.
- **External Storage** : An external storage to store LOB (example : POSIX file system).
- **LOB Locator** : The path name of a file stored in external storage.
- **LOB Data** : Details of a file in a specific location of LOB Locator.

File Name

When storing LOB data in external storage, the following naming convention will be applied:

```
{table_name}_{unique_name}
```

- *table_name* : It is inserted as a prefix and able to store the **LOB** data of many tables in one external storage.
- *unique_name* : The random name created by the DB server.

Default Storage

- **LOB** data is stored in the local file system of the DB server. LOB data is stored in the path specified in the **-lob-base-path option** value of **cubrid createdb**; if this value is omitted, the data will be stored in the [db-vol path]/lob path where the database volume will be created. For more details, see Database Creation and Storage Creation and Management.
- If the relevant path is deleted despite a **LOB** data file path being registered in the database location file (**databases.txt**), please note that the utility that operates in database server (cub_server) and standalone will not function normally.

BLOC/CLOB

BLOB

- A type that stores binary data outside the database.
- The maximum length of **BLOB** data is the maximum file size creatable in an external storage.
- In SQL statements, the **BLOB** type expresses the input and output value in a bit array. That is, it is compatible with the **BIT**(n) and **BIT VARYING**(n) types, and only an explicit type change is allowed. If data lengths differ from one another, the maximum length is truncated to fit the smaller one.
- When converting the **BLOB** type value to a binary value, the length of the converted data cannot exceed 1GB. When converting binary data to the **BLOB** type, the size of the converted data cannot exceed the maximum file size provided by the **BLOB** storage.

CLOB

- A type that stores character string data outside the database.
- The maximum length of **CLOB** data is the maximum file size creatable in an external storage.
- In SQL statements, the CLOB type expresses the input and output value in a character string. That is, it is compatible with the **CHAR**(n), **VARCHAR**(n), **NCHAR**(n), and **NCHAR VARYING**(n) types. However, only an explicit type change is allowed, and if data lengths are different from one another, the maximum length is truncated to fit to the smaller one.
- When converting the **CLOB** type value to a character string, the length of the converted data cannot exceed 1 GB. When converting a character string to the **CLOB** type, the size of the converted data cannot exceed the maximum file size provided by the **CLOB** storage.

Creating and Altering Columns

Description

BLOB/**CLOB** type columns can be created/added/deleted by using a **CREATE TABLE** statement or an **ALTER TABLE** statement.

Note

- You cannot create the index file for a **LOB** type column.
- You cannot define the **PRIMARY KEY**, **FOREIGN KEY**, **UNIQUE**, and **NOT NULL** constraints for a **LOB** type column. However, **SHARED** property cannot be defined and **DEFAULT** property can only be defined by the **NULL** value.
- **LOB** type column/data cannot be the element of collection type data.
- If you are deleting a record containing a **LOB** type column, all files located inside a **LOB** column value (Locator) and the external storage will be deleted. When a record containing a LOB type column is deleted in a basic key table, and a record of a foreign key table that refers to the foregoing details is deleted at the same time, all **LOB** files located in a **LOB** column value (Locator) and the external storage will be deleted. However, if the relevant table is deleted by using a **DROP TABLE** statement, or a **LOB** column is deleted by using an **ALTER TABLE...DROP** statement, only a **LOB** column value (**LOB** Locator) is deleted, and the **LOB** files inside the external storage which a **LOB** column refers to will not be deleted.

Example

```
//creating a table and CLOB column
CREATE TABLE doc_t (doc_id VARCHAR(64) PRIMARY KEY, content CLOB);

//an error occurs when UNIQUE constraint is defined on CLOB column
ALTER TABLE doc_t ADD CONSTRAINT content_unique UNIQUE(content); -- Error

/an error occurs when creating an index on CLOB column
CREATE INDEX ON doc_t (content); -- Error //creating a table and BLOB column CREATE TABLE
image_t (image_id VARCHAR(36) PRIMARY KEY, doc_id VARCHAR(64) NOT NULL, image BLOB);

//an error occurs when adding a BOLB column with NOT NULL constraint
ALTER TABLE image_t ADD COLUMN thumbnail BLOB NOT NULL; -- Error

//an error occurs when adding a BLOB column with DEFAULT attribute
ALTER TABLE image_t ADD COLUMN thumbnail2 BLOB DEFAULT BIT_TO_BLOB(X'010101'); -- Error
```

Storing and Updating Columns

Description

In a **BLOB**/**CLOB** type column, each **BLOB**/**CLOB** type value is stored, and if binary or character string data is input, you must explicitly change the types by using each **BIT_TO_BLOB**( )/**CHAR_TO_CLOB**( ) function.

If a value is input in a **LOB** column by using an **INSERT** statement, a file is created in an external storage internally and the relevant data is stored; the relevant file path (Locator) is stored in an actual column value.

If a record containing a **LOB** column uses a **DELETE** statement, a file to which the relevant **LOB** column refers will be deleted simultaneously. If a **LOB** column value is changed using an **UPDATE** statement, the column value will be changed following the operation below, according to whether a new value is **NULL** or not.

- If a **LOB** type column value is changed to a value that is not **NULL** : If a Locator that refers to an external file is already available in a **LOB** column, the relevant file will be deleted. A new file is created afterwards. After storing a value that is not **NULL**, a Locator for a new file will be stored in a **LOB** column value.
- If changing a **LOB** type column value to **NULL** : If a Locator that refers to an external file is already available in a **LOB** column, the relevant file will be deleted. And then **NULL** is stored in a **LOB** column value.

Example

```
//inserting data after explicit type conversion into CLOB type column
INSERT INTO doc_t (doc_id, content) VALUES ('doc-1', CHAR_TO_CLOB('This is a Dog'));
INSERT INTO doc_t (doc_id, content) VALUES ('doc-2', CHAR_TO_CLOB('This is a Cat'));
```

```
//inserting data after explicit type conversion into BLOB type column
INSERT INTO image_t VALUES ('image-0', 'doc-0', BIT_TO_BLOB(X'000001'));
INSERT INTO image_t VALUES ('image-1', 'doc-1', BIT_TO_BLOB(X'000010'));
INSERT INTO image_t VALUES ('image-2', 'doc-2', BIT_TO_BLOB(X'000100'));

//inserting data from a sub-query result
INSERT INTO image_t SELECT 'image-1010', 'doc-1010', image FROM image_t WHERE image_id =
'image-0';

//updating CLOB column value to NULL
UPDATE doc_t SET content = NULL WHERE doc_id = 'doc-1';

//updating CLOB column value
UPDATE doc_t SET content = CHAR_TO_CLOB('This is a Dog') WHERE doc_id = 'doc-1';

//updating BLOB column value
UPDATE image_t SET image = (SELECT image FROM image_t WHERE image_id = 'image-0') WHERE
image_id = 'image-1';

//deleting BLOB column value and its referencing files
DELETE FROM image_t WHERE image_id = 'image-1010';
```

Getting Column Values

Description

When you get a **LOB** type column, the data stored in a file to which the column refers will be displayed. You can execute an explicit type change by using **CAST** operator, **CLOB_TO_CHAR**( ) function, and **BLOB_TO_BIT**( ) function.

Note

- If the query is executed in CSQL, a column value (Locator) will be displayed, instead of the data stored in a file. To display the data to which a **BLOB**/**CLOB** column refers, it must be changed to strings by using **CLOB_TO_CHAR**( ) function.
- To use the string process function, the strings need to be converted by using the **CLOB_TO_CHAR**( ) function.
- You cannot specify a **LOB** column in **GROUP BY** clause and **ORDER BY** clause.
- Comparison operators, relational operators, **IN**, **NOT IN** operators cannot be used to compare **LOB** columns. However, **IS NULL** expression can be used to compare whether it is a **LOB** column value (Locator) or **NULL**. This means that **TRUE** will be returned when a column value is **NULL**, and if a column value is **NULL**, there is no file to store **LOB** data.
- When a **LOB** column is created, and the file is deleted after data input, a **LOB** column value (Locator) will become a state that is referring to an invalid file. As such, using **CLOB_TO_CHAR**( ), **BLOB_TO_BIT**( ), **CLOB_LENGTH**( ), and **BLOB_LENGTH**( ) functions on the columns that have mismatching **LOB** Locator and a **LOB** data file enables them to display **NULL**.

Example

```
//displaying locator value when selecting CLOB and BLOB column in CSQL interpreter
csql> SELECT doc_t.doc_id, content, image FROM doc_t, image_t WHERE doc_t.doc_id =
image_t.doc_id;
csql> ;xr

=== <Result of SELECT Command in Line 1> ===

  doc_id                content               image
==================================================================
  'doc-1'               file:/home1/data1/ces_658/doc_t.00001282208855807171_7329  file:/
home1/data1/ces_318/image_t.00001282208855809474_7474
  'doc-2'               file:/home1/data1/ces_180/doc_t.00001282208854194135_5598  file:/
home1/data1/ces_519/image_t.00001282208854205773_1215

2 rows selected.

//using string functions after coercing its type by CLOB_TO_CHAR( )
csql> SELECT CLOB_TO_CHAR(content), SUBSTRING(CLOB_TO_CHAR(content), 10) FROM doc_t;
csql> ;xr
```

```
=== <Result of SELECT Command in Line 1> ===

   clob_to_char(content)  substring( clob_to_char(content) from 10)
============================================
  'This is a Dog'       ' Dog'
  'This is a Cat'       ' Cat'

2 rows selected.

csql> SELECT CLOB_TO_CHAR(content) FROM doc_t WHERE CLOB_TO_CHAR(content) LIKE '
%Dog%';
csql> ;xr

=== <Result of SELECT Command in Line 1> ===

   clob_to_char(content)
======================
  'This is a Dog'

csql> SELECT CLOB_TO_CHAR(content) FROM doc_t ORDER BY CLOB_TO_CHAR(content)
csql> ;xr

=== <Result of SELECT Command in Line 1> ===

   clob_to_char(content)
======================
  'This is a Cat'
  'This is a Dog'

2 rows selected.

//an error occurs when LOB column specified in WHERE/ORDER BY/GROUP BY clauses
SELECT * FROM doc_t WHERE content LIKE 'This%'; --Error
SELECT * FROM doc_t ORDER BY content; -- Error
```

Functions and Operators

CAST Operator

By using **CAST** operator, you can execute an explicit type change between **BLOB**/**CLOB** type and binary type/string type. For more details, see CAST Operator.

Syntax

```
CAST (<bit_type_column_or_value> AS CLOB)
CAST (<bit_type_column_or_value> AS BLOB)
CAST (<char_type_column_or_value> AS BLOB)
CAST (<char_type_column_or_value> AS CLOB)
```

LOB Data Process and Type Change Functions

The next table shows the functions provided to process and change BLOB/CLOB types.

Function Expression	Description
CLOB_TO_CHAR (<*clob_type_column*>)	Changes number type, date/time type, and **CLOB** type to **VARCHA**R type.
BLOB_TO_BIT (<*blob_type_column*>)	Changes **BLOB** type to **VARYING BIT** type.
CHAR_TO_CLOB(<*char_type_column_or_value*>)	Changes text string type (**CHAR**, **VARCHAR**, **NCHAR**, **NVACHAR**) to **CLOB** type.
BIT_TO_BLOB(<*blob_type_column_or_value*>)	Changes bit array type (**BIT**, **VARYING BIT**) to **BLOB** type.
CHAR_TO_BLOB(<*char_type_colulmn_or_value*>)	Changes text string type (**CHAR**, **VARCHAR**, **NCHAR**, **NVACHAR**) to **BLOB** type.
CLOB_FROM_FILE(<*file_pathname*>)	Reads file details from the file path of **VARCHAR** type and changes to **CLOB** type data. <*file_pathname*> is analyzed to a

	path of server which is operated by the DB client, such as CAS or CSQL. If a path is specified targeting this, the upper path will be the current work direction of the process. The statement that calls this function will not cache execution plans.
BLOB_FROM_FILE(<*file_pathname*>)	Reads file details from the file path of **VARCHAR** type, and changes to BLOB type data. The file path specified in is interpreted using the same method as the **CLOB_FROM_FILE**( ) function.
CLOB_LENGTH(<*clob_column*>)	Returns the length of LOB data stored in a **CLOB** file in bytes.
BLOB_LENGTH(<*blob_column*>)	Returns the length of LOB data stored in a **BLOB** file in bytes.
<*blob_or_clob_column*> **IS NULL**	Use an **IS NULL** expression to compare whether it is a **LOB** column value (Locator) or **NULL**; returns **TRUE** if **NULL**.

Creating and Managing Storage

LOB File Path Specification

By default, the **LOB** data file is stored in the <db-volumn-path>/lob directory where database volume is created. However, if the **--lob-base-path** option of **cubrid createdb** utility is used when creating the database, a **LOB** data file can be stored in the directory specified by option value. However, if there is no directory specified by option value, attempt to create a directory, and display an error message if it fails to create the directory. For more details, see the **--lob-base-path** option in Creating Database.

```
# image_db volume is created in the current work directory, and a LOB data file will be
stored.
cubrid createdb image_db #

# LOB data file is stored in the “/home1/data1” path within a local file system.
cubrid createdb --lob-base-path=”file:/home1/data1” image_db
```

Check LOB File Store Directory

```
# You can check a directory where a LOB file will be stored by executing the cubrid
spacedb utility.
cubrid spacedb image_db

Space description for database 'image_db' with pagesize 4096. (log pagesize: 4096)

Volid Purpose total_pages free_pages  Vol Name
    0 GENERIC        5000       4780  /home1/data1/image_db
Space description for temporary volumes for database 'image_db' with pagesize 4096.

Volid Purpose total_pages free_pages  Vol Name
32766    TEMP         239        139  /home1/data1/image_db_t32766
LOB space description file:/home1/data1
```

Change or Expand LOB File Store Directory

Secure disk space to create additional file storage, expand the **lob-base-path** of **databases.txt**, and change to the disk location. Restart the database server to apply the changes made to **databases.txt**. However, even if you change the **lob-base-path** of **databases.txt**, access to the **LOB** data saved in a previous storage is possible.

```
# You can change to a new directory from the lob-base-path of databases.txt file.
sh> cat $CUBRID_DATABASES/databases.txt
#db-name         vol-path             db-host         log-path         lob-base-path
image_db         /home1/data1         localhost       /home1/data1     file:/home1/data2
```

Backup and Recovery of LOB Files

While backup/recovery is not supported for **LOB** data files, **LOB** type column value (Locator) is supported with such service.

Copying a Database with LOB Files

If you are copying a database by using the **cubrid copydb** utility, you must configure the **databases.txt** additionally, as the **LOB** file directory path will not be copied if the related option is not specified. For more details, see the **-B** and **--copy-lob-path** options in Copying/Moving Database.

Supporting and Recovering Transactions

Definition

Commit/rollback for **LOB** data changes are supported. That is, it ensures the validation of mapping between **LOB** Locator and actual **LOB** data within transactions, and it supports recovery during DB errors. This means that an error will be displayed in case of mapping errors between **LOB** Locator and **LOB** data due to the rollback of the relevant transactions, as the database is terminated during transactions. See the example below.

Example

```
csql> ;AUTOCOMMIT OFF
csql> CREATE TABLE doc_t (doc_id VARCHAR(64) PRIMARY KEY, content CLOB);
csql> INSERT INTO doc_t VALUES ('doc-10', CHAR_TO_CLOB('This is content'));
csql> commit;
csql> UPDATE doc_t SET content = CHAR_TO_CLOB('This is content 2') where doc_id = 'doc-10';
csql> rollback;
csql> SELECT doc_id, CLOB_TO_CHAR(content) FROM doc_t WHERE doc_id = 'doc-10';

=== <Result of SELECT Command in Line 1> ===
  doc_id         content
=========================================================
  'doc-10'       'This is content '

csql> INSERT INTO doc_t VALUES ('doc-11', CHAR_TO_CLOB ('This is content'));
csql> commit; csql> UPDATE doc_t SET content = CHAR_TO_CLOB('This is content 3') WHERE
doc_id = 'doc-11';

//system crash occurred and then restart server
csql> SELECT doc_id, CLOB_TO_CHAR(content) FROM doc_t WHERE doc_id = 'doc-11'; -- Error :
LOB Locator references to the previous LOB data because only LOB Locator is rollbacked.
```

Note

- When selecting **LOB** data in an application through a driver such as JDBC, the driver can get ResultSet from DB server and fetch the record while changing the cursor location on Resultset. That is, only Locator, the **LOB** column value, is stored at the time ResultSet is imported, and **LOB** data that is referred by a File Locator will be fetched from the file Locator at the time a record is fetched. Therefore, if **LOB** data is updated between two different points of time, there could be an error, as the mapping of **LOB** Locator and actual **LOB** data will be invalid.
- Since backup/recovery is supported only for **LOB** type column value (Locator), an error is likely to occur, as the mapping of **LOB** Locator and LOB data is invalid if recovery is performed based on a specific point of time.
- If the DB is operated in different equipment like S1 and S2, and you want to store **LOB** data in the DB of S1 equipment to S2 equipment, you must read the **LOB** data which the **LOB** column value of S1 equipment is referring to, and **INSERT LOB**. The **LOB** column value (Locator) of S1 equipment is valid only in the relevant local system.

Caution Up to CUBRID 2008 R3.0, Large Objects are processed by using **glo** (Generalized Large Object) classes. However, the **glo** classes has been deprecated since the CUBRID 2008 R3.1. Instead of it, **LOB**/**CLOB** data type is supported. Hence, both DB schema and application must be modified when upgrading CUBRID in an environment using the previous version of **glo** classes.

Collection Data Type

Definition and Characteristics

Definition

Allowing multiple data values to be stored in a single attribute is an extended feature of relational database. Elements of a collection are possible to have different domain each other. The domain can be one of the primitive data types or classes excluding virtual classes. For example, **SET** (INTEGER, tbl_1) can specify an integer or a set of row values of the user-defined class tbl_1 as a domain. When a domain list is not specified (e.g. **SET** ( )), all data types are allowed as elements including user-defined classes.

The data of a collection-type column with at least two domain lists can be retrieved by using the **csql** utility or the C-API. It cannot be retrieved in CUBRID manager or CUBRID API (JDBC, ODBC, OLEDB, PHP, CCI).

Collection Types Supported by CUBRID

Type	Description	Definition	Input Data	Stored Data

SET	A union which does not allow duplicates	col_name SET VARCHAR(20)
		col_name SET (int, VARCHAR(20))
			{'c','c','c','b','b', 'a'}
			{3,3,3,2,2,1,0,'c','c','c','b','b', 'a'}
				{'a','b','c'}
				{0,1,2,3,'a','b','c'}

MULTISET	A union which allows duplicates	col_name MULTISET VARCHAR(20)
		col_name MULTISET (int, VARCHAR(20))
			{'c','c','c','b','b', 'a'}
			{3,3,3,2,2,1,0,'c','c','c','b','b', 'a'}
				{'a','b','b','c','c','c'}
				{0,1,2,2,3,3,3,'a','b','b', 'c','c','c'}

LIST
SEQUENCE
SEQUENCE
	A union which allows duplicates and stores data in the order of input	col_name LIST VARCHAR(20)
		col_name LIST (int, VARCHAR(20))
			{'c','c','c','b','b', 'a'}
			{3,3,3,2,2,1,0,'c','c','c','b','b', 'a'}
				{'c','c','c','b','b','a'}
				{3,3,3,2,2,1,0,'c','c','c','b','b','a'}

As you see the table above, the value specified as a collection type can be inputted with braces ('{', '}') each value is separated with a comma (,).

Characteristics

Coercions

If the specified domains are identical, the collection types can be cast explicitly by using the **CAST** operator. The following table shows the collection types that allow explicit coercions.

Explicit Coercions

		TO		
		SET	**MULTISET**	**LIST**
FROM	**SET**	-	O	O
	MULTISET	O	-	X
	LIST	O	O	-

SET

Description

SET is a set type in which each element has different values. Elements of a **SET** can have many different data types or even instances of different classes.

Example

```
csql> CREATE TABLE set_tbl ( col_1 set(int, CHAR(1)));
csql> INSERT INTO set_tbl VALUES ({3,3,3,2,2,1,0,'c','c','c','b','b','a'});
csql> INSERT INTO set_tbl VALUES ({NULL});
csql> INSERT INTO set_tbl VALUES ({''});
csql> SELECT * FROM set_tbl;
csql> ;xr

=== <Result of SELECT Command in Line 1> ===
  col_1
======================
{0, 1, 2, 3, 'a', 'b', 'c'}
{NULL}
{' '}

csql> SELECT CAST(col_1 AS MULTISET), CAST(col_1 AS LIST) FROM set_tbl;
csql> ;xr

=== <Result of SELECT Command in Line 1> ===

   cast(col_1 as multiset)   cast(col_1 as sequence)
============================================
  {0, 1, 2, 3, 'a', 'b', 'c'}  {0, 1, 2, 3, 'a', 'b', 'c'}
  {NULL}  {NULL}
  {' '}  {' '}

csql> INSERT INTO set_tbl VALUES ('');
ERROR: Cannot coerce '' to type set.
ERROR: Incompatible data type on attribute col_1.
```

MULTISET

Description

MULTISET is a collection type in which duplicated elements are allowed. Elements of a **MULTISET** can have many different data types or even instances of different classes.

Example

```
csql> CREATE TABLE multiset_tbl ( col_1 multiset(int, CHAR(1)));
csql> INSERT INTO multiset_tbl VALUES ({3,3,3,2,2,1,0,'c','c','c','b','b', 'a'});
csql> SELECT * FROM multiset_tbl;
csql> ;xr

=== <Result of SELECT Command> ===
  col_1
======================
  {0, 1, 2, 2, 3, 3, 3, 'a', 'b', 'b', 'c', 'c', 'c'}

csql> SELECT CAST(col_1 AS SET), CAST(col_1 AS LIST) FROM multiset_tbl;
csql> ;xr

=== <Result of SELECT Command in Line 1> ===

   cast(col_1 as set)   cast(col_1 as sequence)
============================================
  {0, 1, 2, 3, 'a', 'b', 'c'}  {3, 3, 3, 2, 2, 1, 0, 'c', 'c', 'c', 'b', 'b', 'a
'}
```

LIST/SEQUENCE

Description

LIST (=**SEQUENCE**) is a collection type in which the input order of elements is preserved, and duplications are allowed. Elements of a **LIST** can have many different data types or even instances of different classes.

Example

```
csql> CREATE TABLE list_tbl ( col_1 list(int, CHAR(1)));
csql> INSERT INTO list_tbl VALUES ({3,3,3,2,2,1,0,'c','c','c','b','b', 'a'});
csql> SELECT * FROM list_tbl;
csql> ;xr

=== <Result of SELECT Command in Line 1> ===
  col_1
======================
{3, 3, 3, 2, 2, 1, 0, 'c', 'c', 'c', 'b', 'b', 'a'}

csql> SELECT CAST(col_1 AS SET), CAST(col_1 AS MULTISET) FROM list_tbl;
csql> ;xr

=== <Result of SELECT Command in Line 1> ===
   cast(col_1 as set)  cast(col_1 as multiset)
============================================
  {0, 1, 2, 3, 'a', 'b', 'c'}  {0, 1, 2, 2, 3, 3, 3, 'a', 'b', 'b', 'c', 'c', 'c
'}
```

Table Definition

CREATE TABLE

Table Definition

Description

To create a table, use the **CREATE TABLE** syntax.

Syntax

```
CREATE {TABLE | CLASS} < table_name>
                                      [ < subclass_definition> ]
                                      [ ( < column_definition> [,<
table_constraint> ]... ) ]
                                      [ CLASS ATTRIBUTE ( <
column_definition_comma_list> ) ]
                                      [ METHOD < method_definition_comma_list> ]
                                      [ FILE < method_file_comma_list> ]
                                      [ INHERIT < resolution_comma_list> ]
                                      [ REUSE_OID ]
< column_definition> ::=
column_name column_type [[ < default_or_shared> ] |  [ < column_constraint> ]]...

< default_or_shared> ::=
{SHARED < value_specification>   | DEFAULT < value_specification> } |
AUTO_INCREMENT [(seed, increment)]

< column_constraint> ::=
NOT NULL | UNIQUE | PRIMARY KEY | FOREIGN KEY < referential definition>

< table_constraint> ::=
[ CONSTRAINT < constraint_name> ] UNIQUE [ KEY | INDEX ]( column_name_comma_list ) |
[ { KEY | INDEX } [ < constraint_name>   ]( column_name_comma_list ) |
[ PRIMARY KEY ( column_name_comma_list )] |
[ < referential_constraint> ]

< referential_constraint> ::=
FOREIGN KEY ( column_name_comma_list ) < referential definition>

< referential definition> ::=
REFERENCES [ referenced_table_name ] ( column_name_comma_list )
[ < referential_triggered_action> ...  ]

< referential_triggered_action> ::=
{ ON UPDATE < referential_action> } |
{ ON DELETE < referential_action> } |
{ ON CACHE OBJECT cache_object_column_name }

< referential_action> ::=
CASCADE | RESTRICT | NO ACTION | SET NULL

< subclass_definition> ::=
{ UNDER | AS SUBCLASS OF } table_name_comma_list

< method_definition> ::=
[ CLASS ] method_name
[ ( [ argument_type_comma_list ] ) ]
[ result_type ]
[ FUNCTION   function_name ]

< resolution> ::=
[ CLASS ] { column_name | method_name } OF super class_name
[ AS alias ]
```

- *table_name* : Specifies the name of the table to be created (maximum : 255 bytes).

- *column_definition* :
- *column_name* : Specifies the name of the column to be created.
- *column_type* : Specifies the data type of the column.
    - [**SHARED** *value* | **DEFAULT** *value*] : Specifies the initial value of the column (see Column Definition for more information).
- *column_constraints* : Specifies the constraint of the column. Available constraints are **NOT NULL**, **UNIQUE**, **PRIMARY KEY** and **FOREIGN KEY** (see Constraint Definition for more information).

Example

```
CREATE TABLE olympic (
    host_year              INT      NOT NULL PRIMARY KEY,
    host_nation          VARCHAR(40) NOT NULL,
    host_city              VARCHAR(20) NOT NULL,
    opening_date        DATE              NOT NULL,
    closing_date        DATE              NOT NULL,
    mascot                    VARCHAR(20) ,
    slogan                    VARCHAR(40) ,
    introduction        VARCHAR(1500)
)
```

Column Definition

A column is a set of data values of a particular simple type, one for each row of the table.

```
<column_definition>::=
column_name column_type [[  <default_or_shared> ] | [ <column_constraint> ]]...

<default_or_shared>::=
{SHARED <value_specification> | DEFAULT <value_specification> } |
AUTO_INCREMENT [(seed, increment)]

<column_constraint>::=
NOT NULL | UNIQUE | PRIMARY KEY | FOREIGN KEY  <referential definition>
```

Column Name

Description

How to create a column name, see Identifier.

You can alter created column name by using RENAME COLUMN clause of the **ALTER TABLE**.

Example

The following is an example of creating the manager2 table that has the following two columns: full_name and age.

```
CREATE TABLE manager2 (full_name VARCHAR(40), age INT );
```

Caution

- The first character of a column name must be an alphabet. The maximum length is 255 characters.
- The column name must be unique in the table.

Setting the Column Initial Value (SHARED, DEFAULT)

Description

SHARED and **DEFAULT** are attributes related to the initial value of the column. You can change the value of **SHARED** and **DEFAULT** in the **ALTER TABLE** statement.

- **SHARED** : Column values are identical in all rows. If a value different from the initial value is **INSERT**ed, the column value is updated to a new one in every row.
- **DEFAULT** : The initial value set when the **DEFAULT** attribute was defined is saved even if the column value is not specified when a new row is inserted. Note that if you set **SYS_TIMESTAMP** as a **DEFAULT** value when

creating a table, the **TIMESTAMP** value at the point of **CREATE TABLE**, not the point at which the data is **INSERT**ed, is specified by default. Therefore, you must specify the **SYS_TIMESTAMP** value for the **VALUES** of the **INSERT** statement when entering data.

Example

```
CREATE TABLE colval_tbl
( id INT, name VARCHAR SHARED 'AAA', phone VARCHAR DEFAULT '000-0000');
INSERT INTO colval_tbl(id) VALUES (1),(2);
SELECT * FROM colval_tbl;
;xr

=== <Result of SELECT Command in Line 3> ===

           id  name                  phone
=========================================================
            1  'AAA'                 '000-0000'
            2  'AAA'                 '000-0000'

--updating column values on every row
INSERT INTO colval_tbl(id, name) VALUES (3,'BBB');
INSERT INTO colval_tbl(id) VALUES (4),(5);
SELECT * FROM colval_tbl;
;xr

=== <Result of SELECT Command in Line 3> ===

           id  name                  phone
=========================================================
            1  'BBB'                 '000-0000'
            2  'BBB'                 '000-0000'
            3  'BBB'                 '000-0000'
            4  'BBB'                 '000-0000'
            5  'BBB'                 '000-0000'

5 rows selected.

--changing DEFAULT value in the ALTER TABLE statement
ALTER TABLE colval_tbl CHANGE phone DEFAULT '111-1111'
INSERT INTO colval_tbl(id) VALUES (6);
SELECT * FROM colval_tbl;
;xr

=== <Result of SELECT Command in Line 1> ===

           id  name                  phone
=========================================================
            1  'BBB'                 '000-0000'
            2  'BBB'                 '000-0000'
            3  'BBB'                 '000-0000'
            4  'BBB'                 '000-0000'
            5  'BBB'                 '000-0000'
            6  'BBB'                 '111-1111'


6 rows selected.
```

AUTO INCREMENT

Description

You can define the **AUTO_INCREMENT** attribute for the column to automatically give serial numbers to column values. This can be defined only for **SMALLINT**, **INTEGER**, **BIGINT**(*p*,0), and **NUMERIC**(*p*,0) domains.

DEFAULT, **SHARED** and **AUTO_INCREMENT** cannot be defined for the same column. Make sure the value entered directly by the user and the value entered by the auto increment attribute do not conflict with each other.

Syntax

```
AUTO_INCREMENT [(seed, increment)]
```

- *seed* : The initial value from which the number starts. Only positive integers are allowed. The default is 1.
- *increment* : The increment value of each row. Only positive integers are allowed. The default value 1.

Example

```
--AUTO_INCREMENT works only when no value is inserted on compat_mode=cubrid
CREATE TABLE auto_tbl(id INT AUTO_INCREMENT, name VARCHAR);
INSERT INTO auto_tbl VALUES(NULL, 'AAA'),(NULL, 'BBB'),(NULL, 'CCC');
INSERT INTO auto_tbl(name) VALUES ('DDD'),('EEE');
SELECT * FROM auto_tbl;
;xr

=== <Result of SELECT Command in Line 4> ===

           id  name
===================================
            1  'AAA'
            2  'BBB'
            3  'CCC'
            4  'DDD'
            5  'EEE'

5 rows selected.
```

Caution

- Even if a column has auto increment, the **UNIQUE** constraint is not satisfied.
- If **NULL** is specified in the column where auto increment is defined, the value of auto increment is stored.
- The initial value and the final value obtained by auto increment cannot exceed the minimum and maximum values allowed in the given domain.
- Because auto increment has no cycle, an error occurs when the maximum value of the type exceeds, and no rollback is executed. Therefore, you must delete and recreate the column in such cases.
- For example, if a table is created as below, the maximum value of A is 32767. Because an error occurs if the value exceeds 32767, you must make sure that the maximum value of the column A does not exceed the maximum value of the type when creating the initial table.

```
create table tbl1(A smallint auto_increment, B char(5));
```

Constraint Definition

Description

You can define **NOT NULL**, **UNIQUE**, **PRIMARY KEY**, **FOREIGN KEY** as the constraints. You can also create an index by using **INDEX** or **KEY**.

```
<column_constraint>::=
NOT NULL | UNIQUE | PRIMARY KEY | FOREIGN KEY <referential_definition>

<table_constraint> ::=
[ CONSTRAINT <constraint_name> ] UNIQUE [ KEY | INDEX ]( column_name_comma_list ) |
[ { KEY | INDEX } [ <constraint_name> ]( column_name_comma_list ) |
[ PRIMARY KEY ( column_name_comma_list )] |
[<referential_constraint>]

<referential_constraint>::=
FOREIGN KEY ( column_name_comma_list ) <referential_definition>

<referential_definition>::=
REFERENCES [ referenced_table_name ] ( column_name_comma_list )
[ <referential_triggered_action>... ]

<referential_triggered_action>::=
{ ON UPDATE <referential_action>} |
{ ON DELETE <referential_action>} |
{ ON CACHE OBJECT cache_object_column_name }

<referential_action>::=
CASCADE | RESTRICT | NO ACTION | SET NULL
```

NOT NULL Constraints

Description

A column for which the **NOT NULL** constraint has been defined must have a certain value that is not **NULL**. The **NOT NULL** constraint can be defined for all columns. An error occurs if you try to insert a **NULL** value into a column with the **NOT NULL** constraint by using the **INSERT** or **UPDATE** statement.

Example

```
CREATE TABLE const_tbl1(id INT NOT NULL, INDEX i_index(id ASC), phone VARCHAR);

CREATE TABLE const_tbl2(id INT NOT NULL PRIMARY KEY, phone VARCHAR);
INSERT INTO const_tbl2 (NULL,'000-0000');
;xr

In line 2, column 25,

ERROR: syntax error, unexpected Null
```

UNIQUE Constraint

Description

The **UNIQUE** constraint enforces a column to have a unique value. An error occurs if a new record that has the same value as the existing one is added by this constraint.

You can place a **UNIQUE** constraint on either a column or a set of columns. If the **UNIQUE** constraint is defined for multiple columns, the uniqueness is ensured not for each column, but the combination of multiple columns.

Example

If a **UNIQUE** constraint is defined on a set of columns, this ensures the uniqueness of the values in all the columns. As shown below, the second INSERT statement succeeds because the value of column a is the same, but the value of column b is unique. The third INSERT statement causes an error because the values of column a and b are the same as those in the first INSERT statement.

```
--UNIQUE constraint is defined on a single column only
CREATE TABLE const_tbl5(id INT UNIQUE, phone VARCHAR);
INSERT INTO const_tbl5(id) VALUES (NULL), (NULL);
INSERT INTO const_tbl5 VALUES (1, '000-0000');
SELECT * FROM const_tbl5;
;xr


=== <Result of SELECT Command in Line 4> ===

           id  phone
===================================
         NULL  NULL
         NULL  NULL
            1  '000-0000'


3 rows selected.

INSERT INTO const_tbl5 VALUES (1, '111-1111');

ERROR: Operation would have caused one or more unique constraint violations.


--UNIQUE constraint is defined on several columns
CREATE TABLE const_tbl6(id INT, phone VARCHAR, CONSTRAINT UNIQUE(id,phone));
INSERT INTO const_tbl6 VALUES (1,NULL), (2,NULL), (1,'000-0000'), (1,'111-1111');
SELECT * FROM const_tbl6;
;xr

=== <Result of SELECT Command in Line 1> ===
           id  phone
===================================
            1  NULL
            2  NULL
            1  '000-0000'
            1  '111-1111'


4 rows selected.
```

PRIMARY KEY Constraint

Description

A key in a table is a set of column(s) that uniquely identifies each row. A candidate key is a set of columns that uniquely identifies each row of the table. You can define one of such candidate keys a primary key. That is, the column defined as a primary key is uniquely identified in each row.

Example

```
CREATE TABLE const_tbl7(
id INT NOT NULL,
phone VARCHAR,
CONSTRAINT pk_id PRIMARY KEY(id));

CONSTRAINT keyword
CREATE TABLE const_tbl8(
id INT NOT NULL PRIMARY KEY,
phone VARCHAR);

--primary key is defined on multiple columns
CREATE TABLE const_tbl8 (
host_year    INT NOT NULL,
event_code   INT NOT NULL,
athlete_code INT NOT NULL,
medal        CHAR(1)  NOT NULL,
score        VARCHAR(20),
unit         VARCHAR(5),
PRIMARY KEY(host_year, event_code, athlete_code, medal)
);
```

FOREIGN KEY Constraint

Description

A foreign key is a column or a set of columns that references the primary key in other tables in order to maintain reference relationship. The foreign key and the referenced primary key must have the same data type. Consistency between two tables is maintained by the foreign key referencing the primary key, which is called referential integrity.

Syntax

```
[ CONSTRAINT < constraint_name > ]
FOREIGN KEY ( column_name_comma_list )
REFERENCES [ referenced_table_name ] ( column_name_comma_list )
[ <referential_triggered_action> ]

<referential_triggered_action> :
ON UPDATE <referential_action>
[ ON DELETE <referential_action> [ ON CACHE OBJECT cache_object_column_name ]]

<referential_action> :
CASCADE | RESTRICT | NO ACTION | SET NULL
```

- *constraint_name* : Specifies the name of the table to be created.
- *column_name* : Specifies the name of the column to be defined as a foreign key after the **FOREIGN KEY** keyword. There is no limit on the number of foreign keys to be defined (the number of columns), but it must be the same number as that of the referred primary keys.

- *referenced_table_name* : Specifies the name of the table to be referenced.
- *column_name* : Specifies the name of the referred primary key column after the **FOREIGN KEY** keyword.
- *referential_triggered_action* : Specifies the trigger action that responds to a certain operation in order to maintain referential integrity. **ON UPDATE**, **ON DELETE** or **ON CACHE OBJECT** can be specified. Each action can be defined multiple times, and the definition order is not significant.
  - **ON UPDATE** : Defines the action to be performed when attempting to update the primary key referenced by the foreign key. You can use either **NO ACTION**, **RESTRICT**, or **SET NULL** option. The default is **RESTRICT**.
  - **ON DELETE** : Defines the action to be performed when attempting to delete the primary key referenced by the foreign key. You can use **NO ACTION**, **RESTRICT**, **CASCADE**, or **SET NULL** option. The default is **RESTRICT**.
  - **ON CACHE OBJECT** : You can search an object using a direct object reference in object-oriented model. **ON CACHE OBJECT** option supports this feature in association with referential integrity (foreign key). **ON CACHE OBJECT** option adds an OID reference to a foreign key configuration. The OID is used as a CACHE point for the foreign key to the primary key table. Such OID is managed by the system internally; it cannot be changed by users.
    To define the **ON CACHE OBJECT** option, you must have defined a column whose domain is the table with a primary key and specified the column in the *cache_object_column_name*.
    The attribute defined with **ON CACHE OBJECT** can use the OID the same way as the one of the existing object type and can maintain the OID reference when it is duplicated.
- *referential_ action* : You can define an option that determines whether to maintain the value of the foreign key when the primary key value is deleted or updated.
  - **CASCADE** : If the primary key is deleted, the foreign key is deleted as well. This option is supported only for the **ON DELETE** operation.
  - **RESTRICT** : Prevents the value of the primary key from being deleted or updated, and rolls back any transaction that has been attempted.
  - **SET NULL** : When a specific record is being deleted or updated, the column value of the foreign key is updated to **NULL**.
  - **NO ACTION** : Its behavior is the same as that of the **RESTRICT** option.

Example

```
--creaing two tables where one is referencing the other
CREATE TABLE a_tbl(
id INT NOT NULL DEFAULT 0 PRIMARY KEY,
phone VARCHAR(10));

CREATE TABLE b_tbl(
ID INT NOT NULL,
name VARCHAR(10) NOT NULL,
CONSTRAINT pk_id PRIMARY KEY(id),
CONSTRAINT fk_id FOREIGN KEY(id) REFERENCES a_tbl(id)
ON DELETE CASCADE ON UPDATE RESTRICT);

INSERT INTO a_tbl VALUES(1,'111-1111'), (2,'222-2222'), (3, '333-
3333');                              INSERT INTO b_tbl
VALUES(1,'George'),(2,'Laura'),(3,'Max');
SELECT a.id, b.id, a.phone, b.name FROM a_tbl a, b_tbl b WHERE a.id=b.id;
;xr

=== <Result of SELECT Command in Line 1> ===

           id           id                   phone                 name
==============================================================================
            1            1                   '111-1111'            'George'
            2            2                   '222-2222'            'Laura'
            3            3                   '333-3333'            'Max'

--when deleting primay key value, it cascades foreign key value
DELETE FROM a_tbl WHERE id=3;
;xr

1 rows affected.
```

```
Current transaction has been committed.

1 command(s) successfully processed.

SELECT a.id, b.id, a.phone, b.name FROM a_tbl a, b_tbl b WHERE a.id=b.id;
;xr

=== <Result of SELECT Command in Line 1> ===

           id           id                   phone                 name
==============================================================================
            1            1                   '111-1111'            'George'
            2            2                   '222-2222'            'Laura'

--when attempting to update primay key value, it restricts the operation
UPDATE  a_tbl SET id = 10 WHERE phone = '111-1111';
;xr

In the command from line 1,

ERROR: Update/Delete operations are restricted by the foreign key 'fk_id'.


0 command(s) successfully processed.
```

Caution

- In a referential constraint, the name of the primary key table to be referenced and the corresponding column names are defined. If the list of column names are is not specified, the primary key of the primary key table is specified in the defined order.
- The number of primary keys in a referential constraint must be identical to that of foreign keys. The same column name cannot be used multiple times for the primary key in the referential constraint.

KEY or INDEX

Description

KEY and **INDEX** are used interchangeably. They create an index that uses the corresponding column as a key. You can specify the index name. If omitted, a name is assigned automatically.

Example

```
CREATE TABLE const_tbl3(id INT, phone VARCHAR, INDEX(id DESC, phone ASC));

CREATE TABLE const_tbl4(id INT, phone VARCHAR, KEY i_key(id DESC, phone ASC));
```

Column Option

Description

You can specify options such as **ASC** or **DESC** after the column name when defining **UNIQUE** or **INDEX** for a specific column. This keyword is specified to save the index value in ascending or descending order.

Syntax

```
column_name [ASC|DESC]
```

Example

```
CREATE TABLE const_tbl(
id VARCHAR,
name VARCHAR,
CONSTRAINT UNIQUE INDEX(id DESC, name ASC)
);

INSERT INTO const_tbl VALUES('1000', 'john'), ('1000','johnny'), ('1000', 'jone');
INSERT INTO const_tbl VALUES('1001', 'johnny'), ('1001','john'), ('1001', 'jone');
```

```
SELECT * FROM const_tbl WHERE id > '100';
===================================================
          id    name
          1001     john
          1001     johnny
          1001     jone
          1000     john
          1000     johnny
          1000     jone
```

Table Option (REUSE_OID)

Description

You can specify the **REUSE_OID** option when creating a table, so that OIDs that have been deleted due to the deletion of records (**DELETE**) can be reused when a new record is inserted (**INSERT**). Such a table is called an OID reusable or a non-referable table.

OID (Object Identifier) is an object identifier represented by physical location information such as the volume number, page number and slot number. By using such OIDs, CUBRID manages the reference relationships of objects and searches, saves or deletes them. When an OID is used, accessibility is improved because the object in the heap file can be directly accessed without referring to the table. However, the problem of decreased reusability of the storage occurs when there are many **DELETE/ INSERT** operations because the object's OID is kept to maintain the reference relationship with the object even if it is deleted.

If you specify the **REUSE_OID** option when creating a table, the OID is also deleted when data in the table is deleted, so that another **INSERT**ed data can use it. OID reusable tables cannot be referred to by other tables, and OID values of the objects in the OID reusable tables cannot be viewed.

Example

```
--creating table with REUSE_OID option specified
CREATE TABLE reuse_tbl (a INT PRIMARY KEY) REUSE_OID;
INSERT INTO reuse_tbl VALUES (1);
INSERT INTO reuse_tbl VALUES (2);
INSERT INTO reuse_tbl VALUES (3);
;xr

3 rows affected.

--an error occurs when column type is a OID reusable table itself
CREATE TABLE tbl_1 ( a reuse_tbl);
;xr

In line 1, column 34,

ERROR: The class 'reuse_tbl' is marked as REUSE_OID and is non-referable. Non-referable
classes can't be the domain of an attribute and their instances' OIDs cannot be returned.

--an error occurs when a table references a OID reusable table
CREATE TABLE tbl_2
(b int, FOREIGN KEY(b) REFERENCES reuse_tbl(a) ON CACHE OBJECT oid_value);
INSERT INTO tbl_2(b) VALUES(1);
SELECT oid_value.a FROM tbl_2;
;xr

ERROR: The class 'reuse_tbl' is marked as REUSE_OID and is non-referable. Non-referable
classes can't be the domain of an attribute and their instances' OIDs cannot be returned.
```

Caution

- OID reusable tables cannot be referred to by other tables.
- Updatable views cannot be created for OID reusable tables.
- OID reusable tables cannot be specified as class attribute domains of other tables.
- OID values of the objects in the OID reusable tables cannot be read.

- Instance methods cannot be called from OID reusable tables. Also, instance methods cannot be called if a subclass inherited from the class where the method is defined is defined as an OID reusable table.
- OID reusable tables are supported only by CUBRID 2008 R2.2 or above, and backward compatibility is not ensured. That is, the database in which the OID reusable table is located cannot be accessed from a lower version database.
- OID reusable tables can be managed as partitioned tables and can be replicated.

CREATE TABLE LIKE

Description

You can create a table that has the same schema as an existing table by using the **CREATE TABLE...LIKE** statement.

Column attribute, table constraint, and index are replicated from the existing table. An index name created from the existing table changes according to a new table name, but an index name defined by a user is replicated as it is. Therefore, you should be careful at a query statement that is supposed to use a specific index created by using the **USING INDEX**.

You cannot create the column definition because the **CREATE TABLE...LIKE** statement replicates the schema only.

Syntax

```
CREATE {TABLE | CLASS} <new_table_name> LIKE <old_table_name>
```

- *new_table_name* : A table name to be created.
- *old_table_name* : The name of the original table that already exists in the database. The following tables cannot be specified as original tables in the **CREATE TABLE…LIKE** statement.
- Partition table
- Table that contains an **AUTO_INCREMENT** column
- Table that uses inheritance or methods

Example

```
CREATE TABLE a_tbl(
id INT NOT NULL DEFAULT 0 PRIMARY KEY,
phone VARCHAR(10));
INSERT INTO a_tbl VALUES(1,'111-1111'), (2,'222-2222'), (3, '333-3333');

--creating an empty table with the same schema as a_tbl
CREATE TABLE new_tbl LIKE a_tbl;
SELECT * FROM new_tbl;
;xr

=== <Result of SELECT Command in Line 1> ===

There are no results.

0 rows selected.

;schema a_tbl
=== <Help: Schema of a Class> ===

<Class Name>
    a_tbl

<Attributes>
    id                     INTEGER DEFAULT 0 NOT NULL
    phone                  CHARACTER VARYING(10)

<Constraints>
    PRIMARY KEY pk_a_tbl_id ON a_tbl (id)
Current transaction has been committed.
;schema new_tbl
=== <Help: Schema of a Class> ===

<Class Name>
    new_tbl
```

```
<Attributes>
    id                     INTEGER DEFAULT 0 NOT NULL
    phone                  CHARACTER VARYING(10)

<Constraints>
    PRIMARY KEY pk_new_tbl_id ON new_tbl (id)

Current transaction has been committed.
```

CREATE TABLE AS SELECT

Description

You can create a new table that contains the result records of the **SELECT** statement by using the **CREATE TABLE...AS SELECT** statement. You can define column and table constraints for the new table. The following rules are applied to reflect the result records of the **SELECT** statement.

- If *col_1* is defined in the new table and the same column *col_1* is specified in select_statement, the result record of the **SELECT** statement is stored as *col_1* value in the new table. Type casting is attempted if the column names are identical but the columns types are different.
- If *col_1* and *col_2* are defined in the new table, *col_1*, col_2 and *col_3* are specified in the column list of the *select_statement* and there is a containment relationship between all of them, *col_1*, *col_2* and *col_3* are created in the new table and the result data of the **SELECT** statement is stored as values for all columns. Type casting is attempted if the column names are identical but the columns types are different.
- If columns *col_1* and *col_2* are defined in the new table and *col_1* and *col_3* are defined in the column list of *select_statement* without any containment relationship between them, *col_1*, *col_2* and *col_3* are created in the new table, the result data of the **SELECT** statement is stored only for *col_1* and *col_3* which are specified in *select_statement*, and **NULL** is stored as the value of *col_2*.
- Column aliases can be included in the column list of *select_statement*. In this case, new column alias is used as a new table column name. It is recommended to use an alias because invalid column name is created, if an alias does not exist when a function calling or an expression is used.
- The **REPLACE** option is valid only when the **UNIQUE** constraint is defined in a new table column (*col_1*). When duplicate values exist in the result record of *select_statement*, a **UNIQUE** value is stored for *col_1* if the **REPLACE** option has been defined, or an error message is displayed if the **REPLACE** option is omitted due to the violation of the **UNIQUE** constraint.

Syntax

```
CREATE {TABLE | CLASS} <table_name>
                   [( <column_definition> [,<table_constraint>]... )]
                   [REPLACE] AS <select_statement>
```

- *table_name* : A name of the table to be created.
- *column_definition* : Defines a column. If it is omitted, the column schema of **SELECT** statement is replicated; however, the constraint or the **AUTO_INCREMENT** attribute is not replicated.
- *table_constraint* : Defines table constraint.
- *select_statement* : A **SELECT** statement targeting a source table that already exists in the database.

Example

```
CREATE TABLE a_tbl(
id INT NOT NULL DEFAULT 0 PRIMARY KEY,
phone VARCHAR(10));
INSERT INTO a_tbl VALUES(1,'111-1111'), (2,'222-2222'), (3, '333-3333');

--creating a table without column definition
CREATE TABLE new_tbl1 AS SELECT * FROM a_tbl;
SELECT * FROM new_tbl1;
;xr

=== <Result of SELECT Command in Line 1> ===

           id  phone
===================================
            1  '111-1111'
            2  '222-2222'
            3  '333-3333'


3 rows selected.

--all of column values are replicated from a_tbl
CREATE TABLE new_tbl2
(id INT NOT NULL AUTO_INCREMENT PRIMARY KEY, phone VARCHAR) AS SELECT * FROM a_tbl;
SELECT * FROM new_tbl2;
;xr

=== <Result of SELECT Command in Line 1> ===

           id  phone
===================================
            1  '111-1111'
            2  '222-2222'
            3  '333-3333'

3 rows selected.

--some of column values are replicated from a_tbl and the rest is NULL
CREATE TABLE new_tbl3
(id INT, name VARCHAR) AS SELECT id, phone FROM a_tbl;
SELECT * FROM new_tbl3
;xr

=== <Result of SELECT Command in Line 1> ===

  name                             id  phone
=========================================================
  NULL                              1  '111-1111'
  NULL                              2  '222-2222'
  NULL                              3  '333-3333'

3 rows selected.

--column alias in the select statement should be used in the column definition
CREATE TABLE new_tbl4
(id1 int, id2 int)AS SELECT t1.id id1, t2.id id2 FROM new_tbl1 t1, new_tbl2 t2;
SELECT * FROM new_tbl4;
;xr

=== <Result of SELECT Command in Line 1> ===

          id1          id2
==========================
            1            1
            1            2
            1            3
            2            1
            2            2
            2            3
            3            1
            3            2
            3            3


9 rows selected.

--REPLACE is used on the UNIQUE column
CREATE TABLE new_tbl5(id1 int UNIQUE) REPLACE AS SELECT * FROM new_tbl4;
SELECT * FROM new_tbl5;
;xr

=== <Result of SELECT Command in Line 1> ===

          id1          id2
==========================
            1            3
            2            3
            3            3
```

```
3 rows selected.
```

ALTER TABLE

Overview

Description

You can modify the structure of a table by using the **ALTER** statement. You can perform operations on the target table such as adding/deleting columns, creating/deleting indexes, and type casting existing columns as well as changing table names, column names and constraints. **TABLE** and **CLASS** are used interchangeably **VIEW** and **VCLASS**, and **COLUMN** and **ATTRIBUTE** as well.

Syntax

```
ALTER [ < class_type> ] < table_name> < alter_clause>

< class_type> : TABLE | CLASS | VCLASS | VIEW

< alter_clause> :     ADD < alter_add> [ INHERIT < resolution_comma_list> ] |
                                      ADD { KEY | INDEX } [index_name] (< index_col_name> )
|
                                      ALTER [ COLUMN ] column_name SET DEFAULT <
value_specifiation> |
                                      DROP < alter_drop> [ INHERIT <
resolution_comma_list> ] |
                                      DROP { KEY | INDEX }  index_name |
                                      DROP FOREIGN KEY  constraint_name |
                                      DROP PRIMARY
KEY  |
                                      RENAME < alter_rename> [ INHERIT <
resolution_comma_list> ] |
                                      CHANGE < alter_change> |
                                      INHERIT < resolution_comma_list>

< alter_add> : [ ATTRIBUTE | COLUMN ] [(]< class_element_comma_list> [)] [ FIRST | AFTER
old_column_name  ] |
                                CLASS ATTRIBUTE < column_definition_comma_list> |
                                CONSTRAINT < constraint_name > < column_constraint>
( column_name )|
                                FILE < file_name_comma_list> |
                                METHOD < method_definition_comma_list> |
                                QUERY < select_statement> |
                                super class < class_name_comma_list>

< alter_change> : FILE < file_path_name> AS < file_path_name> |
                                      METHOD < method_definition_comma_list> |
                                      QUERY [ < unsigned_integer_literal> ] <
select_statement> |
                                      < column_name> DEFAULT < value_specifiation>

< alter_drop> : [ ATTRIBUTE | COLUMN | METHOD ]
                                  < column_name_comma_list> |
                                  FILE < file_name_comma_list> |
                                  QUERY [ < unsigned_integer_literal> ] |
                                  super class < class_name_comma_list> |
                                  CONSTRAINT < constraint_name>

< alter_rename> : [ ATTRIBUTE | COLUMN | METHOD ]
                                      < old_column_name> AS < new_column_name> |
                                      FUNCTION OF < column_name> AS < function_name>
                                      FILE < file_path_name> AS < file_path_name>

< resolution> : { column_name | method_name } OF < super class_name>
                                  [ AS alias ]

< class_element> : < column_definition> | < table_constraint>
< column_constraint> : UNIQUE [ KEY ] | PRIMARY KEY | FOREIGN KEY

< index_col_name> ::=
column_name [(length)] [ ASC | DESC ]
```

Caution

The table name can be changed only by the table owner, **DBA** and **DBA** members. The other users must be granted to change the name by the owner or **DBA** (see Granting Authorization for more information on authorization).

ADD COLUMN Clause

Description

You can add a new column by using the **ADD COLUMN** clause. You can specify the location of the column to be added by using the **FIRST** or **AFTER** keyword.

Syntax

```
ALTER [ TABLE | CLASS | VCLASS | VIEW ] table_name
ADD [ COLUMN | ATTRIBUTE ] [(]<column_definition>[)] [ FIRST | AFTER old_column_name ]

column_definition ::=
column_name column_type
    { [ NOT NULL | NULL ] |
      [ { SHARED <value_specification> | DEFAULT <value_specification> }
          | AUTO_INCREMENT [(seed, increment)] ] |
      [ UNIQUE [ KEY ] |
          [ PRIMARY KEY | FOREIGN KEY REFERENCES
              [ referenced_table_name ]( column_name_comma_list )
              [ <referential_triggered_action> ... ]
          ]
      ] } ...

<referential_triggered_action> ::=
{ ON UPDATE <referential_action> } |
{ ON DELETE <referential_action> } |
{ ON CACHE OBJECT cache_object_column_name }

<referential_action> ::=
CASCADE | RESTRICT | NO ACTION | SET NULL
```

- *table_name* : Specifies the name of a table that has a column to be added.
- *column_definition* : Specifies the name, data type, and constraints of a column to be added.
- **AFTER** *oid_column_name* : Specifies the name of an existing column before the column to be added.

Example

```
CREATE TABLE a_tbl;
ALTER TABLE a_tbl ADD COLUMN age INT DEFAULT 0 NOT NULL;
INSERT INTO a_tbl(age) VALUES(20),(30),(40);
ALTER TABLE a_tbl ADD COLUMN name VARCHAR FIRST;
ALTER TABLE a_tbl ADD COLUMN id INT NOT NULL AUTO_INCREMENT UNIQUE;
ALTER TABLE a_tbl ADD COLUMN phone VARCHAR(13) DEFAULT '000-0000-0000' AFTER name;

SELECT * FROM a_tbl;
csql> ;xr

=== <Result of SELECT Command in Line 1> ===

  name                  phone                         age           id
======================================================================
  NULL                  '000-0000-0000'                20          NULL
  NULL                  '000-0000-0000'                30          NULL
  NULL                  '000-0000-0000'                40          NULL

--adding multiple columns
ALTER TABLE a_tbl ADD COLUMN (age1 int, age2 int, age3 int);
;xr
```

```
1 command(s) successfully processed.
```

ADD CONSTRAINT Clause

Description

You can add a new constraint by using the **ADD CONSTRAINT** clause.

Syntax

```
ALTER [ TABLE | CLASS | VCLASS | VIEW ] table_name
ADD CONSTRAINT < constraint_name > column_constraint ( column_name_comma_list )

column_constraint ::=
UNIQUE [ KEY ] |
PRIMARY KEY |
FOREIGN KEY REFERENCES [referenced_table_name]( column_name_comma_list )
                       [ <referential_triggered_action> ... ]

<referential_triggered_action> ::=
{ ON UPDATE <referential_action> } |
{ ON DELETE <referential_action> } |
{ ON CACHE OBJECT cache_object_column_name }

<referential_action> ::=
CASCADE | RESTRICT | NO ACTION | SET NULL
```

- *table_name* : Specifies the name of a table that has a constraint to be added.
- *constraint_name* : Specifies the name of a constraint to be added, or it can be omitted. If omitted, a name is automatically assigned.
- *column_constraint* : Defines a constraint for the specified column. For more information, see Constraint Definition .

Example

```
ALTER TABLE a_tbl ADD CONSTRAINT PRIMARY KEY(id);
ALTER TABLE a_tbl ADD CONSTRAINT UNIQUE u_key1(id);
```

ADD INDEX Clause

Description

You can define the index attributes for a specific column by using the **ADD INDEX** clause.

Syntax

```
ALTER [ TABLE | CLASS ] table_name ADD { KEY | INDEX } [index_name] (<index_col_name>)

<index_col_name> ::=
column_name [(length)] [ ASC | DESC ]
```

- *table_name* : Specifies the name of a table to be modified.
- *index_name* : Specifies the name of an index. If omitted, a name is automatically assigned.
- *index_col_name* : Specifies the column that has an index to be defined. **ASC** or **DESC** can be specified for a column option; *prefix_length* of an index key also can be specified for a column option.

Example

```
ALTER TABLE a_tbl ADD INDEX (age ASC), ADD INDEX(phone DESC);
;schema a_tbl

=== <Help: Schema of a Class> ===

 <Class Name>

     a_tbl

<Attributes>
```

```
     name                     CHARACTER VARYING(1073741823) DEFAULT ''
     phone                    CHARACTER VARYING(13) DEFAULT '111-1111'
     age                      INTEGER
     id                       INTEGER AUTO_INCREMENT  NOT NULL

 <Constraints>

     UNIQUE u_a_tbl_id ON a_tbl (id)
     INDEX i_a_tbl_age ON a_tbl (age)
     INDEX i_a_tbl_phone_d ON a_tbl (phone DESC)

Current transaction has been committed.
```

ALTER COLUMN ... SET DEFAULT Clause

Description

You can specify a new default value for a column that has no default value or modify the existing default value by using the **ALTER COLUMN** ... **SET DEFAULT**. You can use the **CHANGE** clause to change the default value of multiple columns with a single statement. For more information, see the CHANGE Clause.

Syntax

```
ALTER [ TABLE | CLASS ] table_name ALTER [COLUMN] column_name SET DEFAULT value
```

- *table_name* : Specifies the name of a table that has a column whose default value is to be modified.
- *column_name* : Specifies the name of a column whose default value is to be modified.
- *value* : Specifies a new default value.

Example

```
;schema a_tbl

=== <Help: Schema of a Class> ===
    a_tbl

<Attributes>
    name                    CHARACTER VARYING(1073741823)
    phone                   CHARACTER VARYING(13) DEFAULT '000-0000-0000'
    age                     INTEGER
    id                      INTEGER AUTO_INCREMENT NOT NULL

<Constraints>
    UNIQUE u_a_tbl_id ON a_tbl (id)

Current transaction has been committed.

ALTER TABLE a_tbl ALTER COLUMN name SET DEFAULT '';
ALTER TABLE a_tbl ALTER COLUMN phone SET DEFAULT '111-1111';

;schema a_tbl

=== <Help: Schema of a Class> ===


 <Class Name>

     a_tbl

 <Attributes>

     name                     CHARACTER VARYING(1073741823) DEFAULT ''
     phone                    CHARACTER VARYING(13) DEFAULT '111-1111'
     age                      INTEGER
     id                       INTEGER AUTO_INCREMENT  NOT NULL

 <Constraints>
```

```
     UNIQUE u_a_tbl_id ON a_tbl (id)
```

CHANGE Clause

Description

You can specify or change the default of a column by using the **DEFAULT** keyword of the **CHANGE** clause.

Syntax

```
ALTER [ TABLE | CLASS | VCLASS | VIEW ] table_name CHANGE [ CLASS ] column_name DEFAULT
value [,  column_name DEFAULT value, ...]
```

- *table_name* : Specifies the name of a table that has a column whose default name is to be modified .
- *column_name* : Specifies the name of a column that will have a new default value.
- *value* : Specifies a new default value.

Example

```
ALTER TABLE a_tbl CHANGE name DEFAULT 'abc', phone DEFAULT '000-0000'

schema a_tbl

=== < Help: Schema of a Class> ===


  < Class Name>

          a_tbl

  < Attributes>

          name                              CHARACTER VARYING(1073741823) DEFAULT 'abc'
          phone                             CHARACTER VARYING(13) DEFAULT '000-0000'
          age                               INTEGER
          id                                INTEGER AUTO_INCREMENT   NOT NULL

  < Constraints>

          UNIQUE u_a_tbl_id ON a_tbl (id)
          INDEX i_a_tbl_age ON a_tbl (age)
          INDEX i_a_tbl_phone_d ON a_tbl (phone DESC)
```

RENAME COLUMN Clause

Description

You can change the name of the column by using the **RENAME COLUMN** clause.

Syntax

```
ALTER [ TABLE | CLASS | VCLASS | VIEW ] table_name
RENAME [ COLUMN | ATTRIBUTE ] old_column_name { AS  |  TO  } new_column_name
```

- *table_type* : Specifies the name of a table that has a column to be renamed.
- *old_column_name* : Specifies the name of a column.
- *new_column_name* : Specifies a new column name after the **AS** keyword.

Example

```
ALTER TABLE a_tbl  RENAME COLUMN name AS name1
```

DROP COLUMN Clause

Description

You can delete a column in a table by using the **DROP COLUMN** clause. You can specify multiple columns to delete simultaneously by separating them with commas (,).

Syntax

```
ALTER [ TABLE | CLASS | VCLASS | VIEW ] table_name
DROP [ COLUMN | ATTRIBUTE ] column_name, ...
```

- *table_name* : Specifies the name of a table that has a column to be deleted.
- *column_ name* : Specifies the name of a column to be deleted. Multiple columns can be specified by separating them with commas (,).

Example

```
ALTER TABLE a_tbl DROP COLUMN age1,age2,age3;
```

DROP CONSTRAINT Clause

Description

You can drop the constraints pre-defined for the table, such as **UNIQUE**, **PRIMARY KEY** and **FOREIGN KEY** by using the **DROP CONSTRAINT** clause. In this case, you must specify a constraint name. You can check these names by using the CSQL command (**;schema table_name**).

Syntax

```
ALTER [ TABLE | CLASS ] table_name
DROP CONSTRAINT constraint_name
```

- *table_name* : Specifies the name of a table that has a constraint to be dropped.
- *constraint_name* : Specifies the name of a constraint to be dropped.

Example

```
ALTER TABLE a_tbl DROP CONSTRAINT pk_a_tbl_id;
ALTER TABLE a_tbl DROP CONSTRAINT fk_a_tbl_id;
ALTER TABLE a_tbl DROP CONSTRAINT u_a_tbl_id;
```

DROP INDEX Clause

Description

You can delete an index defined for a column by using the **DROP INDEX** clause.

Syntax

```
ALTER [ TABLE | CLASS ] table_name DROP INDEX index_name
```

- *table_name* : Specifies the name of a table that has an index attribute to be deleted.
- *index_name* : Specifies the name of an index to be deleted.

Example

```
ALTER TABLE a_tbl DROP INDEX i_a_tbl_age;
```

DROP PRIMARY KEY Clause

Description

You can delete a primary key constraint defined for a table by using the **DROP PRIMARY KEY** clause. You do have to specify the name of the primary key constraint because only one primary key can be defined by table.

Syntax

```
ALTER [ TABLE | CLASS ] table_name DROP INDEX PRIMARY KEY
```

- *table_name* : Specifies the name of a table that has a primary key constraint to be deleted.

Example

```
ALTER TABLE a_tbl DROP PRIMARY KEY;
```

DROP FOREIGN KEY Clause

Description

You can drop a foreign key constraint defined for a table using the **DROP FOREIGN KEY** clause.

Syntax

```
ALTER [ TABLE | CLASS ] table_name DROP FOREIGN KEY constraint_name
```

- *table_name* : Specifies the name of a table whose constraint is to be deleted.
- *constraint_name* : Specifies the name of foreign key constraint to be deleted.

Example

```
ALTER TABLE a_tbl DROP FOREIGN KEY fk_a_tbl_id;
```

DROP TABLE

Description

You can drop an existing table by the **DROP** statement. Multiple tables can be dropped by a single **DROP** statement. All rows of table are also dropped.

Syntax

```
DROP [ TABLE | CLASS | VIEW | VCLASS ] < table_specification_comma_list>

< table_specification> ::=
< single_table_spec> | ( < single_table_spec_comma_list> )

< single_table_spec> ::=
|[ ONLY ] table_name
| ALL table_name [ ( EXCEPT table_name, ... ) ]
```

- *table_name* : Specifies the name of the table to be dropped. You can delete multiple tables simultaneously by separating them with commas.
- If a super class name is specified after the **ONLY** keyword, only the super class, not the subclasses inheriting from it, is deleted. If a super class name is specified after the **ALL** keyword, the super classes as well as the subclasses inheriting from it are all deleted. You can specify the list of subclasses not to be deleted after the **EXCEPT** keyword.
- If subclasses that inherit from the super class specified after the **ALL** keyword are specified after the **EXCEPT** keyword, they are not deleted.

Example

The following is an example of dropping the history table.

```
DROP TABLE history
```

RENAME TABLE

Description

You can change the name of a table by using the **RENAME TABLE** statement and specify a list of the table name to change the names of multiple tables. You can use **TO** instead of **AS**.

Syntax

```
RENAME  [ TABLE | CLASS | VIEW | VCLASS ] old_table_name { AS | TO } new_table_name [,
old_table_name { AS | TO } new_table_name, ... ]
```

- *old_table_name* : Specifies the old table name to be renamed.
- *new_table_name* : Specifies a new table name.

Example

```
RENAME TABLE a_tbl AS aa_tbl;
RENAME TABLE a_tbl TO aa_tbl, b_tbl TO bb_tbl;
```

Caution

The table name can be changed only by the table owner, **DBA** and **DBA** members. The other users must be granted to change the name by the owner or **DBA** (see Granting Authorization for more information on authorization).

Index Definition

CREATE INDEX

Description

Use the **CREATE INDEX** statement to create an index in the specified table.

Syntax

```
CREATE [ REVERSE ] [ UNIQUE ] INDEX [ index_name ]
ON table_name ( column_name[(prefix_length)] [ASC | DESC] [ {, column_name[(prefix_length)]
[ASC | DESC]} ...] ) [ ; ]
```

- **REVERSE** : Creates an index in the reverse order. A reverse index helps to increase sorting speed in descending order.
- **UNIQUE** : Creates an index with unique values.
- *index_name* : Specifies the name of the index to be created. The index name must be unique in the table. If omitted, a name is automatically assigned.
- *prefix_length* : When you specify an index for character- or bit string-type column, you can create an index by specifying the beginning part of the column name as a prefix. You can specify the length of the prefix in bytes in parentheses next to the column name. You cannot specify *prefix_length* in a multiple column index or a **UNIQUE** index.
- *table_name* : Specifies the name of the table where the index is to be created.
- *column_name* : Specifies the name of the column where the index is to be applied. To create a composite index, specify two or more column names.
- **ASC** | **DESC** : Specifies the sorting order of columns. In case of a **REVERSE** index, **ASC** is ignored and **DESC** is applied.

Example 1

The following is an example of creating a single column index. In this example, 1-byte long prefix is specified for the nation_code column when creating an index.

```
CREATE INDEX ON game(nation_code(1));
CREATE INDEX game_date_idx ON game(game_date);
```

Example 2

The following is an example of creating a reverse index.

```
CREATE REVERSE INDEX old_index ON participant(gold);
```

Example 3

The following is an example of creating a multiple column index.

```
CREATE INDEX name_nation_idx ON athlete(name, nation_code);
```

ALTER INDEX

Description

Use the **ALTER INDEX** statement to rebuild an index. (That is, drop and rebuild an index.) There are the following two ways to specify an index to be rebuilt:

- Specifying it as the name of the index
- Specifying it as the name of the table or the column where the index is specified

Syntax

```
ALTER [ REVERSE ] [ UNIQUE ] INDEX index_name
[ON { ONLY } table_name ( column_name [ {, column_name } ...) ] REBUILD [ ; ]

ALTER [ REVERSE ] [ UNIQUE ] INDEX
ON { ONLY } table_name ( column_name [ {, column_name } ...) REBUILD [ ; ]
```

- **REVERSE** : Creates an index in the reverse order. A reverse index helps to increase sorting speed in descending order.
- **UNIQUE** : Creates an index with unique values.
- *index_name* : Specifies the name of the index to be altered. The index name must be unique in the table.
- *table_name* : Specifies the name of the table where the index is to be created.
- *column_name* : Specifies the name of the column where the index is to be applied. To create a multiple column index, specify two or more column names.

Example

The following are examples of many ways of re-creating indexes:

```
ALTER INDEX i_game_medal ON game(medal) REBUILD;
ALTER INDEX game_date_idx REBUILD;
```

DROP INDEX

Description

Use the **DROP INDEX** statement to drop an index. There are the following two ways to specify the index to be dropped:

- To specify the name of the index
- To specify the name of the table or the column where the index is specified

Syntax

```
DROP [ REVERSE ] [ UNIQUE ] INDEX index_name
[ON table_name ( column_name [ {, column_name } ...) ] [    ]

DROP [ REVERSE ] [ UNIQUE ] INDEX
  ON table_name ( column_name [ {, column_name } ...) [ ]
```

- **REVERSE** : Specifies that the index to be dropped is a reverse index.
- **UNIQUE** : Specifies that the index to be dropped is a unique index.
- *index_name* : Specifies the name of the index to be dropped.
- *table_name* : Specifies the name of the table whose index is to be dropped.
- *column_name* : Specifies the name of the column whose index is to be dropped.

Example

The following are examples of many ways of dropping indexes:

```
DROP INDEX ON game(medal)

DROP INDEX game_date_idx

DROP REVERSE INDEX gold_index ON participant(gold)

DROP INDEX name_nation_idx ON athlete(name, nation_code)
```

VIEW

CREATE VIEW

Overview

Description

A view is a virtual table that does not exist physically. You can create a view by using an existing table or a query. **VIEW** and **VCLASS** are used interchangeably.

Use **CREATE VIEW** statement to create a view.

Syntax

```
CREATE [OR REPLACE] {VIEW | VCLASS} <view_name>
                           [ <subclass_definition> ]
                           [ ( <view_column_def_comma_list> ) ]
                           [ CLASS ATTRIBUTE
                             ( <column_definition_comma_list> ) ]
                           [ METHOD <method_definition_comma_list> ]
                           [ FILE <method_file_comma_list> ]
                           [ INHERIT <resolution_comma_list> ]
                           [ AS <select_statement> ]
                           [ WITH CHECK OPTION ]

<view_column_definition> ::= <column_definition> | <column_name>

<column_definition> :
column_name column_type [ <default_or_shared> ] [ <column_constraint_list>]

<default_or_shared> :
{SHARED [ <value_specification> ] | DEFAULT <value_specification> } |
AUTO_INCREMENT [ (seed, increment) ]

<column_constraint> :
NOT NULL | UNIQUE | PRIMARY KEY | FOREIGN KEY REFERENCES...

<subclass_definition> :
{ UNDER | AS SUBCLASS OF } table_name_comma_list

<method_definition> :
[ CLASS ] method_name
[ ( [ argument_type_comma_list ] ) ]
[ result_type ]
[ FUNCTION function_name ]

<resolution> :
[ CLASS ] { column_name | method_name } OF superclass_name
[ AS alias ]
```

- **OR REPLACE** : If the keyword **OR REPLACE** is specified after **CREATE**, the existing virtual table is replaced by a new one without displaying any error message, even when the view_name overlaps with the existing virtual table name.
- *view_name* : Specify the name of the table to be created. Must be unique in the database.
- *view_column_definition*
  - *column_name* : Defines a column of the virtual table.
- *column_type* : Specifies the data type of the column.

AS *select_statement* : A valid **SELECT** statement must be specified. A virtual table is created on this basis.

WITH CHECK OPTION : If this option is specified, the update or insert operation is possible only when the condition specified in the **WHERE** clause of the *select_statement* is satisfied. Therefore, this option is used to disallow the update of a virtual table that violates the condition.

Example

```
CREATE TABLE a_tbl(
id INT NOT NULL,
phone VARCHAR(10))
INSERT INTO a_tbl VALUES(1,'111-1111'), (2,'222-2222'), (3, '333-3333'), (4, NULL), (5,
NULL)

--creating a new view based on AS select_statement from a_tbl
CREATE VIEW b_view AS SELECT * FROM a_tbl WHERE phone IS NOT NULL WITH CHECK OPTION
SELECT * FROM b_view
xr

=== < Result of SELECT Command in Line 1> ===

                      id   phone
===================================
                        1   '111-1111'
                        2   '222-2222'
                        3   '333-3333'


3 rows selected.

--WITH CHECK OPTION doesn’t allow to update column value which violates WHERE clause
UPDATE b_view SET phone=NULL
xr

In line 1, column 72,

ERROR: Check option exception on view b_view.

--creating view which name is as same as existing view name
CREATE OR REPLACE VIEW b_view AS SELECT * FROM a_tbl ORDER BY id DESC
xr

Current transaction has been committed.

1 command(s) successfully processed.

--the existing view has been replaced as a new view by OR REPLACE keyword
SELECT * FROM b_view
xr

=== < Result of SELECT Command in Line 1> ===

                      id   phone
===================================
                        5   NULL
                        4   NULL
                        3   '333-3333'
                        2   '222-2222'
                        1   '111-1111'


5 rows selected.
```

Condition for Creating Updatable VIEW

Description

To update data in a virtual table, it must be updatable because an option is needed to define data.

A virtual table is updatable if it satisfies the following conditions:

- The **FROM** clause must include only one table or updatable virtual table. However, two tables included in parentheses as in **FROM** (class_x, class_y) can be updated because they represent one table.
- The **DISTINCT** or **UNIQUE** statement must not be included.
- The **GROUP BY... HAVING** statement must not be included.
- Aggregate functions such as **SUM**( ) or **AVG**( ) must not be included.

- The entire query must consist of queries that can be updated by **UNION ALL**, not by **UNION**. However, the table must exist only in one of the queries that constitute **UNION ALL**.
- If an row is inserted into a virtual table created by using the **UNION ALL** statement, the system determines which table the row will be inserted into. This cannot be done by the user. To control this, the user must manually insert the row or create a separate virtual table for insertion.

Even when all rules above are satisfied, each column of the updatable virtual table may not be updatable. For a column to be updatable, the following rules must be observed:

- Path expressions must not be updatable.
- Columns of number type with an arithmetic operator must not be updatable.

Even though the column defined in the virtual table is updatable, the virtual table can be updated only when there is an appropriate update privilege granted on the table included in the **FROM** clause. Also, there must be an access privilege on the virtual table. The way to grant an access privilege on a virtual table is the same as on a table. For more information on granting authorizations, see the Granting Authorization section.

ALTER VIEW

ADD QUERY Clause

Description

You can add a new query to a query specification by using the **ADD QUERY** clause of the **ALTER VIEW** statement. 1 is assigned to the query defined when a virtual table was created, and 2 is assigned to the query added by the **ADD QUERY** clause.

Syntax

```
ALTER [ VIEW | VCLASS ] view_name
ADD QUERY select_statement
[ INHERIT resolution [ {, resolution }_ ] ]

resolution :
{ column_name | method_name } OF super class_name [ AS alias ]
```

- *view_name* : Specifies the name of the virtual table where the query is to be added.
- *select_statement* : Specifies the query to be added.

Example

```
SELECT * FROM b_view
xr

=== < Result of SELECT Command in Line 1> ===

                      id   phone
===================================
                        1   '111-1111'
                        2   '222-2222'
                        3   '333-3333'
                        4   NULL
                        5   NULL


ALTER VIEW b_view ADD QUERY SELECT * FROM a_tbl WHERE id IN (1,2)
SELECT * FROM b_view
csql> xr

=== < Result of SELECT Command in Line 1> ===

                      id   phone
===================================
                        1   '111-1111'
                        2   '222-2222'
                        3   '333-3333'
                        4   NULL
                        5   NULL
                        1   '111-1111'
                        2   '222-2222'


7 rows selected.
```

AS SELECT Clause

Description

You can change the **SELECT** query defined in the virtual table by using the **AS SELECT** clause in the **ALTER VIEW** statement. This function is working like the **CREATE OR REPLACE** statement. You can also change the query by specifying the query number 1 in the **CHANGE QUERY** clause of the **ALTER VIEW** statement.

Syntax

```
ALTER [ VIEW | VCLASS ] view_name AS select_statement
```

- *view_name* : Specifies the name of the virtual table to be modified.
- *select_statement* : Specifies the new query statement to replace the **SELECT** statement defined when the virtual table is created.

Example

```
ALTER VIEW b_view AS SELECT * FROM a_tbl WHERE phone IS NOT NULL;
SELECT * FROM b_view;
;xr

=== <Result of SELECT Command in Line 1> ===

           id  phone
===================================
            1  '111-1111'
            2  '222-2222'
            3  '333-3333'


3 rows selected.
```

CHANGE QUERY Clause

Description

You can change the query defined in the query specification by using the **CHANGE QUERY** clause reserved word of the **ALTER VIEW** statement.

Syntax

```
ALTER [ VIEW | VCLASS ] view_name
CHANGE QUERY [ integer ] select_statement [ ; ]
```

- *view_name* : Specifies the name of the virtual table to be changed.
- *integer* : Specifies the number value of the query to be changed. The default value is 1.
- *select_statement* : Specifies the new query that will replace the query whose query number is *integer*.

Example

```
--adding select_statement which query number is 2 and 3 for each
ALTER VIEW b_view ADD QUERY SELECT * FROM a_tbl WHERE id IN (1,2);
ALTER VIEW b_view ADD QUERY SELECT * FROM a_tbl WHERE id = 3;
SELECT * FROM b_view;
;xr

=== <Result of SELECT Command in Line 1> ===
           id  phone
===================================
            1  '111-1111'
            2  '222-2222'
            3  '333-3333'
            4  NULL
            5  NULL
            1  '111-1111'
            2  '222-2222'
            3  '333-3333'


8 rows selected.

--altering view changing query number 2
ALTER VIEW b_view CHANGE QUERY 2 SELECT * FROM a_tbl WHERE phone IS NULL;
SELECT * FROM b_view;
;xr

=== <Result of SELECT Command in Line 1> ===

           id  phone
===================================
            1  '111-1111'
            2  '222-2222'
            3  '333-3333'
            4  NULL
            5  NULL
            4  NULL
            5  NULL
            3  '333-3333'
```

DROP QUERY Clause

Description

You can drop a query defined in the query specification by using the **DROP QUERY** of the **ALTER VIEW** statement.

Example

```
ALTER VIEW b_view DROP QUERY 2,3;
SELECT * FROM b_view;
;xr

=== <Result of SELECT Command in Line 1> ===

           id  phone
===================================
            1  '111-1111'
            2  '222-2222'
            3  '333-3333'
            4  NULL
            5  NULL


5 rows selected.
```

DROP VIEW

Description

You can drop a view by using the **DROP VIEW** clause. The way to drop a view is the same as to drop a regular table.

Syntax

```
DROP [ VIEW | VCLASS ] view_name [ { ,view_name , ... } ]
```

- *view_name* : Specifies the name of the virtual table to be dropped.

Example

```
DROP VIEW b_view;
```

RENAME VIEW

Description

You can change the name of a virtual table by using the **RENAME VIEW** statement.

Syntax

```
RENAME [ TABLE |CLASS | VIEW | VCLASS ] old_view_name AS new_view_name [ ; ]
```

- *old_view_name* : Specifies the name of the table to be modified.
- *new_view_name* : Specifies the new name of the virtual table.

Example

The following is an example of renaming a view name to game_2004.

```
RENAME VIEW game_2004 AS info_2004;
```

SERIAL

CREATE SERIAL

Serial is an object that creates a unique sequence number, and has the following characteristics.

- The serial is useful in creating a unique sequence number in multi-user environment.
- Generated serial numbers are not related with table so, you can use the same serial in multiple tables.
- All users including **public** can create a serial object. Once it is created, all users can get the number by using **CURRENT_VALUE** and **NEXT_VALUE**.
- Only owner of a created serial object and **dba** can update or delete a serial object. If an owner is **public**, all users can update or delete it.

Description

You can create a serial object in the database by using the **CREATE SERIAL** statement.

Syntax

```
CREATE SERIAL serial_name
[ START WITH initial ]
[ INCREMENT BY interval]
[ MINVALUE min | NOMINVALUE ]
[ MAXVALUE max | NOMAXVALUE ]
[ CACHE integer | NOCACHE ]
```

- serial_identifier : Specifies the name of the serial to be generated.
- START WITH initial : Specifies the initial value of serial with 38 digits or less. In the ascending serial, that is its minimum value. In the descending serial, this is its maximum value.
- INCREMENT BY interval : Specifies the increment of the serial. You can specify any integer with 38 digits or less except for zero at interval. The absolute value of the interval must be smaller than the difference between MAXVALUE and MINVALUE. If a negative number is specified, the serial is in descending order otherwise, it is in ascending order. The default value is 1.
- MINVALUE : Specifies the minimum value of the serial, with 38 digits or less. MINVALUE must be smaller than or equal to the initial value and smaller than the maximum value.
- NOMINVALUE : 1 is set automatically as a minimum value for the ascending serial -(10)38 for the descending serial.
- MAXVALUE : Specifies the maximum number of the serial with 38 digits or less. MAXVALUE must be smaller than or equal to the initial value and greater than the minimum value.
- NOMAXVALUE : (10)37 is set automatically as a maximum value for the ascending serial -1 for the descending serial.
- CYCLE : Specifies that the serial will be generated continuously after reaching the maximum or minimum value. When a serial in ascending order reaches the maximum value, the minimum value is created as the next value; when a serial in descending order reaches the minimum value, the maximum value is created as the next value.
- NOCYCLE : Specifies that the serial will not be generated any more after reaching the maximum or minimum value. The default value is NOCYCLE.
- CACHE : Saves as many serials as the number specified by "integer" in the cache to improve the performance of the serials and fetches a serial value when one is requested. If all cached values are used up, as many serials as "integer" are fetched again from the disk to the memory. If the database server stops accidently, all cached serial values are deleted. For this reason, the serial values before and after the restart of the database server may be discontinuous. Because the transaction rollback dose not affect the cached serial values, the request for the next serial will return the next value of the value used (or fetched) lastly when the transaction is rolled back. The "integer" after the CACHE keyword cannot be omitted. If the "integer" is equal to or smaller than 1, the serial cache is not applied.
- NOCACHE : Does not use the serial cache feature. The serial value is updated every time and a new serial value is fetched from the disk upon each request.

Example 1

```
--creating serial with default values
CREATE SERIAL order_no

--creating serial within a specific range
CREATE SERIAL order_no START WITH 10000 INCREMENT BY 2 MAXVALUE 20000
--creating serial with specifying the number of cached serial values
CREATE SERIAL order_no START WITH 10000 INCREMENT BY 2 MAXVALUE 20000 CACHE 3

--selecting serial information from the db_serial class
SELECT * FROM db_serial
xr

=== < Result of SELECT Command in Line 1> ===

    name                       current_val           increment_val                max_val
                 min_val                 cyclic           started             cached_num
             att_name
==========================================================================================
==========================================================
'order_no'           10006                       2

20000                     10000                               0                       1
                            3                       NULL
```

Example 2

The following is an example of creating the athlete_idx table to save athlete codes and names and then creating an instance by using the *order_no*. NEXT_VALUE increases the serial number and returns its value.

```
CREATE TABLE athlete_idx( code INT, name VARCHAR(40) )
CREATE SERIAL order_no START WITH 10000 INCREMENT BY 2 MAXVALUE 20000
INSERT INTO athlete_idx VALUES (order_no.NEXT_VALUE, 'Park')
INSERT INTO athlete_idx VALUES (order_no.NEXT_VALUE, 'Kim')
INSERT INTO athlete_idx VALUES (order_no.NEXT_VALUE, 'Choo')
INSERT INTO athlete_idx VALUES (order_no.CURRENT_VALUE, 'Lee')
SELECT * FROM athlete_idx

=== < Result of SELECT Command in Line 1> ===

                  code   name
===================================
                10000   'Park'
                10002   'Kim'
                10004   'Choo'
                10004   'Lee'
```

ALTER SERIAL

Description

With the **ALTER SERIAL** statement, you can update the increment of the serial value, set or delete its initial or minimum/maximum values, and set its cycle attribute.

Syntax

```
ALTER SERIAL serial_identifier
[ INCREMENT BY interval ]
[ START WITH initial_value ]
[ MINVALUE min | NOMINVALUE ]
[ MAXVALUE max | NOMAXVALUE ]
[ CACHE integer | NOCACHE ]
```

- *serial_identifier* : Specifies the name of the serial to be created.
- **INCREMENT BY** *interval* : Specifies the increment of the serial. For the *interval*, you can specify any integer with 38 digits or less except for zero. The absolute value of the *interval* must be smaller than the difference between **MAXVALUE** and **MINVALUE**. If a negative number is specified, the serial is in descending order; otherwise, it is in ascending order. The default value is **1**.

- **START WITH** *initial_value* : Changes the initial value of Serial.
- **MINVALUE** : Specifies the minimum value of the serial with 38 digits or less. **MINVALUE** must be smaller than or equal to the initial value and smaller than the maximum value.
- **NOMINVALUE** : 1 is set automatically as a minimum value for the ascending serial; -(10)^{36} for the descending serial.
- **MAXVALUE** : Specifies the maximum number of the serial with 38 digits or less. **MAXVALUE** must be smaller than or equal to the initial value and greater than the minimum value.
- **NOMAXVALUE** : (10)^{37} is set automatically as a maximum value for the ascending serial; -1 for the descending serial.
- **CYCLE** : Specifies that the serial will be generated continuously after reaching the maximum or minimum value. If the ascending serial reaches the maximum value, the minimum value is generated as the next value. If the descending serial reaches the minimum value, the maximum value is generated as the next value.
- **NOCYCLE** : Specifies that the serial will not be generated any more after reaching the maximum or minimum value. The default is **NOCYCLE**.
- **CACHE** : Saves as many serials as the number specified by integer in the cache to improve the performance of the serials and fetches a serial value when one is requested. The "integer" after the **CACHE** keyword cannot be omitted. If a number equal to or smaller than 1 is specified, the serial cache is not applied.
- **NOCACHE** : It does not use the serial cache feature. The serial value is updated every time and a new serial value is fetched from the disk upon each request.

Caution In CUBRID 2008 R1.x version, the serial value can be modified by updating the db_serial talbe, a system catalog. However, in CUBRID 2008 R2.0 version or above, the modification of the db_serial table is not allowed but use of the **ALTER SERIAL** statement is allowed. Therefore,if an **ALTER SERIAL** statement is included in the data exported (unloaddb) from CUBRID 2008 R2.0 or above, it is not allowed to import (loaddb) the data in CUBRID 2008 R1.x or below.

Example

```
--altering serial by changing start and incremental values
ALTER SERIAL order_no START WITH 100 INCREMENT BY 2;

--altering serial to operate in cache mode
ALTER SERIAL order_no CACHE 5;

--altering serial to operate in common mode
ALTER SERIAL order_no NOCACHE;
```

DROP SERIAL

Description

With the **DROP SERIAL** statement, you can drop a serial object from the database.

Syntax

```
DROP SERIAL serial_identifier
```

- *serial_identifier* : Specifies the name of the serial to be dropped.

Example

The following is an example of dropping the *order_no* serial.

```
DROP SERIAL order_no;
```

Use SERIAL

Description

You can access and update a serial by serial name and a reserved word pair.

Syntax

```
serial_identifier.CURRENT_VALUE
serial_identifier.NEXT_VALUE
```

- *serial_identifier*.**CURRENT_VALUE** : Returns the current serial value.
- *serial_identifier*.**NEXT_VALUE** : Increments the serial value and returns the result.

Example

The following is an example to create a table athlete_idx where athlete numbers and names are stored and to create the instances by using a serial order_no.

```
CREATE TABLE athlete_idx( code INT, name VARCHAR(40) );
INSERT INTO athlete_idx VALUES (order_no.NEXT_VALUE, 'Park');
INSERT INTO athlete_idx VALUES (order_no.NEXT_VALUE, 'Kim');
INSERT INTO athlete_idx VALUES (order_no.NEXT_VALUE, 'Choo');
INSERT INTO athlete_idx VALUES (order_no.NEXT_VALUE, 'Lee');SELECT * FROM athlete_idx;

=== <Result of SELECT Command in Line 1> ===

         code  name
===================================
        10000  'Park'
        10002  'Kim'
        10004  'Choo'
        10006  'Lee'
```

Caution

- When you use a serial for the first time after creating it, **NEXT_VALUE** returns the initial value. Subsequently, the sum of the current value and the increment are returned.

Operators and Functions

Logical Operators

Description

For the logical operator, predicates are always specified as the operand. **TRUE**, **FALSE** or **NULL** is returned as the result of the operation. The following table shows the logic operators supported by CUBRID.

Logical Operators Supported by CUBRID

Logical Operator	Description	Condition
AND, **&&**	If all operands are **TRUE**, it returns **TRUE**.	a **AND** b
OR, ||	If none of operands is **NULL** and one or more operand is **TRUE**, it returns **TRUE**. If **pipes_as_concat** is no that is a parameter related to SQL statements, a double pipe symbol can be used as **OR** operator.	a **OR** b
XOR	If none of operand is **NULL** and each of operand has a different value, it returns **TRUE**.	a **XOR** b
NOT, **!**	A unary operator. If a operand is **FALSE**, it returns **TRUE**. If it is **TRUE**, returns **FALSE**.	**NOT** a

True Table of Logical Operators

a	b	a AND b	a OR b	NOT a	a XOR b
TRUE	**TRUE**	TRUE	TRUE	FALSE	FALSE
TRUE	**FALSE**	FALSE	TRUE	FALSE	TRUE
TRUE	**NULL**	NULL	TRUE	FALSE	NULL
FALSE	**FALSE**	FALSE	FALSE	TRUE	FALSE
FALSE	**NULL**	FALSE	NULL	TRUE	NULL

Comparison Operators

Description

The comparison operator compares the operand on the left and on the right, and returns **TRUE**(1) or **FALSE**(0). Operands of the comparison operation must be of the same data type. Therefore, implicit type casting by the system or implicit type casting by the user is required.

The following table shows the comparison operators supported by CUBRID and their return values.

Comparison Operators Supported by CUBRID

Comparison Operator	Description	Predicate	Return Value
=	A general equal sign. It compares whether the values of the left and right operands are the same. Returns **NULL** if one or more operand is NULL.	1=>2 1=NULL	0 NULL
<=>	A NULL-safe equal sign. It compares whether the values of the left and right operands are the same including **NULL**. Returns 1 if both operands are **NULL**.	1<==>2 1<=> NULL	0 0
<>, !=	The value of left operand is not equal to that of right operand. If any operand value is **NULL**, **NULL** is returned.	1<>2	1

>	The value of left operand is greater than that of right operand. If any operand value is **NULL**, **NULL** is returned.	1>2	0
<	The value of left operand is less than that of right operand. If any operand value is **NULL**, **NULL** is returned.	1<2	1
>=	The value of left operand is greater than or equal to that of right operand. If any operand value is **NULL**, **NULL** is returned.	1>=2	0
<=	The value of left operand is less than or equal to that of right operand. If any operand value is **NULL**, **NULL** is returned.	1<=2	1
IS *boolean_value*	Compares whether the value of the left operand is the same as boolean value of the right. The boolean value may be **TRUE**, **FALSE** (or **NULL**).	1 IS FALSE	0
IN NOT *boolean_value*	Compares whether the value of the left operand is the same as boolean value of the right. The boolean value may be **TRUE**, **FALSE** (or **NULL**).	1 IS NOT FALSE	1

Syntax 1

```
expression  comparison_operator  expression

expression :
• bit string
• character string
• numeric value
• date-time value
• collection value
• NULL

comparison_operator :
=
| <=>
| <>
| !=
| >
| <
| >=
| <=
```

Syntax 2

```
expression IS [NOT] boolean_value

expression :
• bit string
• character string
• numeric value
• date-time value
• collection value
• NULL

boolean_value :
< UNKNOWN | NULL>
| TRUE
| FALSE
```

- *expression* : Declares an expression to be compared.
- *bit string* : A Boolean operation can be performed on bit strings, and all comparison operators can be used for comparison between bit strings. If you compare two expressions with different lengths, 0s are padded at the end of the shorter one.
- *character string* : When compared by a comparison operator, two character strings must have the same character sets. The comparison is determined by the collation sequence of the character code set. If you compare two

character strings with different lengths, blanks are padded at the end of the shorter one before comparison so that they have the same length.

- *numeric value* : The Boolean operator can be performed for all numeric values and any types of comparison operator can be used. When two different numeric types are compared, the system implicitly performs type casting. For example, when an **INTEGER** value is compared with a **DECIMAL** value, the system first casts **INTEGER** to **DECIMAL** before it performs comparison. When you compare a **FLOAT** value, you must specify the range instead of an exact value because the processing of **FLOAT** is dependent on the system.
- *date-time value* : If two date-time values with the same type are compared, the order is determined in time order. That is, when comparing two date-time values, the earlier date is considered to be smaller than the later date. You cannot compare date-time values with different type by using a comparison operator; therefore, you must explicitly convert it. However, comparison operation can be performed between DATE, TIMESTAMP, and DATETIME because they are implicitly converted.
- *collection value* : When comparing two sequences each element of the two sequences is compared in the order that is specified at the time of sequence creation. Comparison between sets or multisets is overloaded by an appropriate operator. You can perform comparison operations on sets, multisets, lists or sequence sets by using a containment operator explained later in this chapter. For more information, see Containment Operators.
- **NULL** : The **NULL** value is not included in the value range of any data type. Therefore, comparison between **NULL** values is only allowed to determine if the given value is **NULL** or not. An implicit type cast does not take place when a **NULL** value is assigned to a different data type. For example, when an attribute of **INTEGER** type has a **NULL** and is compared with a floating point type, the **NULL** value is not coerced to **FLOAT** before comparison is made. A comparison operation on the **NULL** value does not return a result.

Example

```
EVALUATE (1 <> 0); -- 1 is outputted because it is TRUE.
EVALUATE (1 != 0); -- 1 is outputted because it is TRUE.
EVALUATE (0.01 = '0.01'); -- An error occurs because a numeric data type is compared with
a character string type.
EVALUATE (1 = NULL); -- NULL is outputted.
EVALUATE (1 <=> NULL); -- 0 is outputted because it is FALSE.
EVALUATE (1.000 = 1); -- 1 is outputted because it is TRUE.
EVALUATE ('cubrid' = 'CUBRID'); -- 0 is outputted because it is case sensitive.
EVALUATE ('cubrid' = 'cubrid'); -- 1 is outputted because it is TRUE.
EVALUATE (SYSTIMESTAMP = CAST(SYSDATETIME AS TIMESTAMP)); -- 1 is outputted after casting
the type explicitly and then performing comparison operator.
EVALUATE (SYSTIMESTAMP = SYSDATETIME)); 0 is outputted after casting the type implicitly
and then performing comparison operator.
EVALUATE (SYSTIMESTAMP <> NULL); -- NULL is returned without performing comparison
operator.
EVALUATE (SYSTIMESTAMP IS NOT NULL); -- 1 is returned because it is no NULL.
```

Arithmetic Operators

Arithmetic Operators

Description

For arithmetic operators, there are binary operators for addition, subtraction, multiplication, or division, and unary operators to represent whether the number is positive or negative. The unary operators to represent the numbers' positive/negative status have higher priority over the binary operators.

Arithmetic Operators Supported by CUBRID

Arithmetic Operator	Description	Operator	Return Value
+	Addition	1+2	3
-	Subtraction	1-2	-1
*	Multiplication	1*2	2
/	Division. Returns quotient.	1/2.0	0.500000000
DIV	Division. Returns quotient.	1 DIV 2	0

%, **MOD**	Division. Returns quotient. An operator must be an integer type, and it always returns integer. If an operand is real number, the **MOD** function can be used.	1 % 2 1 MOD 2	1

Syntax

```
expression  mathematical_operator  expression
expression :
• bit string
• character string
• numeric value
• date-time value
• collection value
• NULL

mathematical_operator :
• set_arithmetic_operator
• arithmetic_operator

arithmetic_operator :
• +
• -
• *
• /, DIV
• %, MOD

set_arithmetic_operator :
• UNION                   (Union)
• DIFFERENCE         (Difference)
• INTERSECT | INTERSECTION   (Intersection)
```

- *expression* : Declares the mathematical operation to be calculated.
- *mathematical_operator* : A operator that performs an operation the arithmetic and the set operators are applicable.
- *set_arithmetic_operator* : A set arithmetic operator that performs operations such as union, difference and intersection on collection type operands.
- *arithmetic_operator* : An operator to perform the four fundamental arithmetic operations.

Arithmetic Operations and Type Casting of Numeric Data Types

Description

All numeric data types can be used for arithmetic operations. The result type of the operation differs depending on the data types of the operands and the type of the operation. The following table shows the result data types of addition/subtraction/multiplication for each operand type.

Result Data Type by Operand Type

	INT	**NUMERIC**	**FLOAT**	**DOUBLE**	**MONETARY**
INT	**INT** (**BIGINT**)	**NUMERIC**	**FLOAT**	**DOUBLE**	**MONETARY**
NUMERIC	**NUMERIC**	**NUMERIC** (*p* and *s* are also converted)	**DOUBLE**	**DOUBLE**	**MONETARY**
FLOAT	**FLOAT**	**DOUBLE**	**FLOAT**	**DOUBLE**	**MONETARY**
DOUBLE	**DOUBLE**	**DOUBLE**	**DOUBLE**	**DOUBLE**	**MONETARY**
MONETARY	**MONETARY**	**MONETARY**	**MONETARY**	**MONETARY**	**MONETARY**

Note that the result type of the operation does not change if all operands are of the same data type but type casting occurs exceptionally in division operations. An error occurs when a denominator, i.e. a divisor, is 0.

If one of the operands is a **MONETARY** type, all operation results are cast to **MONETARY** type because a **MONETARY** type uses the same operation methods as the DOUBLE type.

The following table shows the total number of digits (*p*) and the number of digits after the decimal point (*s*) of the operation results when all operands are of the **NUMERIC** type.

Result of NUMERIC Type Operation

Operation	Maximum Precision	Maximum Scale
N(p1, s1) + N(p2, s2)	max(p1-s1, p2-s2)+max(s1, s2) +1	max(s1, s2)
N(p1, s1) - N(p2, s2)	max(p1-s1, p2-s2)+max(s1, s2)	max(s1, s2)
N(p1, s1) * N(p2, s2)	p1+p2+1	s1+s2
N(p1, s1) / N(p2, s2)	Let Pt = p1+max(s1, s2) + s2 - s1 when s2 > 0 and Pt = p1 in other cases; St = s1 when s1 > s2 and s2 in other cases; the number of decimal places is min(9-St, 38-Pt) + St when St < 9 and St in other cases.	

Example

```
--int * int
SELECT 123*123;
=============
        15129

-- int * int returns overflow error
SELECT (1234567890123*1234567890123);

-- int * numeric returns numeric type
SELECT (1234567890123*CAST(1234567890123 AS NUMERIC(15,2)));
======================
  1524157875322755800955129.00

-- int * float returns float type
SELECT (1234567890123*CAST(1234567890123 AS FLOAT));
===============================================
                                  1.524158e+024

-- int * double returns double type
SELECT (1234567890123*CAST(1234567890123 AS DOUBLE));
================================================
                          1.524157875322756e+024

-- numeric * numeric returns numeric type
SELECT (CAST(1234567890123 AS NUMERIC(15,2))*CAST(1234567890123 AS NUMERIC(15,2)));
======================
  1524157875322755800955129.0000

-- numeric * float returns double type
SELECT (CAST(1234567890123 AS NUMERIC(15,2))*CAST(1234567890123 AS FLOAT));
=======================================================================
                                                 1.524157954716582e+024

-- numeric * double returns double type
SELECT (CAST(1234567890123 AS NUMERIC(15,2))*CAST(1234567890123 AS DOUBLE));
========================================================================
                                                  1.524157875322756e+024

-- float * float returns float type
SELECT (CAST(1234567890123 AS FLOAT)*CAST(1234567890123 AS FLOAT));
===============================================================
                                                  1.524158e+024
-- float * double returns float type
SELECT (CAST(1234567890123 AS FLOAT)*CAST(1234567890123 AS DOUBLE));
================================================================
                                          1.524157954716582e+024

-- double * double returns float type
SELECT (CAST(1234567890123 AS DOUBLE)*CAST(1234567890123 AS DOUBLE));
=================================================================
                                           1.524157875322756e+024

-- int / int returns int type without type conversion or rounding
```

```
SELECT 100100/100000;
===============
              1

-- int / int returns int type without type conversion or rounding
SELECT 100100/200200;
===============
              0

-- int / zero returns error
SELECT 100100/(100100-100100);
ERROR: Attempt to divide by zero.
```

Arithmetic Operations and Type Casting of DATE/TIME Data Types

Description

If all operands are date/time type, only a subtraction operation is allowed and its return value is **INT**. Note that the unit of the operation differs depending on the types of the operands. Both addition and subtraction operations are allowed in case of date/time and integer types In this case, operation units and return values are date/time data type.

The following table shows operations allowed for each operand type, and their result types.

Allowable Operation and Result Data Type by Operand Type

	TIME **(in seconds)**	**DATE** **(in day)**	**TIMESTAMP** **(in seconds)**	**DATETIME** **(in milliseconds)**	**INT**
TIME	A subtraction is allowed. **INT**	X	X	X	An addition and a subtraction are allowed. **INT**
DATE	X	A subtraction is allowed. **INT**	A subtraction is allowed. **INT**	A subtraction is allowed. **INT**	An addition and a subtraction are allowed. **DATE**
TIMESTAMP	X	A subtraction is allowed. **INT**	A subtraction is allowed. **INT**	A subtraction is allowed. **INT**	An addition and a subtraction are allowed. **TIMESTAMP**
DATETIME	X	A subtraction is allowed. **INT**	A subtraction is allowed. **INT**	A subtraction is allowed. **INT**	An addition and a subtraction are allowed. **DATETIME**
INT	An addition and a subtraction are allowed. **TIME**	An addition and a subtraction are allowed. **DATE**	An addition and a subtraction are allowed. **TIMESTAMP**	An addition and a subtraction are allowed. **DATETIME**	All operations are allowed.

Remark

If any of the date/time arguments contains **NULL**, **NULL** is returned.

Example

```
-- initial systimestamp value
SELECT SYSDATETIME;
===============================
  07:09:52.115 PM 01/14/2010

-- time type + 10(seconds) returns time type
SELECT (CAST (SYSDATETIME AS TIME) + 10);
====================================
  07:10:02 PM
```

```
-- date type + 10 (days) returns date type
SELECT (CAST (SYSDATETIME AS DATE) + 10);
====================================
  01/24/2010

-- timestamp type + 10(seconds) returns timestamp type
SELECT (CAST (SYSDATETIME AS TIMESTAMP) + 10);
=========================================
  07:10:02 PM 01/14/2010

-- systimestamp type + 10(milliseconds) returns systimestamp type
SELECT (SYSDATETIME  + 10);
===============================
  07:09:52.125 PM 01/14/2010

SELECT DATETIME '09/01/2009 03:30:30.001 pm'- TIMESTAMP '08/31/2009 03:30:30 pm';
=======================================
  86400001

SELECT TIMESTAMP '09/01/2009 03:30:30 pm'- TIMESTAMP '08/31/2009 03:30:30 pm';
=======================================
  86400
```

Set Operators

Set Arithmetic Operators

Set Arithmetic Operators

To evaluate set operations such as union, difference or intersection for **SET**, **MULTISET** or **LIST (SEQUENCE)** types, you can use +, - or * operators respectively.

The following table shows a summary of how to use these operators.

Result Data Type by Operand Type

	SET	**MULTISET**	**LIST (=SEQUENCE)**
SET	+, -, * : **SET**	+, -, * : **MULTISET**	+, -, * : **MULTISET**
MULTISET	+, -, * : **MULTISET**	+, -, * : **MULTISET**	+, -, * : **MULTISET**
LIST (=SEQUENCE)	+ : **MULTISET** - : **MULTISET** * : **MULTISET**	+ : **MULTISET** - : **MULTISET** * : **MULTISET**	+ : **LIST** - : **MULTISET** * : **MULTISET**

Syntax

```
value_expression set_arithmetic_operator value_expression

value_expression :
• collection value
• NULL

set_arithmetic_operator :
• + (union)
• - (difference)
• * (intersection)
```

Example

```
SELECT ((CAST ({3,3,3,2,2,1} AS SET))+(CAST ({4,3,3,2} AS MULTISET)))
FROM db_root;
======================
  {1, 2, 2, 3, 3, 3, 4}

SELECT ((CAST ({3,3,3,2,2,1} AS MULTISET))+(CAST ({4,3,3,2} AS MULTISET)))
FROM db_root;
```

```
======================
  {1, 2, 2, 2, 3, 3, 3, 3, 3, 4}

SELECT ((CAST ({3,3,3,2,2,1} AS LIST))+(CAST ({4,3,3,2} AS MULTISET)))
FROM db_root;
======================
  {1, 2, 2, 2, 3, 3, 3, 3, 3, 4}

SELECT ((CAST ({3,3,3,2,2,1} AS SET))-(CAST ({4,3,3,2} AS MULTISET)))
FROM db_root;
======================
  {1}

SELECT ((CAST ({3,3,3,2,2,1} AS MULTISET))-(CAST ({4,3,3,2} AS MULTISET)))
FROM db_root;
======================
  {1, 2, 3}

SELECT ((CAST ({3,3,3,2,2,1} AS LIST))-(CAST ({4,3,3,2} AS MULTISET)))
FROM db_root;
======================
  {1, 2, 3}

SELECT ((CAST ({3,3,3,2,2,1} AS SET))*(CAST ({4,3,3,2} AS MULTISET)))
FROM db_root;
======================
  {2, 3}

SELECT ((CAST ({3,3,3,2,2,1} AS MULTISET))*(CAST ({4,3,3,2} AS MULTISET)))
FROM db_root;
======================
  {2, 3, 3}

SELECT ((CAST ({3,3,3,2,2,1} AS LIST))*(CAST ({4,3,3,2} AS MULTISET)))
FROM db_root;
======================
{2, 3, 3}
```

Assigning Collection Value to Variable

For a collection value to be assigned to a variable, the outer query must return a single row as the result. The following is an example of assigning a collection value to a variable. The outer query must return only a single row as follows:

```
SELECT SET(SELECT name
FROM people
WHERE ssn in {'1234', '5678'})
TO :"names"
FROM TABLE people;
```

Statement Set Operators

Description

Statement set operators are used to get union, difference or intersection on the result of more than one query statement specified as an operand. Note that the data types of the data to be retrieved from the target tables of the two query statements must be identical or implicitly castable.

The following table shows statement set operators supported by CUBRID and their examples.

Statement Set Operators Supported by CUBRID

Statement Set Operator	Description	Note
UNION	Union Duplicates are not allowed.	Outputs all instance results containing duplicates with **UNION ALL**
DIFFERENCE	Difference Duplicates are not allowed.	Same as the **EXCEPT** operator Outputs all instance results containing duplicates with **DIFFERENCE ALL**

INTERSECTION	Intersection Duplicates are not allowed.	Same as the **INTERSECTION** operator Outputs all instance results containing duplicates with **INTERSECTION ALL**

Syntax

```
query_term statement_set_operator[qualifier] query_term
[{statement_set_operator[qualifier] query_term}];

query_term :
• query_specification
• subquery

qualifier :
• DISTINCT or DISTINCTROW (A returned instance is a distinct value.)
• UNIQUE (A returned instance is a unique value.)
• ALL (All instances are returned. Duplicates are allowed.)

statement_set_operator :
• UNION (union)
• DIFFERENCE (difference)
• INTERSECTION | INTERSECT (intersection)
```

Example

```
CREATE TABLE nojoin_tbl_1 (ID INT, Name VARCHAR(32));

INSERT INTO nojoin_tbl_1 VALUES (1,'Kim');
INSERT INTO nojoin_tbl_1 VALUES (2,'Moy');
INSERT INTO nojoin_tbl_1 VALUES (3,'Jonas');
INSERT INTO nojoin_tbl_1 VALUES (4,'Smith');
INSERT INTO nojoin_tbl_1 VALUES (5,'Kim');
INSERT INTO nojoin_tbl_1 VALUES (6,'Smith');
INSERT INTO nojoin_tbl_1 VALUES (7,'Brown');

CREATE TABLE nojoin_tbl_2 (id INT, Name VARCHAR(32));

INSERT INTO nojoin_tbl_2 VALUES (5,'Kim');
INSERT INTO nojoin_tbl_2 VALUES (6,'Smith');
INSERT INTO nojoin_tbl_2 VALUES (7,'Brown');
INSERT INTO nojoin_tbl_2 VALUES (8,'Lin');
INSERT INTO nojoin_tbl_2 VALUES (9,'Edwin');
INSERT INTO nojoin_tbl_2 VALUES (10,'Edwin');

--Using UNION to get only distict rows
SELECT id, name FROM nojoin_tbl_1
UNION
SELECT id,name FROM nojoin_tbl_2;

           id  name
===================================
            1  'Kim'
            2  'Moy'
            3  'Jonas'
            4  'Smith'
            5  'Kim'
            6  'Smith'
            7  'Brown'
            8  'Lin'
            9  'Edwin'
           10  'Edwin'

--Using UNION ALL not eliminating duplicate selected rows
SELECT id, name FROM nojoin_tbl_1
UNION ALL
SELECT id,name FROM nojoin_tbl_2;

           id  name
===================================
            1  'Kim'
            2  'Moy'
            3  'Jonas'
            4  'Smith'
            5  'Kim'
            6  'Smith'
            7  'Brown'
            5  'Kim'
            6  'Smith'
            7  'Brown'
            8  'Lin'
            9  'Edwin'
           10  'Edwin'

--Using DEFFERENCE to get only rows returned by the first query but not by the second
SELECT id, name FROM nojoin_tbl_1
DIFFERENCE
SELECT id,name FROM nojoin_tbl_2;

           id  name
===================================
            1  'Kim'
            2  'Moy'
            3  'Jonas'
            4  'Smith'

--Using INTERSECTION to get only those rows returned by both queries
SELECT id, name FROM nojoin_tbl_1
INTERSECT
SELECT id,name FROM nojoin_tbl_2;

           id  name
===================================
            5  'Kim'
            6  'Smith'
            7  'Brown'
```

Containment Operators

Containment Operators

Description

Containment operators are used to check the containment relationship by performing comparison operation on operands of the set data type. Set data types or subqueries can be specified as operands. The operation returns TRUE or FALSE if there is a containment relationship between the two operands of identical/different/subset/proper subset.

The description and returned values about the containment operators supported by CUBRID are as follows:

Containment Operators Supported by CUBRID

Containment Operator	Description	Predicates	Return Value
A **SETEQ** B	A = B Elements in A and B are same each other.	{1,2} SETEQ {1,2,2}	0
A **SETNEQ** B	A ≠ B Elements in A and B are not same each other.	{1,2} SETNEQ {1,2,3}	1
A **SUPERSET** B	A ⊃ B B is a proper subset of A.	{1,2} SUPERSET {1,2,3}	0
A **SUBSET** B	A ⊂ B A is a proper subset of B.	{1,2} SUBSET {1,2,3}	1
A **SUPERSETEQ** B	A ⊇ B B is a subset of A.	{1,2} SUPERSETEQ {1,2,3}	0

A **SUBSETEQ** B	A ⊆ B A is a subset of B.	{1,2} SUBSETEQ {1,2,3}	1

The following table shows than possibility of operation by operand and type conversion if a containment operator is used.

Possibility of Operation by Operand

	SET	**MULTISET**	**LIST(=SEQUENCE)**
SET	Operation possible	Operation possible	Operation possible
MULTISET	Operation possible	Operation possible	Operation possible (**LIST** is converted into **MULTISET**)
LIST(=SEQUENCE)	Operation possible	Operation possible (**LIST** is converted into **MULTISET**)	Some operation possible (**SETEQ**, **SETNEQ**) Error occurs for the rest of operators.

Syntax

```
collection_operand  containment_operator  collection_operand

collection_operand:
• set
• multiset
• sequence(or list)
• subquery
• NULL

containment_operator:
• SETEQ
• SETNEQ
• SUPERSET
• SUBSET
• SUPERSETEQ
• SUBSETEQ
```

- *collection_operand* : This expression that can be specified as an operand is a single SET-valued attribute, an arithmetic expression containing a SET operator or a SET value enclosed in braces. If the type is not specified, the SET value enclosed in braces is treated as a **LIST** type by default.
- Subqueries can be specified as operands. If a column which is not a **SET** type is queried, a SET data type keyword is required for the subquery (e.g. **SET**(*subquery*)). The column retrieved by a subquery must return a single set so that it can be compared with the set of the other operands.
- If the element type is an object, the OIDs, not its contents, are compared. For example, two objects with different OIDs are considered to be different even though they have the same attribute values.
  - **NULL** : Any of operands to be compared is **NULL**, **NULL** is returned.

```
--empty set is a subset of any set
EVALUATE ({} SUBSETEQ (CAST ({3,1,2} AS SET)));
=============
            1

--operation between set type and null returns null
EVALUATE ((CAST ({3,1,2} AS SET)) SUBSETEQ NULL);
=============
            NULL

--{1,2,3} seteq {1,2,3} returns true
EVALUATE ((CAST ({3,1,2} AS SET)) SETEQ (CAST ({1,2,3,3} AS SET)));
=============
            1

--{1,2,3} seteq {1,2,3,3} returns false
EVALUATE ((CAST ({3,1,2} AS SET)) SETEQ (CAST ({1,2,3,3} AS MULTISET)));
=============
            0

--{1,2,3} setneq {1,2,3,3} returns true
EVALUATE ((CAST ({3,1,2} AS SET)) SETNEQ (CAST ({1,2,3,3} AS MULTISET)));
=============
            1

--{1,2,3} subseteq {1,2,3,4} returns true
EVALUATE ((CAST ({3,1,2} AS SET)) SUBSETEQ (CAST ({1,2,4,4,3} AS SET)));
=============
            1

--{1,2,3} subseteq {1,2,3,4,4} returns true
EVALUATE ((CAST ({3,1,2} AS SET)) SUBSETEQ (CAST ({1,2,4,4,3} AS MULTISET)));
=============
            1

--{1,2,3} subseteq {1,2,4,4,3} returns true
EVALUATE ((CAST ({3,1,2} AS SET)) SUBSETEQ (CAST ({1,2,4,4,3} AS LIST)));
=============
            0

--{1,2,3} subseteq {1,2,3,4,4} returns true
EVALUATE ((CAST ({3,1,2} AS SET)) SUBSETEQ (CAST ({1,2,3,4,4} AS LIST)));
=============
            1

--{3,1,2} seteq {3,1,2} returns true
EVALUATE ((CAST ({3,1,2} AS LIST)) SETEQ (CAST ({3,1,2} AS LIST)));
=============
            1
--error occurs because LIST subseteq LIST is not supported
EVALUATE ((CAST ({3,1,2} AS LIST)) SUBSETEQ (CAST ({3,1,2} AS LIST)));
=============
        error
```

SETEQ Operator

Description

The **SETEQ** operator returns **TRUE** if the first operand is the same as the second one. It can perform comparison operator for all collection data type.

Syntax

```
collection_operand SETEQ collection_operand
```

Example

```
--creating a table with SET type address column and LIST type zip_code column

CREATE TABLE contain_tbl (id int primary key, name char(10), address SET varchar(20),
zip_code LIST int)
INSERT INTO contain_tbl VALUES(1, 'Kim', {'country', 'state'},{1, 2, 3})
INSERT INTO contain_tbl VALUES(2, 'Moy', {'country', 'state'},{3, 2, 1})
INSERT INTO contain_tbl VALUES(3, 'Jones', {'country', 'state', 'city'},{1,2,3,4})
INSERT INTO contain_tbl VALUES(4, 'Smith', {'country', 'state', 'city',
'street'},{1,2,3,4})
INSERT INTO contain_tbl VALUES(5, 'Kim', {'country', 'state', 'city', 'street'},{1,2,3,4})
INSERT INTO contain_tbl VALUES(6, 'Smith', {'country', 'state', 'city',
'street'},{1,2,3,5})
INSERT INTO contain_tbl VALUES(7, 'Brown', {'country', 'state', 'city', 'street'},{})

--selecting rows when two collection_operands are same in the WEHRE clause
SELECT id, name, address, zip_code FROM contain_tbl WHERE address SETEQ {'country','state',
'city'}

=== < Result of SELECT Command in Line 1> ===

                      id    name                                  address
            zip_code
```

```
===============================================================================
                        3   'Jones         '                   {'city', 'country',
'state'}   {1, 2, 3, 4}

1 row selected.

--selecting rows when two collection_operands are same in the WEHRE clause
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SETEQ {1,2,3}

=== < Result of SELECT Command in Line 1> ===

                      id   name                                   address
            zip_code
===============================================================================
                        1   'Kim              '                   {'country', 'state'}   {1,
2, 3}

1 rows selected.
```

SETNEQ Operator

Description

The **SETNEQ** operator returns **TRUE(1)** if a first operand is different from a second operand. A comparable operation can be performed for all collection data types.

Syntax

```
collection_operand SETNEQ collection_operand
```

Example

```
--selecting rows when two collection_operands are not same in the WEHRE clause
SELECT id, name, address, zip_code FROM contain_tbl WHERE address SETNEQ
{'country','state', 'city'};

=== <Result of SELECT Command in Line 1> ===

           id  name                  address               zip_code
===============================================================================
            1  'Kim       '          {'country', 'state'}  {1, 2, 3}
            2  'Moy       '          {'country', 'state'}  {3, 2, 1}
            4  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            5  'Kim       '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            6  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 5}
            7  'Brown     '          {'city', 'country', 'state', 'street'}  {}

6 rows selected.

--selecting rows when two collection_operands are not same in the WEHRE clause
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SETNEQ {1,2,3};

=== <Result of SELECT Command in Line 1> ===

           id  name                  address               zip_code
===============================================================================
            2  'Moy       '          {'country', 'state'}  {3, 2, 1}
            3  'Jones     '          {'city', 'country', 'state'}  {1, 2, 3, 4}
            4  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            5  'Kim       '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            6  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 5}
            7  'Brown     '          {'city', 'country', 'state', 'street'}  {}

6 rows selected.
```

SUPERSET Operator

Description

The **SUPERSET** operator returns **TRUE(1)** when a second operand is a proper subset of a first operand; that is, the first one is larger than the second one. If two operands are identical, **FALSE(0)** is returned. Note that **SUPERSET** is not supported if all operands are **LIST** type.

Syntax

```
collection_operand SUPERSET collection_operand
```

Example

```
--selecting rows when the first operand is a superset of the second operand and they are
not same
SELECT id, name, address, zip_code FROM contain_tbl WHERE address SUPERSET
{'country','state','city'};

=== <Result of SELECT Command in Line 1> ===

           id  name                  address               zip_code
===============================================================================
            4  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            5  'Kim       '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            6  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 5}
            7  'Brown     '          {'city', 'country', 'state', 'street'}  {}

4 rows selected.

--SUPERSET operator cannot be used for comparison between LIST and LIST type values
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SUPERSET {1,2,3};

ERROR: ' superset ' operator is not defined on types sequence and sequence.

--Comparing operands with a SUPERSET operator after casting LIST type as SET type
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SUPERSET (CAST ({1,2,3}
AS SET));

=== <Result of SELECT Command in Line 1> ===

           id  name                  address               zip_code
===============================================================================
            3  'Jones     '          {'city', 'country', 'state'}  {1, 2, 3, 4}
            4  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            5  'Kim       '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            6  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 5}

4 rows selected.
```

SUPERSETEQ Operator

Description

The **SUPERSETEQ** operator returns **TRUE(1)** when a second operand is a subset of a first operand; that is, the first one is identical to or larger than the second one. Note that **SUPERSETEQ** is not supported if an operand is **LIST** type.

Syntax

```
collection_operand SUPERSETEQ collection_operand
```

Example

```
--selecting rows when the first operand is a superset of the second operand
SELECT id, name, address, zip_code FROM contain_tbl WHERE address SUPERSETEQ
{'country','state','city'};

=== <Result of SELECT Command in Line 1> ===
           id  name                  address               zip_code
================================================================================
            3  'Jones     '          {'city', 'country', 'state'}  {1, 2, 3, 4}
            4  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            5  'Kim       '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            6  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 5}
            7  'Brown     '          {'city', 'country', 'state', 'street'}  {}

5 rows selected.

--SUPERSETEQ operator cannot be used for comparison between LIST and LIST type values
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SUPERSETEQ {1,2,3};

ERROR: ' superseteq ' operator is not defined on types sequence and sequence.

--Comparing operands with a SUPERSETEQ operator after casting LIST type as SET type
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SUPERSETEQ (CAST
({1,2,3} AS SET));


=== <Result of SELECT Command in Line 1> ===

           id  name                  address               zip_code
================================================================================
            1  'Kim       '          {'country', 'state'}  {1, 2, 3}
            3  'Jones     '          {'city', 'country', 'state'}  {1, 2, 3, 4}
            4  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            5  'Kim       '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 4}
            6  'Smith     '          {'city', 'country', 'state', 'street'}  {1, 2, 3, 5}

5 rows selected.
```

SUBSET Operator

Description

The **SUBSET** operator returns **TRUE**(1) if the second operand contains all elements of the first operand. If the first and the second collection have the same elements, **FALSE**(0) is returned. Note that both operands are the **LIST** type, the **SUBSET** operation is not supported.

Syntax

```
collection_operand SUBSET collection_operand
```

Example

```
--selecting rows when the first operand is a subset of the second operand and they are not
same
SELECT id, name, address, zip_code FROM contain_tbl WHERE address SUBSET
{'country','state','city'};

=== <Result of SELECT Command in Line 1> ===

           id  name                  address               zip_code
================================================================================
            1  'Kim       '          {'country', 'state'}  {1, 2, 3}
            2  'Moy       '          {'country', 'state'}  {3, 2, 1}

2 rows selected.

--SUBSET operator cannot be used for comparison between LIST and LIST type values
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SUBSET {1,2,3};

ERROR: ' subset ' operator is not defined on types sequence and sequence.

--Comparing operands with a SUBSET operator after casting LIST type as SET type
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SUBSET (CAST ({1,2,3}
AS SET));

=== <Result of SELECT Command in Line 1> ===
           id  name                  address               zip_code
================================================================================
            7  'Brown     '          {'city', 'country', 'state', 'street'}  {}

1 rows selected.
```

SUBSETEQ Operator

Description

The **SUBSETEQ** operator returns **TRUE(1)** when a first operand is a subset of a second operand; that is, the second one is identical to or larger than the first one. Note that **SUBSETEQ** is not supported if an operand is **LIST** type.

Syntax

```
collection_operand SUBSETEQ collection_operand
```

Example

```
--selecting rows when the first operand is a subset of the second operand
SELECT id, name, address, zip_code FROM contain_tbl WHERE address SUBSETEQ
{'country','state','city'};

=== <Result of SELECT Command in Line 1> ===

           id  name                  address               zip_code
================================================================================
            1  'Kim       '          {'country', 'state'}  {1, 2, 3}
            2  'Moy       '          {'country', 'state'}  {3, 2, 1}
            3  'Jones     '          {'city', 'country', 'state'}  {1, 2, 3, 4}

3 rows selected.

--SUBSETEQ operator cannot be used for comparison between LIST and LIST type values
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SUBSETEQ {1,2,3};

ERROR: ' subseteq ' operator is not defined on types sequence and sequence.

--Comparing operands with a SUBSETEQ operator after casting LIST type as SET type
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SUBSETEQ (CAST ({1,2,3}
AS SET));

=== <Result of SELECT Command in Line 1> ===

           id  name                  address               zip_code
================================================================================
            1  'Kim       '          {'country', 'state'}  {1, 2, 3}
            7  'Brown     '          {'city', 'country', 'state', 'street'}  {}

2 rows selected.
```

BIT Functions and Operators

Bitwise Operator

A **Bitwise** operator performs operations in bits, and can be used in arithmetic operations. An integer type is specified as the operand and the **BIT** type cannot be specified. An integer of **BIGINT** type (64-bit integer) is returned as the result of the operation. If one or more operand is **NULL**, **NULL** is returned.

The following table shows the bitwise operators supported by CUBRID.

The bitwise operators supported by CUBRID

Bitwise operator	Description	Expression	Return Value
&	Performs AND operation in bits and returns a BIGINT integer.	17 & 3	1

|	Performs OR operation in bits and returns a BIGINT integer.	17 | 3	19
^	Performs XOR operation in bits and returns a BIGINT integer.	17 ^ 3	18
~	A unary operator. It performs complementary operation that reverses (INVERT) the bit order of the operand and returns a BIGINT integer.	~17	-18
<<	Performs the operation of moving bits of the left operand as far to the left as the value of the right operand, and returns a BIGINT integer.	17 << 3	136
>>	Performs the operation of moving bits of the left operand as far to the right as the value of the right operand, and returns a BIGINT integer.	17 >> 3	2

BIT_AND Function

Description

An aggregate function. It performs **AND** operations in bits on every bit of *expr*. The return value is a **BIGINT** type. If there is no row that satisfies the expression, **NULL** is returned.

Syntax

```
BIT_AND(expr)
```

- *expr* : An expression of integer type

Example

```
CREATE TABLE bit_tbl(id int);
INSERT INTO bit_tbl VALUES (1), (2), (3), (4), (5);
SELECT 1&3&5, BIT_AND(id) FROM bit_tbl WHERE id in(1,3,5);
;xr

=== <Result of SELECT Command in Line 1> ===

            1&3&5               bit_and(id)
============================================
                     1                    1
```

BIT_OR Function

Description

An aggregate function. It performs **OR** operations in bits on every bit of *expr*. The return value is a **BIGINT** type. If there is no row that satisfies the expression, **NULL** is returned.

Syntax

```
BIT_OR(expr)
```

- *expr* : An expression of integer type

Example

```
SELECT 1|3|5, BIT_OR(id) FROM bit_tbl WHERE id in(1,3,5);

=== <Result of SELECT Command in Line 1> ===

           1|3|5                   bit_or(id)
============================================
                     7                      7
```

BIT_XOR Function

Description

An aggregate function. It performs **XOR** operations in bits on every bit of *expr*. The return value is a **BIGINT** type. If there is no row that satisfies the expression, **NULL** is returned.

Syntax

```
BIT_XOR(expr)
```

- *expr* : An expression of integer type

Example

```
SELECT 1^2^3^, BIT_XOR(id) FROM bit_tbl WHERE id in(1,3,5);

=== <Result of SELECT Command in Line 1> ===

    `        1^3^5               bit_xor(id)
============================================
                     7                     7
```

BIT_COUNT Function

Description

The **BIT_COUNT** function returns the number of bits of *expr* that have been set to 1; it is not an aggregate function. The return value is a **BIGINT** type.

Syntax

```
BIT_COUNT (expr)
```

- *expr* : An expression of integer type

Example

```
SELECT BIT_COUNT(id) FROM bit_tbl WHERE id in(1,3,5);

=== <Result of SELECT Command in Line 1> ===

   bit_count(id)
================
               1
               2
               2
```

String Functions and Operators

Concatenation Operator

Description

A concatenation operator gets a character string or bit string data type as an operand and returns a concatenated string. The plus sign (+) and double pipe symbol (||) are provided as concatenation operators for character string data. If **NULL** is specified as an operand, a **NULL** value is returned.

If **pipes_as_concat** that is a parameter related to SQL statement is set to **no**, a double pipe (||) symbol is interpreted as an **OR** operator. Therefore, to concatenate, a plus operator (+) or the **CONCAT** function should be used.

Syntax

```
concat_operand1   +   concat_operand1
concat_operand2   ||   concat_operand2
```

```
concat_operand1 :
• bit string
• NULL

concat_operand2 :
• bit string
• character string
• NULL
```

- *concat_operand1* : Left string after concatenation. String or bit type can be specified.
- *concat_operand2* : Right string after concatenation. String or bit type can be specified.

Example

```
SELECT 'CUBRID' || ',' + '2008';
======================
  'CUBRID,2008'

SELECT 'cubrid' || ',' || B'0010' ||B'0000' ||B'0000' ||B'1000'FROM db_root;
======================
  'cubrid,2008'

SELECT ((EXTRACT(YEAR FROM SYS_TIMESTAMP))||(EXTRACT(MONTH FROM SYS_TIMESTAMP)));
======================
  '200812'

SELECT 'CUBRID' || ',' + NULL;
======================
  NULL
```

BIT_LENGTH Function

Description

The **BIT_LENGTH** function returns the length (bits) of a character string or bit string as an integer value. The return value of the **BIT_LENGTH** function may differ depending on the character set, because for the character string, the number of bytes taken up by a single character is different depending on the character set of the data input environment (e.g., EUC-KR: 2*8 bits). For details about character sets supported by CUBRID, see Definition and Characteristics.

Syntax

```
BIT_LENGTH ( string )

string :
• bit string
• character string
• NULL
```

- *string* : Specifies the character string or bit string whose number of bits is to be calculated. If this value is **NULL**, **NULL** is returned.

Example

```
SELECT BIT_LENGTH('');
=================
                0

SELECT BIT_LENGTH('CUBRID');
=======================
                     48

SELECT BIT_LENGTH('큐브리드');
=========================
                       64

SELECT BIT_LENGTH(B'010101010');
===========================
                          9
```

```
CREATE TABLE bit_length_tbl (char_1 CHAR, char_2 CHAR(5), varchar_1 VARCHAR, bit_var_1 BIT
VARYING);
INSERT INTO bit_length_tbl VALUES('', '', '', B''); --Length of empty string
INSERT INTO bit_length_tbl VALUES('a', 'a', 'a', B'010101010'); --English character
INSERT INTO bit_length_tbl VALUES(NULL, '큐', '큐', B'010101010'); --Korean character and
NULL
INSERT INTO bit_length_tbl VALUES(' ', ' 큐', ' 큐', B'010101010'); --Korean character and
space

SELECT BIT_LENGTH(char_1), BIT_LENGTH(char_2), BIT_LENGTH(varchar_1), BIT_LENGTH(bit_var_1)
FROM bit_length_tbl;

=== <Result of SELECT Command in Line 15> ===

=================================================================================
8                   40                        0                       0
8                   40                        8                       9
NULL                40                       16                       9
8                   40                       24                       9
```

CHAR_LENGTH, CHARACTER_LENGTH, LENGTHB, LENGTH Function

Description

CHAR_LENGTH, **LENGTHB**, and **LENGTH** are used interchangeably.

They return the length of a character string (byte) as an integer. The return value may be different depending on the character set (e.g., EUC-KR: 2 bites).

For details about the character sets supported by CUBRID, see Definition and Characteristics.

Syntax

```
CHAR_LENGTH( string )
CHARACTER_LENGTH( string )
LENGTHB( string )
LENGTH( string )

string :
• character string
• NULL
```

- *string* : Specifies the character string whose number of characters is to be calculated. If the character string is **NULL**, **NULL** is returned.

Remark

- The length of each space character that is included in a character string is one byte.
- For multi-byte strings, the length of a single character is calculated as 2 or 3 bytes depending on the character set of the data input environment.
- The length of empty quotes ('') to represent a space character is 0. Note that in a **CHAR**(*n*) type, the length of a space character is *n*, and it is specified as 1 if n is omitted.

Example

```
--character set is euc-kr for Korean characters
SELECT LENGTH('');
==================
                 0

SELECT LENGTH('CUBRID');
==================
                 6

SELECT LENGTH('큐브리드');
==================
                 8
```

```
CREATE TABLE length_tbl (char_1 CHAR, char_2 CHAR(5), varchar_1 VARCHAR, varchar_2
VARCHAR);
INSERT INTO length_tbl VALUES('', '', '', ''); --Length of empty string
INSERT INTO length_tbl VALUES('a', 'a', 'a', 'a'); --English character
INSERT INTO length_tbl VALUES(NULL, '큐', '큐', '큐'); --Korean character and NULL
INSERT INTO length_tbl VALUES(' ', ' 큐', ' 큐', ' 큐'); --Korean character and space

SELECT LENGTH(char_1), LENGTH(char_2), LENGTH(varchar_1), LENGTH(varchar_2) FROM
length_tbl;

=== <Result of SELECT Command in Line 1> ===

================================================================================

1                     5                        0             0
1                     5                        1             1
NULL                  5                        2             2
1                     5                        3             3
```

CHR Function

Description

The **CHR** function returns a character that corresponds to the return value of the expression specified as an argument. It returns 0 if it exceeds range of character code.

Syntax

```
CHR( number_operand )
```

- *number_operand* : Specifies an expression that returns a numeric value.

Example

```
SELECT CHR(68) || CHR(68-2);

=== <Result of SELECT Command in Line 1> ===

   chr(68)|| chr(68-2)
======================
  'DB'
```

CONCAT Function

Description

The **CONCAT** function has at least one argument specified for it and returns a string as a result of concatenating all argument values. The number of parameters that can be specified is unlimited. Automatic type casting takes place if a non-string type is specified as the argument. If any of the arguments is specified as **NULL**, **NULL** is returned.

If you want to insert separators between strings specified as arguments for concatenation, use the CONCAT_WS Function.

Syntax

```
CONCAT( string1, string2 [,string3 [, ... [, stringN]...]])

string :
• character string
• NULL
```

Example

```
SELECT CONCAT('CUBRID', '2008' , 'R3.0')

======================
'CUBRID2008R3.0'
```

```
--it returns null when null is specified for one of parameters
SELECT CONCAT('CUBRID', '2008' , 'R3.0', NULL)

======================
    NULL


--it converts number types and then returns concatenated strings
SELECT CONCAT(2008, 3.0)

======================
    '20083.0'
```

CONCAT_WS Function

Description

The **CONCAT_WS** function has at least two arguments specified for it. The function uses the first argument value as the separator and returns the result.

```
CONCAT_WS( string1, string2 [,string3 [, ... [, stringN]...]])

string :
• character string
• NULL
```

Example

```
SELECT CONCAT_WS(' ', 'CUBRID', '2008' , 'R3.0');
======================
  'CUBRID 2008 R3.0'

--it returns strings even if null is specified for one of parameters
SELECT CONCAT_WS(' ', 'CUBRID', '2008', NULL, 'R3.0');
======================
  'CUBRID 2008 R3.0'

--it converts number types and then returns concatenated strings with separator
SELECT CONCAT_WS(' ',2008, 3.0);
======================
  '2008 3.0'
```

FIELD Function

Description

The **FIELD** function returns the location index value (position) of a string of *string1*, *string2*. The function returns 0 if it does not have a parameter value which is the same as *search_string*. It returns 0 if *search_string* is **NULL** because it cannot perform the comparison operation with the other arguments.

If all arguments specified for **FIELD()** are of string type, string comparison operation is performed: if all of them are of number type, numeric comparison operation is performed. If the type of one argument is different from that of another, a comparison operation is performed by casting each argument to the type of the first argument. If type casting fails during the comparison operation with each argument, the function considers the result of the comparison operation as **FALSE** and resumes the other operations.

Syntax

```
FIELD( search_string, string1 [,string2 [, ... [, stringN]...]])

string :
• character string
• NULL
```

Example

```
SELECT FIELD('abc', 'a', 'ab', 'abc', 'abcd', 'abcde');

==================================================
                                                 3

--it returns 0 when no same string is found in the list
SELECT FIELD('abc', 'a', 'ab', NULL);

================================
                               0

--it returns 0 when null is specified in the first parameter
SELECT FIELD(NULL, 'a', 'ab', NULL);

===============================
                              0
```

INSTR Function

Description

The **INSTR** function, similarly to the **POSITION**, returns the position of a *substring* within *string*; the position. For the **INSTR** function, you can specify the starting position of the search for *substring* to make it possible to search for duplicate *substring*.

Note that the function calculates the starting position and the length of the character string in bytes, not in characters. For a multi-byte character set, the number of bite representing onc character is different, so the return value may not be the same.

Syntax

```
INSTR( string , substring [, position] )

string , substring :
• character string
• NULL
position :
• INT
• NULL
```

- *string* : Specifies the input character string.
- *substring* : Specifies the character string whose position is to be returned.
- *position* : Optional. Represents the position of a *string* where the search begins. If omitted, the default value 1 is applied. The first position of the *string* is specified as 1. If the value is negative, the system counts backward from the end of the *string*.

Example

```
--character set is euc-kr for Korean characters
--it returns position of the first 'b'
SELECT INSTR ('12345abcdeabcde','b');

===================================
                                  7

-- it returns position of the first '나' on double byte charset
SELECT INSTR ('12345가나다라마가나다라마', '나' );
=================================
                                8

-- it returns position of the second '나' on double byte charset
SELECT INSTR ('12345가나다라마가나다라마', '나', 16 );
=================================
                               18
```

```
--it returns position of the 'b' searching from the 8th position
SELECT INSTR ('12345abcdeabcde','b', 8);

===================================
                                 12

--it returns position of the 'b' searching backwardly from the end
SELECT INSTR ('12345abcdeabcde','b', -1);

====================================
                                  12

--it returns position of the 'b' searching backwardly from a specified position
SELECT INSTR ('12345abcdeabcde','b', -8);

====================================
                                   7
```

LCASE, LOWER Function

Description

The **LOWER** function converts uppercase characters that are included in a character string to lowercase characters; it works the same as the **LCASE** function. Note that the **LOWER** function may not work properly in character sets that are not supported by CUBRID. For details about the character sets supported by CUBRID, see Definition and Characteristics.

Syntax

```
LCASE (string )
LOWER ( string )

string :
• character string
• NULL
```

- *string* : Specifies the string in which uppercase characters are to be converted to lowercase. If the value is **NULL**, **NULL** is returned.

Example

```
SELECT LOWER('');

======================
  ''

SELECT LOWER(NULL);

======================
  NULL

SELECT LOWER('Cubrid');

======================
  'cubrid'
```

LEFT Function

Description

The **LEFT** function returns a length number of characters from the leftmost of *string*. If any of the arguments is **NULL**, **NULL** is returned. If a value greater than the *length* of the *string* or a negative number is specified for a length, the entire string is returned.

To extract a length number of characters from the rightmost of the string, use the RIGHT Function.

Syntax

```
LEFT( string , length )

string :
• character string
• NULL

length :
• INT
• NULL
```

Example

```
SELECT LEFT('CUBRID', 3);
======================
  'CUB'
==================

SELECT LEFT('CUBRID', 10);
======================
  'CUBRID'
==================
```

LOCATE Function

Description

The **LOCATE** function returns the location index value of a `substring` within a character string. The third argument *position* can be omitted. If this argument is specified, the function searches for **substring** from the given position and returns the location index value of the first occurrence. If the *substring* cannot be found within the string, 0 is returned.

The **LOCATE** function is working like the POSITION Function, but you cannot use **LOCATE** for bit strings.

Syntax

```
LOCATE ( substring, string [, position] )

string :
• character string
• NULL
```

Example

```
--it returns 1 when substring is empty space
SELECT LOCATE ('', '12345abcdeabcde');
===============================
                              1

--it returns position of the first 'abc'
SELECT LOCATE ('abc', '12345abcdeabcde');
================================
                               6

--it returns position of the second 'abc'
SELECT LOCATE ('abc', '12345abcdeabcde', 8) FROM db_root;
======================================
                                    11

--it returns 0 when no substring found in the string
SELECT LOCATE ('ABC', '12345abcdeabcde');
=================================
                                0
```

LPAD Function

Description

The **LPAD** function pads the left side of a string with a specific set of characters.

Syntax

```
LPAD( char1, n, [, char2 ] )

char1 :
• character string
• string valued column
• NULL

n :
• integer
• NULL

char2 :
• character string
• NULL
```

- *char1* : Specifies the string to pad characters to. If *n* is smaller than the length of *char1*, padding is not performed, and *char1* is truncated to length n and then returned. A single character is processed as 2 or 3 bytes in multi-byte character set environment. If *char1* is truncated up to the first byte representing a character according to a value of *n*, the last byte is removed and a space character (1 byte) is added to the left because the last character cannot be represented normally. If the value is **NULL**, **NULL** is returned.
- *n* : Specifies the total length of *char1* in bytes. Note that the number and the length of the character strings may be different in multi-byte character set environment. If the value is **NULL**, **NULL** is returned.
- *char2* : Specifies the string to pad to the left until the length of *char1* reaches *n*. If it is not specified, empty characters (' ') are used as a default. If the value is **NULL**, **NULL** is returned.

Example

```
--character set is euc-kr for Korean characters
--it returns only 3 characters if not enough length is specified

SELECT LPAD ('CUBRID', 3, '?');

======================
  'CUB'

--on multi-byte charset, it returns the first character only with a left padded space

SELECT LPAD ('큐브리드', 3, '?');
======================
  ' 큐'

--padding spaces on the left till char_length is 10
SELECT LPAD ('CUBRID', 10);

======================
  '    CUBRID'

--padding specific characters on the left till char_length is 10
SELECT LPAD ('CUBRID', 10, '?');
======================
  '????CUBRID'

--padding specific characters on the left till char_length is 10

SELECT LPAD ('큐브리드', 10, '?');
======================
  '??큐브리드'

--padding 4 characters on the left

SELECT LPAD ('큐브리드', LENGTH('큐브리드')+4, '?');
======================
  '????큐브리드'
```

LTRIM Function

Description

The **LTRIM** function removes all specified characters from the left-hand side of a string.

Syntax

```
LTRIM( string [, trim_string])

string :
• character string
• string valued column
• NULL

trim_string :
• character string
• NULL
```

- *string* : Enters a string or string-type column to trim. If this value is **NULL**, **NULL** is returned.
- *trim_string* : You can specify a specific string to be removed in the left side of *string*. If it is not specified, empty characters (' ') is automatically specified so that the empty characters in the left side are removed.

Example

```
--trimming spaces on the left
SELECT LTRIM ('         Olympic         ')

======================
    'Olympic         '

--If NULL is specified, it returns NULL
SELECT LTRIM ('iiiiiOlympiciiiii', NULL)

======================
    NULL

-- trimming specific strings on the left
SELECT LTRIM ('iiiiiOlympiciiiii', 'i')

======================
    'Olympiciiiii'
```

MID Function

Description

The **MID** function extracts a string with the length of *substring_length* from a *position* within the *string* and then returns it. If a negative number is specified as a *position* value, the *position* is calculated in a reverse direction from the end of the *string*. **substring_length** cannot be omitted. If a negative value is specified, the function considers this as 0 and returns an empty string.

The **MID** function is working like the SUBSTR Function, but there are differences in that it cannot be used for bit strings, that the *substring_length* argument must be specified, and that it returns an empty string if a negative number is specified for *substring_length*.

Syntax

```
string :
• character string
• NULL

position :
• integer
• NULL

substring_length :
• integer
• NULL
```

- *string* : Specifies an input character string. If this value is **NULL**, **NULL** is returned.
- *position* : Specifies the starting position from which the string is to be extracted. The position of the first character is 1. It is considered to be 1 even if it is specified as 0. If the input value is **NULL**, **NULL** is returned.
- *substring_lenghth* : Specifies the length of the string to be extracted. If 0 or a negative number is specified, an empty string is returned; if **NULL** is specified, **NULL** is returned.

Example

```
CREATE TABLE mid_tbl(a VARCHAR);
INSERT INTO mid_tbl VALUES('12345abcdeabcde');

--it returns empty string when substring_length is 0
SELECT MID(a, 6, 0), SUBSTR(a, 6, 0), SUBSTRING(a, 6, 0) FROM mid_tbl;

==================================================================
  ''                    ''                    ''

--it returns 4-length substrings counting from the 6th position
SELECT MID(a, 6, 4), SUBSTR(a, 6, 4), SUBSTRING(a, 6, 4) FROM mid_tbl;

==================================================================
  'abcd'                'abcd'                'abcd'

--it returns a empty string when substring_length < 0
SELECT MID(a, 6, -4), SUBSTR(a, 6, -4), SUBSTRING(a, 6, -4) FROM mid_tbl;

==================================================================
  ''                    NULL                  'abcdeabcde'

--it returns 4-length substrings at 6th position counting backward from the end
SELECT MID(a, -6, 4), SUBSTR(a, -6, 4), SUBSTRING(a, -6, 4) FROM mid_tbl;

==================================================================
  'eabc'                'eabc'                '1234'
```

OCTET_LENGTH Function

Description

The **OCTET_LENGTH** function returns the length (byte) of a character string or bit string as an integer. Therefore, it returns 1 (byte) if the length of the bit string is 8 bits, but 2 (bytes) if the length is 9 bits.

Syntax

```
OCTET_LENGTH ( string )

string :
• bit string
• character string
• NULL
```

- *string* : Specifies the character or bit string whose length is to be returned in bytes. If the value is **NULL**, **NULL** is returned.

Example

```
--character set is euc-kr for Korean characters
SELECT OCTET_LENGTH('');
==================
                 0

SELECT OCTET_LENGTH('CUBRID');
==================
                 6
```

```
SELECT OCTET_LENGTH('큐브리드');
==================
                 8

SELECT OCTET_LENGTH(B'010101010');
==================
                 2

CREATE TABLE octet_length_tbl (char_1 CHAR, char_2 CHAR(5), varchar_1 VARCHAR, bit_var_1
BIT VARYING);
INSERT INTO octet_length_tbl VALUES('', '', '', B''); --Length of empty string
INSERT INTO octet_length_tbl VALUES('a', 'a', 'a', B'010101010'); --English character
INSERT INTO octet_length_tbl VALUES(NULL, '큐', '큐', B'010101010'); --Korean character and
NULL
INSERT INTO octet_length_tbl VALUES(' ', ' 큐', ' 큐', B'010101010'); --Korean character
and space

SELECT OCTET_LENGTH(char_1), OCTET_LENGTH(char_2), OCTET_LENGTH(varchar_1),
OCTET_LENGTH(bit_var_1) FROM octet_length_tbl;
=== <Result of SELECT Command in Line 16> ===

================================================================================
1                      5                         0                       0
1                      5                         1                       2
NULL                   5                         2                       2
1                      5                         3                       2
```

POSITION Function

Description

The **POSITION** function returns the position of a character string corresponding to *substring* within a character string corresponding to *string*. Note that it returns the position in bytes, not in characters. Therefore, the return values may differ because the number of bytes representing a single character is different in multi-byte character sets.

An expression that returns a character string or a bit string can be specified as an argument of this function. The return value is an integer greater than or equal to 0. This function returns the position value in bytes for a character string, and in bits for a bit string.

The **POSITION** function is occasionally used in combination with other functions. For example, if you want to extract a certain string from another string, you can use the result of the **POSITION** function as an input to the **SUBSTRING** function.

Syntax

```
POSITION ( substring IN string )

substring :
• bit string
• character string
• NULL
```

- *substring* : Specifies the character string whose position is to be returned. If the value is an empty character, 1 is returned. If the value is **NULL**, **NULL** is returned.

Example

```
--character set is euc-kr for Korean characters

--it returns 1 when substring is empty space
SELECT POSITION ('' IN '12345abcdeabcde');
===============================
                              1

--it returns position of the first 'b'
SELECT POSITION ('b' IN '12345abcdeabcde');
================================
                               7
```

```
-- it returns position of the first '나' on double byte charset
SELECT POSITION ('나' IN '12345가나다라마가나다라마');
=================================
                                8

--it returns 0 when no substring found in the string
SELECT POSITION ('f' IN '12345abcdeabcde');
=================================
                                0

SELECT POSITION (B'1' IN B'000011110000');
=================================
                                5
```

REPLACE Function

Description

The **REPLACE** function searches for a character string, *search_string*, within a given character string, *string*, and replaces it with a character string, *replacement_string*. If the string to be replaced, *replacement_string* is omitted, all *search_strings* retrieved from *string* are removed. If **NULL** is specified as an argument, **NULL** is returned.

Syntax

```
REPLACE( string, search_string [, replacement_string ] )

string :
• character string
• NULL

search_string :
• character string
• NULL

replacement_string :
• character string
• NULL
```

- *string* : Specifies the original string. If the value is **NULL**, **NULL** is returned.
- *search_string* : Specifies the string to be searched. If the value is **NULL**, **NULL** is returned.
- *replacement_string* : Specifies the string to replace the *search_string*. If this value is omitted, *string* is returned with the *search_string* removed. If the value is **NULL**, **NULL** is returned.

Example

```
--it returns NULL when an argument is specified with NULL value
SELECT REPLACE('12345abcdeabcde','abcde',NULL);

======================
  NULL

--not only the first substring but all substrings into 'ABCDE' are replaced
SELECT REPLACE('12345abcdeabcde','abcde','ABCDE');

======================
  '12345ABCDEABCDE'

--it removes all of substrings when replace_string is omitted
SELECT REPLACE('12345abcdeabcde','abcde');

======================
  '12345'
```

REVERSE Function

Description

The **REVERSE** function returns *string* converted in the reverse order.

Syntax

```
REVERSE( string )

string :
• character string
• NULL
```

- *substring* : Specifies an input character string. If the value is an empty string, empty value is returned. If the value is **NULL**, **NULL** is returned.

Example

```
SELECT REVERSE('CUBRID');

======================
  'DIRBUC'
```

RIGHT Function

Description

The **RIGHT** function returns a *length* number of characters from the rightmost of a *string*. If any of the arguments is **NULL**, **NULL** is returned. If a value greater than the length of the *string* or a negative number is specified for a *length*, the entire string is returned.

To extract a length number of characters from the leftmost of the string, use the LEFT Function.

Syntax

```
RIGHT( string , length )

string :
• character string
• NULL

length :
• INT
• NULL
```

Example

```
SELECT RIGHT('CUBRID', 3);
======================
  'RID'
==================

SELECT RIGHT ('CUBRID', 10);
======================
  'CUBRID'
==================
```

RPAD Function

Description

The **RPAD** function pads the right side of a string with a specific set of characters.

Syntax

```
RPAD( char1, n, [, char2 ] )
```

```
char1 :
• character string
• string valued column
• NULL

n :
• integer
• NULL

char2 :
• character string
• NULL
```

- *char1* : Specifies the string to pad characters to. If n is smaller than the length of *char1*, padding is not performed, and *char1* is truncated to length n and then returned. A single character is processed as 2 or 3 bytes in multi-byte character set environment. If *char1* is truncated up to the first byte representing a character according to a value of *n*, the last byte is removed and an empty character (1 byte) is added to the left because the last character cannot be represented normally. If the value is **NULL**, **NULL** is specified.
- *n* : Specifies the total length of *char1* in bytes. Note that the number and the length of the character strings may be different in multi-byte character set environment. If the value is **NULL**, **NULL** is specified.
- *char2* : Specifies the string to pad to the right until the length of *char1* reaches *n*. If it is not specified, empty characters (' ') are used as a default. If the value is **NULL**, **NULL** is returned.

Example

```
--character set is euc-kr for Korean characters

--it returns only 3 characters if not enough length is specified

SELECT RPAD ('CUBRID', 3, '?');

======================
  'CUB'

--on multi-byte charset, it returns the first character only with a right-padded space

SELECT RPAD ('큐브리드', 3, '?');
======================
  '큐 '

--padding spaces on the right till char_length is 10
SELECT RPAD ('CUBRID', 10);

======================
  'CUBRID    '

--padding specific characters on the right till char_length is 10

SELECT RPAD ('CUBRID', 10, '?');
======================
  'CUBRID????'

--padding specific characters on the right till char_length is 10

SELECT RPAD ('큐브리드', 10, '?');
======================
  '큐브리드??'

--padding 4 characters on the right

SELECT RPAD ('큐브리드', LENGTH('큐브리드')+4, '?');
======================
  '큐브리드????'
```

RTRIM Function

Description

The **RTRIM** function removes specified characters from the right-hand side of a string.

Syntax

```
RTRIM( string [, trim_string])

string :
• character string
• string valued column
• NULL

trim_string :
• character string
• NULL
```

- *string* : Enters a string or string-type column to trim. If this value is **NULL**, **NULL** is returned.
- *trim_string* : You can specify a specific string to be removed in the right side of *string*. If it is not specified, empty characters (' ') is automatically specified so that the empty characters in the right side are removed.

Example

```
SELECT RTRIM ('         Olympic         ')

======================
    '         Olympic'

--If NULL is specified, it returns NULL
SELECT RTRIM ('iiiiiOlympiciiiii', NULL)

======================
    NULL

-- trimming specific strings on the right
SELECT RTRIM ('iiiiiOlympiciiiii', 'i')

======================
    'iiiiiOlympic'
```

STRCMP Function

Description

The **STRCMP** function compares two strings, *string1* and *string2*, and returns 0 if they are identical, 1 if *string1* is greater, or -1 if *string1* is smaller. If any of the parameters is **NULL**, **NULL** is returned.

Syntax

```
STRCMP( string1 , string2 )

string :
• character string
• NULL
```

Example

```
SELECT STRCMP('abc', 'abc');

=======================
                      0
SELECT STRCMP ('acc', 'abc');

=======================
                      1

--STRCMP works case-insensitively
```

```
SELECT STRCMP ('ABC','abc');

=======================
                      0
```

SUBSTR Function

Description

The **SUBSTR** function extracts a character string with the length of *substring_length* from a position, *position*, within character string, *string*, and then returns it. If a negative number is specified as a *position* value, the position is calculated in a reverse direction from the end of the string. If *substring_length* is omitted, character strings between the given position, *position*, and the end of the string are extracted, and then returned.

Note that it returns the starting position and the length of character string in bytes, not in characters. Therefore, in a multi-byte character set, you must specify the parameter in consideration of the number of bytes representing a single character.

Syntax

```
SUBSTR( string, position [, substring_length])

string :
• character string
• bit string
• NULL

position :
• integer
• NULL

substring_length :
• integer
```

- *string* : Specifies the input character string. If the input value is **NULL**, **NULL** is returned.
- *position* : Specifies the position from where the string is to be extracted in bytes. Even though the position of the first character is specified as 1 or a negative number, it is considered as 1. If a value greater than the string length or **NULL** is specified, **NULL** is returned.
- *substring_length* : Specifies the length of the string to be extracted in bytes. If this argument is omitted, character strings between the given position, *position*, and the end of them are extracted. **NULL** cannot be specified as an argument value of this function. If 0 is specified, an empty string is returned; if a negative value is specified, **NULL** is returned.

Example

```
--character set is euc-kr for Korean characters

--it returns empty string when substring_length is 0
SELECT SUBSTR('12345abcdeabcde',6, 0);

======================
  ''

--it returns 4-length substrings counting from the position
SELECT SUBSTR('12345abcdeabcde', 6, 4), SUBSTR('12345abcdeabcde', -6, 4);

============================================
  'abcd'                'eabc'


--it returns substrings counting from the position to the end
SELECT SUBSTR('12345abcdeabcde', 6), SUBSTR('12345abcdeabcde', -6);
============================================
  'abcdeabcde'          'eabcde'

-- it returns 4-length substrings counting from 16th position on double byte charset
SELECT SUBSTR ('12345가나다라마가나다라마', 16 , 4);
======================
  '가나'
```

SUBSTRING Function

Description

The **SUBSTRING** function, operating like **SUBSTR**, extracts a character string having the length of *substring_length* from a position, *position*, within character string, *string*, and returns it.

If a negative number is specified to the *position* value, the **SUBSTRING** function calculates the position from the beginning of the string. And **SUBSTR** function calculates the position from the end of the string. If a negative number is specified to the *substring_length* value, the **SUBSTRING** function handles the argument is omitted, but the **SUBSTR** function returns **NULL**.

Syntax

```
SUBSTRING( string, position [, substring_length])
SUBSTRING( string FROM position [FOR substring_length] )

string :
• bit string
• character string
• NULL

position :
• integer
• NULL

substring_length :
• integer
```

- *string* : Specifies the input character string. If the input value is **NULL**, **NULL** is returned.
- *position* : Specifies the position from where the string is to be extracted in bytes. Even though the position of the first character is specified as 1 or a negative number, it is considered as 1. If a value greater than the string length is specified, an empty string is returned. If **NULL**, **NULL** is returned.
- *substring_length* : Specifies the length of the string to be extracted in bytes. If this argument is omitted, character strings between the given position, *position*, and the end of them are extracted. **NULL** cannot be specified as an argument value of this function. If 0 is specified, an empty string is returned; if a negative value is specified, **NULL** is returned.

Example

```
SELECT SUBSTRING('12345abcdeabcde', -6 ,4), SUBSTR('12345abcdeabcde', -6 ,4);
============================================
  '1234'                'eabc'


SELECT SUBSTRING('12345abcdeabcde', 16), SUBSTR('12345abcdeabcde', 16);
============================================
  ''                    NULL

SELECT SUBSTRING('12345abcdeabcde', 6, -4), SUBSTR('12345abcdeabcde', 6, -4);
============================================
  'abcdeabcde'          NULL
```

TRANSLATE Function

Description

The **TRANSLATE** function searches for a character specified as a character string, *from_substring*, within a given character string, *string*, and replaces it with a character specified as a character string, *to_substring*, if exists. Correspondence relationship is determined according to the order of characters specified by *from_substring* and *to_substring*. All characters in *from_substring* that do not have a one to one correspondence relationship with the characters in *to_substring* are removed from the string. The **TRANSLATE** function is working like the **REPLACE** function, but you cannot omit the to_substring argument with this function.

Syntax

```
TRANSLATE( string, from_substring, to_substring )

string :
• character string
• NULL

from_substring :
• character string
• NULL

to_substring :
• character string
• NULL
```

- *string* : Specifies the original string. If the value is **NULL**, **NULL** is returned.
- *from_substring* : Specifies the string to be retrieved. If the value is **NULL**, **NULL** is returned.
- *to_substring* : Specifies the character string in the *from_substring* to be replaced. It cannot be omitted. If the value is **NULL**, **NULL** is returned.

Example

```
--it returns NULL when an argument is specified with NULL value
SELECT TRANSLATE('12345abcdeabcde','abcde', NULL);

======================
  NULL

--it translates 'a','b','c','d','e' into '1', '2', '3', '4', '5' respectively
SELECT TRANSLATE('12345abcdeabcde', 'abcde', '12345');

======================
  '123451234512345'

--it translates 'a','b','c' into '1', '2', '3' respectively and removes 'd's and 'e's
SELECT TRANSLATE('12345abcdeabcde','abcde', '123');

======================
  '12345123123'

--it removes 'a's,'b's,'c's,'d's, and 'e's in the string
SELECT TRANSLATE('12345abcdeabcde','abcde', '');
======================
  '12345'

--it only translates 'a','b','c' into '3', '4', '5' respectively
SELECT TRANSLATE('12345abcdeabcde','ABabc', '12345');

======================
  '12345345de345de'
```

TRIM Function

Description

The **TRIM** function trims leading and/or trailing characters from a character string.

Syntax

```
TRIM ( [ [ LEADING | TRAILING | BOTH ] [ trim_string ] FROM ] string )

trim_string :
• character string
• NULL

string :
• character string literal
• string valued column
• NULL
```

- *trim_string* : You can specify a specific string to be removed that is in front of or at the back of the target string. If it is not specified, an empty character (' ') is automatically specified so that spaces in front of or at the back of the target string are removed.
- *string* : Enters a string or string-type column to trim. If this value is **NULL**, **NULL** is returned.
- [**LEADING** | **TRAILING** | **BOTH**] : You can specify an option to trim a specified string that is in a certain position of the target string. If it is **LEADING**, trimming is performed in front of a character string if it is **TRAILING**, trimming is performed at the back of a character string if it is **BOTH**, trimming is performed in front and at the back of a character string. If the option is not specified, **BOTH** is specified by default.
- The character string of *trim_string* and *string* should have the same character set.

Example

```
--trimming NULL returns NULL
SELECT TRIM (NULL)

======================
    NULL

--trimming spaces on both leading and trailing parts
SELECT TRIM ('         Olympic         ')

======================
    'Olympic'

--trimming specific strings on both leading and trailing parts
SELECT TRIM ('i' FROM 'iiiiiOlympiciiiii')

======================
    'Olympic'

--trimming specific strings on the leading part
SELECT TRIM (LEADING 'i' FROM 'iiiiiOlympiciiiii')

======================
    'Olympiciiiii'

--trimming specific strings on the trailing part
SELECT TRIM (TRAILING 'i' FROM 'iiiiiOlympiciiiii')

======================
    'iiiiiOlympic'
```

UCASE, UPPER Function

Description

The **UCASE** and **UPPER** functions convert lowercase characters that are included in a character string to uppercase characters. Note that the **UPPER** function may not work properly in character sets that are not supported by CUBRID. For details about the character sets supported by CUBRID, see Definition and Characteristics.

Syntax

```
UCASE ( string )
UPPER ( string )

string :
• character string
• NULL
```

- *string* : Specifies the string in which lowercase characters are to be converted to uppercase. If the value is **NULL**, **NULL** is returned.

Example

```
SELECT UPPER('');

======================
  ''
```

```
SELECT UPPER(NULL);

======================
  NULL

SELECT UPPER('Cubrid');

======================
  'CUBRID'
```

Numeric Operators and Functions

ABS Function

Description

The **ABS** function returns the absolute value of a given number. The data type of the return value is the same as that of the argument.

Syntax

```
ABS( number_operand )
```

- *number_operand* : An operator which returns a numeric value

Example

```
--it returns the absolute value of the argument
SELECT ABS(12.3), ABS(-12.3), ABS(-12.3000), ABS(0.0);

================================================================================
  12.3                  12.3                  12.3000               .0
```

ACOS Function

Description

The **ACOS** function returns an arc cosine value of the argument. That is, it returns a value whose cosine is *x* in radian. The return value is a **DOUBLE** type. x must be a value between -1 and 1, inclusive. Otherwise, **NULL** is returned.

Syntax

```
ACOS( x )
```

- *x* : An expression that returns a numeric value.

Example

```
SELECT ACOS(1), ACOS(0), ACOS(-1);

==================================================================================
     0.000000000000000e+00     1.570796326794897e+00     3.141592653589793e+00
```

ASIN Function

Description

The **ASIN** function returns an arc sine value of the argument. That is, it returns a value whose sine is *x* in radian. The return value is a **DOUBLE** type. x must be a value between -1 and 1, inclusive. Otherwise, **NULL** is returned.

Syntax

```
ASIN ( x )
```

- *x* : An expression that returns a numeric value.

Example

```
SELECT ASIN(1), ASIN(0), ASIN(-1);

==============================================================================
     1.570796326794897e+00     0.000000000000000e+00    -1.570796326794897e+00
```

ATAN Function

Description

The **ATAN** function returns a value whose tangent is *x* in radian. The argument *y* can be omitted. If *y* is specified, the function calculates the arc tangent value of *y/x*. The return value is a **DOUBLE** type.

Syntax

```
ATAN ( [y,] x )
```

- *x*, *y* : An expression that returns a numeric value.

Example

```
SELECT ATAN(1), ATAN(-1), ATAN(1,-1);

=== <Result of SELECT Command in Line 1> ===

                   atan(1)                  atan(-1)              atan2(1, -1)
==============================================================================
     7.853981633974483e-01    -7.853981633974483e-01     2.356194490192345e+00
```

ATAN2 Function

Description

The **ATAN2** function returns the arc tangent value of *y/x* in radian. This function is working like the ATAN **Function**. Arguments *x* and *y* must be specified. The return value is a **DOUBLE** type.

Syntax

```
ATAN2 ( y, x )
```

- *x*, *y* : An expression that returns a numeric value.

Example

```
SELECT ATAN2(1,1), ATAN2(-1,-1), ATAN2(Pi(),0);

=== <Result of SELECT Command in Line 1> ===

atan2(1, 1)             atan2(-1, -1)           atan2( pi(), 0)
==============================================================================
 7.853981633974483e-01    -2.356194490192345e+00     1.570796326794897e+00
```

CEIL Function

Description

The **CEIL** function returns the smallest integer that is not less than its argument. The return value is determined based on the valid number of digits that are specified as the *number_operand* argument.

Syntax

```
CEIL( number_operand )
```

- *number_operand* : An expression that returns a numeric value.

Example

```
SELECT CEIL(34567.34567), CEIL(-34567.34567);

============================================
  34568.00000           -34567.00000

SELECT CEIL(34567.1), CEIL(-34567.1);
=============================
         34568.0         -34567.0
```

COS Function

Description

The **COS** function returns a cosine value of the argument. The argument *x* must be a radian value. The return value is a **DOUBLE** type.

Syntax

```
COS( x )
```

- *x* : An expression that returns a numeric value.

Example

```
SELECT COS(pi()/6), COS(pi()/3), COS(pi());

==============================================================================
    8.660254037844387e-01     5.000000000000001e-01     -1.000000000000000e+00
```

COT Function

Description

The **COT** function returns the cotangent value of the argument *x*. That is, it returns a value whose tangent is *x* in radian. The return value is a **DOUBLE** type.

Syntax

```
COT ( x )
```

- *x* : An expression that returns a numeric value.

Example

```
SELECT COT(1), COT(-1), COT(0);

==========================================================================
        6.420926159343306e-01         -6.420926159343306e-01         NULL
```

DEGREES Function

Description

The **DEGREES** function returns the argument *x* specified in radian converted to a degree value. The return value is a **DOUBLE** type.

Syntax

```
DEGREES ( x )
```

- *x* : An expression that returns a numeric value.

Example

```
SELECT DEGREES(pi()/6), DEGREES(pi()/3), DEGREES (pi());
```

```
==============================================================================
    3.000000000000000e+01     5.999999999999999e+01     1.800000000000000e+02
```

DRANDOM/DRAND Function

Description

The **DRANDOM/DRAND** function returns a random double-precision floating point value in the range of between 0.0 and 1.0. A *seed* argument that is **INTEGER** type can be specified. It rounds up real numbers and an error is returned when it exceeds the range of **INTEGER**.

The **DRAND** function performs the operation only once to produce only one random number regardless of the number of rows where the operation is output, but the **DRANDOM** function performs the operation every time the statement is repeated to produce a different random value for each row. Therefore, to output rows in a random order, you must use the **DRANDOM** function in the **ORDER BY** clause.

To obtain a random integer value, use the RANDOM/RAND Function.

Syntax

```
DRANDOM( [seed] )
DRAND( [seed] )
```

Example

```
SELECT DRAND(), DRAND(1), DRAND(1.4);
=== <Result of SELECT Command in Line 1> ===

       drand()                 drand(1)                  drandom(1.4)
==============================================================================
   2.849646518006921e-001 4.163034446537495e-002 4.163034446537495e-002

SELECT * FROM rand_tbl;
=== <Result of SELECT Command in Line 1> ===
           id  name
===================================
            1  'a'
            2  'b'
            3  'c'
            4  'd'
            5  'e'
            6  'f'
            7  'g'
            8  'h'
            9  'i'
           10  'j'

--drandom() returns random values on every row
SELECT DRAND(), DRANDOM() FROM rand_tbl;
=== <Result of SELECT Command in Line 1> ===

   drand()                 drandom()
==============================================================================
   7.638782921842098e-001    1.018707846308786e-001
   7.638782921842098e-001    3.191320535905026e-001
   7.638782921842098e-001    3.461714529862361e-001
   7.638782921842098e-001    6.791894283883175e-001
   7.638782921842098e-001    4.533829767754143e-001
   7.638782921842098e-001    1.714224677266762e-001
   7.638782921842098e-001    1.698049867244484e-001
   7.638782921842098e-001    4.507583849604786e-002
   7.638782921842098e-001    5.279091769157994e-001
   7.638782921842098e-001    7.021088290047914e-001

--selecting rows in random order
SELECT * FROM rand_tbl ORDER BY DRANDOM();

=== <Result of SELECT Command in Line 1> ===
           id  name
===================================
            6  'f'
            2  'b'
            7  'g'
            8  'h'
            1  'a'
            4  'd'
           10  'j'
            9  'i'
            5  'e'
            3  'c'
```

EXP Function

Description

The EXP function returns e^x (the base of natural logarithm) raised to a power.

Syntax

```
EXP( x )
```

- *x :* An operator which returns a numeric value

Example

```
SELECT EXP(1), EXP(0);

=========================================================
    2.718281828459045e+000000     1.000000000000000e+000
SELECT EXP(-1), EXP(2.00);

=========================================================
    3.678794411714423e-001     7.389056098930650e+000
```

FLOOR Function

Description

The **FLOOR** function returns the largest integer that is not greater than its argument. The data type of the return value is the same as that of the argument.

Syntax

```
FLOOR( number_operand )
```

- *number_operand* : An operator which returns a numeric value

Example

```
--it returns the largest integer less than or equal to the arguments
SELECT FLOOR(34567.34567), FLOOR(-34567.34567);

============================================
  34567.00000           -34568.00000

SELECT FLOOR(34567), FLOOR(-34567);
=============================
         34567         -34567
```

FORMAT Function

Description

The **FORMAT** function displays the number *x* by using commas as thousands delimiters, so that its format becomes '#,###,###.#####' and performs rounding after the decimal point to express as many as *dec* digits after it. The return value is a string type.

Syntax

```
FORMAT ( x , dec )
```

- *x* , *dec* : An expression that returns a numeric value.

Example

```
SELECT FORMAT(12000.123456,3), FORMAT(12000.123456,0);

==================================================================
  '12,000.123'                 '12,000'
```

GREATEST Function

Description

The **GREATEST** function compares more than one expression specified as parameters and returns the greatest value. If only one expression has been specified, the expression is returned because there is no expression to be compared with.

Therefore, more than one expression that are specified as parameters must be of the type that can be compared with each other. If the types of the specified parameters are identical, so are the types of the return values; if they are different, the type of the return value becomes a convertible common data type.

That is, the **GREATEST** function compares the values of column 1, column 2 and column 3 in the same row and returns the greatest value while the **MAX** function compares the values of column in all result rows and returns the greatest value.

Syntax

```
GREATEST( expression [, expression]* )
```

- *expression* : Specifies more than one expression. Their types must be comparable each other. One of the arguments is **NULL**, **NULL** is returned.

Example

The following is an example that returns the number of every medals and the highest number that Korea won. (demodb)

```
SELECT gold, silver , bronze, GREATEST (gold, silver, bronze) FROM participant
WHERE nation_code = 'KOR';

=== <Result of SELECT Command in Line 2> ===

gold       silver       bronze  greatest(gold, silver, bronze)
=======================================================================
            9           12            9                              12
            8           10           10                              10
            7           15            5                              15
           12            5           12                              12
           12           10           11                              12
```

LEAST Function

Description

The **LEAST** function compares more than one expression specified as parameters and returns the smallest value. If only one expression has been specified, the expression is returned because there is no expression to be compared with.

Therefore, more than one expression that are specified as parameters must be of the type that can be compared with each other. If the types of the specified parameters are identical, so are the types of the return values; if they are different, the type of the return value becomes a convertible common data type.

That is, the **LEAST** function compares the values of column 1, column 2 and column 3 in the same row and returns the smallest value while the **MIN** function compares the values of column in all result rows and returns the smallest value.

Syntax

```
LEAST( expression [, expression]* )
```

- *expression* : Specifies more than one expression. Their types must be comparable each other. One of the arguments is **NULL**, **NULL** is returned.

Example

The following is an example that returns the number of every medals and the lowest number that Korea won. (demodb)

```
SELECT gold, silver , bronze, LEAST(gold, silver, bronze) FROM participant
WHERE nation_code = 'KOR';


=== <Result of SELECT Command in Line 2> ===

         gold       silver       bronze  least(gold, silver, bronze)
====================================================================
            9           12            9                            9
            8           10           10                            8
            7           15            5                            5
           12            5           12                            5
           12           10           11                           10
```

LN Function

Description

The **LN** function returns the natural log value (base = e) of an antilogarithm *x*. The return value is a **DOUBLE** type. If the antilogarithm is 0 or a negative number, an error is returned.

Syntax

```
LN ( x )
```

- *x* : An expression that returns a positive value.

Example

```
SELECT ln(1), ln(2.72);

=====================================================
     0.000000000000000e+00     1.000631880307906e+00
```

LOG2 Function

Description

The **LOG2** function returns a log value whose antilogarithm is *x* and base is 2. The return value is a **DOUBLE** type. If the antilogarithm is 0 or a negative number, an error is returned.

Syntax

```
LOG2 ( x )
```

- *x* : An expression that returns a positive number.
- fails.

Example

```
SELECT log2(1), log2(8);

==================================================
     0.000000000000000e+00     3.000000000000000e+00
```

LOG10 Function

Description

The **LOG10** function returns the common log value of an antilogarithm *x*. The return value is a **DOUBLE** type. If the antilogarithm is 0 or a negative number, an error is returned.

Syntax

```
LOG10 ( x )
```

- *x* : An expression that returns a positive number.

Example

```
SELECT log10(1), log10(1000)

==========================================================================
     0.000000000000000e+00     3.000000000000000e+00
```

MOD Function

Description

The **MOD** function returns the remainder of the first parameter *m* divided by the second parameter *n*. If *n* is 0, *m* is returned without the division operation being performed.

Note that if the dividend, the parameter m of the **MOD** function, is a negative number, the function operates differently from a typical operation (classical modulus) method.

Result of MOD

m	**n**	**MOD(m, n)**	**Classical Modulus** **m-n*FLOOR(m/n)**
11	4	3	3
11	-4	3	-1
-11	4	-3	1
-11	-4	-3	-3
11	0	11	Divided by 0 error

Syntax

```
MOD(m, n)
```

- *m* : Represents the dividend. It is an expression that returns a numeric value.
- *n* : Represents the divisor. It is an expression that returns a numeric value.

Example

```
--it returns the reminder of m divided by n
SELECT MOD(11, 4), MOD(11, -4), MOD(-11, 4), MOD(-11, -4), MOD(11,0);

=================================================================
           3             3            -3              -3           11


SELECT MOD(11.0, 4), MOD(11.000, 4), MOD(11, 4.0), MOD(11, 4.000);
```

```
==========================================================================
  3.0                   3.000                 3.0                   3.000
```

PI Function

Description

The **PI** function returns the π value, and is expressed in double-precision.

Syntax

```
PI()
```

Example

```
SELECT PI(), PI()/2;

=====================================================
     3.141592653589793e+00     1.570796326794897e+00
```

POW, POWER Function

Description

The **POW** function returns *x* to the power of *y*. **POW** and **POWER** are used interchangeably. The return value is a **DOUBLE** type.

Syntax

```
POW( x, y )
POWER( x, y )
```

- *x* : It represents the base. It is an expression that returns a numeric value. An expression that returns a numeric value.
- *y* : It represents the exponent. An expression that returns a numeric value. If the base is a negative number, an integer must specified as the exponent.

Example

```
SELECT POWER(2, 5), POWER(-2, 5), POWER(0, 0), POWER(1,0);
=======================================================================================
 3.200000000000000e+01    -
3.200000000000000e+01     1.000000000000000e+00     1.000000000000000e+00


--it returns an error when the negative base is powered by a non-int exponent
SELECT POWER(-2, -5.1), POWER(-2, -5.1);
ERROR
```

POW, POWER Function

Description

The **POW** function returns *x* to the power of *y*. **POW** and **POWER** are used interchangeably. The return value is a **DOUBLE** type.

Syntax

```
POW( x, y )
POWER( x, y )
```

- *x* : It represents the base. It is an expression that returns a numeric value. An expression that returns a numeric value.
- *y* : It represents the exponent. An expression that returns a numeric value. If the base is a negative number, an integer must specified as the exponent.

Example

```
SELECT POWER(2, 5), POWER(-2, 5), POWER(0, 0), POWER(1,0);
======================================================================================
 3.200000000000000e+01    -
3.200000000000000e+01     1.000000000000000e+00     1.000000000000000e+00


--it returns an error when the negative base is powered by a non-int exponent
SELECT POWER(-2, -5.1), POWER(-2, -5.1);
ERROR
```

RADIANS Function

Description

The **RADIANS** function returns the argument *x* specified in degrees converted to a radian value. The return value is a **DOUBLE** type.

Syntax

```
RADIANS ( x )
```

- *x* : An expression that returns a numeric value.

Example

```
SELECT RADIANS(90), RADIANS(180), RADIANS(360);

==============================================================================
     1.570796326794897e+00     3.141592653589793e+00     6.283185307179586e+00
```

RANDOM/RAND Function

Description

The **RANDOM/RAND** function returns any integer value between 0^2^{31} and a *seed* argument that is **INTEGER** type can be specified. It rounds up real numbers and an error is returned when it exceeds the range of **INTEGER**.

The **RAND** function performs the operation only once to produce only one random number regardless of the number of rows where the operation is output, but the **RANDOM** function performs the operation every time the statement is repeated to produce a different random value for each row. Therefore, to output rows in a random order, you must use the **RANDOM** function.

To obtain a random real number, use the DRANDOM/DRAND Function.

Syntax

```
RANDOM( [seed] )
RAND( [seed] )
```

Example

```
SELECT RAND(), RAND(1), RAND(1.4);
=== <Result of SELECT Command in Line 1> ===

       rand()      rand(1)      random(1.4)
====================================================
    1526981144      89400484      89400484

--creating a new table
SELECT * FROM rand_tbl;
=== <Result of SELECT Command in Line 1> ===

           id  name
===================================
            1  'a'
            2  'b'
            3  'c'
            4  'd'
            5  'e'
            6  'f'
            7  'g'
            8  'h'
            9  'i'
           10  'j'

--random() returns random values on every row
SELECT RAND(),RANDOM() FROM rand_tbl;
=== <Result of SELECT Command in Line 1> ===

       rand()       random()
============================
   2078876566     1753698891
   2078876566     1508854032
   2078876566      625052132
   2078876566      279624236
   2078876566     1449981446
   2078876566     1360529082
   2078876566     1563510619
   2078876566     1598680194
   2078876566     1160177096
   2078876566     2075234419


--selecting rows in random order
SELECT * FROM rand_tbl ORDER BY RANDOM();
=== <Result of SELECT Command in Line 1> ===

           id  name
===================================
            6  'f'
            1  'a'
            5  'e'
            4  'd'
            2  'b'
            7  'g'
           10  'j'
            9  'i'
            3  'c'
            8  'h'
```

ROUND Function

Description

The **ROUND** function returns the specified argument, *number_operand*, rounded to the number of places after the decimal point specified by the *integer*. If the *integer* argument is a negative number, it rounds to a place before the decimal point, that is, at the integer part.

Syntax

```
ROUND( number_operand, integer )
```

- *number_operand* : An expression that returns a numeric value
- *integer* : Specifies the place to round to. If a positive integer *n* is specified, the number is represented to the nth place after the decimal point; if a negative integer *n* is specified, the number is rounded to the *n*th place before the decimal point.
- The return value has the same type as the *number_operand*.

Example

```
--it rounds a number to one decimal point when the second argument is omitted
SELECT ROUND(34567.34567), ROUND(-34567.34567);

============================================
  34567.00000           -34567.00000
```

```
--it rounds a number to three decimal point
SELECT ROUND(34567.34567, 3), ROUND(-34567.34567, 3)  FROM db_root;

============================================
  34567.34600           -34567.34600

--it rounds a number three digit to the left of the decimal point
SELECT ROUND(34567.34567, -3), ROUND(-34567.34567, -3);

============================================
  35000.00000           -35000.00000
```

SIGN Function

Description

The **SIGN** function returns the sign of a given number. It returns 1 for a positive value, -1 for a negative value, and 0 for zero.

Syntax

```
SIGN(number_operand)
```

- *number_operand* : An operator which returns a numeric value

Example

```
--it returns the sign of the argument

SELECT SIGN(12.3), SIGN(-12.3), SIGN(0);

========================================
            1            -1            0
```

SIN Function

Description

The **SIN** function returns a sine value of the parameter. The argument *x* must be a radian value. The return value is a **DOUBLE** type.

Syntax

```
SIN ( x )
```

- *x* : An expression that returns a numeric value.

Example

```
SELECT SIN(pi()/6), SIN(pi()/3), SIN(pi());

==============================================================================
    4.999999999999999e-01     8.660254037844386e-01     1.224646799147353e-16
```

SQRT Function

Description

The **SQRT** function returns the square root of *x* as a **DOUBLE** type.

Syntax

```
SORT ( x )
```

- *x* : An expression that returns a numeric value. An error is returned if this value is a negative number.

Example

```
SELECT SQRT(4), SQRT(16.0);

====================================================
     2.000000000000000e+00     4.000000000000000e+00
```

TAN Function

Description

The **TAN** function returns a tangent value of the argument. The argument *x* must be a radian value. The return value is a **DOUBLE** type.

Syntax

```
TAN ( x )
```

- *x* : An expression that returns a numeric value.

Example

```
SELECT TAN(pi()/6), TAN(pi()/3), TAN(pi()/4);

tan( pi()/6)              tan( pi()/3)              tan( pi()/4)
==============================================================================
     5.773502691896257e-01     1.732050807568877e+00     9.999999999999999e-01
```

TRUNC, TRUNCATE Function

Description

The **TRUNC** and **TRUNCATE** function truncates the numbers of the specified argument *x* to the right of the *dec* position. If the *dec* argument is a negative number, it displays 0s to the *dec*-th position left to the decimal point. Note that the *dec* argument of the **TRUNC** function can be omitted, but that of the **TRUNCATE** function cannot be omitted. If the *dec* argument is a negative number, it displays 0s to the *dec*-th position left to the decimal point. The number of digits of the return value to be represented follows the argument *x*.

Syntax

```
TRUNC( x[, dec] )
TRUNCATE( x, dec )
```

- *x* : An expression that returns a numeric value.
- *dec* : The place to be truncated is specified. If a positive integer *n* is specified, the number is represented to the *n*-th place after the decimal point; if a negative integer *n* is specified, the number is truncated to the *n*-th place before the decimal point. It truncates to the first place after the decimal point if the *dec* argument is 0 or omitted. Note that the *dec* argument cannot be omitted in the **TRUNCATE** function.

Example

```
--it returns a number truncated to 0 places
SELECT TRUNC(34567.34567), TRUNCATE(34567.34567, 0);

============================================
  34567.00000            34567.00000

--it returns a number truncated to three decimal places
SELECT TRUNC(34567.34567, 3), TRUNC(-34567.34567, 3)  FROM db_root;

============================================
  34567.34500           -34567.34500

--it returns a number truncated to three digits left of the decimal point
SELECT TRUNC(34567.34567, -3), TRUNC(-34567.34567, -3);

============================================
  34000.00000           -34000.00000
```

Date/Time Operators and Functions

EXTRACT Operator

Description

The **EXTRACT** operator extracts the values from *date-time_argument* and then converts the value type into **INTEGER**.

Syntax

```
EXTRACT ( field FROM date-time_argument )

field :
• YEAR
• MONTH
• DAY
• HOUR
• MINUTE
• SECOND
• MILLISECOND

date-time_argument :
• expression
```

- *field* : Specifies a value to be extracted from date-time expression.
- *date-time argument* : An expression that returns a value of date-time. This expression must be one of **TIME**, **DATE**, **TIMESTAMP**, or **DATETIME** types. If the value is **NULL**, **NULL** is returned.

Example

```
SELECT EXTRACT(MONTH FROM DATETIME '2008-12-25 10:30:20:123' );
==========================================================
                                                        12

SELECT EXTRACT(HOUR FROM DATETIME '2008-12-25 10:30:20:123' );
==========================================================
                                                        10

SELECT EXTRACT(MILLISECOND FROM DATETIME '2008-12-25 10:30:20:123' );
==========================================================
                                                       123
```

ADDDATE, DATE_ADD Function

Description

The **ADDDATE** function performs an addition or subtraction operation on a specific **DATE** value; **ADDDATE** and **DATE_ADD** are used interchangeably. They return the DATETIME type in the following cases: 1) The first argument is **DATETIME** or **TIMESTAMP** type, 2) The first argument is **DATE** type, or 3) Less than **DAY** unit is specified for the value of INTERVAL.

Therefore, to return value of **DATETIME** type, you should convert the value of first argument by using the **CAST** function. Even though the date resulting from the operation exceeds the last day of the month, the function returns a valid **DATE** value considering the last date of the month.

Syntax

```
ADDDATE(date, INTERVAL expr unit)
DATE_ADD(date, INTERVAL expr unit)
ADDDATE(date, days)
```

- *date* : It is a **DATE**, **TIMETIME**, or **TIMESTAMP** expression that represents the start date. If an invalid **DATE** value such as '2006-07-00' is specified, an error is returned.

- *expr* : It represents the interval value to be added to the start date. If a negative number is specified next to the **INTERVAL** keyword, the interval value is subtracted from the start date.
- *unit* : It represents the unit of the interval value specified in the *expr* expression. See the following table to specify the format for the interpretation of the interval value. If the value of *expr* is less than the number requested in the *unit*, it is specified from the smallest unit. For example, if it is HOUR_SECOND, three values such as 'HOURS:MINUTES:SECONDS' are required. In the case, if only two values such as "1:1" are given, it is regarded as 'MINUTES:SECONDS'.

expr value for unix

Unit Value	**expr Value**
MILLISECOND	MILLISECONDS
SECOND	SECONDS
MINUTE	MINUTES
HOUR	HOURS
DAY	DAYS
WEEK	WEEKS
MONTH	MONTHS
QUARTER	QUARTERS
YEAR	YEARS
SECOND_MILLISECOND	'SECONDS.MILLISECONDS'
MINUTE_MILLISECOND	'MINUTES:SECONDS.MILLISECONDS'
MINUTE_SECOND	'MINUTES:SECONDS'
HOUR_MILLISECOND	'HOURS:MINUTES:SECONDS.MILLISECONDS'
HOUR_SECOND	'HOURS:MINUTES:SECONDS'
HOUR_MINUTE	'HOURS:MINUTES'
DAY_MILLISECOND	'DAYS HOURS:MINUTES:SECONDS.MILLISECONDS'
DAY_SECOND	'DAYS HOURS:MINUTES:SECONDS'
DAY_MINUTE	'DAYS HOURS:MINUTES'
DAY_HOUR	'DAYS HOURS'
YEAR_MONTH	'YEARS-MONTHS'

Example

```
SELECT SYSDATE, ADDDATE(SYSDATE,INTERVAL 24 HOUR), ADDDATE(SYSDATE, 1);

=== <Result of SELECT Command in Line 1> ===

   SYS_DATE    date_add( SYS_DATE , INTERVAL 24 HOUR)   adddate( SYS_DATE , 1)
==============================================================================
  03/30/2010  12:00:00.000 AM 03/31/2010               03/31/2010

--it substracts days when argument < 0
SELECT SYSDATE, ADDDATE(SYSDATE,INTERVAL -24 HOUR), ADDDATE(SYSDATE, -1);

=== <Result of SELECT Command in Line 1> ===

   SYS_DATE    date_add( SYS_DATE , INTERVAL -24 HOUR)   adddate( SYS_DATE , -1)
==============================================================================
  03/30/2010  12:00:00.000 AM 03/29/2010               03/29/2010

--when expr is not fully specified for unit
select sys_datetime, adddate(sys_datetime, interval '1:20' HOUR_SECOND);

===<Results of SELECT Command in Line ===
    SYS_DATETIME                 date_add( SYS_DATETIME ,         INTERVAL
'1:20'         HOUR_SECOND)
=====================================================================================
    06:18:24.149 PM 06/28/2010   06:19:44.149 PM 06/28/2010
```

ADD_MONTH Function

The **ADD_MONTHS** function adds a *month* value to the expression *date_parameter* of **DATE** type, and it returns a **DATE** type value. If the day (*dd*) of the value specified as the parameter exists within the month of the result value of the operation, it returns the given day (*dd*); otherwise returns the last day of the given month (*dd*). If the result value of the operation exceeds the expression range of the **DATE** type, it returns an error.

Syntax

```
ADD_MONTHS ( date_argument , month )

date_argument :
• date
• NULL

month :
• integer
• NULL
```

- *date_argument* : Specifies an expression of **DATE** type. To specify a **TIMESTAMP** or **DATETIME** value, an explicit casting to **DATE** type is required. If the value is **NULL**, **NULL** is returned.
- *month* : Specifies the number of the months to be added to the *date_argument*. Both positive and negative values can be specified. If the given value is not an integer type, conversion to an integer type by an implicit casting (rounding to the first place after the decimal point) is performed. If the value is **NULL**, **NULL** is returned.

Example

```
--it returns DATE type value by adding month to the first argument

SELECT ADD_MONTHS(DATE '2008-12-25', 5), ADD_MONTHS(DATE '2008-12-25', -5);

======================================================================
  05/25/2009                         07/25/2008


SELECT ADD_MONTHS(DATE '2008-12-31', 5.5), ADD_MONTHS(DATE '2008-12-31', -5.5);

==========================================================================
  06/30/2009                           06/30/2008

SELECT ADD_MONTHS(CAST (SYS_DATETIME AS DATE), 5), ADD_MONTHS(CAST (SYS_TIMESTAMP AS DATE),
5);

===============================================================================
  07/03/2010                                     07/03/2010
```

CURDATE, CURRENT_DATE, CURRENT_DATE(), SYS_DATE, SYSDATE

Description

CURDATE(), **CURRENT_DATE**, **CURRENT_DATE**, **SYS_DATE**, and **SYSDATE** are used interchangeably, and they return the current date as the **DATE** type (*MM*/*DD*/*YYYY* or *YYYY-MM-DD*). The unit is day.

Syntax

```
CURDATE()
CURRENT_DATE()
CURRENT_DATE
SYS_DATE
SYSDATE
```

Example

```
--it returns the current date in DATE type
SELECT CURDATE(), CURRENT_DATE(), CURRENT_DATE, SYS_DATE, SYSDATE;

=== <Result of SELECT Command in Line 1> ===

   SYS_DATE    SYS_DATE    SYS_DATE    SYS_DATE    SYS_DATE
============================================================
  04/01/2010  04/01/2010  04/01/2010  04/01/2010  04/01/2010

--it returns the date 60 days added to the current date
SELECT CURDATE()+60;

=== <Result of SELECT Command in Line 1> ===

   SYS_DATE +60
==============
   05/31/2010
```

CURRENT_DATETIME, CURRENT_DATETIME(), NOW(), SYS_DATETIME, SYSDATETIME

Description

CURRENT_DATETIME, **CURRENT_DATETIME**(), **NOW**() **SYS_DATETIME**, and **SYSDATETIME** are used interchangeably, and they return the current date and time in **DATETIME** type. The unit is millisecond.

Syntax

```
CURRENT_DATETIME
CURRENT_DATETIME()
NOW()
SYS_DATETIME
SYSDATETIME
```

Example

```
--it returns the current date and time in DATETIME type
SELECT NOW(), SYS_DATETIME;

=== <Result of SELECT Command in Line 1> ===

   SYS_DATETIME                   SYS_DATETIME
==============================================================
  04:08:09.829 PM 02/04/2010     04:08:09.829 PM 02/04/2010

--it returns the timestamp value 1 hour added to the current sys_datetime value
SELECT TO_CHAR(SYSDATETIME+3600*1000, 'YYYY-MM-DD HH:MI');

======================
  '2010-02-04 04:08'
```

CURTIME(), CURRENT_TIME, CURRENT_TIME(), SYS_TIME, SYSTIME

Description

CURTIME(), **CURRENT_TIME**, **CURRENT_TIME**(), **SYS_TIME**, and **SYSTIME** are used interchangeably, and they return the current time as the **TIME** type (*HH*:*MI*:SS). The unit is second.

Syntax

```
CURTIME()
CURRENT_TIME
CURRENT_TIME()
SYS_TIME
SYSTIME
```

Example

```
--it returns the current time in TIME type
SELECT CURTIME(), CURRENT_TIME(), CURRENT_TIME, SYS_TIME, SYSTIME;

=== <Result of SELECT Command in Line 1> ===

   SYS_TIME     SYS_TIME     SYS_TIME     SYS_TIME     SYS_TIME
=================================================================
  04:37:34 PM  04:37:34 PM  04:37:34 PM  04:37:34 PM  04:37:34 PM

--it returns the time value 1 hour added to the current sys_time
SELECT CURTIME()+3600;

=== <Result of SELECT Command in Line 1> ===

   SYS_TIME +3600
=================
   05:37:34 PM
```

CURRENT_TIMESTAMP, CURRENT_TIMESTAMP(), SYS_TIMESTAMP, SYSTIMESTAMP

Description

CURRENT_TIMESTAMP, **CURRENT_TIMESTAMP**(), **SYS_TIMESTAMP**, and **SYSTIMESTAMP** are used interchangeably, and they return the current date and time as the **TIMESTAMP** type. LOCALTIME, LOCALTIME(), LOCALTIMESTAMP, LOCALTIMESTAMP() provide the same function. The unit is second.

If you define **DEFAULT** value for column initial value and specify the initial value to **SYS_DATETIME**, the default value is specified to the timestamp at the time of creating a table, not inserting a table. Note that the default value is not specified in case of INSERT. Therefore, you must specify **SYS_DATETIME** in the **VALUES** of **INSERT** statement upon inserting data.

Syntax

```
CURRENT_TIMESTAMP
CURRENT_TIMESTAMP()
SYS_TIMESTAMP
SYSTIMESTAMP
LOCALTIME
LOCALTIME()
LOCALTIMESTAMP
LOCALTIMESTAMP()
```

Example

```
--it returns the current date and time in TIMESTAMP type
SELECT LOCALTIME, SYS_TIMESTAMP
==============================================================================
    07:00:48 PM 04/01/2010         07:00:48 PM 04/01/2010

--it returns the timestamp value 1 hour added to the current sys_timestamp value
SELECT CURRENT_TIMESTAMP()+3600
===========================
    08:02:42 PM 04/01/2010
```

DATE Function

Description

The **DATE** function extracts the date part from the specified argument, and returns it as *MM/DD/YYYY*' format string. Arguments that can be specified are **DATE**, **TIMESTAMP** and **DATETIME** types. The return value is a **VARCHAR** type.

Syntax

```
DATE(date)
```

- *date* : The **DATE**, **TIMESTAMP** or **DATETIME** can be specified.

Example

```
SELECT DATE('2010-02-27 15:10:23');
======================
  '02/27/2010'

SELECT DATE(NOW());
======================
  '04/01/2010'
```

DATEDIFF Function

Description

The **DATEDIFF** function returns the difference between two arguments as an integer representing the number of days. Arguments that can be specified are **DATE**, **TIMESTAMP** and **DATETIME** types and it return value is only **INTEGER** type.

Syntax

```
DATEDIFF (date1,date2)
```

- *date1*, *date2* : The **DATE**, **TIMESTAMP** or **DATETIME** type or date/time format string can be specified. If invalid string is specified, an error is returned.

Example

```
SELECT DATEDIFF('2010-2-28 23:59:59','2010-03-02');
===============================================
                                             -2

SELECT DATEDIFF('2010/12/31', SYSDATETIME);

In the command from line 1,

ERROR: Conversion error in date format.
```

DATE_SUB(), SUBDATE()

Description

DATE_SUB and **SUBDATE**() are used interchangeably, and they perform an addition or subtraction operation on a specific **DATE** value. The return value is a **DATE** or **DATETIME** type. If the date resulting from the operation exceeds the last day of the month, the function returns a valid **DATE** value considering the last date of the month.

Syntax

```
DATE_SUB (date, INTERVAL expr unit)
SUBDATE(date, INTERVAL expr unit)
SUBDATE(date, days)
```

- *date* : It is a **DATE** or **TIMESTAMP** expression that represents the start date. If an invalid **DATE** value such as '2006-07-00' is specified, **NULL** is returned.
- *expr* : It represents the interval value to be subtracted from the start date. If a negative number is specified next to the **INTERVAL** keyword, the interval value is added to the start date.
- *unit* : It represents the unit of the interval value specified in the *exp* expression. To check the expr argument for the unit value, see the table of ADDDATE, DATE_ADD Function.

Example

```
SELECT SYSDATE, SUBDATE(SYSDATE,INTERVAL 24 HOUR), SUBDATE(SYSDATE, 1);

=== <Result of SELECT Command in Line 1> ===

   SYS_DATE    date_sub( SYS_DATE , INTERVAL 24 HOUR)   subdate( SYS_DATE , 1)
```

```
==============================================================================
  03/30/2010  12:00:00.000 AM 03/29/2010               03/29/2010

--it adds days when argument < 0
SELECT SYSDATE, SUBDATE(SYSDATE,INTERVAL -24 HOUR), SUBDATE(SYSDATE, -1);

=== <Result of SELECT Command in Line 1> ===

   SYS_DATE    date_sub( SYS_DATE , INTERVAL -24 HOUR)   subdate( SYS_DATE , -1)
==============================================================================
  03/30/2010  12:00:00.000 AM 03/31/2010               03/31/2010
```

LAST_DAY Function

Description

The **LAST_DAY** function returns the last day of the given month as a **DATE** type.

Syntax

```
LAST_DAY ( date_argument )

date_argument :
• date
• NULL
```

- *date_argument* : Specifies an expression of **DATE** type. To specify a **TIMESTAMP**or **DATETIME** value, explicit casting to **DATE** is required. If the value is **NULL**, **NULL** is returned.

Example

```
--it returns last day of the momth in DATE type
SELECT LAST_DAY(DATE '1980-02-01'), LAST_DAY(DATE '2010-02-01');

============================================================
  02/29/1980                    02/28/2010

--it returns last day of the momth when explicitly casted to DATE type
SELECT LAST_DAY(CAST (SYS_TIMESTAMP AS DATE)), LAST_DAY(CAST (SYS_DATETIME AS DATE));

================================================================================
  02/28/2010                                 02/28/2010
```

LOCALTIME, LOCALTIME(), LOCALTIMESTAMP, LOCALTIMESTAMP()

Description

LOCALTIME, **LOCALTIME**(), **LOCALTIMESTAMP**, and **LOCALTIMESTAMP**() are used interchangeably and they return the current date and time in **TIMESTAMP**. CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP, SYS_TIMESTAMP, SYSTIMESTAMP provide the same function. The unit is second.

Syntax

```
LOCALTIME
LOCALTIME()
LOCALTIMESTAMP
LOCALTIMESTAMP()
CURRENT_TIMESTAMP()
CURRENT_TIMESTAMP
SYS_TIMESTAMP
SYSTIMESTAMP
```

Example

```
--it returns the current date and time in TIMESTAMP type
SELECT LOCALTIME, SYS_TIMESTAMP;
===================================================
  07:00:48 PM 04/01/2010     07:00:48 PM 04/01/2010
```

```
--it returns the timestamp value 1 hour added to the current sys_timestamp value
SELECT CURRENT_TIMESTAMP ()+3600;
===========================
  08:02:42 PM 04/01/2010
```

MONTHS_BETWEEN Function

Description

The **MONTHS_BETWEEN** function returns the difference between the given **DATE** value. The return value is **DOUBLE** type. An integer value is returned if the two dates specified as parameters are identical or are the last day of the given month; otherwise, a value obtained by dividing the day difference by 31 is returned.

Syntax

```
MONTHS_BETWEEN(date_argument, date_argument)

date_argument :
• date
• NULL
```

- *date_argument* : Specifies an expression of **DATE** type. To specify a **TIMESTAMP**or **DATETIME** value, explicit casting to **DATE** is required. If the value is **NULL**, **NULL** is returned.

Example

```
--it returns the negative months when the first argument is the previous date
SELECT MONTHS_BETWEEN(DATE '2008-12-31', DATE '2010-6-30');
======================================================
                               -1.800000000000000e+001

--it returns integer values when each date is the last dat of the month
SELECT MONTHS_BETWEEN(DATE '2010-6-30', DATE '2008-12-31');
======================================================
                                1.800000000000000e+001

--it returns months between two arguments when explicitly casted to DATE type
SELECT MONTHS_BETWEEN(CAST (SYS_TIMESTAMP AS DATE), DATE '2008-12-25');
====================================================================
                                              1.332258064516129e+001

--it returns months between two arguments when explicitly casted to DATE type
SELECT MONTHS_BETWEEN(CAST (SYS_DATETIME AS DATE), DATE '2008-12-25');
===================================================================
                                             1.332258064516129e+001
```

TIMESTAMP Function

Description

The **TIMESTAMP** function converts a **DATE** or **TIMESTAMP** type expression to a **DATETIME** type.

If the **DATA** format string ('YYYY-MM-DD' or 'MM/DD/YYYY) or **TIMESTAMP** format string ('YYYY-MM-DD HH:MI:SS' or 'HH:MI:SS MM/DD/ YYYY') is specified as the first argument, the function returns it as **DATETIME**.

If the **TIME** format string ('HH:MI:SS') is specified as the second, the function adds it to the first argument and returns the result as a **DATETIME** type. If the second argument is not specified, 12:00:00.000 AM is specified by default.

Syntax

```
TIMESTAMP(date [,time])
```

- *date* : The **DATE** or **TIMESTAMP** type string can be specified.
- *time* : The **TIME** type (HH:MI:SS) can be specified.

Example

```
SELECT TIMESTAMP('2009-12-31'), TIMESTAMP('2009-12-31','12:00:00');
```

```
======================================================================
  12:00:00.000 AM 12/31/2009     12:00:00.000 PM 12/31/2009

SELECT TIMESTAMP('2010-12-31 12:00:00','12:00:00');
================================================
  12:00:00.000 AM 01/01/2011

SELECT TIMESTAMP('13:10:30 12/25/2008');
====================================
  01:10:30.000 PM 12/25/2008
```

UNIX_TIMESTAMP Function

Description

The arguments of the **UNIX_TIMESTAMP** function can be omitted. If they are omitted, the function returns the interval between '1970-01-01 00:00:00' UTC and the current system date/time in seconds as a **INTEGER** value. If the date argument is specified, the function returns the interval between '1970-01-01 00:00:00' UTC and the specified date/time in seconds.

Syntax

```
UNIX_TIMESTAMP( [date] )
```

- *date* : **DATE** or **TIMESTAMP** type strings, or strings in the '*YYYYMMDD*' format, can be specified.

Example

```
SELECT UNIX_TIMESTAMP('1970-01-02'), UNIX_TIMESTAMP();

   unix_timestamp('1970-01-02')   unix_timestamp()
===================================================
                            540000         1270196737
```

Data Type Casting Functions and Operators

CAST Operator

Description

The **CAST** operator can be used to explicitly cast one data type to another in the **SELECT** statement. A query list or a value expression in the **WHERE** clause can be cast to another data type.

The following table shows a summary of explicit type conversions (casts) using the **CAST** operator in CUBRID.

	EN	AN	VC	FC	VB	FB	BLOB	CLOB	D	T	UT	DT	S	MS	SQ
EN	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No	No	No	No	No
AN	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No	No	No	No	No
VC	Yes	Yes	Yes[1]	Yes*	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
FC	Yes	Yes	Yes*	Yes*	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
VB	No	No	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No
FB	No	No	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No
BLOB	No	No	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No	No
CLOB	No	No	Yes	Yes	Yes	Yes	No	Yes	No	No	No	No	No	No	No
D	No	No	Yes	Yes	No	No	No	No	Yes	No	Yes	Yes	No	No	No
T	No	No	Yes	Yes	No	No	No	No	No	Yes	No	No	No	No	No
UT	No	No	Yes	Yes	No	No	No	No	Yes	Yes	Yes	Yes	No	No	No

DT	No	No	Yes	Yes	No	No	No	No	Yes	Yes	Yes	Yes	No	No	No
S	No	No	No	No	No	No	No	No	No	No	No	No	Yes	Yes	Yes
MS	No	No	No	No	No	No	No	No	No	No	No	No	Yes	Yes	Yes
SQ	No	No	No	No	No	No	No	No	No	No	No	No	Yes	Yes	Yes

[1] In this case, the **CAST** operation is allowed only when the value expression and the data type to be cast have the same character code set.

Data Type Key

- **EN** : Exact numeric data type (**INTEGER**, **SMALLINT**, **BIGINT, NUMERIC**, **DECIMAL**)
- **AN** : Approximate numeric data type (**FLOAT/REAL**, **DOUBLE PRECISION**, **MONETARY**)
- **VC** : Variable-length character string (**VARCHAR**(*n*), **NCHAR VARYING**(*n*))
- **FC** : Fixed-length character string (**CHAR**(*n*), **NCHAR**(*n*))
- **VB**: Variable-length bit string (**BIT VARYING**(*n*))
- **FB** : Fixed-length bit string (**BIT**(*n*))
- **BLOB** : Binary data that is stored outside DB
- **CLOB** : String data that is stored inside DB
- **D** : Date (**DATE**)
- **T** : Time (**TIME**)
- **UT** : Timestamp (**TIMESTAMP**)
- **S** : Set (**SET**)
- **MS** : Multiset (**MULTISET**)
- **SQ** : Sequence set (**LIST**, **SEQUENCE**)

Syntax

```
CAST (cast_operand AS cast_target)

cast_operand :
• value expression
• NULL

cast_target :
• data type
```

- *cast_operand* : Declares the value to cast to a different data type.
- *cast_target* : Specifies the type to cast to.

Example

The following is an example of explicitly casting and returning a **VARCHAR** record in kg unit to a **FLOAT**.

```

--operation after casting character as INT type returns 2
SELECT (1+CAST ('1' AS INT));
===========================
                          2

--cannot cast the string which is out of range as SMALLINT
SELECT (1+CAST('1234567890' AS SMALLINT));

ERROR

--operation after casting returns 1+1234567890
SELECT (1+CAST('1234567890' AS INT));
====================================
                          1234567891

--'1234.567890' is casted to 1235 after rounding up
SELECT (1+CAST('1234.567890' AS INT));
```

```
--'1234.567890' is casted to string containing only first 5 letters.
SELECT (CAST('1234.567890' AS CHAR(5)));
======================
  '1234.'

--numeric type can be casted to CHAR type only when enough length is specified
SELECT (CAST(1234.567890 AS CHAR(5)));

ERROR

--numeric type can be casted to CHAR type only when enough length is specified
SELECT (CAST(1234.567890 AS CHAR(11)));
======================
  '1234.567890'

--numeric type can be casted to CHAR type only when enough length is specified
SELECT (CAST(1234.567890 AS VARCHAR));
======================
  '1234.567890'

--string can be casted to time/date types only when its literal is correctly specified
SELECT (CAST('2008-12-25 10:30:20' AS TIMESTAMP));
=============================================
  10:30:20 AM 12/25/2008

SELECT (CAST('10:30:20' AS TIME));
==================================================
  10:30:20 AM

--string can be casted to TIME type when its literal is same as TIME’s.
SELECT (CAST('2008-12-25 10:30:20' AS TIME));

ERROR

--string can be casted to TIME type after specifying its type of the string
SELECT (CAST(TIMESTAMP'2008-12-25 10:30:20' AS TIME));
==================================================
  10:30:20 AM

SELECT CAST('abcde' AS BLOB);
======================
file:/home1/user1/db/tdb/lob/ces_743/ces_temp.00001283232024309172_1342
```

Remark

- **CAST** is allowed only between data types having the same character set.
- If you cast an approximate data type to integer type, the number is rounded to zero decimal places.
- If you cast a numeric data type to string character type, it should be longer than the length of significant digits + decimal point. An error occurs otherwise.
- If you cast a character string type A to a character string type B, B should be longer than the A. The end of character string is truncated otherwise.
- If you cast a character string type A to a date-time date type B, it is converted only when literal of A and B type match one another. An error occurs otherwise.
- You must explicitly do type casting for numeric data stored in a character string so that an arithmetic operation can be performed.

DATE_FORMAT Function

Description

The **DATE_FORMAT** function converts the value of strings with **DATE** format ('*YYYY-MM-DD*' or '*MM*/*DD*/*YYYY*') or that of date/time data type (**DATE**, **TIMESTAMP**, **DATETIME**) to specified date/time format and then return the value with the **VARCHAR** data type.

Syntax

```
DATE_FORMAT(date, format)
```

- *date* : A value of strings with the **DATE** format ('*YYYY-MM-DD*' or '*MM/DD/YYYY*') or that of date/time data type (**DATE**, **TIMESTAMP**, **DATETIME**) can be specified .
- *format* : Specifies the output format. Use a string that contains '%' as a specifier. See the following table to specify the format. Date/Time formats described in the following Date/Time Format 2 table are used in **DATE_FORMAT** function, and TIME_FORMAT Function, and STR_TO_DATE Function.

Default Date/Time Format

Date/Time Type	Default Output Format
DATE	'MM/DD/YYYY'
TIME	'HH:MI:SS AM'
TIMESTAMP	'HH:MI:SS AM MM/DD/YYYY'
DATETIME	'HH:MI:SS.FF AM MM/DD/YYYY

Date/Time Format 2

format Value	Meaning
%a	Weekday, English abbreviation (Sun, ..., Sat)
%b	Month, English abbreviation (Jan, ..., Dec)
%c	Month (1, ..., 12)
%D	Day of the month, English ordinal number (1st, 2nd, 3rd, ...)
%d	Day of the month, two-digit number (01, ..., 31)
%e	Day of the month (1, ..., 31)
%f	Microseconds, three-digit number (000, ..., 999)
%H	Hour, 24-hour based, number with at least two--digit (00, ..., 23, ..., 100, ...)
%h	Hour, 12-hour based two-digit number (01, ..., 12)
%I	Hour, 12-hour based two-digit number (01, ..., 12)
%i	Minutes, two-digit number (00, ..., 59)
%j	Day of year, three-digit number (001, ..., 366)
%k	Hour, 24-hour based, number with at least one-digit (0, ..., 23, ..., 100, ...)
%l	Hour, 12-hour based (1, ..., 12)
%M	Month, English string (January, ..., December)
%m	Month, two-digit number (01, ..., 12)
%p	AM or PM
%r	Time, 12-hour based, hour:minute:second (hh:mm:ss AM or hh:mm:ss PM)
%S	Seconds, two-digit number (00, ..., 59)
%s	Seconds, two-digit number (00, ..., 59)
%T	Time, 24-hour based, hour:minute:second (hh:mm:ss)
%U	Week, two-digit number, week number of the year with Sunday being the first day Week

	(00, ..., 53)
%u	Week, two-digit number, week number of the year with Monday being the first day (00, ..., 53)
%V	Week, two-digit number, week number of the year with Sunday being the first day Week (00, ..., 53) (Available to use in combination with %X)
%v	Week, two-digit number, week number of the year with Monday being the first day (00, ..., 53) (Available to use in combination with %X)
%W	Weekday, English string (Sunday, ..., Saturday)
%w	Day of the week, number index (0=Sunday, ..., 6=Saturday)
%X	Year, four-digit number calculated as the week number with Sunday being the first day of the week (0000, ..., 9999) (Available to use in combination with %V)
%x	Year, four-digit number calculated as the week number with Monday being the first day of the week (0000, ..., 9999) (Available to use in combination with %V)
%Y	Year, four-digit number (0001, ..., 9999)
%y	Year, two-digit number (00, 01, ..., 99)
%%	Output the special character "%" as a string
%x	Output an arbitrary character x as a string out of English letters that are not used as format specifiers.

Example

```
SELECT DATE_FORMAT('2009-10-04 22:23:00', '%W %M %Y');
======================
  'Sunday October 2009'


SELECT DATE_FORMAT('2007-10-04 22:23:00', '%H:%i:%s');
======================
  '22:23:00'

SELECT DATE_FORMAT('1900-10-04 22:23:00', '%D %y %a %d %m %b %j');
======================
  '4th 00 Thu 04 10 Oct 277'


SELECT DATE_FORMAT('1999-01-01', '%X %V');
======================
  '1998 52'
```

STR_TO_DATE Function

Description

The **STR_TO_DATE** function interprets a string given as a parameter according to the specified format and converts it to a Date/Time value. This function works in the opposite manner as the DATE_FORMAT function. The type of the return value is determined according to the Date or Time part included in the string. The return type can be **DATETIME**, **DATE** or **TIME**. If an invalid value is included in the string, or if the string cannot be interpreted by applying the format specifier defined in the format, **STR_TO_DATE()** returns **NULL**.

Syntax

```
STR_TO_DATE(string, format)
```

- *string* : All string types can be specified.
- *format* : Specifies format that is used to interpreter strings. The string format that contains '%' is used as a specifier. Form more information, see the table of Date/Time Format 2.

Example

```
SELECT STR_TO_DATE('01,5,2013','%d,%m,%Y');
=======================================
  05/01/2013

SELECT STR_TO_DATE('May 1, 2013','%M %d,%Y');
=========================================
  05/01/2013


SELECT STR_TO_DATE('a09:30:17','a%h:%i:%s');
========================================
  09:30:17 AM

SELECT STR_TO_DATE('09:30:17a','%h:%i:%s');
=======================================
  09:30:17 AM
```

TIME_FORMAT Function

Description

The **TIME_FORMAT** function converts the value of strings with **TIME** format ('*HH-MI-SS)* or that of date/time data type (**DATE**, **TIMESTAMP**, **DATETIME**) to specified date/time format and then return the value with the **VARCHAR** data type.

Syntax

```
TIME_FORMAT(time, format)
```

- *time* : A value of string with **TIME** (*HH*:*MI*:*SS*) or that of date/time data type ((**DATE**, **TIMESTAMP**, **DATETIME**) an be specified.
- *format* : Specifies the output format. Use a string that contains '%' as a specifier. See the table of the Date/Time Format 2 table. If un-related format specifier is used, the English letters themselves are displayed.

Example

```
SELECT TIME_FORMAT('22:23:00', '%H %i %s');
======================
  '22 23 00'

SELECT TIME_FORMAT('25:59:00', '%H %h %i %s %f');
======================
  '25 01 59 00 000'

SELECT SYSTIME, TIME_FORMAT(SYSTIME, '%T');
===================================
  08:46:53 PM  '20:46:53'
```

TO_CHAR Function (date_time)

Description

The **TO_CHAR** function converts the value of strings with **TIME** format (*HH*:*MI*:*SS*) or that of date/time type **(TIME**, **TIMESTAMP**, **DATETIME**) by Date/Time Format 1 and then return the value with the **VARCHAR** data type. If a format argument is not specified, it converts the value based by default format. If a format which is not corresponding to the given value, an error is returned.

Syntax

```
TO_CHAR( date_time [, format [, date_lang_string_literal ]] )

date_time :
• date
• time
• timestamp
• datetime
• NULL

format :
• character strings (see Date/Time Format 1 )
• NULL

date_lang_string_literal : (see date_lang_string_literal)

• 'en_US'
• 'ko_KR'
```

- *date_time* : Specifies an expression that returns date-time type string. If the value is **NULL**, **NULL** is returned.
- *format* : Specifies a format of return value. If a format is not specified, the default format is used. If the value is **NULL**, **NULL** is returned.
- *date_lang_string_literal* : Specifies a language applied to a return value (refer to date_lang_string_literal). The default value is 'en_US'. You can modify the value by specifying the **CUBRID_DATE_LANG** environment variable.

Default Date/Time Format

Date/Time Type	Default Output Format
DATE	'MM/DD/YYYY'
TIME	'HH:MI:SS AM'
TIMESTAMP	'HH:MI:SS AM MM/DD/YYYY'
DATETIME	'HH:MI:SS.FF AM MM/DD/YYYY'

Date/Time Format 1

Format Element	Description
CC	Century
YYYY, **YY**	Year with 4 numbers, Year with 2 numbers
Q	Quarter (1, 2, 3, 4; January - March = 1)
MM	Month (01-12; January = 01) Note : MI represents the minute of hour.
MONTH	Month in characters
MON	Abbreviated month name
DD	Day (1 - 31)
DAY	Day of the week in characters
DY	Abbreviated day of the week
D or **d**	Day of the week in numbers (1 - 7)
AM or **PM**	AM/PM
A.M. or **P.M.**	AM/PM with periods
HH or **HH12**	Hour (1 -12)
HH24	Hour (0 - 23)
MI	Minute (0 - 59)
SS	Second (0 - 59)

FF	Millsecond (0-999)
- / , . ; : "text"	Punctuation and quotation marks are represented as they are in the result

Example of date_lang_string_literal

Format Element	**Date_lang_string_literal**	
	'en_US'	**'ko_KR'**
MONTH	JANUARY	1 월
MON	JAN	1
DAY	MONDAY	월요일
DY	MON	월
Month	January	1 월
Mon	Jan	1
Day	Monday	월요일
Dy	Mon	월
month	january	1 월
mon	jan	1
day	monday	월요일
Dy	mon	월
AM	AM	오전
Am	Am	오전
am	am	오전
A.M.	A.M.	오전
A.m.	A.m.	오전
a.m.	a.m.	오전
PM	AM	오전
Pm	Am	오전
pm	am	오전
P.M.	A.M.	오전
P.m.	A.m.	오전
p.m.	a.m	오전

The Number of Digits Format

Format Element	**Number of Digits**
MONTH(Month, month)	9 (ko_KR : 4)
MON(Mon, mon)	3 (ko_KR : 2)

DAY(Day, day)	9 (ko_KR : 6)
DY(Dy, dy)	3 (ko_KR : 2)
HH12, HH24	2
"text"	The length of the text
Other formats	Same as the length of the format

Example

```
--creating a table having date/time type columns
CREATE TABLE datetime_tbl(a TIME, b DATE, c TIMESTAMP, d DATETIME);
INSERT INTO datetime_tbl VALUES(SYSTIME, SYSDATE, SYSTIMESTAMP, SYSDATETIME);

--selecting a VARCHAR type string from the data in the specified format
SELECT TO_CHAR(b, 'DD, DY , MON, YYYY') FROM datetime_tbl;
======================
  '04, THU , FEB, 2010'

SELECT TO_CHAR(c, 'HH24:MI, DD, MONTH, YYYY') FROM datetime_tbl;
======================
  '16:50, 04, FEBRUARY , 2010'

SELECT TO_CHAR(c, 'HH24:MI:FF, DD, MONTH, YYYY') FROM datetime_tbl;
ERROR: Invalid format.

SELECT TO_CHAR(d, 'HH12:MI:SS:FF pm, YYYY-MM-DD-DAY') FROM datetime_tbl;
======================
  '04:50:11:624 pm, 2010-02-04-THURSDAY '


SELECT TO_CHAR(TIMESTAMP'2009-10-04 22:23:00', 'Day Month yyyy');
======================
  'Sunday October 2009'
```

TO_CHAR Function (number)

Description

The **TO_CHAR** function converts a Number Format or numeric data type to a character string according to the number format and returns it. The type of the return value is **VARCHAR**. If the number format has not been specified as an argument, all significant digits are converted to a character string according to the default format.

Syntax

```
TO_CHAR(number_argument[, format_argument ])

number_argument :
• numeric(decimal)
• integer
• smallint
• bigint
• float(real)
• double
• NULL

format_argument :
• character strings (see Number Format)
• NULL
```

- *number_argument* : Specifies an expression that returns numeric data type string. If the value is **NULL**, **NULL** is returned.
- *format_argument* : Specifies a format of return value. If format is not specified, all significant digits are returned as character string by default. If the value is **NULL**, **NULL** is returned.

Number Format

Format Element	**Example**	**Description**

9	9999	The number of 9's represents the number of significant digits to be returned. If the number of significant digits specified in the format is not sufficient, only the decimal part is rounded. If it is less than the number of digits in an integer, # is outputted. If the number of significant digits specified in the format is sufficient, the part preceding the integer part is filled with space characters and the decimal part is filled with 0.
0	0999	If the number of significant digits specified in the format is sufficient, the part preceding the integer part is filled with 0, not space characers before the value is returned.
S	S9999	Outputs the negative/positive sign in the specified position. These signs can be used only at the beginning of character string.
C	C9999	Returns the ISO currency code at the specified position.
,(comma)	9,999	Returns a comma (",") at the specified position. Multiple commas are allowed in the format.
.(percimal point)	9.999	Outputs the decimal point (".") that distinguishes the integer and the decimal part at a specified position. Only one decimal point is allowed in the format.
EEEE	9.99EEEE	Returns a scientific notation number.

Example

```
--selecting a string casted from a number in the specified format
SELECT TO_CHAR(12345,'S999999'), TO_CHAR(12345,'S099999');

============================================
  ' +12345'             '+012345'


SELECT TO_CHAR(1234567,'C9,999,999,999');
======================
  '    $1,234,567'

SELECT TO_CHAR(123.4567,'99'), TO_CHAR(123.4567,'99999.999');
=============================================================
  '##'                     '123.45670'            '  123.457'


SELECT TO_CHAR(1.234567,'99.999EEEE'), TO_CHAR(1.234567E-4);

============================================
  '1.235E+00'           '0.0001234567'
```

TO_DATE Function

Description

The **TO_DATE** function interprets a character string based on the date format given as an argument, converts it to a **DATE** type value, and returns it. For the format, see TO_CHAR Function (date_time). If a format is not specified, the "MM/DD/YYYY" format is applied by default.

Syntax

```
TO_DATE(string_argument[,format_argument[,date_lang_string_literal]])

string_argument :
• character strings
• NULL

format_argument :
• character strings (see Date/Time Format 1)
• NULL
```

```
date_lang_string_literal : (see date_lang_string_literal)
• 'en_US'
• 'ko_KR'
```

- *string_argument* : Specifies an expression that returns character string. If the value is **NULL**, **NULL** is returned.
- *format_argument* : Specifies a format of return value to be converted as **DATE** type. See the "Default Date-Time Format" table of TO_CHAR Function (date_time). If the value is **NULL**, **NULL** is returned.
- *date_lang_string_literal* : Specifies the language for the input value to be applied. You can modify the value by using the **CUBRID_DATE_LANG** environment.

Example

```
--selecting a date type value casted from a string in the specified format

SELECT TO_DATE('12/25/2008');
===============================================
  12/25/2008

SELECT TO_DATE('25/12/2008', 'DD/MM/YYYY');
===============================================
  12/25/2008

SELECT TO_DATE('081225', 'YYMMDD');
===============================================
  12/25/2008

SELECT TO_DATE('2008-12-25', 'YYYY-MM-DD');
===============================================
  12/25/2008
```

TO_DATETIME Function

Description

The **TO_DATETIME** function interprets a character string based on the date-time format given as an argument, converts it to a **DATETIME** type value, and returns it. For the format, see TO_CHAR Function (date_time). If format is not specified, the "HH:MI:SS.FF [am|pm] MM/DD/YYYY" format is applied by default.

Syntax

```
TO_DATETIME(string_argument[,format_argument[,date_lang_string_literal]])

string_argument :
• character strings
• NULL

format_argument :
• character strings (see the table Date/Time_Format_1)
• NULL

date_lang_string_literal : (see the table Example_of_date_lang_string_literal)
• 'en_US'
• 'ko_KR'
```

- *string_argument* : Specifies an expression that returns character string. If the value is **NULL**, **NULL** is returned.
- *format_argument* : Specifies a format of return value to be converted as **DATETIME** type. See the "Default Date-Time Format" table of TO_CHAR Function (date_time). If the value is **NULL**, **NULL** is returned.
- *date_lang_string_literal* : Specifies the language for the input value to be applied. You can modify the value by using the **CUBRID_DATE_LANG** environment.

Example

```
--selecting a datetime type value casted from a string in the specified format

SELECT TO_DATETIME('13:10:30 12/25/2008');
=====================================
  01:10:30.000 PM 12/25/2008
```

```
SELECT TO_DATETIME('08-Dec-25 13:10:30.999', 'YY-Mon-DD HH24:MI:SS.FF');
=====================================
  01:10:30.999 PM 12/25/2008

SELECT TO_DATETIME('DATE: 12-25-2008 TIME: 13:10:30.999', '"DATE:" MM-DD-YYYY "TIME:"
HH24:MI:SS.FF');
=====================================
  01:10:30.999 PM 12/25/2008
```

TO_NUMBER Function

Description

The **TO_NUMBER** function interprets a character string based on the number format given as an argument, converts it to a **NUMERIC** type value, and returns it. If the number format is not specified, returns all significant digits that are included in the character string as **NUMERIC** type numbers by default.

Syntax

```
TO_NUMBER(string_argument[, format_argument ])

string_argument :
• character strings
• NULL

format_argument :
• character strings
• NULL
```

- *string_argument* : Specifies an expression that returns character string. If the value is **NULL**, **NULL** is returned.
- *format_argument* : Specifies a format of return value to be converted as **NUMBER** type. See the "Number Format" table of TO_CHAR Function (number). If the value is **NULL**, an error is returned.

Example

```
--selecting a number casted from a string in the specified format
SELECT TO_NUMBER('-1234');
============================================
  -1234


SELECT TO_NUMBER('12345','999999');

============================================
  12345


SELECT TO_NUMBER('$12,345.67','C99,999.999');
======================
  12345.670


SELECT TO_NUMBER('12345.67','99999.999');
============================================
  12345.670
```

TO_TIME Function

Description

The **TO_TIME** function interprets a character string based on the time format given as an argument, converts it to a **TIME** type value, and returns it. For the format, see TO_CHAR Function (date_time). If a format is not specified, the "HH:MI:SS" format is applied by default.

Syntax

```
TO_TIME(string_argument[,format_argument [,date_lang_string_literal]]):
```

```
string_argument :
• character strings
• NULL

format_argument :
• character strings (refer to Date/Time Format 1)
• NULL

date_lang_string_literal : (refer to date_lang_string_literal)
• 'en_US'
• 'ko_KR'
```

- *string_argument* : Specifies an expression that returns character string. If the value is **NULL**, **NULL** is returned.
- *format_argument* : Specifies a format of return value to be converted as **TIME** type. See the "Default Date-Time Format" table of TO_CHAR Function (date_time). If the value is **NULL**, **NULL** is returned.
- *date_lang_string_literal* : Specifies the language for the input value to be applied. You can modify the value by using the **CUBRID_DATE_LANG** environment.

Example

```
--selecting a time type value casted from a string in the specified format

SELECT TO_TIME ('13:10:30');
=============================================
  01:10:30 PM

SELECT TO_TIME('HOUR: 13 MINUTE: 10 SECOND: 30', '"HOUR:" HH24 "MINUTE:" MI "SECOND:" SS');
=============================================
  01:10:30 PM

SELECT TO_TIME ('13:10:30', 'HH24:MI:SS');
=============================================
  01:10:30 PM

SELECT TO_TIME ('13:10:30', 'HH12:MI:SS');
ERROR //a string and the format not matched
```

TO_TIMESTAMP Function

Description

The **TO_TIMESTAMP** function interprets a character string based on the time format given as an argument, converts it to a **TIMESTAMP** type value, and returns it. For the format, see TO_CHAR Function (date_time). If a format is not specified, the "HH:MI[:SS] [am|pm] MM/DD/YYYY" format is applied by default.

Syntax

```
TO_TIMESTAMP(string_argument[, format_argument[,date_lang_string_literal]])

string_argument :
• character strings
• NULL

format_argument :
• character strings (refer to Date/Time Format 1 table)
• NULL

date_lang_string_literal : (refer to date_lang_string_literal table)
• 'en_US'
• 'ko_KR'
```

- *string_argument* : Specifies an expression that returns character string. If the value is **NULL**, **NULL** is returned.
- *format_argument* : Specifies a format of return value to be converted as **TIMESTAMP** type. See the "Default Date-Time Format" table of TO_CHAR Function (date_time). If the value is **NULL**, **NULL** is returned.
- *date_lang_string_literal* : Specifies the language for the input value to be applied. You can modify the value by using the **CUBRID_DATE_LANG** environment.

Example

```
--selecting a timestamp type value casted from a string in the specified format

SELECT TO_TIMESTAMP('13:10:30 12/25/2008');
======================================
  01:10:30 PM 12/25/2008

SELECT TO_TIMESTAMP('08-Dec-25 13:10:30', 'YY-Mon-DD HH24:MI:SS');
======================================
  01:10:30 PM 12/25/2008

SELECT TO_TIMESTAMP('YEAR: 2008 DATE: 12-25 TIME: 13:10:30', '"YEAR:" YYYY "DATE:" MM-DD
"TIME:" HH24:MI:SS');
======================================
  01:10:30 PM 12/25/2008
```

Aggregate Functions

AVG Function

Description

The **AVG** function calculates the arithmetic average of the value of an expression representing all rows. Only one *expression* is specified as a parameter. You can get the average without duplicates by using the **DISTINCT** or **UNIQUE** keyword in front of the expression or the average of all values by omitting the keyword or by using **ALL**.

Syntax

```
AVG ( [ { DISTINCT | DISTINCTROW } | UNIQUE | ALL ] expression )
```

- *expression* : Specifies an expression that returns a numeric value. A collection expression cannot be specified.
- **ALL** : Calculates an average value for all data (default).
- **DISTINCT** or **UNIQUE** : Calculates an average value without duplicates.

Example

The following is an example that returns the average number of gold medals Korea won in Olympics. (demodb)

```
SELECT AVG(gold)
FROM participant
WHERE nation_code = 'KOR';
```

Result value : 9

COUNT Function

Description

The **COUNT** function returns the number of of rows returned by a query. If an asterisk (*) is specified, the number of all rows satisfying the condition (including the rows with the **NULL** value) is returned. If the **DISTINCT** or **UNIQUE** keyword is specified in front of the expression, only the number of rows that have a unique value (excluding the rows with the **NULL** value) is returned after duplicates have been removed. Therefore, the value returned is always an integer and **NULL** is never returned.

A column that has collection type and object domain (user-defined class or multimedia class) can also be specified in the *expression*.

Syntax

```
COUNT ( * | [ { DISTINCT | DISTINCTROW } | UNIQUE | ALL ] expression )
```

- *expression* : Specifies an expression.
- **ALL** : Gets the number of rows given in the *expression* (default).
- **DISTINCT** or **UNIQUE** : Gets the number of rows without duplicates.

Example

The following is an example that returns the number of Olympic Games that had a mascot. (demodb)

```
SELECT COUNT(*)
FROM olympic
WHERE mascot IS NOT NULL;
```

Result value : 9

MAX Function

Description

The **MAX** function gets the greatest value of expressions of all rows. Only one *expression* is specified.

For expressions that return character strings, the string that appears later in alphabetical order becomes the maximum value; for those that return numbers, the greatest value becomes the maximum value.

Syntax

```
MAX ( [ { { DISTINCT | DISTINCTROW } | UNIQUE | ALL ] expression )
```

- *expression* : Specifies an expression that returns a numeric or string value. A collection expression cannot be specified.
- **ALL** : Gets the maximum value for all data (default).
- **DISTINCT** or **UNIQUE** : Gets the maximum value without duplicates.

Example

The following is an example that returns the maximum number of gold medals Korea won in the Olympics. (demodb)

```
SELECT MAX(gold) FROM participant WHERE nation_code = 'KOR';

=== <Result of SELECT Command in Line 1> ===

    max(gold)
=============
           12
```

MIN Function

Description

The **MIN** function gets the smallest value of expressions of all rows. Only one *expression* is specified.

For expressions that return character strings, the string that appears earlier in alphabetical order becomes the minimum value; for those that return numbers, the smallest value becomes the minimum value.

Syntax

```
MIN ( [ { DISTINCT | DISTINCTROW } | UNIQUE | ALL ] expression )
```

- *expression* : Specifies an expression that returns a numeric or string value. A collection expression cannot be specified.
- **ALL** : Gets the minimum value for all data (default).
- **DISTINCT** or **UNIQUE** : Gets the maximum value without duplicates.

Example

The following is an example that returns the minimum number of gold medals Korea won in the Olympics. (demodb)

```
SELECT MIN(gold) FROM participant WHERE nation_code = 'KOR';

=== <Result of SELECT Command in Line 1> ===

    min(gold)
=============
            7
```

STDDEV Function

Description

The **STDDEV** function returns a standard deviation of the expression values of all rows. Only one *expression* is specified as a parameter. You can get the standard deviation without duplicates by inserting the **DISTINCT** or **UNIQUE** keyword in front of the expression, or get the standard deviation of all values by omitting the keyword or by using **ALL**.

The return value may be different from the actual evaluation value because it follows the type of the expression specified as a parameter.

Syntax

```
STDDEV( [ { DISTINCT | DISTINCTROW } | UNIQUE | ALL] expression )
```

- *expression* : Specifies an expression that returns a numeric value.
- **ALL** : Calculates the standard deviation for all data (default).
- **DISTINCT** or **UNIQUE** : Calculates the standard deviation without duplicates.

Example

The following is an example that returns the standard deviation of gold medals Korea won in the Olympics. (demodb)

```
SELECT host_year, gold FROM participant WHERE nation_code = 'KOR';

=== <Result of SELECT Command in Line 1> ===

    host_year         gold
==========================
         2004            9
         2000            8
         1996            7
         1992           12
         1988           12

SELECT STDDEV(gold), STDDEV(CAST (gold AS FLOAT)) FROM participant
WHERE nation_code = 'KOR';

=== <Result of SELECT Command in Line 1> ===

  stddev(gold)  stddev( cast(gold as float))
============================================
             2                 2.302172e+000
```

SUM Function

Description

The **SUM** function returns the sum of expressions of all rows. Only one *expression* is specified as a parameter. You can get the sum without duplicates by inserting the **DISTINCT** or **UNIQUE** keyword in front of the expression, or get the sum of all values by omitting the keyword or by using **ALL**.

Syntax

```
SUM ( [ { DISTINCT | DISTINCTROW } | UNIQUE | ALL ] expression )
```

You can specify a single-value expression as a input for **SUM** function.

- *expression* : Specifies an expression that returns a numeric value.
- **ALL** : Gets the sum for all data (default).
- **DISTINCT** or **UNIQUE** : Gets the sum of unique values without duplicates

Example

The following is an example that outputs the top 10 countries and the total number of gold medals based on the sum of gold medals won in the Olympics. (demodb)

```
SELECT nation_code, SUM(gold) FROM participant GROUP BY nation_code
ORDER BY SUM(gold) DESC
FOR ORDERBY_NUM() BETWEEN 1 AND 10 ;

=== <Result of SELECT Command in Line 1> ===

  nation_code             sum(gold)
===================================
  'USA'                         190
  'CHN'                          97
  'RUS'                          85
  'GER'                          79
  'URS'                          55
  'FRA'                          53
  'AUS'                          52
  'ITA'                          48
  'KOR'                          48
  'EUN'                          45


10 rows selected.
```

VARIANCE Function

Description

The **VARIANCE** function returns a variance of expression values of all rows. Only one *expression* is specified as a parameter. You can get the variance without duplicates by using the **DISTINCT** or **UNIQUE** keyword in front of the expression or the variance of all values by omitting the keyword or by using **ALL**.

The return value may be different from the actual evaluation value because it follows the type of the expression specified as a parameter.

The following is a formula that is applied to the function.

$$\frac{\left[\mathrm{SUM}(x^2) - \frac{(\mathrm{SUM}(x))^2}{n}\right]}{(n-1)}$$

Syntax

```
VARIANCE( [DISTINCT | UNIQUE | ALL] expression )
```

- *expression* : Specifies an expression that returns a numeric value.
- **ALL** : Gets the variance for all values (default).
- **DISTINCT** or **UNIQUE** : Gets the variance of unique values without duplicates.

Example

The following is an example that returns the variance of the number of gold medals Korea won from 1988 to 2004 in the Olympics. (demodb)

```
SELECT VARIANCE(gold), VARIANCE(CAST (gold AS FLOAT)) FROM participant
WHERE nation_code = 'KOR';
=== <Result of SELECT Command in Line 1> ===

  variance(gold)  variance( cast(gold as float))
================================================
               5                   5.299995e+000
```

Click Counter Functions

INCR, DECR Function

Description

The **INCR** function increments the column's value given as a parameter for a SELECT statement by 1. The **DECR** function decrements the value of the column by 1.

Syntax

```
SELECT [ qualifier ] select_expression
[ { TO | INTO } variable [ {, variable }...; ] ]
...;
select_expression :
 *
 table_name. *
 [expression | counter_expression] [ {, expression |
counter_expression}...]

counter_expression :
INCR(path_expression)
```

The **INCR** and **DECR** functions are called "click counters" and can be effectively used to increase the number of post views for a Bulletin Board System (BBS) type of web service. In a scenario where you want to SELECT a post and immediately increase the number of views by 1 using an **UPDATE** statement, you can view the post and increment the number at the same time by using the **INCR** function in a single **SELECT** statement.

The **INCR** function increments the column value specified as an argument. Only integer type numbers can be used as arguments. If the value is **NULL**, the **INCR** function returns the **NULL**. That is, a value must be valid in order to be incremented by the **INCR** function. The **DECR** function decrements the column value specified as a parameter.

If an **INCR** function is specified in the SELECT statement, the **COUNTER** value is incremented by 1 and the query result is displayed with the values before the increment. Furthermore, the **INCR** function does not increment the value of the tuple affected by the query process but rather the one affected by the final result.

Remark

- The **INCR/DECR** function executes independent of user-defined transactions and is applied automatically to the database by the top operation internally used in the system, apart from the transaction's **COMMIT/ROLLBACK**.
- When multiple **INCR/DECR** functions are specified in a single SELECT statement, the failure of any of the **INCR/DECR** functions leads to the failure of all of them.
- The **INCR/DECR** functions apply only to top-level SELECT statements. **SUB SELECT** statements such as **INSERT** ... **SELECT** ... statement and **UPDATE** table **SET** col = **SELECT** ... statement are not supported. The following is an example where the **INCR** function is not allowed.

  ```
  SELECT b.content, INCR(b.read_count) FROM (SELECT * FROM board WHERE id = 1) AS b
  ```

- If the **SELECT** statement with **INCR/DECR** function(s) returns more than one row as a result, it is treated as an error. The final result must have only one row to be considered valid.
- The **INCR/DECR** function can be used only in numerical domains. Applicable domains are limited to integer data types such as **SMALLINT** and **INTEGER**. They cannot be used in other domains.
- When the **INCR** function is called, the value to be returned will be the current value, while the value to be stored will be the current value + 1. Execute the following statement to select the value to be stored as the result :

  ```
  SELECT content, INCR(read_count) + 1 FROM board WHERE id = 1;
  ```

- If the defined maximum value of the domain is exceeded, the **INCR** function initializes the column value to 0. Likewise, the column value is also initialized to 0 when the **DECR** function applies to the minimum value.
- Data inconsistency can occur because the **INCR/DECR** functions are executed regardless of **UPDATE** trigger. The example below shows the database inconsistency in that situation.

  ```
  CREATE TRIGGER event_tr BEFORE UPDATE ON event EXECUTE REJECT;
  SELECT INCR(players) FROM event WHERE gender='M';
  ```

Example

Suppose that the following three rows of data were inserted into the 'board' table.

```
CREATE TABLE board (
id  INT, title  VARCHAR(100), content  VARCHAR(4000), read_count  INT );
INSERT INTO board VALUES (1, 'aaa', 'text...', 0);
INSERT INTO board VALUES (2, 'bbb', 'text...', 0);
INSERT INTO board VALUES (3, 'ccc', 'text...', 0);
```

The following is an example of incrementing the value of the 'read_count' column in a data whose 'id' value is 1 using the **INCR** function.

```
SELECT content, INCR(read_count) FROM board WHERE id = 1;
=== <Result of SELECT Command in Line 1> ===

  content                read_count
===================================
  'text...'                       0
```

In the example, the column value becomes read_count + 1 as a result of the **INCR** function in the **SELECT** statement. You can check the result using the following **SELECT** statement.

```
SELECT content, read_count FROM board WHERE id = 1;
=== <Result of SELECT Command in Line 1> ===

  content                read_count
===================================
  'text...'                       1
```

ROWNUM Functions

ROWNUM/INST_NUM() Function

Description

The **ROWNUM** function returns the number representing the order of the records that will be generated by the query result. The first result record is assigned 1, and the second result record is assigned 2.

ROWNUM and **INST_NUM()** can be used in the **SELECT** statement, and **GROUPBY_NUM()** can be used in the **SELECT** statement with **GROUP BY** clauses. The **ROWNUM** function can be used to limit the number of result records of the query in several ways. For example, it can be used to search only the first 10 records or to return even or odd number records.

The **ROWNUM** function has a result value as an integer, and can be used wherever an expression is valid such as the **SELECT** or **WHERE** clause. However, it is not allowed to compare the result of the **ROWNUM** function with the attribute or the correlated subquery.

Syntax

```
INST_NUM()
ROWNUM
```

Remark

- The **ROWNUM** function specified in the **WHERE** clause works the same as the **INST_NUM()** function. Whereas **INST_NUM()** is a scalar function, **GROUPBY_NUM()** is a kind of an aggregate function. In a **SELECT** statement with a **GROUP BY** clause, **GROUPBY_NUM()** must be used instead of **INST_NUM()**.
- The **ROWNUM** function belongs to each **SELECT** statement. That is, if a **ROWNUM** function is used in a subquery, it returns the sequence of the subquery result while it is being executed. Internally, the result of the **ROWNUM** function is generated right before the searched record is written to the query result set. At this moment, the counter value that generates the serial number of the result set records increases.
- If an **ORDER BY** clause is included in the **SELECT** statement, the value of the **ROWNUM** function specified in the **WHERE** clause is generated before sorting for the **ORDER BY** clause. If a **GROUP BY** clause is included in the **SELECT** statement, the value of the **GROUPBY_NUM()** function specified in the **HAVING** clause is calculated after the query results are grouped. After the sorting process is completed using the **ORDER BY** clause,

you need to use the **ORDERBY_NUM()** function in the **ORDER BY** clause in order to get a sequence of the result records.

- The **ROWNUM** function can also be used in SQL statements such as **INSERT**, **DELETE** and **UPDATE** in addition to the **SELECT** statement. For example, as in the query **INSERT INTO** *table_name* **SELECT** ... **FROM** ... **WHERE** ..., you can search for part of the row from one table and then insert it into another by using the **ROWNUM** function in the **WHERE** clause.

Example

The following is an example (demodb) that returns country names ranked first to fourth based on the number of gold medals in the 1988 Olympics.

```
--Limiting 4 rows using ROWNUM in the WHERE condition
SELECT  * FROM
(SELECT nation_code FROM participant WHERE host_year = 1988
     ORDER BY gold DESC) AS T
WHERE ROWNUM <5;

=== <Result of SELECT Command in Line 4> ===

  nation_code
======================
  'URS'
  'GDR'
  'USA'
  'KOR'

4 rows selected.

--Limiting 4 rows using FOR ORDERBY_NUM()
SELECT ROWNUM, nation_code FROM participant WHERE host_year = 1988
ORDER BY gold DESC
FOR ORDERBY_NUM() < 5;

=== <Result of SELECT Command in Line 3> ===

       rownum  nation_code
===================================
          156  'URS'
          155  'GDR'
          154  'USA'
          153  'KOR'


4 rows selected.

--Unexpected results : ROWNUM operated before ORDER BY
SELECT ROWNUM, nation_code FROM participant
WHERE host_year = 1988 AND ROWNUM < 5
ORDER BY gold DESC;

=== <Result of SELECT Command in Line 3> ===

       rownum  nation_code
===================================
            1  'ZIM'
            2  'ZAM'
            3  'ZAI'
            4  'YMD'


4 rows selected.
```

GROUPBY_NUM() Function

Description

The **GROUPBY_NUM()** function is used with the **ROWNUM()** or **INST_NUM()** function to limit the number of result rows. The difference is that the **GROUPBY_NUM()** function is combined after the **GROUP BY ... HAVING** clause to give order to a result that has been already sorted. In addition, while the **INST_NUM()** function is a scalar function, the **GROUPBY_NUM()** function is kind of an aggregate function.

That is, when retrieving only some of the result rows by using **ROWNUM** in a condition clause of the **SELECT** statement that includes the **GROUP BY** clause, **ROWNUM** is applied first and then group sorting by **GROUP BY** is performed. On the other hand, when retrieving only some of the result rows by using the **GROUPBY_NUM()** function, **ROWNUM** is applied to the result of group sorting by **GROUP BY**.

Syntax

```
GROUPBY_NUM()
```

Example

The following is an example that searches for the fastest record in the previous five Olympic Games from the history table. (demodb)

```
--Group-ordering first and then limiting rows using GROUPBY_NUM()
SELECT host_year, MIN(score) FROM history
GROUP BY host_year HAVING GROUPBY_NUM() BETWEEN 1 AND 5;

=== <Result of SELECT Command in Line 2> ===

    host_year  min(score)
===================================
         1968  '8.9'
         1980  '01:53.0'
         1984  '13:06.0'
         1988  '01:58.0'
         1992  '02:07.0'


5 rows selected.

--Limiting rows first and then Group-ordering using ROWNUM
SELECT host_year, MIN(score) FROM history
WHERE ROWNUM BETWEEN 1 AND 5 GROUP BY host_year;

=== <Result of SELECT Command in Line 2> ===

    host_year  min(score)
===================================
         2000  '03:41.0'
         2004  '01:45.0'


2 rows selected.
```

ORDERBY_NUM() Function

Description

The **ORDERBY_NUM()** function is used with the **ROWNUM()** or **INST_NUM()** function to limit the number of result rows. The difference is that the **ORDERBY_NUM()** function is combined after the ORDER BY clause to give order to a result that has been already sorted.

That is, when retrieving only some of the result rows by using **ROWNUM** in a condition clause of the **SELECT** statement that includes the **ORDER BY** clause, **ROWNUM** is applied first and then group sorting by **ORDER BY** is performed. On the other hand, when retrieving only some of the result rows by using the **ORDER_NUM()** function, **ROWNUM** is applied to the result of sorting by **ORDER BY**.

Syntax

```
FOR ORDERBY_NUM()
```

Example

The following is an example of searching athlete names ranked 3rd to 5th and their records in the history table. (demodb)

```
--Ordering first and then limiting rows using FOR ORDERBY_NUM()
SELECT athlete, score FROM history
ORDER BY score FOR ORDERBY_NUM() BETWEEN 3 AND 5;

=== <Result of SELECT Command in Line 1> ===

  athlete               score
============================================
  'Luo Xuejuan'         '01:07.0'
  'Rodal Vebjorn'       '01:43.0'
  'Thorpe Ian'          '01:45.0'


3 rows selected.

--Limiting rows first and then Ordering using ROWNUM
SELECT athlete, score FROM history
WHERE ROWNUM BETWEEN 3 AND 5 ORDER BY score;

=== <Result of SELECT Command in Line 1> ===

  athlete               score
============================================
  'Thorpe Ian'          '01:45.0'
  'Thorpe Ian'          '03:41.0'
  'Hackett Grant'       '14:43.0'


3 rows selected.
```

Information Functions

CURRENT_USER, USER, USER(), SYSTEM_USER()

Description

CURRENT_USER and **USER** are used interchangeably. They return the user name that is currently logged in to the database as a string.

USER() and **SYSTEM_USER()** are used interchangeably. They return the user name with a host name.

Syntax

```
CURRENT_USER
USER
USER()
SYSTEM_USER()
```

Example

```
--selecting the current user on the session
SELECT USER;

=== <Result of SELECT Command in Line 1> ===

   CURRENT_USER
======================
  'PUBLIC'

SELECT USER(), CURRENT_USER;

=== <Result of SELECT Command in Line 1> ===

   user()                CURRENT_USER
============================================
  'PUBLIC@cdbs006.cub'  'PUBLIC'
```

```
--selecting all users of the current database from the system table
SELECT name, id, password FROM db_user;

=== <Result of SELECT Command in Line 1> ===

  name                           id  password
=========================================================
  'DBA'                        NULL  NULL
  'PUBLIC'                     NULL  NULL
  'SELECT_ONLY_USER'           NULL  db_password
  'ALMOST_DBA_USER'            NULL  db_password
  'SELECT_ONLY_USER2'          NULL  NULL
```

DATABASE(), SCHEMA()

Description

DATEBASE() returns the name of the currently-connected database as a **VARCHAR** type string. **DATABASE** and **SCHEMA** are used interchangeably.

Syntax

```
DATABASE()
SCHEMA()
```

Example

```
SELECT DATABASE(), SCHEMA();
;xr

=== <Result of SELECT Command in Line 1> ===

   database()             schema()
============================================
  'demodb'              'demodb'
```

DEFAULT Function

Description

The **DEFAULT** function returns a default value defined for a column. If a default value has not been specified for the given column, **NULL** or an error is returned. If any of constraints is not defined or the **UNIQUE** constraint is defined for the column where a default value is not defined, **NULL** is returned. If **NOT NULL** or **PRIMARY KEY** constraint is defined, an error is returned.

Syntax

```
DEFAULT(column_name)
```

Example

```
CREATE TABLE info_tbl(id INT DEFAULT 0, name VARCHAR)
INSERT INTO info_tbl VALUES (1,'a'),(2,'b'),(NULL,'c');

3 rows affected.

SELECT id, DEFAULT(id) FROM info_tbl;

=== <Result of SELECT Command in Line 1> ===

           id   default(id)
=============================
            1             0
            2             0
         NULL             0
```

LIST_DBS Function

Description

The **LIST_DBS** function outputs the list of all databases in the CUBRID database server, separated by blanks.

Syntax

```
LIST_DBS()
```

Example

```
SELECT LIST_DBS();

=== <Result of SELECT Command in Line 1> ===

  dbs
======================
  'testdb demodb'
```

ROW_COUNT Function

Description

The **ROW_COUNT** function returns the number of rows updated (**UPDATE**, **INSERT**, **DELETE)** by the previous statement. Note that the **ROW_COUNT** function execution area at the SQL level is limited to the client session in which the SQL was created. If this function is called after executing SQL with the **;run** or **;xrun** command, it returns -1.

Syntax

```
ROW_COUNT()
```

Example

```
SELECT * FROM info_tbl;

=== <Result of SELECT Command in Line 1> ===

           id  name
===================================
            1  'a'
            2  'b'
         NULL  'c'

INSERT INTO info_tbl VALUES (4,'d'),(5, 'e');
SELECT ROW_COUNT();
;xr

=== <Result of SELECT Command in Line 2> ===

   row_count()
===============
              2

DELETE FROM info_tbl WHERE id IN (4,5);
SELECT ROW_COUNT();
;xr

=== <Result of SELECT Command in Line 2> ===

   row_count()
===============
              2


SELECT ROW_COUNT();

=== <Result of SELECT Command in Line 1> ===
   row_count()
===============
             -1
```

Conditional Expressions and Functions

CASE

Description

The **CASE** expression uses the SQL statement to perform an **IF** ... **THEN** statement. When a result of comparison expression specified in a **WHEN** clause is true, a value specified in **THEN** value is returned. A value specified in an **ELSE** clause is returned otherwise. If no **ELSE** clause exists, **NULL** is returned.

Syntax

```
CASE control_expression simple_when_list
[ else_clause ]
END
CASE searched_when_list
[ else_clause ]
END

simple_when :
WHEN expression THEN result

searched_when :
WHEN search_condition THEN result

else_clause :
ELSE result

result :
expression | NULL
```

The CASE expression must end with the END keyword. A *control_expression* argument and an *expression argument* in *simple_when* expression should be comparable data types. The data types of *result* specified in the **THEN** ... **ELSE** statement should all same, or they can be convertible to common data type.

The data type for a value returned by the **CASE** expression is determined based on the following rules.

- If data types for result specified in the **THEN** statement are all same, a value with the data type is returned.
- If data types can be convertible to common data type even though they are not all same, a value with the data type is returned.
- If any of values for *result* is a variable length string, a value data type is a variable length string. If values for *result* are all a fixed length string, the longest character string or bit string is returned.
- If any of values for result is an approximate numeric data type, a value with a numeric data type is returned. The number of digits after the decimal point is determined to display all significant digits.

Example

```
--creating a table
CREATE TABLE case_tbl( a INT);
INSERT INTO case_tbl VALUES (1);
INSERT INTO case_tbl VALUES (2);
INSERT INTO case_tbl VALUES (3);
INSERT INTO case_tbl VALUES (NULL);

--case operation with a search when clause
SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
FROM case_tbl;

===================================
            1  'one'
            2  'two'
            3  'other'
         NULL  'other'

--case operation with a simple when clause
SELECT a,
       CASE a WHEN 1 THEN 'one'
              WHEN 2 THEN 'two'
              ELSE 'other'
       END
FROM case_tbl;

===================================
            1  'one'
            2  'two'
            3  'other'
         NULL  'other'


--result types are converted to a single type containing all of significant figures
SELECT a,
       CASE WHEN a=1 THEN 1
            WHEN a=2 THEN 1.2345
            ELSE 1.234567890
       END
FROM case_tbl;

===================================
            1  1.000000000
            2  1.234500000
            3  1.234567890
         NULL  1.234567890

--an error occurs when result types are not convertible
SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 1.2345
       END
FROM case_tbl;
===================================
ERROR
```

COALESCE Function

Description

The **COALESCE** function has more than one expression as an argument. If a first argument is non-**NULL**, it is returned if it is **NULL**, a second argument is returned. If all expressions which have an argument are **NULL**, **NULL** is returned. Therefore, this function is generally used to replace NULL with other default value. All expressions with an argument must be identical or each data type must be convertible to one another.

Syntax

```
COALESCE(expression [, ...])

result :
expression | NULL
```

COALESCE(*a*, *b*) works the same as the **CASE** statement as follows:

```
CASE WHEN a IS NOT NULL
THEN a
ELSE b
END
```

Example

```
SELECT * FROM case_tbl
```

```
=== < Result of SELECT Command in Line 1> ===

                        a
=============
                        1
                        2
                        3
                  NULL

--substituting a default value 10.0000 for NULL valuse
SELECT a, COALESCE(a, 10.0000) FROM case_tbl

=== < Result of SELECT Command in Line 1> ===

                        a   coalesce(a, 10.0000)
===================================
                        1   1.0000
                        2   2.0000
                        3   3.0000
                  NULL   10.0000
```

DECODE Function

Description

As well as a **CASE** expression, the **DECODE** function performs the same functionality as the **IF** ... **THEN** ... **ELSE** statement. It compares the *expression* argument with *search* argument, and returns the *result* corresponding to *search* that has the same value. It returns *default* if there is no *search* with the same value, and returns **NULL** if *default* is omitted. An expression argument and a search argument to be comparable should be same or convertible each other. The number of digits after the decimal point is determined to display all significant digits including valid number of all *result*.

Syntax

```
DECODE( expression, search, result [, search, result]* [, default] )

result :
result | default | NULL
```

DECODE(*a*, *b*, *c*, *d*, *e*) works the same as the **CASE** statement as follows:

```
CASE WHEN a = b THEN c
WHEN a= c THEN d
ELSE e
END
```

Example

```
SELECT * FROM case_tbl

=== < Result of SELECT Command in Line 1> ===

                        a
=============
                        1
                        2
                        3
                  NULL

--Using DECODE function to compare expression and search values one by one
SELECT a, DECODE(a, 1, 'one', 2, 'two', 'other') FROM case_tbl

===================================
                        1   'one'
                        2   'two'
                        3   'other'
                  NULL   'other'


--result types are converted to a single type containing all of significant figures
```

```
SELECT a, DECODE(a, 1, 1, 2, 1.2345, 1.234567890) FROM case_tbl

===================================
                        1   1.000000000
                        2   1.234500000
                        3   1.234567890
                  NULL   1.234567890

--an error occurs when result types are not convertible
SELECT a, DECODE(a, 1, 'one', 2, 'two', 1.2345) FROM case_tbl

===================================
ERROR
```

IF Function

Description

The **IF** function returns *expression2* if the value of the arithmetic expression specified as the first parameter is **TRUE**, or *expression3* if the value is **FALSE** or **NULL**. *expression2* and *expression3* which are returned as the result must be the same or of a convertible common type. If one is explicitly **NULL**, the result of the function follows the type of the non-**NULL** parameter.

Syntax

```
IF( expression1, expression2, expression3 )

result :
exrpession2 | expression3
```

IF(*a*, *b*, *c*) works the same as the **CASE** statement as follows:

```
CASE WHEN a IS TRUE THEN b
ELSE c
END
```

Example

```
SELECT * FROM case_tbl;

=== <Result of SELECT Command in Line 1> ===

            a
=============
            1
            2
            3
         NULL

--IF function returns the second expression when the fist is TRUE
SELECT a, IF(a=1, 'one', 'other') FROM case_tbl;

            a   if(a=1, 'one', 'other')
===================================
            1  'one'
            2  'other'
            3  'other'
         NULL  'other'


--If function in WHERE clause
SELECT * FROM case_tbl WHERE IF(a=1, 1, 2) = 1;

            a
=============
            1


1 rows selected.
```

IFNULL, NVL Function

Description

The **IFNULL** function is working like the **NVL** function; however, only the **NVL** function supports set data type as well. The **IFNULL** function (which has two arguments) returns *expr1* if the value of the first expression is not **NULL** or returns *expr2*, otherwise. The data type of the result is determined as the type which can be converted from both *expr1* and *expr2* types; see the table below.

expr1 Type	expr2 Type	Type of NVL Return Value	Type of IFNULL Return Value
?	?	Error	VARCHAR
?	X	X type	VARCHAR
CHAR	VARCHAR	VARCHAR	VARCHAR
CHAR	NCHAR	Error	VARCHAR
VARCHAR	NCHAR	VARCHAR	VARCHAR
String type	Number type	Error	VARCHAR
String type	Date/Time type	Error	VARCHAR
Number type	Date/Time type	Error	VARCHAR
Date/Time type	Date/Time type	Common convertible type	VARCHAR
Number type	Number type	Common convertible type	Common convertible type
Collection type	Collection type	Common convertible type	Error
Collection type	Others	Error	Error

Syntax

```
IFNULL( expr1, expr2 )

result :
expr1 | expr2
```

IFNULL(*a*, *b*, *c*) works the same as the **CASE** statement as follows:

```
CASE WHEN a IS NULL THEN b
ELSE a
END
```

Example

```
SELECT * FROM case_tbl;

=== <Result of SELECT Command in Line 1> ===

            a
=============
            1
            2
            3
         NULL

--returning a specific value when a is NULL
SELECT a, NVL(a, 10.0000) FROM case_tbl;

=== <Result of SELECT Command in Line 1> ===

            a  nvl(a, 10.0000)
===================================
            1  1.0000
            2  2.0000
            3  3.0000
         NULL  10.0000

--IFNULL can be used instead of NVL and return values are converted to the string type
SELECT a, IFNULL(a, 'UNKNOWN') FROM case_tbl;
```

```
            a   ifnull(a, 'UNKNOWN')
===================================
            1  '1'
            2  '2'
            3  '3'
         NULL  'UNKNOWN'
```

NULLIF Function

Description

The **NULLIF** function returns **NULL** if the two expressions specified as the parameters are identical, and returns the first parameter value otherwise.

Syntax

```
NULLIF(expr1, expr2)

result :
expr1 | NULL
```

NULLIF(*a*, *b*) is the same of the **CASE** statement.

```
CASE
WHEN a = b THEN NULL
ELSE a
END
```

Example

```
SELECT * FROM case_tbl;

=== <Result of SELECT Command in Line 1> ===

            a
=============
            1
            2
            3
         NULL

--returning NULL value when a is 1
SELECT a, NULLIF(a, 1) FROM case_tbl;

=== <Result of SELECT Command in Line 1> ===

            a  nullif(a, 1)
===========================
            1          NULL
            2             2
            3             3
         NULL          NULL

--returning NULL value when arguments are same
SELECT NULLIF (1, 1.000)  FROM db_root;

=== <Result of SELECT Command in Line 1> ===

  nullif(1, 1.000)
======================
  NULL

--returning the first value when arguments are not same
SELECT NULLIF ('A', 'a')  FROM db_root;

=== <Result of SELECT Command in Line 1> ===

  nullif('A', 'a')
======================
  'A'
```

NVL2 Function

Description

Three parameters are specified for the **NVL** function. The function returns the second expression (*expr2*) if the first expression (*expr1*) is not **NULL**, and the third expression (*expr2*) if it is **NULL**.

Syntax

```
NVL2( expr1, expr2, expr3 )

result :
expr2 | expr3
```

Example

```
SELECT * FROM case_tbl;

=== <Result of SELECT Command in Line 1> ===

            a
=============
            1
            2
            3
         NULL

--returning a specific value of INT type
SELECT a, NVL2(a, a+1, 10.5678) FROM case_tbl;

=== <Result of SELECT Command in Line 1> ===

            a  nvl2(a, a+1, 10.5678)
====================================
            1                      2
            2                      3
            3                      4
         NULL                     11
```

Conditional Expressions

Basic Conditional Expressions

A conditional expression is an expression that is included in the **WHERE** clause of the **SELECT**, **UPDATE** and **DELETE** statements, and in the **HAVING** clause of the **SELECT** statement. There are simple comparison, **ANY**/**SOME**/**ALL**, **BETWEEN**, **EXISTS**, **IN**/**NOT IN**, **LIKE** and **IS NULL** conditional expressions, depending on the kinds of the operators combined.

A simple comparison conditional expression compares two comparable data values. Expressions or subqueries are specified as operands, and the conditional expression always returns **NULL** if one of the operands is **NULL**. The following table shows operators that can be used in the simple comparison conditional expressions. For details, see Comparison Operator.

Operators for Conditional Expressions

Comparison Operator	Description	Conditional Expression	Return Value
=	A value of left operand is the same as that of right operand.	1=2	0
<>, !=	A value of left operand is not the same as that of right operand.	1<>2	1
>	A value of left operand is greater than that of right operand.	1>2	0
<	A value of left operand is less than	1<2	1

	that of right operand.		
>=	A value of left operand is equal to or greater than that of right operand.	1>=2	0
<=	A value of left operand is equal to or less than that of right operand.	1<=2	1

ANY/SOME/ALL Conditional Expressions

Description

Group conditional expressions that include quantifiers such as **ANY/SOME/ALL** perform comparison operation on one data value and on some or all values included in the list. A conditional expression that includes **ANY** or **SOME** returns **TRUE** if the value of the data on the left satisfies simple comparison with at least one of the values in the list specified as an operand on the right. A group conditional expression that includes **ALL** returns **TRUE** if the value of the data on the left satisfies simple comparison with all values in the list on the right.

When a comparison operation is performed on **NULL** in a group conditional expression that includes **ANY** or **SOME**, **UNKNOWN** or **TRUE** is returned as the result; when a comparison operation is performed on **NULL** in a group conditional expression that includes **ALL**, **UNKNOWN** or **FALSE** is returned.

Syntax

```
expression comp_op SOME expression
expression comp_op ANY expression
expression comp_op ALL expression
```

- *comp_op* : A comparison operator >, = or <= can be used.
- *expression* (left) : A single-value column, path expression, constant value or arithmetic function that produces a single value can be used.
- *expression* (right) : A column name, path expression, list (set) of constant values or subquery can be used. A list is a set represented within braces ({}). If a subquery is used, *expression* (left) and comparison operation on all results of the subquery execution is performed.

Example

```
--creating a table

CREATE TABLE condition_tbl (id int primary key, name char(10), dept_name VARCHAR, salary
INT);
INSERT INTO condition_tbl VALUES(1, 'Kim', 'devel', 4000000);
INSERT INTO condition_tbl VALUES(2, 'Moy', 'sales', 3000000);
INSERT INTO condition_tbl VALUES(3, 'Jones', 'sales', 5400000);
INSERT INTO condition_tbl VALUES(4, 'Smith', 'devel', 5500000);
INSERT INTO condition_tbl VALUES(5, 'Kim', 'account', 3800000);
INSERT INTO condition_tbl VALUES(6, 'Smith', 'devel', 2400000);
INSERT INTO condition_tbl VALUES(7, 'Brown', 'account', NULL);

--selecting rows where department is sales or devel
SELECT * FROM condition_tbl WHERE dept_name = ANY{'devel','sales'};

=== <Result of SELECT Command in Line 1> ===

           id  name                  dept_name                  salary
======================================================================
            1  'Kim       '          'devel'                   4000000
            2  'Moy       '          'sales'                   3000000
            3  'Jones     '          'sales'                   5400000
            4  'Smith     '          'devel'                   5500000
            6  'Smith     '          'devel'                   2400000

--selecting rows comparing NULL value in the ALL group conditions
SELECT * FROM condition_tbl WHERE salary > ALL{3000000, 4000000, NULL};

=== <Result of SELECT Command in Line 1> ===
There are no results.

--selecting rows comparing NULL value in the ANY group conditions
SELECT * FROM condition_tbl WHERE salary > ANY{3000000, 4000000, NULL};

=== <Result of SELECT Command in Line 1> ===

           id  name                  dept_name                  salary
======================================================================
            1  'Kim       '          'devel'                   4000000
            3  'Jones     '          'sales'                   5400000
            4  'Smith     '          'devel'                   5500000
            5  'Kim       '          'account'                 3800000

--selecting rows where salary*0.9 is less than those salary in devel department
SELECT * FROM condition_tbl WHERE (
(0.9 * salary) < ALL (SELECT salary FROM condition_tbl
WHERE dept_name = 'devel')
);

=== <Result of SELECT Command in Line 1> ===

           id  name                  dept_name                  salary
======================================================================
            6  'Smith     '          'devel'                   2400000
```

BETWEEN Conditional Expression

Description

The **BETWEEN** conditional expression makes a comparison to determine whether the data value on the left exists between two data values specified on the right. It returns **TRUE** even when the data value on the left is the same as a boundary value of the comparison target range. If **NOT** comes before the **BETWEEN** keyword, the result of a **NOT** operation on the result of the **BETWEEN** operation is returned.

i **BETWEEN** *g* **AND** *m* and the compound condition *i* **>= g AND** *i* <= *m* have the same effect.

Syntax

```
expression [ NOT ] BETWEEN expression AND expression
```

- *expression* : A column name, path expression, constant value, arithmetic expression or aggregate function can be used. For a character string expression, the conditions are evaluated in alphabetical order. If **NULL** is specified for at least one of the expressions, the **BETWEEN** predicate returns **UNKNOWN** as the result.

Example

```
--selecting rows where 3000000 <= salary <= 4000000
SELECT * FROM condition_tbl WHERE salary BETWEEN 3000000 AND 4000000;

SELECT * FROM condition_tbl WHERE (salary >= 3000000) AND (salary <= 4000000);

=== <Result of SELECT Command in Line 1> ===

           id  name                  dept_name                  salary
======================================================================
            1  'Kim       '          'devel'                   4000000
            2  'Moy       '          'sales'                   3000000
            5  'Kim       '          'account'                 3800000

--selecting rows where salary < 3000000 or salary > 4000000
SELECT * FROM condition_tbl WHERE salary NOT BETWEEN 3000000 AND 4000000;

=== <Result of SELECT Command in Line 1> ===

           id  name                  dept_name                  salary
======================================================================
            3  'Jones     '          'sales'                   5400000
            4  'Smith     '          'devel'                   5500000
            6  'Smith     '          'devel'                   2400000
```

```
--selecting rows where name starts from A to E
SELECT * FROM condition_tbl WHERE name BETWEEN 'A' AND 'E';

=== <Result of SELECT Command in Line 1> ===

           id  name                  dept_name                  salary
======================================================================
            7  'Brown     '          'account'                    NULL
```

EXISTS Conditional Expression

Description

The **EXISTS** conditional expression returns **TRUE** if one or more results of the execution of the subquery specified on the right exist, and returns **FALSE** if the result of the operation is an empty set.

Syntax

```
EXISTS expression
```

- *expression* : Specifies a subquery and compares to determine whether the result of the subquery execution exists. If the subquery does not produce any result, the result of the conditional expression is **FALSE**.

Example

```
--selecting rows using EXISTS and subquery
SELECT 'raise' FROM db_root WHERE EXISTS(
SELECT * FROM condition_tbl WHERE salary < 2500000);

=== <Result of SELECT Command in Line 2> ===

  'raise'
======================
  'raise'

--selecting rows using NOT EXISTS and subquery
SELECT 'raise' FROM db_root WHERE NOT EXISTS(
SELECT * FROM condition_tbl WHERE salary < 2500000);

=== <Result of SELECT Command in Line 2> ===

There are no results.

0 rows selected.
```

IN Conditional Expression

Description

The **IN** conditional expression compares to determine whether the single data value on the left is included in the list specified on the right. That is, the predicate returns **TRUE** if the single data value on the left is an element of the expression specified on the right. If **NOT** comes before the **IN** keyword, the result of a **NOT** operation on the result of the **IN** operation is returned.

Syntax

```
expression [ NOT ] IN expression
```

- *expression* (left) : A single-value column, path expression, constant value or arithmetic function that produces a single value can be used.
- *expression* (right) : A column name, path *expression*, list (set) of constant values or subquery can be used. A list is a set represented within parentheses (()) or braces ({}). If a subquery is used, comparison with expression(left) is performed for all results of the subquery execution.

Example

```
--selecting rows where department is sales or devel
SELECT * FROM condition_tbl WHERE dept_name IN {'devel','sales'};

SELECT * FROM condition_tbl WHERE dept_name = ANY{'devel','sales'};

=== <Result of SELECT Command in Line 1> ===

           id  name                  dept_name                  salary
======================================================================
            1  'Kim       '          'devel'                   4000000
            2  'Moy       '          'sales'                   3000000
            3  'Jones     '          'sales'                   5400000
            4  'Smith     '          'devel'                   5500000
            6  'Smith     '          'devel'                   2400000

--selecting rows where department is neither sales nor devel
SELECT * FROM condition_tbl WHERE dept_name NOT IN {'devel','sales'};

=== <Result of SELECT Command in Line 1> ===

           id  name                  dept_name                  salary
======================================================================
            5  'Kim       '          'account'                 3800000
            7  'Brown     '          'account'                    NULL
```

IS NULL Conditional Expression

Description

The **IS NULL** conditional expression compares to determine whether the expression specified on the left is **NULL**, and if it is **NULL**, returns **TRUE** and it can be used in the conditional expression. If **NOT** comes before the **NULL** keyword, the result of a **NOT** operation on the result of the **IS NULL** operation is returned.

Syntax

```
expression IS [ NOT ] NULL
```

- *expression* : A single-value column, path expression, constant value or arithmetic function that produces a single value can be used.

Example

```
SELECT * FROM condition_tbl WHERE salary IS NULL;

=== <Result of SELECT Command in Line 1> ===

           id  name                  dept_name                  salary
======================================================================
            7  'Brown     '          'account'                    NULL

--selecting rows where salary is NOT NULL
SELECT * FROM condition_tbl WHERE salary IS NOT NULL;

=== <Result of SELECT Command in Line 1> ===

           id  name                  dept_name                  salary
======================================================================
            1  'Kim       '          'devel'                   4000000
            2  'Moy       '          'sales'                   3000000
            3  'Jones     '          'sales'                   5400000
            4  'Smith     '          'devel'                   5500000
            5  'Kim       '          'account'                 3800000
            6  'Smith     '          'devel'                   2400000

--simple conparison operation returns NULL when operand is NULL
SELECT * FROM condition_tbl WHERE salary = NULL;

=== <Result of SELECT Command in Line 1> ===
There are no results.

0 rows selected.
```

ISNULL Function

Description

The **ISNULL** function performs a comparison to determine if the result of the expression specified as an argument is **NULL**. The function returns 1 if it is **NULL** or 0 otherwise. You can check if a certain value is **NULL**. This function is working like the **ISNULL** expression.

Syntax

```
ISNULL(expression)
```

- *expression* : An arithmetic function that has a single-value column, path expression, constant value is specified.

Example

```
--Using ISNULL function to select rows with NULL value
SELECT * FROM condition_tbl WHERE ISNULL(salary);

=== <Result of SELECT Command in Line 1> ===

           id  name                  dept_name                  salary
======================================================================
            7  'Brown     '          'account'                    NULL
```

LIKE Conditional Expression

Description

The **LIKE** conditional expression compares patterns between character string data, and returns **TRUE** if a character string whose pattern matches the search word is found. Pattern comparison target domains are **CHAR**, **VARCHAR** and **STRING**. The **LIKE** search cannot be performed on an **NCHAR** or **BIT** type. If **NOT** comes before the **LIKE** keyword, the result of a **NOT** operation on the result of the **LIKE** operation is returned.

A wild card string corresponding to any character or character string can be included in the search word on the right of the **LIKE** operator. % (percent) and _ (underscore) can be used. .% corresponds to any character string whose length is 0 or greater, and _ corresponds to one character. An escape character is a character that is used to search for a wild card character itself, and can be specified by the user as another character whose length is 1. See below for an example of using a character string that includes wild card or escape characters.

Syntax

```
expression [ NOT ] LIKE expression [ ESCAPE char]
```

- *expression* (left) : Specify the data type column of the character string. Pattern comparison, which is case-sensitive, starts from the first character of the column.
- *expression* (right) : Enter the search word. A character string with a length of 0 or greater is required. Wild card characters (% or _) can be included as the pattern of the search word. The length of the character string is 0 or greater.
- **ESCAPE** *char* : If the string pattern of the search word includes "_" or "%" itself, an ESCAPE character must be specified. For example, if you want to search for the character string "10%" after specifying backslash (\) as the ESCAPE character, you must specify "10\%" for the expression (right). If you want to search for the character string "C:\", you can specify "C:\\" for the expression (right).

Remark

LIKE search may not work properly for data entered in multi-byte character set environment such as utf-8. This is because byte units for string comparison operation differ depending on the character sets. You can get normal results by adding a parameter(**single_byte_compare**=yes) to the **cubrid.conf** file that enables string comparison in a single-byte units, and restarting the DB.

For details about character sets supported in CUBRID, see Definition and Characteristics. For details about the single_byte_compare parameter, see Other Parameters.

Example

```
--selection rows where name contains lower case 's', not upper case
SELECT * FROM condition_tbl WHERE name LIKE '%s%';

=== <Result of SELECT Command in Line 1> ===

           id  name                  dept_name                  salary
======================================================================
            3  'Jones     '          'sales'                   5400000

--selection rows where second letter is 'O' or 'o'
SELECT * FROM condition_tbl WHERE UPPER(name) LIKE '_O%';

=== <Result of SELECT Command in Line 1> ===

           id  name                  dept_name                  salary
======================================================================
            2  'Moy       '          'sales'                   3000000
            3  'Jones     '          'sales'                   5400000

--selection rows where name is 3 characters

SELECT * FROM condition_tbl WHERE name LIKE '___';

=== <Result of SELECT Command in Line 1> ===

           id  name                  dept_name                  salary
======================================================================
            1  'Kim       '          'devel'                   4000000
            2  'Moy       '          'sales'                   3000000
            5  'Kim       '          'account'                 3800000
```

Data Manipluation

SELECT

Overview

Description

The **SELECT** statement specifies columns that you want to retrieve from a table.

Syntax

```
SELECT [ <qualifier> ] <select_expressions>
    [ { TO | INTO } <variable_comma_list> ]
    [ FROM <extended_table_specification_comma_list> ]
    [ WHERE <search_condition> ]
    [ GROUP BY {col_name | expr} [ ASC | DESC ],...[ WITH ROLLUP ] ]
    [ HAVING  <search_condition> ]
    [ ORDER BY {col_name | expr} [ ASC | DESC ],... [ FOR <orderby_for_condition> ] ]
    [ LIMIT [offset,] row_count ]
    [ USING INDEX { index name [,index_name,...] | NONE }]

<qualifier> ::= ALL | DISTINCT | DISTINCTROW | UNIQUE

<select_expressions> ::= * | <expression_comma_list>

<extended_table_specification_comma_list> ::=
<table specification> [ {, <table specification> | <join table specification> }... ]

<table_specification> ::=
 <single_table_spec> [ <correlation> ] [ WITH (lock_hint) ]|
 <metaclass_specification> [ <correlation> ] |
 <subquery> <correlation> |
 TABLE ( <expression> ) <correlation>

<correlation> ::= [ AS ] <identifier> [ ( <identifier_comma_list> ) ]

<single_table_spec> ::= [ ONLY ] <table_name> |
                      ALL <table_name> [ EXCEPT <table_name> ]

<metaclass_specification> ::= CLASS <class_name>

<join_table_specification> ::=
[ INNER | [ LEFT | RIGHT [ OUTER ] ] JOIN <table specification> ON <search condition>

lock_hint :
READ UNCOMMITTED

<orderby_for_condition> ::=
ORDERBY_NUM() { BETWEEN int AND int } |
    { { = | =< | < | > | >= } int } |
    IN ( int, ...)
```

- *qualifier* : A qualifier. It can be omitted. When omitted, it is set to **ALL**.
    - **ALL** : Retrieves all records of the table.
    - **DISTINCT** : Retrieves only records with unique values without allowing duplicates. **DISTINCT** and **DISTINCTROW** are used interchangeably.
    - **UNIQUE** : Like **DISTINCT**, retrieves only records with unique values without allowing duplicates.
- *select_expression* :
    - * : By using **SELECT** * statement, you can retrieve all the columns from the table specified in the **FROM** clause.
    - *expression_comma_list* : *expression* can be a path expression, variable or table name. All general expressions including arithmetic operations can also be used. Use a comma (,) to separate each expression in the list.

You can specify aliases by using the **AS** keyword for columns or expressions to be queried. Specified aliases are used as column names in **GROUP BY**, **HAVING**, **ORDER BY** and **FOR** clauses. The position index of a column is assigned based on the order in which the column was specified. The starting value is 1.
As **AVG**, **COUNT**, **MAX**, **MIN**, or **SUM**, an aggregate function that manipulates the retrieved data can also be used in the *expression*. As the aggregate function returns only one result, you cannot specify a general column which has not been grouped by an aggregate function in the **SELECT** column list.

- *table_name.* * : Specifying the table name and using * has the same effect as specifying all columns for the given table.
- *variable* : The data retrieved by the *select_expression* can be saved in more than one variables.
- [:]*identifier* : By using the :identifier after **TO** (or **INTO**), you can save the data to be retrieved in the ':identifier' variable.

Example 1

The following is an example of retrieving host countries of the Olympic Games without any duplicates. This example is performed on the olympic table of demodb.

The **DISTINCT** or **UNIQUE** keyword allows only unique values in the query result set. For example, when there are multiple **olympic** records whose **host_nation** values are 'Greece', you can use such keywords to display only one value in the query result.

```
SELECT DISTINCT host_nation FROM olympic;

=== <Result of SELECT Command in Line 1> ===
  host_nation
======================
  'Australia'
  'Belgium'
  'Canada'
  'Finland'
  'France'
...
18 rows selected.
```

Example 2

The following is an example that defines an alias to a column to be queried, and sorts the result record by using the column alias in the **ORDER BY** clause. At this time, the number of the result records is limited to 5 by using the **LIMIT** clause and FOR **ORDERBY_NUM()**.

```
SELECT host_year as col1, host_nation as col2 FROM olympic ORDER BY col2 LIMIT 5;

=== <Result of SELECT Command in Line 1> ===

         col1  col2
===================================
         2000  'Australia'
         1956  'Australia'
         1920  'Belgium'
         1976  'Canada'
         1948  'England'

5 rows selected.

SELECT CONCAT(host_nation, ', ', host_city) AS host_place FROM olympic
ORDER BY host_place FOR ORDERBY_NUM() BETWEEN 1 AND 5;
;xr

=== <Result of SELECT Command in Line 1> ===

  host_place
======================
  'Australia,  Melbourne'
  'Australia,  Sydney'
  'Belgium,  Antwerp'
  'Canada,  Montreal'
  'England,  London'
```

```
5 rows selected.
```

FROM Clause

General

Description

The **FROM** clause specifies the table in which data is to be retrieved in the query. If no table is referenced, the **FROM** clause can be omitted. Retrieval paths are as follows:

- Single table
- Subquery
- Derived table

Syntax

```
SELECT [ <qualifier> ] <select_expressions>
                          [ FROM <table_specification> [ {, <table specification>
| <join table specification> }... ]]


<select_expressions> ::= * | <expression_comma_list>

<table_specification> ::=
 <single_table_spec> [ <correlation> ] [ WITH (lock_hint) ] |
 <metaclass_specification> [ <correlation> ] |
 <subquery> <correlation> |
 TABLE ( <expression> ) <correlation>

<correlation> ::= [ AS ] <identifier> [ ( <identifier_comma_list> ) ]

<single_table_spec> ::= [ ONLY ] <table_name> |
                      ALL <table_name> [ EXCEPT <table_name> ]

<metaclass_specification> ::= CLASS <class_name>

lock_hint ::=
READ UNCOMMITTED
```

- *select_expressions* : One or more columns or expressions to query is specified. Use * to query all columns in the table. You can also specify an alias for a column or an expression to be queried by using the AS keyword. This keyword can be used in **GROUP BY**, **HAVING**, **ORDER BY** and **FOR** clauses. The position index of the column is given according to the order in which the column was specified. The starting value is 1.
- *table_specification* : At least one table name is specified after the **FROM** clause. Subqueries and derived tables can also be used in the **FROM** clause. For more information on subquery derived tables, see Subquery Derived Table.
- *lock_hint* : You can set **READ UNCOMMITTED** for the table isolation level. **READ UNCOMMITTED** is a level where dirty reads are allowed; see Transaction Isolation Level for more information on the CUBRID transaction isolation level.

Example

```
--FROM clause can be omitted in the statement
SELECT 1+1 AS sum_value;
;xr

=== <Result of SELECT Command in Line 1> ===

    sum_value
=============
            2


1 rows selected.

--db_root can be used as a dummy table
SELECT 1+1 AS sum_value FROM db_root;
```

```
csql> ;xr

=== <Result of SELECT Command in Line 1> ===

    sum_value
=============
            2


1 rows selected.

SELECT CONCAT('CUBRID', '2008' , 'R3.0') AS db_version;
;xr

=== <Result of SELECT Command in Line 1> ===

  db_version
======================
  'CUBRID2008R3.0'
```

Derived Table

In the query statement, subqueries can be used in the table specification of the **FROM** clause. Such subqueries create derived tables where subquery results are treated as tables. A correlation specification must be used when a subquery that creates a derived table is used.

Derived tables are also used to access the individual element of an attribute that has a set value. In this case, an element of the set value is created as an instance in the derived table.

Subquery Derived Table

Description

Each instance in the derived table is created from the result of the subquery in the **FROM** clause. A derived table created form a subquery can have any number of columns and records.

Syntax

```
FROM (subquery) [ AS ] derived_table_name [( column_name [ {, column_name }_ ] )]
```

- The number of *column_name* and the number of columns created by the *subquery* must be identical.

Example 1

The following is an example of retrieving the sum of the number of gold medals won by Korea and that of silver medals won by Japan. This example shows a way of getting an intermediate result of the subquery and processing it as a single result, by using a derived table. The query returns the sum of the **gold** values whose **nation_code** is 'KOR' and the **silver** values whose **nation_code** column is 'JPN'.

```
SELECT SUM(n) FROM (SELECT gold FROM participant WHERE nation_code='KOR'
UNION ALL SELECT silver FROM participant WHERE nation_code='JPN') AS t(n);
=== <Result of SELECT Command in Line 2> ===
  sum(n)
========
      82
1 rows selected.
```

Example 2

Subquery derived tables can be useful when combined with outer queries. For example, a derived table can be used in the **FROM** clause of the subquery used in the **WHERE** clause.

The following is a query example that shows **nation_code**, **host_year** and **gold** fields of the instances whose number of gold medals is greater than average sum of the number of silver and bronze medals when one or more sliver or bronze medals were won. In this example, the query (the outer **SELECT** clause) and the subquery (the inner **SELECT** clause) share the **nation_code** attribute.

```
SELECT nation_code, host_year, gold
FROM participant p
WHERE gold > ( SELECT AVG(s)
            FROM ( SELECT silver + bronze
            FROM participant
            WHERE nation_code = p.nation_code
            AND silver > 0
            AND bronze > 0
          ) AS t(s));
=== <Result of SELECT Command in Line 1> ===
  nation_code          host_year          gold
=========================================
  'JPN'                        2004                16
  'CHN'                       2004               32
  'DEN'                       1996                 4
  'ESP'                       1992                13
4 rows selected.
```

WHERE Clause

Description

In a query, a column can be processed based on conditions. The **WHERE** clause specifies a search condition for data.

Syntax

```
WHERE search_condition

search_condition :
• comparison_predicate
• between_predicate
• exists_predicate
• in_predicate
• null_predicate
• like_predicate
• quantified predicate
• set_predicate
```

The **WHERE** clause specifies a condition that determines the data to be retrieved by *search_condition* or a query. Only data for which the condition is true is retrieved for the query results. (**NULL** value is not retrieved for the query results because it is evaluated as unknown value.)

- *search_condition* : It is described in detail in the following sections.
- Basic Conditional Expression
- BETWEEN Conditional Expression
- EXISTS Conditional Expression
- IN Conditional Expression
- IS NULL Conditional Expression
- LIKE Conditional Expression
- ANY/SOME/ALL Conditional Expressions

The logical operator **AND** or **OR** can be used for multiple conditions. If **AND** is specified, all conditions must be true. If **OR** is specified, only one needs to be true. If the keyword **NOT** is preceded by a condition, the meaning of the condition is reserved. The following table shows the order in which logical operators are evaluated.

Priority	Operator	Function
1	**()**	Logical expressions in parentheses are evaluated first.
2	NOT	Negates the result of the logical expression.
3	AND	All conditions in the logical expression must be true.
4	OR	One of the conditions in the logical expression must be true.

GROUP BY ... HAVING Clause

Description

The **GROUP BY** clause is used to group the result retrieved by the **SELECT** statement based on a specific column. This clause is used to sort by group or to get the aggregation by group using the aggregation function. Herein, a group consists of records that have the same value for the column specified in the **GROUP BY** clause.

You can also set a condition for group selection by including the **HAVING** clause after the **GROUP BY** clause. That is, only groups satisfying the condition specified by the **HAVING** clause are queried out of all groups that are grouped by the **GROUP BY** clause.

By SQL standard, you cannot specify a column (hidden column) not defined in the **GROUP BY** clause to the SELECT column list. However, by using extended CUBRID grammars, you can specify the hidden column to the SELECT column list. If you do not use the extended CUBRID grammars, the **only_full_group_by** parameter should be set to **yes**. For more information, see Statement/Type-Related Parameters.

Syntax

```
SELECT ...
GROUP BY { col_name | expr | position } [ ASC | DESC ],... [ WITH ROLLUP ][ ORDER BY NULL ]
                        [ HAVING < search_condition> ]
```

- *col_name* | *expr* | *position* : Specify one or more column names, expressions or aliases. Items are separated by commas. Columns are sorted on this basis.
- [**ASC**| **DESC**] : Specify the **ASC** or **DESC** sorting option after the columns specified in the **GROUP BY** clause. If the sorting option is not specified, the default value is **ASC**.
- *search_condition* : Specify the search condition in the **HAVING** clause. In the **HAVING** clause you can refer to the hidden columns not specified in the **GROUP BY** clause as well as to columns and aliases specified in the **GROUP BY** clause and columns used in aggregate functions.
- **WITH ROLLUP** : If you specify the **WITH ROLLUP** modifier in the **GROUP BY** clause, the aggregate information of the result value of each GROUPed BY column is displayed for each group, and the total of all result rows is displayed at the last row.
- **ORDER BY NULL** : You can avoid the sorting overhead caused by **GROUP BY** by specifying the **ORDER BY NULL** modifier in the GROUP BY clause.

Example

```
--creating a new table
CREATE TABLE sales_tbl
(dept_no int, name VARCHAR(20) PRIMARY KEY, sales_month int, sales_amount int DEFAULT 100)
INSERT INTO sales_tbl VALUES
(201, 'George' , 1, 450),
(201, 'Laura'   , 2, 500),
(301, 'Max'       , 4, 300),
(501, 'Stephan', 4, DEFAULT),
(501, 'Chang'   , 5, 150),
(501, 'Sue'       , 6, 150),
(NULL, 'Yoka'     ,4, NULL)

--selecting rows grouped by dept_no with ORDER BY NULL modifier
SELECT dept_no, avg(sales_amount) FROM sales_tbl
GROUP BY dept_no ORDER BY NULL
xr

=== < Result of SELECT Command in Line 1> ===

            dept_no   avg(sales_amount)
================================
                  NULL                            NULL
                    201                              475
                    301                              300
                    501                              133


--conditions in WHERE clause operate first before GROUP BY
```

```
SELECT dept_no, avg(sales_amount) FROM sales_tbl
WHERE sales_amount > 100 GROUP BY dept_no
xr

=== < Result of SELECT Command in Line 1> ===

            dept_no   avg(sales_amount)
================================
                    201                               475
                    301                               300
                    501                               150

--conditions in HAVING clause operate last after GROUP BY
SELECT dept_no, avg(sales_amount) FROM sales_tbl
WHERE sales_amount > 100 GROUP BY dept_no HAVING avg(sales_amount) > 200
xr

=== < Result of SELECT Command in Line 1> ===

            dept_no   avg(sales_amount)
================================
                    201                               475
                    301                               300

--selecting and sorting rows with using column alias
SELECT dept_no AS a1, avg(sales_amount) AS a2 FROM sales_tbl
WHERE sales_amount > 200 GROUP BY a1 HAVING a2 > 200 ORDER BY a2
xr

=== < Result of SELECT Command in Line 1> ===

                      a1                     a2
==========================
                    301                   300
                    201                   475


2 rows selected.

--selecting rows grouped by dept_no with WITH ROLLUP modifier
SELECT dept_no AS a1, name AS a2, avg(sales_amount) AS a3 FROM sales_tbl
WHERE sales_amount > 100 GROUP BY a1,a2 WITH ROLLUP
xr

=== < Result of SELECT Command in Line 1> ===

                      a1   a2                                                        a3
================================================
                    201   'George'                                          450
                    201   'Laura'                                             500
                    201   NULL                                                    475
                    301   'Max'                                                 300
                    301   NULL                                                    300
                    501   'Chang'                                             150
                    501   'Sue'                                                 150
                    501   NULL                                                    150
                  NULL   NULL                                                   310


9 rows selected.
```

ORDER BY Clause

Description

The **ORDER BY** clause sorts the query result set in ascending or descending order. If you do not specify a sorting option such as **ASC** or **DESC**, the result set in ascending order by default. If you do not specify the **ORDER BY** clause, the order of records to be queried may vary depending on query.

Syntax

```
SELECT ...
ORDER BY {col_name | expr | position} [ASC | DESC],...]
    [ FOR <orderby_for_condition> ] ]

<orderby_for_condition> ::=
ORDERBY_NUM() { BETWEEN int AND int } |
    { { = | =< | < | > | >= } int } |
    IN ( int, ...)
```

- *col_name* | *expr* | *position* : Specify an column name, expression, or alias. One or more column names, expressions or aliases can be specified. Items are separated by commas. A column that is not specified in the list of **SELECT** columns can be specified.
- [**ASC**| **DESC**] : **ASC** means sorting in ascending order, and **DESC** is sorting in descending order. If the sorting option is not specified, the default value is **ASC**.

Example

```
--selecting rows sorted by ORDER BY clause
SELECT * FROM sales_tbl ORDER BY dept_no DESC, name ASC;
;xr

=== <Result of SELECT Command in Line 1> ===

      dept_no  name                  sales_month  sales_amount
==============================================================
          501  'Chang'                         5           150
          501  'Stephan'                       4           100
          501  'Sue'                           6           150
          301  'Max'                           4           300
          201  'George'                        1           450
          201  'Laura'                         2           500
         NULL  'Yoka'                          4          NULL


7 rows selected.

--sorting reversely and limiting result rows by LIMIT clause
SELECT dept_no AS a1, avg(sales_amount) AS a2 FROM sales_tbl
GROUP BY a1 ORDER BY a2 DESC LIMIT 0,3;
;xr

=== <Result of SELECT Command in Line 1> ===

           a1           a2
==========================
          201          475
          301          300
          501          133


3 rows selected.

--sorting reversely and limiting result rows by FOR clause
SELECT dept_no AS a1, avg(sales_amount) AS a2 FROM sales_tbl
GROUP BY a1 ORDER BY a2 DESC FOR ORDERBY_NUM() BETWEEN 1 AND 3;
;xr

=== <Result of SELECT Command in Line 1> ===

           a1           a2
==========================
          201          475
          301          300
          501          133


3 rows selected.
```

LIMIT Clause

Description

The **LIMIT** clause can be used to limit the number of records displayed. It takes one or two arguments. You can specify a very big integer for *row_count* to output to the last row, starting from a specific row.

The **LIMIT** clause can be used as a prepared statement. In this case, the bind parameter (?) can be used instead of an argument.

INST_NUM() and **ROWNUM** cannot be included in the **WHERE** clause in a query that contains the **LIMIT** clause. Also, **LIMIT** cannot be used together with FOR **ORDERBY_NUM()** or **HAVING GROUPBY_NUM()**.

Syntax

```
LIMIT [offset,] row_count
```

- *offset* : Specify the offset value of the starting row to be output. The offset value of the starting row of the result set is 0; it can be omitted and the default value is 0.
- *row_count* : Specify the number of records to be output. You can specify an integer greater than 0.

Example

```
--LIMIT clause can be used in prepared statement
PREPARE STMT FROM 'SELECT * FROM sales tbl LIMIT ?, ?';
EXECUTE STMT USING 0, 10;

--selecting rows with LIMIT clause
SELECT * FROM sales_tbl WHERE sales_amount > 100 LIMIT 5;
;xr

=== <Result of SELECT Command in Line 1> ===

      dept_no  name                  sales_month  sales_amount
==============================================================
          201  'George'                        1           450
          201  'Laura'                         2           500
          301  'Max'                           4           300
          501  'Chang'                         5           150
          501  'Sue'                           6           150

5 rows selected.

--LIMIT clause can be used in subquery
SELECT t1.* FROM
(SELECT * FROM sales_tbl AS t2 WHERE sales_amount > 100 LIMIT 5) AS t1 LIMIT 1,3;
;xr

=== <Result of SELECT Command in Line 1> ===

      dept_no  name                  sales_month  sales_amount
==============================================================
          201  'Laura'                         2           500
          301  'Max'                           4           300
          501  'Chang'                         5           150


3 rows selected.
```

USING INDEX Clause

Description

The **USING INDEX** clause allows indexes to be specified in the query so that the query processor can choose an appropriate index.

The **USING INDEX** clause must be specified after the **WHERE** clause in the **SELECT**, **DELETE** or **UPDATE** statement.

Syntax

```
SELECT . . . FROM . . . WHERE . . .
[USING INDEX { NONE | index_spec [ {, index_spec } ...] } ] [ ; ]
DELETE FROM . . . WHERE . . .
[USING INDEX { NONE | index_spec [ {, index_spec } ...] } ] [ ; ]
UPDATE . . . SET . . . WHERE . . .
[USING INDEX { NONE | index_spec [ {, index_spec } ...] } ] [ ; ]
index_spec :
 [table_name.]index_name [(+)]
```

- **NONE** : If **NONE** is specified, a sequential scan is selected.
- (+) : If (+) is specified after the index name, an index scan using the specified index is selected.

The **USING INDEX** clause forces a sequential/index scan to be used or an index that does not degrade the performance to be included.

If a list of index names is specified in the **USING INDEX** clause, the query optimizer calculates the query execution cost only for the specified index, and then creates an optimized execution plan by comparing the index scan cost of the listed indexes and the sequential scan cost (CUBRID performs query optimization based on the cost in choosing the execution plan).

USING INDEX can be useful when you want to get the result in the desired order without using **ORDER BY**. When index scan is performed by CUBRID, the results are created in the order they were saved in the index. When there are more than one indexes in one table, you can use **USING INDEX** to get the query results in a given order of indexes.

Example

The following is an example of creating an index based on the table creation statement of the **athlete** table.

```
CREATE TABLE athlete (
   code             SMALLINT    NOT NULL PRIMARY KEY,
   name             VARCHAR(40) NOT NULL,
   gender           CHAR(1)     ,
   nation_code      CHAR(3)     ,
   event            VARCHAR(30) ,
   );
CREATE UNIQUE INDEX athlete_idx ON athlete(code, nation_code);
CREATE INDEX char_idx ON athlete(gender, nation_code);
```

For the following query, the query optimizer can choose an index scan that uses the **athlete_idx** index.

```
SELECT * FROM athlete WHERE gender='M' AND nation_code='USA';
```

As in the query below, if **USING INDEX char_idx** is specified, the query optimizer calculates the index scan cost only for the given index specified by **USING INDEX**.

If the index scan cost is less than the sequential scan cost, an index scan is performed.

```
SELECT * FROM athlete WHERE gender='M' AND nation_code='USA'
USING INDEX char_idx(+);
```

To forcefully specify an index scan that uses the **char_idx** index, place (+) after the index name.

```
SELECT * FROM athlete WHERE gender='M' AND nation_code='USA'
USING INDEX char_idx(+);
```

To allow a sequential scan to be selected, specify **NONE** in the **USING INDEX** clause as follows:

```
SELECT * FROM athlete WHERE gender='M' AND nation_code='USA'
USING INDEX NONE;
```

If more than one indexes were specified in the **USING INDEX** clause as shown below, the query optimizer chooses an appropriate one from the specified indexes.

```
SELECT * FROM athlete WHERE gender='M' AND nation_code='USA'
USING INDEX char_idx, athlete_idx;
```

Outer Join

Description

A join is a query that combines the rows of two or more tables or virtual tables (views). In a join query, a condition that compares the columns that are common in two or more tables is called a join condition. Rows are retrieved from each joined table, and are combined only when they satisfy the specified join condition.

A join query using an equality operator (=) is called an equi-join, and one without any join condition is called a cartesian product. Meanwhile, joining a single table is called a self join. In a self join, table **ALIAS** is used to distinguish columns, because the same table is used twice in the **FROM** clause.

A join that outputs only rows that satisfy the join condition from a joined table is called an inner or a simple join, whereas a join that outputs both rows that satisfy and do not satisfy the join condition from a joined table is called an outer join. An outer join is divided into a left outer join which outputs all rowss of the left table as the result, a right outer join which outputs all rowss of the right table as the result and a full outer join which outputs all rows of both tables. If there is no column value that corresponds to a table on one side in the result of an outer join query, all rowss are returned as **NULL**.

Syntax

```
FROM table_specification [{, table_specification | join_table_specification}...]

table_specification :
table_specification [ correlation ]
CLASS table_name [ correlation ]
subquery correlation
TABLE (expression) correlation

join_table_specification :
[ INNER | {LEFT | RIGHT} [ OUTER ] ] JOIN table_specification
join_condition

join_condition :
ON search_condition
```

- *oin_table_specification*
  - { **LEFT** | **RIGHT** } [**OUTER**] **JOIN** : **LEFT** is used for a left outer join query, and **RIGHT** is for a right outer join query.

CUBRID does not support full outer joins. Path expressions that include subqueries and sub-columns cannot be used in the join conditions of an outer join.

Join conditions of an outer join are specified in a different way from those of an inner join. In an inner join, join conditions are expressed in the **WHERE** clause; in an outer join, they appear after the **ON** keyword in the **FROM** clause. Other retrieval conditions can be used in the **WHERE** or **ON** clause, but the retrieval result can differ depending on whether the condition is used in the **WHERE** or **ON** clause.

The table execution order is fixed according to the order specified in the **FROM** clause. Therefore, when using an outer join, you should create a query statement in consideration of the table order. It is recommended to use standard statements using { **LEFT** | **RIGHT** } [**OUTER**] **JOIN**, because using an Oracle-style join query statements by specifying an outer join operator (+) in the **WHERE** clause, even if possible, might lead the execution result or plan in an unwanted direction.

Example 1

The following is an example of retrieving the years and host countries of the Olympic Games since 1950 where a world record has been set. The following query retrieves instances whose values of the **host_year** column in the **history** table are greater than 1950.

```
SELECT DISTINCT h.host_year, o.host_nation FROM history h, olympic o
WHERE h.host_year=o.host_year AND o.host_year>1950;
=== <Result of SELECT Command in Line 2> ===
    host_year  host_nation
```

```
===================================
         1968  'Mexico'
         1980  'U.S.S.R.'
         1984  'United States of America'
         1988  'Korea'
         1992  'Spain'
         1996  'United States of America'
         2000  'Australia'
         2004  'Greece'
8 rows selected.
```

Example 2

The following is an example of retrieving the years and host countries of the Olympic Games since 1950 where a world record has been set, but including the Olympic Games where any world records haven't been set in the result. This example can be expressed in the following right outer join query. In this example, all instances whose values of the **host_year** column in the **history** table are not greater than 1950 are also retrieved. All instances of **host_nation** are included because this is a right outer join. **host_year** that does not have a value is represented as **NULL**.

```
SELECT DISTINCT h.host_year, o.host_nation
FROM history h RIGHT OUTER JOIN olympic o ON h.host_year=o.host_year WHERE
o.host_year>1950;
=== <Result of SELECT Command in Line 3> ===
    host_year  host_nation
===================================
         NULL  'Australia'
         NULL  'Canada'
         NULL  'Finland'
         NULL  'Germany'
         NULL  'Italy'
         NULL  'Japan'
         1968  'Mexico'
         1980  'U.S.S.R.'
         1984  'United States of America'
         1988  'Korea'
         1992  'Spain'
         1996  'United States of America'
         2000  'Australia'
         2004  'Greece'
14 rows selected.
```

Example 3

A right outer join query can be converted to a left outer join query by switching the position of two tables in the **FROM** clause. The right outer join query in the previous example can be expressed as a left outer join query as follows:

```
SELECT DISTINCT h.host_year, o.host_nation
FROM olympic o LEFT OUTER JOIN history h ON h.host_year=o.host_year WHERE o.host_year>1950;
=== <Result of SELECT Command in Line 3> ===
    host_year  host_nation
===================================
         NULL  'Australia'
         NULL  'Canada'
         NULL  'Finland'
         NULL  'Germany'
         NULL  'Italy'
         NULL  'Japan'
         1968  'Mexico'
         1980  'U.S.S.R.'
         1984  'United States of America'
         1988  'Korea'
         1992  'Spain'
         1996  'United States of America'
         2000  'Australia'
         2004  'Greece'
14 rows selected.
```

In this example, **h.host_year=o.host_year** is an outer join condition, and **o.host_year > 1950** is a search condition. If the search condition is used not in the **WHERE** clause but in the **ON** clause, the meaning and the result will be different. The following query also includes instances whose values of **o.host_year** are not greater than 1950.

```
SELECT DISTINCT h.host_year, o.host_nation
FROM olympic o LEFT OUTER JOIN history h ON h.host_year=o.host_year AND
o.host_year>1950;

=== <Result of SELECT Command in Line 3> ===
    host_year  host_nation
===================================
         NULL  'Australia'
         NULL  'Belgium'
         NULL  'Canada'
...
         1996  'United States of America'
         2000  'Australia'
         2004  'Greece'
22 rows selected.
```

Example 4

Outer joins can also be represented by using **(+)** in the **WHERE** clause. The above example is a query that has the same meaning as the example using the **LEFT OUTER JOIN**. The **(+)** syntax is not ISO/ANSI standard, so it can lead to ambiguous situations. It is recommended to use the standard syntax **LEFT OUTER JOIN** (or **RIGHT OUTER JOIN**) if possible.

```
SELECT DISTINCT h.host_year, o.host_nation FROM history h, olympic o
WHERE o.host_year=h.host_year(+) AND o.host_year>1950;
=== <Result of SELECT Command in Line 2> ===
    host_year  host_nation
===================================
         NULL  'Australia'
         NULL  'Canada'
         NULL  'Finland'
         NULL  'Germany'
         NULL  'Italy'
         NULL  'Japan'
         1968  'Mexico'
         1980  'U.S.S.R.'
         1984  'United States of America'
         1988  'Korea'
         1992  'Spain'
         1996  'United States of America'
         2000  'Australia'
         2004  'Greece'
14 rows selected.
```

Subquery

A subquery can be used wherever expressions such as **SELECT** or **WHERE** clause can be used. If the subquery is represented as an expression, it must return a single column; otherwise it can return multiple rows. Subqueries can be divided into single-row subqueries and multiple-row subqueries depending on how they are used.

Single-Row Subquery

Description

A single-row subquery outputs an row that has a single column. If no row is returned by the subquery, the subquery expression has a **NULL** value. If the subquery is supposed to return more than one rows, an error occurs.

Example

The following is an example of retrieving the history table as well as the host country where a new world record has been set. This example shows a single-row subquery used as an expression. In this example, the subquery returns **host_nation** values for the rows whose values of the **host_year** column in the **olympic** table are the same as those of the **host_year** column in the **history** table. If there are no values that meet the condition, the result of the subquery is **NULL**.

```
SELECT h.host_year, (SELECT host_nation FROM olympic o WHERE o.host_year=h.host_year),
h.event_code, h.score, h.unit from history h;
=== <Result of SELECT Command in Line 1> ===
    host_year (SELECT host_nation FROM olympic o WHERE
o.host_year=h.host_year)   event_code  score  unit
==========================================================================================
===================
         2004  'Greece'                                                           20283
'07:53.0' 'time'
         2004  'Greece'                                                           20283
'07:53.0' 'time'
         2004  'Greece'                                                           20281
'03:57.0' 'time'
         2004  'Greece'                                                           20281
'03:57.0' 'time'
         2004  'Greece'                                                           20281
'03:57.0' 'time'
         2004  'Greece'                                                           20281
'03:57.0' 'time'
         2004  'Greece'                                                           20326
'210'     'kg'
         2000  'Australia'                                                        20328
'225'     'kg'
         2004  'Greece'                                                           20331
'237.5'   'kg'
...
147 rows selected.
```

Multiple-Row Subquery

Description

A multiple-row subquery returns one or more rows that contain the specified column. The result of the mutiple-row subquery can be used to create a set, a multiset or a list/sequence set using an appropriate keyword (**SET**, **MULTISET**, **LIST** or **SEQUENCE**).

Example

The following is an example of retrieving countries and their capital cities from the nation table, and returning lists of host countries and host cities of the Olympic Games. In this example, the subquery result is used to create a list from the values of the **host_city** column in the **olympic** table. This query returns **name** and **capital** value for **nation** table, as well as a set that contains **host_city** values of the **olympic** table with **host_nation** value. If the **name** value is an empty set in the query result, it is excluded. If there is no **olympic** table that has the same value as the **name**, an empty set is returned.

```
SELECT name, capital, list(SELECT host_city FROM olympic WHERE host_nation = name) FROM
nation;

=== <Result of SELECT Command in Line 1> ===
  name                  capital               sequence((SELECT host_city FROM olympic
WHERE host_nation=name))
==================================================================
  'Somalia'             'Mogadishu'           {}
  'Sri Lanka'           'Sri Jayewardenepura Kotte'  {}
  'Sao Tome & Principe'  'Sao Tome'            {}
...
  'U.S.S.R.'            'Moscow'              {'Moscow'}
  'Uruguay'             'Montevideo'          {}
  'United States of America'  'Washington.D.C'      {'Atlanta ', 'St. Louis', 'Los
Angeles', 'Los Angeles'}
  'Uzbekistan'          'Tashkent'            {}
  'Vanuatu'             'Port Vila'           {}
215 rows selected.
```

Such multiple-row subquery expressions can be used anywhere a set value expression is allowed. However, they cannot be used where a set constant value is required as in the **DEFAULT** specification in the class attribute definition.

If the **ORDER BY** clause is not used explicitly in the subquery, the order of the multiple-row query result is not set. Therefore, the order of the multiple-row subquery result that creates a sequence set must be specified by using the **ORDER BY** clause.

Hierarchical Query

START WITH ... CONNECT BY Clause

Description

This clause is used to obtain a set of data organized in a hierarchy. The **START WITH ... CONNECT BY** clause is used in combination with the **SELECT** clause in the following form.

Syntax

```
SELECT column_list
    FROM table_joins | tables
    [WHERE join_conditions and/or filtering_conditions]
    [START WITH condition]
    CONNECT BY [NOCYCLE] condition
```

START WITH Clause

The START WITH clause will filter the rows from which the hierarchy will start. The rows that satisfy the START WITH condition will be the root nodes of the hierarchy. If START WITH is omitted, then all the rows will be considered as root nodes.

Note If START WITH clause is omitted or the rows that satisfy the START WITH condition does not exist, all of rows in the table are considered as root nodes; which means that hierarchy relationship of sub rows which belong each root is searched. Therefore, some of results can be duplicate.

CONNECT BY [NOCYCLE] or PRIOR Operator

- PRIOR : The CONNECT BY condition is tested for a pair of rows. If it evaluates to true, the two rows satisfy the parent-child relationship of the hierarchy. We need to specify the columns that are used from the parent row and the columns that are used from the child row. We can use the PRIOR operator when applied to a column, which will refer to the value of the parent row for that column. If PRIOR is not used for a column, the value in the child row is used.
- NOCYCLE : In some cases, the resulting rows of the table joins may contain cycles, depending on the CONNECT BY condition. Because cycles cause an infinite loop in the result tree construction, CUBRID detects them and either returns an error doesn't expand the branches beyond the point where a cycle is found (if the NOCYCLE keyword is specified).
  This keyword may be specified after the CONNECT BY keywords. It makes CUBRID run a statement even if the processed data contains cycles.
  If a CONNECT BY statement causes a cycle at runtime and the NOCYCLE keyword is not specified, CUBRID will return an error and the statement will be canceled. When specifying the NOCYCLE keyword, if CUBRID detects a cycle while processing a hierarchy node, it will set the CONNECT_BY_ISCYCLE attribute for that node to the value of 1 and it will stop further expansion of that branch.

Example

For the following samples, you will need the following structures:

Table tree

ID	MgrID	Name	BirthYear
1	NULL	KIM	1963
2	NULL	Moy	1958
3	1	Jonas	1976
4	1	Simth	1974

5	2	Verma	1973
6	2	Foster	1972
7	6	Brown	1981

Target tree_cycle

ID	MgrID	Name
1	NULL	Kim
2	11	Moy
3	1	Jonas
4	1	Smith
5	3	Verma
6	3	Foster
7	4	Brown
8	4	Lin
9	2	Edwin
10	9	Audrey
11	10	Stone

```
-- Creating tree table and then inserting data
CREATE TABLE tree(ID INT, MgrID INT, Name VARCHAR(32), BirthYear INT);

INSERT INTO tree VALUES (1,NULL,'Kim', 1963);
INSERT INTO tree VALUES (2,NULL,'Moy', 1958);
INSERT INTO tree VALUES (3,1,'Jonas', 1976);
INSERT INTO tree VALUES (4,1,'Smith', 1974);
INSERT INTO tree VALUES (5,2,'Verma', 1973);
INSERT INTO tree VALUES (6,2,'Foster', 1972);
INSERT INTO tree VALUES (7,6,'Brown', 1981);

-- Creating tree_cycle table and then inserting data
CREATE TABLE tree_cycle(ID INT, MgrID INT, Name VARCHAR(32));

INSERT INTO tree_cycle VALUES (1,NULL,'Kim');
INSERT INTO tree_cycle VALUES (2,11,'Moy');
INSERT INTO tree_cycle VALUES (3,1,'Jonas');
INSERT INTO tree_cycle VALUES (4,1,'Smith');
INSERT INTO tree_cycle VALUES (5,3,'Verma');
INSERT INTO tree_cycle VALUES (6,3,'Foster');
INSERT INTO tree_cycle VALUES (7,4,'Brown');
INSERT INTO tree_cycle VALUES (8,4,'Lin');
INSERT INTO tree_cycle VALUES (9,2,'Edwin');
INSERT INTO tree_cycle VALUES (10,9,'Audrey');
INSERT INTO tree_cycle VALUES (11,10,'Stone');

-- Executing a hierarchy query with CONNECT BY clause
SELECT id, mgrid, name
    FROM tree
    CONNECT BY PRIOR id=mgrid
    ORDER BY id;

id  mgrid       name
======================
1   null        Kim
2   null        Moy
3   1       Jonas
3   1       Jonas
4   1       Smith
4   1       Smith
5   2       Verma
```

```
5   2       Verma
6   2       Foster
6   2       Foster
7   6       Brown
7   6       Brown
7   6       Brown

-- Executing a hierarchy query with START WITH clause
SELECT id, mgrid, name
    FROM tree
    START WITH mgrid IS NULL
    CONNECT BY prior id=mgrid
    ORDER BY id;

id  mgrid       name
=============================
1   null        Kim
2   null        Moy
3   1       Jonas
4   1       Smith
5   2       Verma
6   2       Foster
7   6       Brown
```

Hierarchical Query for Table Join

Join Conditions

The table joins are evaluated first using the join conditions, if any. The conditions found in the **WHERE** clause are classified as join conditions or filtering conditions. All the conditions in the **FROM** clause are classified as join conditions. Only the join conditions are evaluated; the filtering conditions are kept for later evaluation. We recommended placing all join conditions in the **FROM** clause only so that conditions that are intended for joins are not mistakenly classified as filtering conditions.

Query Results

The resulting rows of the table joins are filtered according to the **START WITH** condition to obtain the root nodes for the hierarchy. If no **START WITH** condition is specified, then all the rows resulting from the table joins will be considered as root nodes.

After the root nodes are obtained, CUBRID will select the child rows for the root nodes. These are all nodes from the table joins that respect the **CONNECT BY** condition. This step will be repeated for the child nodes to determine their child nodes and so on until no more child nodes can be added.

In addition, CUBRID evaluates the **CONNECT BY** clause first and all the rows of the resulting hierarchy tress by using the filtering condition in the **WHERE** clause.

Example

The example illustrates how joins can be used in **CONNECT BY** queries. The joins are evaluated before the **CONNECT BY** condition and the join result will be the starting table on which the two clauses (**START WITH** clause and **CONNECT BY** clause).

```
-- Creating tree2 table and then inserting data
CREATE TABLE tree2(id int, treeid int, job varchar(32));

INSERT INTO tree2 VALUES(1,1,'Partner');
INSERT INTO tree2 VALUES(2,2,'Partner');
INSERT INTO tree2 VALUES(3,3,'Developer');
INSERT INTO tree2 VALUES(4,4,'Developer');
INSERT INTO tree2 VALUES(5,5,'Sales Exec.');
INSERT INTO tree2 VALUES(6,6,'Sales Exec.');
INSERT INTO tree2 VALUES(7,7,'Assistant');
INSERT INTO tree2 VALUES(8,null,'Secretary');

-- Executing a hierarchical query onto table joins
SELECT t.id,t.name,t2.job,level
```

```
    FROM tree t
        inner join tree2 t2 on t.id=t2.treeid
    START WITH t.mgrid is null
    CONNECT BY prior t.id=t.mgrid
    ORDER BY t.id;

id  name        job     level
================================================
1   Kim     Partner     1
2   Moy     Partner     1
3   Jonas       Developer   2
4   Smith       Developer   2
5   Verma       Sales Exec. 2
6   Foster      Sales Exec. 2
7   Brown       Assistant   3
```

Pseudo-Columns Available When Using the CONNECT BY Clause

LEVEL

This pseudo-column represents the level of the node in the hierarchy. Root nodes are considered to be at level 1, their children level 2 and so on.

The **LEVEL** pseudo-column may be used in the **SELECT** list, **WHERE** clause, **ORDER BY** clause, **GROUP BY ... HAVING** clauses and also in aggregate functions.

The following is an example of executing a hierarchical query with **LEVEL**.

```
-- Executing a hierarchical query with LEVEL
SELECT id, mgrid, name, LEVEL
    FROM tree
    WHERE LEVEL=2
    START WITH mgrid IS NULL
    CONNECT BY PRIOR id=mgrid
    ORDER BY id;
id    mgrid        name        level
=========================================
3    1            Jonas       2
4    1            Smith       2
5    2            Verma       2
6    2            Foster      2
```

CONNECT_BY_ISLEAF

This pseudo-column indicates whether a hierarchical node is a leaf node or not. If the value for a row is 1, then the associated node is a leaf node.; otherwise, it will have the value 0 indicating that the node has children.

In this example, the **CONNECT_BY_ISLEAF** shows that the rows with the IDs 3, 4, 5 and 7 have no children.

```
-- Executing a hierarchical query with CONNECT_BY_ISLEAF
SELECT id, mgrid, name, CONNECT_BY_ISLEAF
      FROM tree
      START WITH mgrid IS NULL
      CONNECT BY PRIOR id=mgrid
      ORDER BY id;

id    mgrid        name        connect_by_isleaf
===========================================================
1    null          Kim         0
2    null          Moy         0
3    1             Jonas       1
4    1             Smith       1
5    2             Verma       1
6    2             Foster      0
7    6             Brown       1
```

CONNECT_BY_ISCYCLE

This pseudo-column indicates that a cycle was detected while processing the node, meaning that a child was also found to be an ancestor. A value of 1 for a row means a cycle was detected; the pseudo-column's value is 0, otherwise.

The **CONNECT_BY_ISCYCLE** pseudo-column may be used in the **SELECT** list, **WHERE** clause, **ORDER BY** clause, **GROUP BY** and **HAVING** clauses and also in aggregate functions (when the **GROUP BY** class exists in the statement).

Note This pseudo-column is available only when the **NOCYCLE** keyword is used in the statement.

The following is an example of executing a hierarchical query with **CONNECT_BY_ISCYCE** operator.

```
-- --Executing a hierarchical query with CONNECT_BY_ISCYCLE
SELECT id, mgrid, name, CONNECT_BY_ISCYCLE
    FROM tree_cycle
    START WITH name in ('Kim', 'Moy')
    CONNECT BY NOCYCLE PRIOR id=mgrid
    ORDER BY id;

id  mgrid       name        connect_by_iscycle
==========================================================
1   null        Kim     0
2   11      Moy     0
3   1       Jonas       0
4   1       Smith       0
5   3       Verma       0
6   3       Foster      0
7   4       Brown       0
8   4       Lin     0
9   2       Edwin       0
10  9       Audrey      0
11  10      Stone       1
```

Operators Available When Using the CONNECT BY Clause

CONNECT_BY_ROOT Operator

This operator can be applied to columns and it returns the parent row or root row values for that column. This operator may be used in the **SELECT** list, **Where** clause and **ORDER BY** clause. When using the **CONNECT BY** clause some column operators become available.

The following is an example of executing a hierarchical query with **CONNECT_BY_ROOT** operator.

```
-- Executing a hierarchical query with CONNECT_BY_ROOT operator
SELECT id, mgrid, name, CONNECT_BY_ROOT id
    FROM tree
    START WITH mgrid IS NULL
    CONNECT BY PRIOR id=mgrid
    ORDER BY id;

id  mgrid       name        connect_by_root id
==========================================================
1   null        Kim     1
2   null        Moy     2
3   1       Jonas       1
4   1       Smith       1
5   2       Verma       2
6   2       Foster      2
7   6       Brown       2
```

PRIOR Operator

This operator may be applied to a column; it will return the parent node value for that column. For a root node, the operator will return the **NULL** value if it is applied to a column. This operator may be used in the **SELECT** list, **WHERE** clause, **ORDER BY** clause and also in the **CONNECT BY** clause.

The following is an example of executing a hierarchical query with **PRIOR** operator.

```
-- Executing a hierarchical query with PRIOR operator
SELECT id, mgrid, name, PRIOR id as "prior_id"
    FROM tree
    START WITH mgrid IS NULL
    CONNECT BY PRIOR id=mgrid
    ORDER BY id;

id  mgrid       name        prior_id
========================================
1   null        Kim     null
2   null        Moy     null
3   1       Jonas       1
4   1       Smith       1
5   2       Verma       2
6   2       Foster  2
7   6       Brown       6
```

Functions Available When Using the CONNECT BY Clause

Description

The **SYS_CONNECT_BY_PATH** function returns the branch of the node in the hierarchy. It returns a string that represents the concatenation of all the values obtained by evaluating the scalar expression for all the parents of a row, including that row, separated by the separator character, ordered ascending by level.

This function may be used in the **SELECT** list, **WHERE** clause and **ORDER BY** clause.

Syntax

```
SYS_CONNECT_BY_PATH (column_name, separator_char)
```

Example

The following is an example of executing a hierarchical query with **SYS_CONNECT_BY_PATH** function.

```
--Executing a hierarchical query with SYS_CONNECT_BY_PATH function
SELECT id, mgrid, name, SYS_CONNECT_BY_PATH(name,'/') as [hierarchy]
    FROM tree
    START WITH mgrid IS NULL
    CONNECT BY PRIOR id=mgrid
    ORDER BY id;

id  mgrid       name        hierarchy
=================================================
1   null        Kim     /Kim
2   null        Moy     /Moy
3   1       Jonas       /Kim/Jonas
4   1       Smith       /Kim/Smith
5   2       Verma       /Moy/Verma
6   2       Foster      /Moy/Foster
7   6       Brown       /Moy/Foster/Brown
```

Ordering Data with the Hierarchical Query

Description

The **ORDER SIBLINGS BY** clause will cause the ordering of the rows while preserving the hierarchy ordering so that the child nodes with the same parent will be stored according to the column list.

Syntax

```
ORDER SIBLINGS BY col_1 [ASC|DESC] [, col_2 [ASC|DESC] […[, col_n [ASC|DESC]]…]]
```

Example 1

The following example shows how to output information about seniors and subordinates in a company in the order of birth year.

The result with hierarchical query shows parent and child nodes in a row according to the column list specified in ORDER SIBLINGS BY statement by default. Sibling nodes that share the same parent node have outputted in a specified order.

```
-- Outputting a parent node and its child nodes, which sibling nodes that share the same
parent are sorted in the order of birth year.
SELECT id, mgrid, name, birthyear, level
FROM tree
START WITH mgrid IS NULL
CONNECT BY PRIOR id=mgrid
ORDER SIBLINGS BY birthyear;

id        mgrid  name                    birthyear        level
===========================================================================
2         NULL  'Moy'                        1958           1
6            2  'Foster'                     1972           2
7            6  'Brown'                      1981           3
5            2  'Verma'                      1973           2
1         NULL  'Kim'                        1963           1
4            1  'Smith'                      1974           2
3            1  'Jonas'                      1976           2
```

Example 2

The following example shows how to output information about seniors and subordinates in a company in the order of joining. For the same level, the employee ID numbers are assigned in the order of joining. id indicates employee ID numbers (parent and child nodes) and mgrid indicates the employee ID numbers of their seniors.

```
-- Outputting siblings in a row
SELECT id, mgrid, name, LEVEL
FROM tree
START WITH mgrid IS NULL
CONNECT BY PRIOR id=mgrid
ORDER SIBLINGS BY id;

id  mgrid       name        level
===============================================
1   null        Kim     1
3   1       Jonas       2
4   1       Smith       2
2   null        Moy     1
5   2       Verma       2
6   2       Foster      2
7   6       Brown       3
```

Scenario of Using Hierarchical Query

First of all let's start by giving a rough SQL translation of the **SELECT** statement with a **CONNECT BY** clause. For this we can consider that we have a table that contains a recurrent reference. We can consider that table to have two columns named ID and ParentID; ID is the primary key for the table and ParentID is a foreign-key to the same table. Naturally, the root nodes will have a ParentID value of **NULL**.

Now let us consider the fact that we want to get the full rows and a column with the level of the row in the hierarchy tree. For this we can write something similar to by querying with **UNION ALL**.

```
SELECT L1.ID, L1.ParentID, ..., 1 AS [Level]
    FROM tree_table AS L1
    WHERE L1.ParentID IS NULL
UNION ALL
SELECT L2.ID, L2.ParentID, ..., 2 AS [Level]
    FROM tree_table AS L1
        INNER JOIN tree_table AS L2 ON L1.ID=L2.ParentID
    WHERE L1.ParentID IS NULL
UNION ALL
SELECT L3.ID, L3.ParentID, ..., 3 AS [Level]
    FROM tree_table AS L1
        INNER JOIN tree_table AS L2 ON L1.ID=L2.ParentID
        INNER JOIN tree_table AS L3 ON L2.ID=L3.ParentID
    WHERE L1.ParentID IS NULL
UNION ALL ...
```

The problem with our approach is that we do not know how many levels we have. This could be rewritten in a stored procedure with a cycle until no new rows are retrieved, but we will have to check the tree for cycles at every step. Using a **SELECT** statement with a **CONNECT BY** clause we can rewrite this as follows.

This query will return the full hierarchy with the level of each row in the hierarchy.

```
SELECT ID, ParentID, ..., Level
    FROM tree_table
    START WITH ParentID IS NULL
    CONNECT BY ParentID=PRIOR ID
```

If we want to avoid the potential error caused by cycles we can write it as follows:

```
SELECT ID, ParentID, ..., Level
    FROM tree_table
    START WITH ParentID IS NULL
    CONNECT BY NOCYCLE ParentID=PRIOR ID
```

Performance of Hierarchical Query

Although this form is shorter and clearer, please keep in mind that it has its limitations regarding speed. If the result of the query contains all the rows of the table, the **CONNECT BY** form might be slower as it has to do additional processing (such as cycle detection, pseudo-column bookkeeping and others). However, if the result of the query only contains a part of the table rows, the **CONNECT BY** form might be faster.
For example, if we have a table with 20,000 records and we want to retrieve a sub-tree of roughly 1,000 records, a **SELECT** statement with a **START WITH ... CONNECT BY** clause will run up to 30% faster than an equivalent **UNION ALL** with **SELECT** statements.

INSERT

Overview

Description

You can insert a new record into a table in a database by using the **INSERT** statement. CUBRID supports **INSERT...VALUES**, **INSERT...SET** and **INSERT...SELECT** statements.

INSERT...VALUES and **INSERT...SET** statements are used to insert a new record based on the value that is explicitly specified while the **INSERT...SELECT** statement is used to insert query result records obtained from different tables. Use the **INSERT VALUES** or **INSERT...SELECT** statement to insert multiple rows by using the single **INSERT** statement.

Syntax

```
<INSERT … VALUES statement>
INSERT [INTO] table_name [(column_name, ...)]
    {VALUES | VALUE}({expr | DEFAULT}, ...)[,({expr | DEFAULT}, ...),...]
    [ON DUPLICATE KEY UPDATE column_name = expr, ... ]
INSERT [INTO] table_name DEFAULT [ VALUES ]
INSERT [INTO] table_name VALUES()

<INSERT … SET statement>
INSERT [INTO] table_name
    SET column_name = {expr | DEFAULT}[, column_name = {expr | DEFAULT},...]
    [ON DUPLICATE KEY UPDATE column_name = expr, ... ]

<INSERT … SELECT statement>
INSERT [INTO] table_name [(column_name, ...)]
    SELECT...
    [ON DUPLICATE KEY UPDATE column_name = expr, ... ]
```

- *table_name* : Specify the name of the target table into which you want to insert a new record.
- *column_name* : Specify the name of the column into which you want to insert the value. If you omit to specify the column name, it is considered that all columns defined in the table have been specified. Therefore, you must specify the values for all columns next to the **VALUES** keyword. If you do not specify all the columns defined in the table, a **DEFAULT** value is assigned to the non-specified columns; if the **DEFAULT** value is not defined, a **NULL** value is assigned.

- *expr* | **DEFAULT** : Specify values that correspond to the columns next to the **VALUES** keyword. Expressions or the **DEFAULT** keyword can be specified as a value. At this time, the order and number of the specified column list must correspond to the column value list. The column value list for a single record is described in parentheses.
- **DEFAULT** : You can use the **DEFAULT** keyword to specify a default value as the column value. If you specify **DEFAULT** in the column value list next to the **VALUES** keyword, a default value column is saved for the given column: if you specify **DEFAUL**T before the **VALUES** keyword, default values are saved for all columns in the table. **NULL** is saved for the column whose default value has not been defined.
- **ON DUPLICATE KEY UPDATE** : In case constraints are violated because a duplicated value for a column where **PRIMARY KEY** or **UNIQUE** attribute is defined is inserted, the value that makes constraints violated is changed into a specific value by performing the action specified in the **ON DUPLICATE KEY UPDATE** statement.

Example

```
CREATE TABLE a_tbl1(
id INT UNIQUE,
name VARCHAR,
phone VARCHAR DEFAULT '000-0000');

--insert default values with DEFAULT keyword before VALUES
INSERT INTO a_tbl1 DEFAULT VALUES;

--insert multiple rows
INSERT INTO a_tbl1 VALUES (1,'aaa', DEFAULT),(2,'bbb', DEFAULT);

--insert a single row specifying column values for all
INSERT INTO a_tbl1 VALUES (3,'ccc', '333-3333');

--insert two rows specifying column values for only
INSERT INTO a_tbl1(id) VALUES (4), (5);

--insert a single row with SET clauses
INSERT INTO a_tbl1 SET id=6, name='eee';
INSERT INTO a_tbl1 SET id=7, phone='777-7777';

SELECT * FROM a_tbl1;
;xr

=== <Result of SELECT Command in Line 1> ===

           id  name                  phone
=========================================================
         NULL  NULL                  '000-0000'
            1  'aaa'                 '000-0000'
            2  'bbb'                 '000-0000'
            3  'ccc'                 '333-3333'
            4  NULL                  '000-0000'
            5  NULL                  '000-0000'
            6  'eee'                 '000-0000'
            7  NULL                  '777-7777'


8 rows selected.
```

INSERT ... SELECT Statement

Description

If you use the **SELECT** query in the **INSERT** statement, you can insert query results obtained from at least one table. The **SELECT** statement can be used in place of the **VALUES** keyword, or be included as a subquery in the column value list next to **VALUES**. If you specify the **SELECT** statement in place of the **VALUES** keyword, you can insert multiple query result records into the column of the table at once. However, there should be only one query result record if the **SELECT** statement is specified in the column value list.

In this way, you can extract data from another table that satisfies a certain retrieval condition, and insert it into the target table by combining the **SELECT** statement with the **INSERT** statement.

Syntax

```
INSERT [INTO] table_name [(column_name, ...)]
    SELECT...
    [ON DUPLICATE KEY UPDATE column_name = expr, ... ]
```

Example

```
--creating an empty table which schema replicated from a_tbl1
CREATE TABLE a_tbl2 LIKE a_tbl1;

--inserting multiple rows from SELECT query results
INSERT INTO a_tbl2 SELECT * FROM a_tbl1 WHERE id IS NOT NULL;

7 rows affected

--inserting column value with SELECT subquery specified in the value list
INSERT INTO a_tbl2 VALUES(8, SELECT name FROM a_tbl1 WHERE name <'bbb', DEFAULT);

SELECT * FROM a_tbl2;
csql> ;xr

=== <Result of SELECT Command in Line 1> ===

           id  name                  phone
=========================================================
            1  'aaa'                 '000-0000'
            2  'bbb'                 '000-0000'
            3  'ccc'                 '333-3333'
            4  NULL                  '000-0000'
            5  NULL                  '000-0000'
            6  'eee'                 '000-0000'
            7  NULL                  '777-7777'
            8  'aaa'                 '000-0000'


8 rows selected.
```

ON DUPLICATE KEY UPDATE Statement

Description

In a situation in which a duplicate value is inserted into a column for which the **UNIQUE** index or the **PRIMARY KEY** constraint has been set, you can update to a new value without outputting the error by specifying the **ON DUPLICATE KEY UPDATE** clause in the **INSERT** statement.

However, the **ON DUPLICATE KEY UPDATE** clause cannot be used in a table in which a trigger for **INSERT** or **UPDATE** has been activated, or in a nested **INSERT** statement.

Syntax

```
<INSERT … VALUES statement>
<INSERT … SET statement>
<INSERT … SELECT statement>
    INSERT ...
    [ON DUPLICATE KEY UPDATE column_name = expr, ... ]
```

- *column_name* = *expr* : Specifies the name of the column whose value you want to change next to **ON DUPLICATE KEY UPDATE** and a new column value by using the equal sign.

Example

```
--creating a new table having the same schema as a_tbl1
CREATE TABLE a_tbl3 LIKE a_tbl1;
INSERT INTO a_tbl3 SELECT * FROM a_tbl1 WHERE id IS NOT NULL and name IS NOT NULL;
SELECT * FROM a_tbl3;
;xr

=== <Result of SELECT Command in Line 1> ===
           id  name                  phone
=========================================================
            1  'aaa'                 '000-0000'
            2  'bbb'                 '000-0000'
            3  'ccc'                 '333-3333'
            6  'eee'                 '000-0000'

--insert duplicated value violating UNIQUE constraint
INSERT INTO a_tbl3 VALUES(2, 'bbb', '222-2222');
;xr

In the command from line 1,

ERROR: Operation would have caused one or more unique constraint violations.

--insert duplicated value with specifying ON DUPLICATED KEY UPDATE clause
INSERT INTO a_tbl3 VALUES(2, 'bbb', '222-2222')
ON DUPLICATE KEY UPDATE phone = '222-2222';

SELECT * FROM a_tbl3 WHERE id=2;
;xr

=== <Result of SELECT Command in Line 1> ===

           id  name                  phone
=========================================================
            2  'bbb'                 '222-2222'


1 rows selected.
```

UPDATE

Description

You can update the column value of a record saved in the target table to a new one by using the **UPDATE** statement. Specify the name of the column to update and a new value in the **SET** clause, and specify the condition to be used to extract the record to be updated in the WHERE Clause. You can also specify the number of records to be updated in the **LIMIT** clause.

Syntax

```
UPDATE table_name SET column_name = {expr | DEFAULT} [, column_name = {expr | DEFAULT}...]
    [WHERE search_condition]
    [LIMIT row_count]
```

- *table_name* : Specify the name of the table to be updated.
- *column_name* : Specify the columns to be updated.
- *expr* | **DEFAULT** : Specify a new value for the column, and specify an expression or the **DEFAULT** keyword as the value. You can also specify the **SELECT** query, which returns a single result record.
- *search_condition* : You can update the column value only for the record that satisfies the condition by specifying one in the WHERE Clause.
- *row_count* : Specify the number of records to be updated after the LIMIT Clause. An integer greater than 0 can be specified.

Remark

One column can be updated only once in the same **UPDATE** statement.

Example

```
--creating a new table having all records copied from a_tbl1
CREATE TABLE a_tbl5 AS SELECT * FROM a_tbl1;
SELECT * FROM a_tbl5 WHERE name IS NULL;
;xr

=== <Result of SELECT Command in Line 1> ===
           id  name                  phone
=========================================================
         NULL  NULL                  '000-0000'
            4  NULL                  '000-0000'
            5  NULL                  '000-0000'
            7  NULL                  '777-7777'


4 rows selected.

UPDATE a_tbl5 SET name='yyy', phone='999-9999' WHERE name IS NULL LIMIT 3;
SELECT * FROM a_tbl5;
;xr

=== <Result of SELECT Command in Line 1> ===

           id  name                  phone
=========================================================
         NULL  'yyy'                 '999-9999'
            1  'aaa'                 '000-0000'
            2  'bbb'                 '000-0000'
            3  'ccc'                 '333-3333'
            4  'yyy'                 '999-9999'
            5  'yyy'                 '999-9999'
            6  'eee'                 '000-0000'
            7  NULL                  '777-7777'


8 rows selected.
```

REPLACE

Description

The **REPLACE** statement is working like **INSERT**, but the difference is that it inserts a new record after deleting the existing record without displaying the error when a duplicate value is inserted into a column for which **PRIMARY KEY** and **UNIQUE** constraints have defined. You must have both **INSERT** and **DELETE** privileges to use the **REPLACE** statement, because it performs insertion or insertion after deletion operations.

The **REPLACE** statement determines whether a new record causes the duplication of **PRIMARY KEY** or **UNIQUE** index column values. Therefore, for performance reasons, it is recommended to use the **INSERT** statement for a table for which a **PRIMARY KEY** or **UNIQUE** index has not been defined. The **REPLACE** statement is an extension of the SQL standard. See the following regarding the use of this statement.

- The **REPLACE** statement cannot contain subqueries.
- The **REPLACE** statement cannot be used for tables for which an **INSERT** or **DELETE** trigger has been set.
- An assignment statement such as **SET** *col_name* = *col_name* + 1 is not valid. Change such a statement to **SET** *col_name* = **DEFAULT**(*col_name*) + 1. Here, a non-NULL default value should be set for the *col_name* column.

Syntax

```
<REPLACE … VALUES statement>
REPLACE [INTO] table_name [(column_name, ...)]
    {VALUES | VALUE}({expr | DEFAULT}, ...)[,({expr | DEFAULT}, ...),...]

<REPLACE … SET statement>
REPLACE [INTO] table_name
    SET column_name = {expr | DEFAULT}[, column_name = {expr | DEFAULT},...]

<REPLACE … SELECT statement>
REPLACE [INTO] table_name [(column_name, ...)]
    SELECT...
```

- *table_name* : Specify the name of the target table into which you want to insert a new record.
- *column_name* : Specify the name of the column into which you want to insert the value. If you omit to specify the column name, it is considered that all columns defined in the table have been specified. Therefore, you must specify

the value for the column next to **VALUES**. If you do not specify all the columns defined in the table, a **DEFAULT** value is assigned to the non-specified columns; if the **DEFAULT** value is not defined, a NULL value is assigned.

- *expr* | **DEFAULT** : Specify values that correspond to the columns after **VALUES**. Expressions or the **DEFAULT** keyword can be specified as a value. At this time, the order and number of the specified column list must correspond to the column value list. The column value list for a single record is described in parentheses.

Example

```
--creating a new table having the same schema as a_tbl1
CREATE TABLE a_tbl4 LIKE a_tbl1;
INSERT INTO a_tbl4 SELECT * FROM a_tbl1 WHERE id IS NOT NULL and name IS NOT NULL;
SELECT * FROM a_tbl4;
;xr

=== <Result of SELECT Command in Line 1> ===

           id  name                  phone
=========================================================
            1  'aaa'                 '000-0000'
            2  'bbb'                 '000-0000'
            3  'ccc'                 '333-3333'
            6  'eee'                 '000-0000'

--insert duplicated value violating UNIQUE constraint
REPLACE INTO a_tbl4 VALUES(1, 'aaa', '111-1111'),(2, 'bbb', '222-2222');
REPLACE INTO a_tbl4 SET id=6, name='fff', phone=DEFAULT;

SELECT * FROM a_tbl4;
;xr

=== <Result of SELECT Command in Line 1> ===

           id  name                  phone
=========================================================
            3  'ccc'                 '333-3333'
            1  'aaa'                 '111-1111'
            2  'bbb'                 '222-2222'
            6  'fff'                 '000-0000'


4 rows selected.
```

DELETE

Description

You can delete records in the table by using the **DELETE** statement. You can specify delete conditions by combining the statement with the WHERE Clause . If you want to limit the number of records to be deleted, you can do so by specifying the number of records to be deleted after the LIMIT Clause. In this case, only **row_count** records are deleted even when the number of records satisfying the WHERE Clause exceeds **row_count**.

Syntax

```
DELETE FROM <table_specification> [ WHERE <search_condition> ] [LIMIT row_count]

<table_specification> ::= <table_hierarchy> | ( <table_hierarchy_comma_list )

<table_hierarchy> ::= [ ONLY ] <table_name> |
                      ALL <table_name> [ EXCEPT <table_specification> ]
```

- *table_name* : Specifies the name of the table that contains the data to be deleted.
- *search_condition* : Delete only the data that meets the *search_condition* by using the WHERE Clause. If it is not specified, all the data in the table will be deleted.
- *row_count* : Specify the number of records to be deleted after the LIMIT Clause. An integer greater than 0 can be specified.

Example

```
CREATE TABLE a_tbl(
id INT NOT NULL,
phone VARCHAR(10));
INSERT INTO a_tbl VALUES(1,'111-1111'), (2,'222-2222'), (3, '333-3333'), (4, NULL), (5,
NULL);
;xr

5 rows affected

DELETE FROM a_tbl WHERE phone IS NULL LIMIT 1;
;xr

1 rows affected.

--delete one record only from a_tbl
SELECT * FROM a_tbl;
;xr

=== <Result of SELECT Command in Line 1> ===

           id  phone
===================================
            1  '111-1111'
            2  '222-2222'
            3  '333-3333'
            5  NULL


4 rows selected.

--delete all records from a_tbl
DELETE FROM a_tbl;
;xr

4 rows affected.
```

TRUNCATE

Description

You can delete all records in the specified table by using the **TRUNCATE** statement.

This statement internally delete first all indexes and constraints defined in a table and then deletes all records. Therefore, it performs the job faster than using the **DELETE FROM** *table_name* statement without a **WHERE** clause.

If the **PRIMARY KEY** constraint is defined in the table and this is referred by one or more **FOREIGN KEY**, it follows the **FOREIGN KEY ACTION**. If the **ON DELETE** action of **FOREIGN KEY** is **RESTRICT** or **NO_ACTION**, the **TRUNCATE** statement returns an error. If it is **CASCADE**, it deletes **FOREIGN KEY**. The **TRUNCATE** statement initializes the **AUTO INCREMENT** column of the table. Therefore, if data is inserted, the **AUTO INCREMENT** column value increases from the initial value.

Syntax

```
TRUNCATE [ TABLE ] <table_name>
```

- *table_name* : Specify the name of the table that contains the data to be deleted.

Example

```
CREATE TABLE a_tbl(A INT AUTO_INCREMENT(3,10) PRIMARY KEY);
INSERT INTO a_tbl VALUES (NULL),(NULL),(NULL);
SELECT * FROM a_tbl;

=============
            3
            13
            23
```

```
--AUTO INCREMENT column value increases from the initial value after truncating the table
TRUNCATE TABLE a_tbl;
INSERT INTO a_tbl VALUES (NULL);
SELECT * FROM a_tbl;
=============
            3
```

DO

Description

The **DO** statement executes the specified expression, but does not return the result. Generally, the execution speed of the **DO** statement is higher than that of the **SELECT** expression statement, because the database server does not return the operation result or errors.

Syntax

```
DO expression
```

- *expression* : Specify an expression.

PREPARED STATEMENT

Overview

You can execute it directly at the SQL level in CUBRID 2008 R3.0 or higher because the prepared statement is provided at the server side from CUBRID 2008 R3.0. However, the prepared statement area at the SQL level is limited to the client session in which the SQLs are created and the previous prepared statement cannot be executed after SQL statements are execited with **;run** or **;xrun**.

Provide the following SQL statements to use the prepared statement.

- Prepare the SQL statement to execute.

  ```
  PREPARE stmt_name FROM preparable_stmt
  ```

- Execute the prepared statement.

  ```
  EXECUTE stmt_name [USING value [, value] …]
  ```

- Drop the prepared statement.

  ```
  {DEALLOCATE | DROP} PREPARE stmt_name
  ```

PREPARE Statement

Description

The **PREPARE** statement prepares the query specified in *preparable_stmt* of the **FROM** clause, and assigns the name to be used later when the SQL statement is referenced to *stmt_name*.

Syntax

```
PREPARE stmt_name FROM preparable_stmt
```

- *stmt_name* : The prepared statement is specified. If an SQL statement with the same *stmt_name* exists in the given client session, clear the existing prepared statement and prepare a new SQL statement. If the **PREPARE** statement is not executed properly due to an error in the given SQL statement, it is processed as if the *stmt_name* assigned to the SQL statement does not exist.
- *preparable_stmt* : You must use only one SQL statement. Multiple SQL statements cannot be specified. You can use a question mark (?) as a bind parameter in the *preparable_stmt* statement and it should not be enclosed with quotes.

Example

```
--prepare and execute a statement without any parameter marker
PREPARE stmt1 FROM 'SELECT CURRENT_TIMESTAMP ()';
EXECUTE stmt1;
;ru

Current transaction has been committed.

=== <Result of SELECT Command in Line 2> ===

   SYS_TIMESTAMP
===========================
  03:49:28 PM 04/02/2010

--prepare and execute a statement with a parameter marker
PREPARE stmt1 FROM 'SELECT POWER(?,2)*PI()';
EXECUTE stmt1 USING 2;
;ru

=== <Result of SELECT Command in Line 2> ===

     power( ?:0 , 2)* pi()
==========================
     1.256637061435917e+01
```

EXECUTE Statement

Description

The **EXECUTE** statement executes the prepared statement. You can bind the data value after the **USING** clause if a bind parameter (?) is included in the prepared statement. You cannot specify user-defined variables like an attribute in the **USING** clause. An value such as literal and an input parameter only can be specified.

Syntax

```
EXECUTE stmt_name [USING value [, value] …]
```

- *stmt_name* : The name given to the prepared statement to be executed is specified. An error message is displayed if the *stmt_name* is not valid, or if the prepared statement does not exist.
- *value* : The data to bind is specified if there is a bind parameter in the prepared statement. The number and the order of the data must correspond to that of the bind parameter. If it does not, an error message is displayed.

Example

```
;clear
EXECUTE stmt1
;ru

ERROR: A prepared statement with the name stmt1 does not exist.

--prepare and execute a statement
PREPARE stmt2 FROM 'SELECT MID(?,?,?)';
EXECUTE stmt2 USING '12345abcdeabcde',6,5;
EXECUTE stmt2 USING '12345abcdeabcde',6,10;
;ru

Current transaction has been committed.

=== <Result of SELECT Command in Line 1> ===

   mid( ?:0 ,  ?:1 ,  ?:2 )
======================
  'abcde'

=== <Result of SELECT Command in Line 2> ===

   mid( ?:0 ,  ?:1 ,  ?:2 )
======================
  'abcdeabcde'
```

DEALLOCATE PREPARE, DROP PREPARE Statement

Description

DEALLOCATE PREPARE and **DROP PREPARE** are used interchangeably and they clear the prepared statement. All prepared statements are cleared automatically by the server when the client session is terminated even if the **DEALLOCATE PREPARE** or **DROP PREPARE** statement is not executed.

Syntax

```
{DEALLOCATE | DROP} PREPARE stmt_name
```

- *stmt_name* : The name given to the prepared statement to be cleared is specified. An error message is displayed if the *stmt_name* is not valid, or if the prepared statement does not exist.

Example

```
DEALLOCATE PREPARE stmt1;
DEALLOCATE PREPARE stmt2;
;xr

2 command(s) successfully processed.
```

Transaction and Lock

Overview

This chapter covers issues relating to concurrency and restore, as well as how to commit or roll back transactions.

In multi-user environment, controlling access and update is essential to protect database integrity and ensure that a user's transaction will have accurate and consistent data. Without appropriate control, data could be updated incorrectly in the wrong order.

To control parallel operations on the same data, data must be locked during transaction, and unacceptable access to the data by another transaction must be blocked until the end of the transaction. In addition, any updates to a certain class must not be seen by other users before they are committed. If updates are not committed, all queries entered after the last commit or rollback of the update can be invalidated.

All examples introduced here were executed by csql. Outputs in the examples are displayed in *italics*.

Database Transaction

Overview

A database transaction groups CUBRID queries into a unit of consistency (for ensuring valid results in multi-user environment) and restore (for making the results of committed transactions permanent and ensuring that the aborted transactions are canceled in the database despite any failure, such as system failure). A transaction is a collection of one or more queries that access and update the database.

CUBRID allows multiple users to access the database simultaneously and manages accesses and updates to prevent inconsistency of the database. For example, if data is updated by one user, the changes made by this transaction are not seen to other users or the database until the updates are committed. This principle is important because the transaction can be rolled back without being committed.

You can delay permanent updates to the database until you are confident of the transaction result. Also, you can remove (**ROLLBACK**) all updates in the database if an unsatisfactory result or failure occurs in the application or computer system during the transaction. The end of the transaction is determined by the **COMMIT WORK** or **ROLLBACK WORK** statement. The **COMMIT WORK** statement makes all updates permanent while the **ROLLBACK WORK** statement cancels all updates entered in the transaction. For more information, see the Transaction Commit and Transaction Rollback sections.

Transaction Commit

Description

Updates that occurred in the database are not permanently stored until the **COMMIT WORK** statement is executed. "Permanently stored" means that storing the updates in the disk is completed; The **WORK** keyword can be omitted. In addition, other users of the database cannot see the updates until they are permanently applied. For example, when a new row is inserted into a class, only the user who inserted the row can access it until the database transaction is committed. (If the **UNCOMMITTED INSTANCES** isolation level is used, other users can see inconsistent uncommitted updates.)

All locks obtained by the transaction are released after the transaction is committed.

Syntax

```
[;]COMMIT [ WORK ]
```

If you place a semicolon (;) before the statement, the statement is considered as a session command and is executed immediately. If you don't, the statement is considered as a query statement and the execution is delayed until ;x[run] is executed.

Example

The database transaction in the following example consists of three **UPDATE** statements and changes three column values of seats from the stadium. To compare the results, check the current values and names before the update is made. Since, by default, csql runs in an autocommit mode, the following example is executed after setting the autocommit mode to off.

```
;autocommit off
AUTOCOMMIT IS OFF
select name, seats
from stadium where code in (30138, 30139, 30140);
;xrun
=== <Result of SELECT Command in Line 1>===
    name                        seats
==================================
    'Athens Olympic Tennis Centre'         3200
    'Goudi Olympic Hall'         5000
    'Vouliagmeni Olympic Centre'         3400
3 rows selected.
update stadium
set seats = seats + 1000
where code in (30138, 30139, 30140);
;xrun
3 rows affected.
select name, seats from stadium where code in (30138, 30139, 30140);
;xrun
=== <Result of SELECT Command in Line 1>===
    name                        seats
===================================
    'Athens Olympic Tennis Centre'         4200
   'Goudi Olympic Hall'         6000
    'Vouliagmeni Olympic Centre'         4400

3 rows selected.
;commit work
```

Note In CUBRID, an auto-commit mode is set by default for transaction management. An auto-commit mode is a mode that commits or rolls back all SQL statements. The transaction is committed automatically if the SQL is executed successfully, or is rolled back automatically if an error occurs.
Such auto commit modes are supported in CUBRID JDBC, ODBC, OLEDB and the CSQL Interpreter. In CUBRID CCI and CUBRID PHP, auto commit modes can be applied only for SELECT statements by setting broker parameters. For details, see Parameter by Broker. For a session command (;AUtocommit) that sets the auto-commit mode in the CSQL interpreter, see Session Commands.

Transaction Rollback

Description

The **ROLLBACK WORK** statement removes all updates to the database since the last transaction. The **WORK** keyword can be omitted. By using this statement, you can cancel incorrect or unnecessary updates before they are permanently applied to the database. All locks obtained during the transaction are released.

Syntax

```
[;]ROLLBACK [ WORK ]
```

If you place a semicolon (;) before the statement, the statement is considered as a session command and is executed immediately. If you don't, the statement is considered as a query statement and the execution is delayed until ;x[run] is executed.

Example

The following example shows two commands that modify the definition and the row of the same table.

```
alter class code
drop s_name;
insert into code (s_name, f_name) values ('D','Diamond');
;xrun

In line 3, column 21,
ERROR: s_name is not defined.
```

The **INSERT** statement fails because the s_name column has been dropped in the definition of code. The data intended to be entered to the code table is correct, but the s_name column is wrongly removed. At this point, you can use the **ROLLBACK WORK** statement to restore the original definition of the code table.

```
;rollback work
```

Later, remove the s_name column by entering the **ALTER TABLE** again and modify the **INSERT** statement. The **INSERT** command must be entered again because the transaction has been aborted. If the database update has been done as intended, commit the transaction to make the changes permanent.

```
alter class code
drop s_name;

insert into code (f_name)
values ('Diamond');

;commit work
```

Savepoint and Partial Rollback

Description

A savepoint is established during the transaction so that database changes made by the transaction are rolled back to the specified savepoint. Such operation is called a partial rollback. In a partial rollback, database operations (insert, update, delete, etc.) after the savepoint are rolled back, and transaction operations before it are not rolled back. The transaction can proceed with other operations after the partial rollback is executed. Or the transaction can be terminated with the **COMMIT WORK** or **ROLLBACK WORK** statement. Note that the savepoint does not commit the changes made by the transaction.

A savepoint can be created at a certain point of the transaction, and multiple savepoints can be used for a certain point. If a partial rollback is executed to a savepoint before the specified savepoint or the transaction is terminated with the **COMMIT WORK** or **ROLLBACK WORK** statement, the specified savepoint is removed. The partial rollback after the specified savepoint can be performed multiple times.

Savepoints are useful because intermediate steps can be created and named to control long and complicated utilities. For example, if you use a savepoint during the update operation, you don't need to perform all statements again when you made a mistake.

Syntax 1

```
SAVEPOINT mark
mark:
_ a SQL identifier
_ a host variable (starting with :)
```

If you make *mark* all the same value when you specify multiple savepoints in a single transaction, only the latest savepoint appears in the partial rollback. The previous savepoints remain hidden until the rollback to the latest savepoint is performed and then appears when the latest savepoint disappears after being used.

Syntax 2

```
ROLLBACK [ WORK ] [ TO [ SAVEPOINT ] mark ] [ ]
mark:
_ a SQL identifier
_ a host variable (starting with :)
```

Previously, the **ROLLBACK WORK** statement canceled all database changes added since the latest transaction. The **ROLLBACK WORK** statement is also used for the partial rollback that rolls back the transaction changes after the specified savepoint.

If *mark* value is not given, the transaction terminates canceling all changes including all savepoints created in the transaction. If *mark* value is given, changes after the specified savepoint are canceled and the ones before it are remained.

Example

The following is an example of rolling back part of the transaction.

First, set savepoints SP1 and SP2.

```
create class athlete2 (name varchar(40), gender char(1), nation_code char(3), event
varchar(30))
insert into athlete2(name, gender, nation_code, event)
values ('Lim Kye-Sook', 'W', 'KOR', 'Hockey')
savepoint SP1

select * from athlete2
insert into athlete2(name, gender, nation_code, event)
values ('Lim Jin-Suk', 'M', 'KOR', 'Handball')

select * from athlete2
savepoint SP2

rename class athlete2 as sportsman
select * from sportsman
rollback work to SP2
xrun
```

In the example above, the name change of the athlete2 table is rolled back by the partial rollback. The following is an example of executing the query with the original name and examining the result.

```
select * from athlete2
delete from athlete2 where name = 'Lim Jin-Suk'
select * from athlete2
rollback work to SP2
xrun
```

In the example above, deleting 'Lim Jin-Suk' is canceled by the rollback work to SP2 statement.

The following is an example of rolling back to SP1.

```
select * from athlete2
rollback work to SP1
select * from athlete2
commit work
xrun
```

Database Concurrency

If there are multiple users with read and write privileges in a database, possibility exists that more than one user will access the database simultaneously. Controlling access and update in multi-user environment is essential to protect database integrity and ensure that users and transactions should have accurate and consistent data. Without appropriate control, data could be updated incorrectly in the wrong order.

Like most commercial database systems, CUBRID adopts serializability, an element that is essential to maintaining data concurrency within the database. Serializability ensures no interference between transactions when multiple transactions are executed at the same time. It is guaranteed more with the higher isolation level. This principle is based on the assumption that database consistency is guaranteed as long as transaction is executed automatically. This will be covered in the Lock Protocol section in detail.

The transaction must ensure database concurrency, and each transaction must guarantee appropriate results. When multiple transactions are being executed at the same time, an event in transaction T1 should not affect an event in transaction T2. This means isolation. Transaction isolation level is the degree to which a transaction is separated from all other concurrent transactions. The higher isolation level means the lower interference from other transactions. The lower isolation level means the higher the concurrency. A database determines whether which lock is applied to tables and records based on these isolation levels. Therefore, can control the level of consistency and concurrency specific to a service by setting appropriate isolation level.

You can set an isolation level by using the SET TRANSACTION ISOLATION LEVEL statement or system parameters provided by CUBRID. For more information, see Concurrency/Lock Parameters.

The read operations that allow interference between transactions with isolation levels are as follows:

- **Dirty read** : A transaction T2 can read D' before a transaction T1 updates data D to D' and commits it.
- **Non-repeatable read** : A transaction T1 can read other value, if a transaction T2 updates data while data is retrieved in the transaction T2 multiple times.
- **Phantom read** : A transaction T1 can read E, if a transaction T2 inserts new record E while data is retrieved in the transaction T1 multiple times.

CUBRID ensures lock/unlock in the unit of row, index, table, or database, and it provides six levels of transaction isolation. Therefore, you can adjust concurrency levels more specifically than other DBMSs.

Isolation Levels Provided by CUBRID

CUBRID Isolation Level(isolation_level)	Other DBMS Isolation Level (isolation_level)	DIRTY READ	UNREPEATABLE READ	PHANTOM READ	Note
SERIALIZABLE (6)	SERIALIZABLE (4)	N	N	N	Guarantees data consistency. Increases overhead due to lock.
REPEATABLE READ CLASS with REPEATABLE READ INSTANCES (5)	REPEATABLE READ (3)	N	N	Y	Not allow to update table schema while the table is selected.
REPEATABLE READ CLASS with READ COMMITTED INSTANCES (4)	READ COMMITTED (2)	N	Y	Y	Not allow to update table schema while the table is selected.
REPEATABLE READ CLASS with READ UNCOMMITTED INSTANCES (3)	READ UNCOMMITTED (1)	Y	Y	Y	Default configuration. Not allow to update table schema while the table is selected.
READ COMMITTED CLASS with READ COMMITTED INSTANCES (2)		N	Y	Y	Allows to update table schema while the table is selected.
READ COMMITTED CLASS with READ		Y	Y	Y	Allows to update table schema

UNCOMMITTED INSTANCES (1)	while the table is selected.

Lock Protocol

Overview

In the two-phase locking protocol used by CUBRID, a transaction obtains a shared lock before it reads an object, and an exclusive lock before it updates the object so that conflicting operations are not executed simultaneously.

If transaction T1 requires a lock, CUBRID checks if the requested lock conflicts with the existing one. If it does, transaction T1 enters a standby state and delays the lock. If another transaction T2 releases the lock, transaction T1 resumes and obtains it. Once the lock is released, the transaction do not require any more new locks.

Granularity Locking

CUBRID uses a granularity locking protocol to decrease the number of locks. In the granularity locking protocol, a database can be modeled as a hierarchy of lockable units: bigger locks have more granular locks.

For example, suppose that a database consists of multiple tables and each table consists of multiple instances. If the database is locked, all tables and instances are implicitly considered to be locked. A lock on a big unit results in less overhead, because only one lock needs to be managed. However, it leads to decreased concurrency because almost all concurrent transactions conflict with each other. The finer the granularity, the better the concurrency; it causes more overhead because more locks need to be managed. CUBRID selects a locking granularity level based on the operation being executed. For example, if a transaction retrieves all instances of a table, the entire tables will be locked, rather than each instance. If the transaction accesses a few instances of the table, the instances are locked individually.

If the locking granularities overlap, effects of a finer granularity are propagated in order to prevent conflicts. That is, if a shared lock is required on an instance of a table, an intention shared lock will be set on the table. If an exclusive lock is required on an instance of a table, an intention exclusive lock will be set on the table. An intention shared lock on a table means that a shared lock can be set on an instance of the table. An intention exclusive lock on a table means that a shared/exclusive lock can be set on an instance of the table. That is, if an intention shared lock on a table is allowed in one transaction, another transaction cannot obtain an exclusive lock on the table (for example, to add a new column). However, the second transaction may obtain a shared lock on the table. If an intention exclusive lock on the table is allowed in one transaction, another transaction cannot obtain a shared lock on the table (for example, a query on an instance of the tables cannot be executed because it is being changed).

A mechanism called lock escalation is used to limit the number of locks being managed. If a transaction has more than a certain number of locks (a number which can be changed by the **lock_escalation** system parameter), the system begins to require locks at the next higher level of granularity. This escalates the locks to a coarser level of granularity. CUBRID performs lock escalation when no transactions have a higher level of granularity in order to avoid a deadlock caused by lock conversion.

Lock Mode Types And Compatibility

CUBRID determines the lock mode depending on the type of operation to be performed by the transaction, and determines whether or not to share the lock depending on the mode of the lock preoccupied by another transaction. Such decisions concerning the lock are made by the system automatically. Manual assignment by the user is not allowed. To check the lock information of CUBRID, use the **cubrid lockdb** *db_name* command. For details, see Checking Lock Status.

- **Shared lock (shared lock, S_LOCK)** : This lock is obtained before the read operation is executed on the object. It can be obtained by multiple transactions for the same object.
  Transaction T1 obtains the shared lock first before it performs the read operation on a certain object X, and releases it immediately after it completes the operation even before transaction T1 is committed. Here, transaction T2 and T3 can perform the read operation on X concurrently, but not the update operation.

- **Exclusive lock (exclusive lock, X_LOCK)** : This lock is obtained before the update operation is executed on the object. It can only be obtained by one transaction.
  Transaction T1 obtains the exclusive lock first before it performs the update operation on a certain object X, and does not release it until transaction T1 is committed even after the update operation is completed. Therefore, transaction T2 and T3 cannot perform the read operation as well on X before transaction T1 releases the exclusive lock.
- **Update lock (update lock, U_LOCK)** : This lock is obtained when the read operation is executed in the expression before the update operation is performed.
  For example, when an UPDATE statement combined with a **WHERE** clause is executed, execute the operation by obtaining the update lock for each tuple and the exclusive lock only for the result tuples that satisfy the condition when performing index search or full scan search in the **WHERE** clause. The update lock is converted to an exclusive lock when the actual update operation is performed. It can be called a quasi-exclusive lock because it does not allow the read lock on the same object for another transaction.
- **Intention lock (intention lock)** : A lock that is set inherently in a higher-level object than X to protect the lock on the object X of a certain level.
  For example, when a shared lock is requested for a certain tuple, prevent a situation from occurring in which the table is locked by another transaction by setting the intention shared lock as well on the table at the higher level in hierarchy. Therefore, the intention lock is not set on tuples at the lowest level, but is set on higher-level objects. The types of intention locks are as follows:
- **Intention shared lock (intention shared lock, IS_LOCK)** : If the intention shared lock is set on the table, which is the higher-level object, as the result of the shared lock set on a certain tuple, another transaction cannot perform operations such as changing the schema of the table (e.g. adding a column or changing the table name) or updating all tuples. However updating some tuples or viewing all tuples is allowed.
- **Intention exclusive lock (intention exclusive lock, IX_LOCK)** : If the intention exclusive lock is set on the table, which is the higher-level object, as the result of the exclusive lock set on a certain tuple, another transaction cannot perform operations such as changing the schema of the table, updating or viewing all tuples. However updating some tuples is allowed.
- **Shared with intent exclusive (shared with intent exclusive, SIX_LOCK)** : This lock is set on the higher-level object inherently to protect the shared lock set on all objects at the lower hierarchical level and the intention exclusive lock on some object at the lower hierarchical level.
  Once the shared intention exclusive lock is set on a table, another transaction cannot change the schema of the table, update all/some tuples or view all tuples. However, viewing some tuples is allowed.

The following table briefly shows the lock compatibility between the locks described below. Compatibility means that the lock requester can obtain a lock while the lock holder is keeping the lock obtained for the object X. N/A means 'not applicable'.

Lock Compatibility

		Lock Holder(lock holder)						
		NULL_LOCK	**IS_LOCK**	**S_LOCK**	**IX_LOCK**	**SIX_LOCK**	**U_LOCK**	**X_LOCK**
Lock Requester (lock requester)	**NULL_LOCK**	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
	IS_LOCK	TRUE	TRUE	TRUE	TRUE	TRUE	N/A	FALSE
	S_LOCK	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE
	IX_LOCK	TRUE	TRUE	FALSE	TRUE	FALSE	N/A	FALSE
	SIX_LOCK	TRUE	TRUE	FALSE	FALSE	FALSE	N/A	FALSE
	U_LOCK	TRUE	N/A	TRUE	N/A	N/A	FALSE	FALSE
	X_LOCK	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

- **NULL_LOCK** : No lock

Example

session 1	**session 2**

`;autocommit off`
`AUTOCOMMIT IS OFF`
	`;autocommit off`
	`AUTOCOMMIT IS OFF`
	`set transaction`

```
set transaction isolation level 4
;xr
Isolation level set to:
REPEATABLE READ SCHEMA, READ COMMITTED
INSTANCES.

1 command(s) successfully processed.
```

```
isolation level 4
;xr
Isolation level
set to:
REPEATABLE READ
SCHEMA, READ
COMMITTED
INSTANCES.

1 command(s)
successfully
processed.

/*
C:\CUBRID>cubrid
lockdb demodb

*** Lock Table
Dump ***
…

Object Lock Table:
        Current
number of objects
which are
locked    = 0
        Maximum
number of objects
which can be
locked = 10000
…
*/
```

```
select nation_code, gold from participant
where nation_code='USA';
;xr

=== <Result of SELECT Command> ===
 nation_code                  gold
======================================
'USA'                          36
'USA'                          37
'USA'                          44
'USA'                          37
'USA'                          36

5 rows selected.

1 command(s) successfully processed.

/*
C:\CUBRID>cubrid lockdb demodb
*** Lock Table Dump ***
…

Object type: Root class.
LOCK HOLDERS:
    Tran_index =   2, Granted_mode
=  IS_LOCK, Count =   1, Nsubgranules =  1

Object type: Class = participant.
LOCK HOLDERS:
    Tran_index =   2, Granted_mode
=  IS_LOCK, Count =   2, Nsubgranules =  0
*/
```

```
update participant
set gold = 11
where nation_code
= 'USA' ;
;xr
```

```
5 rows affected.
```

```
select nation_code, gold from participant
where nation_code='USA';

;xr

/* no results until transaction 2 releases a
lock

C:\CUBRID>cubrid lockdb demodb
*** Lock Table Dump ***
…

Object type: Instance of class
( 0|   551|   7) = participant.
LOCK HOLDERS:
    Tran_index =   3, Granted_mode
=   X_LOCK, Count =   2

…

Object type: Root class.
LOCK HOLDERS:
    Tran_index =   3, Granted_mode
=  IX_LOCK, Count =   1, Nsubgranules =  3
NON_2PL_RELEASED:
    Tran_index =   2, Non_2_phase_lock
=  IS_LOCK

…

Object type: Class = participant.
LOCK HOLDERS:
    Tran_index =   3, Granted_mode
=  IX_LOCK, Count =   3, Nsubgranules =  5
    Tran_index =   2, Granted_mode
=  IS_LOCK, Count =   2, Nsubgranules =  0
*/
```

```
;commit work

Current
transaction has
been committed.
```

```
=== <Result of SELECT Command> ===
nation_code                  gold
=================================
'USA'                          11
'USA'                          11
'USA'                          11
'USA'                          11
'USA'                          11

5 rows selected.

1 command(s) successfully processed.

/*
C:\CUBRID>cubrid lockdb demodb
…

Object type: Root class.
LOCK HOLDERS:
    Tran_index =   2, Granted_mode
=  IS_LOCK, Count =   1, Nsubgranules =  1

Object type: Class = participant.
LOCK HOLDERS:
    Tran_index =   2, Granted_mode
=  IS_LOCK, Count =   3, Nsubgranules =  0
```

```
…
*/
```

```
;commit work

Current transaction has been committed.

/*
C:\CUBRID>cubrid lockdb demodb
…

Object Lock Table:
        Current number of objects which are
locked    = 0
        Maximum number of objects which can
be locked = 10000
*/
```

Transaction Deadlock

A deadlock is a state in which two or more transactions wait at the same time for another transaction's lock to be released. CUBRID resolves the problem by rolling back one of the transactions, because transactions in a deadlock state will hinder the work of another transaction. The transaction to be rolled back is usually the transaction which has made the least updates, i.e. the one that started more recently. As soon as a transaction is rolled back, the lock held by the transaction is released and other transactions in a deadlock are permitted to proceed.

It is impossible to predict such deadlocks, but it is recommended that you reduce the range to which lock is applied by setting the index, shortening the transaction, or setting the transaction isolation level as low in order to decrease such occurrences.

Example

session 1

```
;autocommit off
AUTOCOMMIT IS OFF
set transaction isolation level 6
;xr
Isolation level set to:
SERIALIZABLE

1 command(s) successfully processed.
```

session 2

```
;autocommit off
AUTOCOMMIT IS OFF
set transaction isolation level 6
;xr
Isolation level set to:
SERIALIZABLE

1 command(s) successfully
processed.
```

session 1

```
CREATE TABLE lock_tbl(host_year integer,
nation_code char(3));
INSERT INTO lock_tbl VALUES (2004,
'KOR');
INSERT INTO lock_tbl VALUES (2004,
'USA');
INSERT INTO lock_tbl VALUES (2004,
'GER');
INSERT INTO lock_tbl VALUES (2008,
'GER');
COMMIT;
;xr

6 command(s) successfully processed.

SELECT * FROM lock_tbl;
;xr

=== <Result of SELECT Command> ===

    host_year  nation_code
===================================
         2004  'KOR'
         2004  'USA'
         2004  'GER'
         2008  'GER'

4 rows selected.
```

```
SELECT * FROM lock_tbl;
;xr

=== <Result of SELECT Command> ===

    host_year  nation_code
==================================
=
         2004  'KOR'
         2004  'USA'
         2004  'GER'
         2008  'GER'

4 rows selected.
```

```
DELETE FROM lock_tbl WHERE
host_year=2008;
;xr

/* no result until transaction 2
releases a lock

C:\CUBRID>cubrid lockdb demodb
*** Lock Table Dump ***
…

Object type: Class = lock_tbl.
LOCK HOLDERS:
    Tran_index =   2, Granted_mode
=   S_LOCK, Count =   2, Nsubgranules
=  0

BLOCKED LOCK HOLDERS:
    Tran_index =   1, Granted_mode
=   S_LOCK, Count =   3, Nsubgranules
=  0
    Blocked_mode = SIX_LOCK
    Start_waiting_at = Fri Feb 12
14:22:58 2010
    Wait_for_nsecs = -1

*/
```

```
INSERT INTO lock_tbl VALUES (2004,
'AUS');
;xr

1 rows affected.
```

```
In the command from line 1,

ERROR: Your transaction (index 1, dba@
090205|4760) has been unilaterally
aborted by the system.


0 command(s) successfully processed.


/*
System rolled back the transaction 1 to
resolve a deadlock

C:\CUBRID>cubrid lockdb demodb
*** Lock Table Dump ***

Object type: Class = lock_tbl.
```

```
LOCK HOLDERS:
    Tran_index =   2, Granted_mode =
SIX_LOCK, Count =   3, Nsubgranules =  0
*/
```

Transaction Timeout

CUBRID provides the lock timeout feature, which sets the waiting time for the lock until the transaction lock setting is allowed.

If the lock is allowed within the lock timeout, CUBRID rolls back the transaction and outputs an error message when the timeout has passed. If a transaction deadlock occurs within the lock timeout, CUBRID rolls back the transaction whose waiting time is closest to the timeout.

Setting the Lock Timeout

Description

The system parameter **lock_timeout_in_secs** in the **$CUBRID/conf/cubrid.conf** file or the **SET TRANSACTION** statement sets the timeout (in seconds) during which the application client will wait for the lock and rolls back the transaction and outputs an error message when the specified time has passed. The default value of the **lock_timeout_in_secs** parameter is **-1**, which means the application client will wait indefinitely until the transaction lock is allowed. Therefore, the user can change this value depending on the transaction pattern of the application client. If the lock timeout value has been set to 0, an error message will be displayed as soon as a lock occurs.

Syntax

```
SET TRANSACTION LOCK TIMEOUT timeout_spec [ ; ]
timeout_spec:
- INFINITE
- OFF
- unsigned_integer
- variable
```

- **INFINITE** : Wait indefinitely until the transaction lock is allowed. Has the same effect as setting the system parameter **lock_timeout_in_secs**to -1.
- **OFF** : Do not wait for the lock, but roll back the transaction and display an error message. Has the same effect as setting the system parameter **lock_timeout_in_secs**to 0.
- *unsigned_integer* : Set in seconds. Wait for the transaction lock for the specified time period.
- *variable* : A variable can be specified. Wait for the transaction lock for the value saved by the variable.

Example 1

```
vi $CUBRID/conf/cubrid.conf
…
lock_timeout_in_secs = 10
…
```

Example 2

```
csql> SET TRANSACTION LOCK TIMEOUT 10;
csql> ;xr
```

Checking the Lock Timeout

Description

You can check the lock timeout set for the current application client by using the **GET TRANSACTION** statement, or save this value in a variable.

Syntax

```
GET TRANSACTION LOCK TIMEOUT [ { INTO | TO } variable ] [ ; ]
```

Example

```
csql> GET TRANSACTION LOCK TIMEOUT;
csql> ;xr

=== <Result of GET LOCK TIMEOUT Command in Line 1> ===

         Result
===============

  1.000000e+001
```

Lock Timeout Error Message

Displays the following message if lock timeout occurs in a transaction that was waiting for another transaction's lock to be released. To increase the level of detail of the information displayed in the lock timeout error message, see the description of lock_timeout_message_type in Concurrency/Lock Parameters .

```
ERROR: Your transaction (index 3, cub_user@cdbs006.cub|15668) timed out waiting
on    X_LOCK lock on instance 0|636|34 of class participant. You are waiting for
user(s)  to finish.
```

- Your transaction(index 3 …) : This means that the index of the transaction that was rolled back due to timeout while waiting for the lock is 3. The transaction index is a number that is sequentially assigned when the client connects to the database server. You can also check this number by executing the **cubrid lockdb** utility.
- (…cub_user@cdbs006.cub|15668) : cub_user is the login ID of the client and the part after @ is the name of the host where the client was running. The part after| is the process ID (PID) of the client.
- X_LOCK : This means the exclusive lock set on the object to perform data update. For details, see Lock Mode Types And Compatibility.
- Instance 0|636|34 of class participant : This means that **X_LOCK** has been set on a certain tuple in the table named participant and the OID (unique ID assigned to the given object) of the tuple is 0|636|34.

That is, the above lock error message can be interpreted as meaning that "Because another client is holding **X_LOCK** on a certain tuple in the participant table, transaction 3 which running on the host cdbs006.cub waited for the lock and was rolled back as the timeout has passed."

If you want to check the lock information of the transaction specified in the error message, you can do so by using the **cubrid lockdb** utility to search for the OID value (ex: 0|636|34) of a specific tuple where the **X_LOCK** is set currently to find the transaction ID currently holding the lock, the client program name and the process ID (PID). For details, see Checking Lock Status. You can also check the transaction lock information in the CUBRID Manager. For details, see Database Lock Information.

You can organize the transactions by checking uncommitted queries through the SQL log after checking the transaction lock information in the manner described above. For information on checking the SQL log, see Broker Log or SQL Log.

Also, you can force problematic transactions to quit by using the **cubrid killtran** utility. For details, see Killing Transactions.

Transaction Isolation Level

Overview

The transaction isolation level is determined based on how much interference occurs. The more isolation means the less interference from other transactions and more serializable. The less isolation means the more interference from other transactions and higher level of concurrency. You can control the level of consistency and concurrency specific to a service by setting appropriate isolation level.

Note A transaction can be restored in all supported isolation levels because updates are not committed before the end of the transaction.

SET TRANSACTION ISOLATION LEVEL

Description

You can set the level of transaction isolation by using **isolation_level** and the **SET TRANSACTION** statement in the **$CUBRID/conf/cubrid.conf**. The level of **REPEATABLE READ CLASS** and **READ UNCOMMITTED INSTANCES** are set by default, which indicates the level 3 through level 1 to 6. For more information, see Database Concurrency.

Syntax

```
SET TRANSACTION ISOLATION LEVEL isolation_level_spec [ ; ]
isolation_level_spec:
_ SERIALIZABLE
_ CURSOR STABILITY
_ isolation_level [ { CLASS | SCHEMA } [ , isolation_level INSTANCES ] ]
_ isolation_level [ INSTANCES [ , isolation_level { CLASS | SCHEMA } ] ]
_ variable
isolation_level:
_ REPEATABLE READ
_ READ COMMITTED
_ READ UNCOMMITTED
```

Example 1

```
vi $CUBRID/conf/cubrid.conf
…
isolation_level = 1
…

or

isolation_level = "TRAN_COMMIT_CLASS_UNCOMMIT_INSTANCE"
```

Example 2

```
csql> SET TRANSACTION ISOLATION LEVEL 1;
csql> ;xr

or

csql> SET TRANSACTION ISOLATION LEVEL READ COMMITTED CLASS,READ UNCOMMITTED INSTANCES;
csql> ;xr
```

The following table shows the isolation levels from 1 to 6. It consists of table schema (row) and isolation level. For the unsupported isolation level, see Unsupported Combination of Isolation Level.

Levels of Isolation Supported by CUBRID

Name	Description
SERIALIZABLE (6)	In this isolation level, problems concerning concurrency (e.g. dirty read, non-repeatable read, phantom read, etc.) do not occur.
REPEATABLE READ CLASS with REPEATABLE READ INSTANCES (5)	Another transaction T2 cannot update the schema of table A while transaction T1 is viewing table A. Transaction T1 may experience phantom read for the record R that was inserted by another transaction T2 when it is repeatedly retrieving a specific record.
REPEATABLE READ CLASS with READ COMMITTED INSTANCES (or CURSOR STABILITY) (4)	Another transaction T2 cannot update the schema of table A while transaction T1 is viewing table A. Transaction T1 may experience R read (non-repeatable read) that was updated and committed by another transaction T2 when it is repeatedly retrieving the record R.
REPEATABLE READ CLASS with	Default isolation level. Another transaction T2 cannot update the schema of table A while

READ UNCOMMITTED INSTANCES (3)	transaction T1 is viewing table A. Transaction T1 may experience R' read (dirty read) for the record that was updated but not committed by another transaction T2.
READ COMMITTED CLASS with READ COMMITTED INSTANCES (2)	Transaction T1 may experience A' read (non-repeatable read) for the table that was updated and committed by another transaction T2 while it is viewing table A repeatedly. Transaction T1 may experience R' read (non-repeatable read) for the record that was updated and committed by another transaction T2 while it is retrieving the record R repeatedly.
READ COMMITTED CLASS with READ UNCOMMITTED INSTANCES (1)	Transaction T1 may experience A' read (non-repeatable read) for the table that was updated and committed by another transaction T2 while it is repeatedly viewing table A. Transaction T1 may experience R' read (dirty read) for the record that was updated but not committed by another transaction T2.

If the transaction level is changed in an application client while a transaction is executed, the new level is applied to the rest of the transaction being executed. Therefore, some object locks that have already been obtained may be released during the transaction while the new isolation level is applied. For this reason, it is recommended that the transaction isolation level be modified when the transaction starts (after commit, rollback or system restart) because an isolation level which has already been set does not apply to the entire transaction, but can be changed during the transaction.

GET TRANSACTION ISOLATION LEVEL

Description

You can assign the current isolation level to *variable* by using the **GET TRANSACTION** statement. The following is a statement that verifies the isolation level. *variable*.

Syntax

```
GET TRANSACTION ISOLATION LEVEL [ { INTO | TO } variable ] [ ; ]
```

Example

```
csql> GET TRANSACTION ISOLATION LEVEL;
csql> ;xr

       Result
=============
  READ COMMITTED SCHEMA, READ UNCOMMITTED INSTANCES
```

SERIALIZABLE

The highest isolation level (6). Problems concerning concurrency (e.g. dirty read, non-repeatable read, phantom read, etc.) do not occur.

The following are the rules of this isolation level:

- Transaction T1 cannot read or modify the record being updated by another transaction T2.
- Transaction T1 cannot read or modify the record being viewed by another transaction T2.
- Another transaction T2 cannot insert a new record into table A while transaction T1 is retrieving the records of table A.

This isolation level uses a two-phase locking protocol for shared and exclusive lock: the lock is maintained until the transaction ends even after the operation has been executed.

Example

The following is an example that shows that another transaction cannot access the table or record while one transaction is reading or updating the object when the transaction level of the concurrent transactions is **SERIALIZABLE**.

session 1	session 2

	;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 6 ;xr Isolation level set to: SERIALIZABLE	;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 6 ;xr Isolation level set to: SERIALIZABLE

	--creating a table
	
	CREATE TABLE isol6_tbl(host_year integer, nation_code char(3));
	
	INSERT INTO isol6_tbl VALUES (2008, 'AUS');
	
	COMMIT;
	;xr	

		--selecting records from the table SELECT * FROM isol6_tbl WHERE nation_code = 'AUS'; ;xr === <Result of SELECT Command> === host_year nation_code =================================== 2008 'AUS' 1 rows selected.
	INSERT INTO isol6_tbl VALUES (2004, 'AUS'); ;xr /* unable to insert a row until the tran 2 committed */	
	1 rows affected.	COMMIT; ;xr
		SELECT * FROM isol6_tbl WHERE nation_code = 'AUS'; ;xr /* unable to select rows until tran 1 committed */

	COMMIT;
	;xr
		=== <Result of SELECT Command> ===
		    host_year nation_code
		===================================
		         2008 'AUS'
		         2004 'AUS'
		
		2 rows selected.

	DELETE FROM isol6_tbl
	WHERE nation_code = 'AUS'
	and
	host_year=2008;
	;xr
	
	/* unable to delete rows until tran 2 committed */	

	1 rows deleted. 1 command(s) successfully	COMMIT; ;xr

processed.	
	SELECT * FROM isol6_tbl WHERE nation_code = 'AUS'; ;xr /* unable to select rows until tran 1 committed */
COMMIT; ;xr	=== <Result of SELECT Command> === host_year nation_code =================================== 2004 'AUS' 1 rows selected.
ALTER TABLE isol6_tbl ADD COLUMN gold INT; ;xr /* unable to alter the table schema until tran 2 committed */	/* repeatable read is ensured while tran_1 is altering table schema */ SELECT * FROM isol6_tbl WHERE nation_code = 'AUS'; ;xr === <Result of SELECT Command> === host_year nation_code =================================== 2004 'AUS' 1 rows selected.
1 command(s) successfully processed.	COMMIT; ;xr
	SELECT * FROM isol6_tbl WHERE nation_code = 'AUS'; ;xr /* unable to access the table until tran_1 committed */
COMMIT; ;xr	=== <Result of SELECT Command > === host_year nation_code gold =================================== 2004 'AUS' NULL 1 rows selected.

REPEATABLE READ CLASS with REPEATABLE READ INSTANCES

A relatively high isolation level (5). A dirty or non-repeatable read does not occur, but a phantom read may.

The following are the rules of this isolation level:

- Transaction T1 cannot read or modify the record being updated by another transaction T2.
- Transaction T1 cannot read or modify the record being viewed by another transaction T2.
- Another transaction T2 can insert a new record into table A while transaction T1 is retrieving records of table A. However, transaction T1 and T2 cannot set the lock on the same record.

This isolation level uses a two-phase locking protocol.

Example

The following is an example that shows that phantom read may occur because another transaction can add a new record while one transaction is performing the object read when the transaction level of the concurrent transactions is **REPEATABLE READ CLASS** with **REPEATABLE READ INSTANCES**.

session 1	session 2
;autocommit off AUTOCOMMIT IS OFF	;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 5

SET TRANSACTION ISOLATION LEVEL 5 ;xr Isolation level set to: REPEATABLE READ SCHEMA, REPEATABLE READ INSTANCES.	;xr Isolation level set to: REPEATABLE READ SCHEMA, REPEATABLE READ INSTANCES.

--creating a table
 	
CREATE TABLE isol5_tbl(host_year integer, nation_code char(3));
CREATE UNIQUE INDEX on isol5_tbl(nation_code, host_year);
INSERT INTO isol5_tbl VALUES (2008, 'AUS');
INSERT INTO isol5_tbl VALUES (2004, 'AUS');
 	
COMMIT;
;xr	

	--selecting records from the table SELECT * FROM isol5_tbl WHERE nation_code='AUS'; ;xr === <Result of SELECT Command> === host_year nation_code =================================== 2004 'AUS' 2008 'AUS' 2 rows selected.
INSERT INTO isol5_tbl VALUES (2004, 'KOR'); INSERT INTO isol5_tbl VALUES (2000, 'AUS'); ;xr 2 rows affected. /* able to insert new rows only when locks are not conflicted */	
	SELECT * FROM isol5_tbl WHERE nation_code='AUS'; ;xr /* phantom read may occur when tran 1 committed */

COMMIT;
;xr
	=== <Result of SELECT Command> ===
	    host_year nation_code
	===================================
	         2000 'AUS'
	         2004 'AUS'
	         2008 'AUS'
	
	3 rows selected.

DELETE FROM isol5_tbl
WHERE nation_code = 'AUS' and
host_year=2008;
;xrun
 	
/* unable to delete rows	

until tran 2 committed */	
1 rows affected. 1 command(s) successfully processed.	COMMIT; ;xr
	SELECT * FROM isol5_tbl WHERE nation_code = 'AUS'; ;xr /* unable to select rows until tran 1 committed */

COMMIT;
;xr
	=== <Result of SELECT Command> ===
	    host_year nation_code
	===================================
	         2000 'AUS'
	         2004 'AUS'
	
	2 rows selected.

ALTER TABLE isol5_tbl ADD COLUMN gold INT; ;xr /* unable to alter the table schema until tran 2 committed */	
	/* repeatable read is ensured while tran_1 is altering table schema */ SELECT * FROM isol5_tbl WHERE nation_code = 'AUS'; ;xr === <Result of SELECT Command> === host_year nation_code =================================== 2000 'AUS' 2004 'AUS' 2 rows selected.
1 command(s) successfully processed.	COMMIT; ;xr
	SELECT * FROM isol5_tbl WHERE nation_code = 'AUS'; ;xr /* unable to access the table until tran_1 committed */
COMMIT; ;xr	=== <Result of SELECT Command > === host_year nation_code gold =================================== 2000 'AUS' NULL 2004 'AUS' NULL 2 rows selected.

REPEATABLE READ CLASS with READ COMMITTED INSTANCES

A relatively low isolation level (4). A dirty read does not occur, but non-repeatable or phantom read may. That is, transaction T1 can read another value because insert or update by transaction T2 is allowed while transaction T1 is repeatedly retrieving one object.

The following are the rules of this isolation level:

- Transaction T1 cannot read the record being updated by another transaction T2.
- Transaction T1 can update/insert record to the table being viewed by another transaction T2.

- Transaction T1 cannot change the schema of the table being viewed by another transaction T2.

This isolation level uses a two-phase locking protocol for an exclusive lock. A shared lock on a row is released immediately after it is read; however, an intention lock on a table is released when a transaction terminates to ensure repeatable read on the schema.

Example

The following is an example that shows that a phantom or non-repeatable read may occur because another transaction can add or update a record while one transaction is performing the object read but repeatable read for the table schema update is ensured when the transaction level of the concurrent transactions is **REPEATABLE READ CLASS** with **READ COMMITTED INSTANCES**.

session 1	session 2
;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 4 ;xr Isolation level set to: REPEATABLE READ SCHEMA, READ COMMITTED INSTANCES.	;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 4 ;xr Isolation level set to: REPEATABLE READ SCHEMA, READ COMMITTED INSTANCES.

--creating a table
 	
CREATE TABLE isol4_tbl(host_year integer, nation_code char(3));
INSERT INTO isol4_tbl VALUES (2008, 'AUS');
 	
COMMIT;
;xr	

	--selecting records from the table SELECT * FROM isol4_tbl; ;xr === <Result of SELECT Command> === host_year nation_code =================================== 2008 'AUS' 1 rows selected.
INSERT INTO isol4_tbl VALUES (2004, 'AUS'); INSERT INTO isol4_tbl VALUES (2000, 'NED'); ;xr 2 rows affected. /* able to insert new rows even if tran 2 uncommitted */	
	SELECT * FROM isol4_tbl; ;xr /* phantom read may occur when tran 1 committed */
COMMIT; ;xr	=== <Result of SELECT Command> === host_year nation_code ===================================

	2008 'AUS' 2004 'AUS' 2000 'NED' 3 rows selected.

INSERT INTO isol4_tbl
VALUES (1994, 'FRA');
;xr
 	
1 rows affected.	

	SELECT * FROM isol4_tbl; ;xr /* unrepeatable read may occur when tran 1 committed */

DELETE FROM isol4_tbl
WHERE nation_code = 'AUS' and
host_year=2008;
;xr
 	
1 rows affected.
 	
/* able to delete rows while tran 2 is selecting rows*/	

COMMIT; ;xr	=== <Result of SELECT Command> === host_year nation_code =================================== 2004 'AUS' 2000 'NED' 1994 'FRA' 3 rows selected.
ALTER TABLE isol4_tbl ADD COLUMN gold INT; ;xr /* unable to alter the table schema until tran 2 committed */	
	/* repeatable read is ensured while tran_1 is altering table schema */ SELECT * FROM isol4_tbl; ;xr === <Result of SELECT Command> === host_year nation_code =================================== 2004 'AUS' 2000 'NED' 1994 'FRA' 3 rows selected.
1 command(s) successfully processed.	COMMIT; ;xr
	SELECT * FROM isol4_tbl; ;xr /* unable to access the table until tran_1 committed */
COMMIT; ;xr	=== <Result of SELECT Command > === host_year nation_code gold

	===================================
	  2004 'AUS' NULL
	  2000 'NED' NULL
	  1994 'FRA' NULL
	
	3 rows selected.

REPEATABLE READ CLASS with READ UNCOMMITTED INSTANCES

The default isolation of CUBRID (3). The concurrency level is high. A dirty, non-repeatable or phantom read may occur for the tuple, but repeatable read is ensured for the table. That is, transaction T2 can read an object while transaction T1 is updating one.

The following are the rules of this isolation level:

- Transaction T1 can read the record being updated by another transaction T2.
- Transaction T1 can update/insert record to the table being viewed by another transaction T2.
- Transaction T1 cannot change the schema of the table being viewed by another transaction T2.

This isolation level uses a two-phase locking protocol for an exclusive lock. However, the shared lock on the tuple is released immediately after it is retrieved, and the update lock on the tuple is released immediately after the update is executed. The intention lock on the table is released when the transaction ends to ensure repeatable reads.

Example

The following is an example that shows that another transaction can read dirty data uncommitted by one transaction but repeatable reads are ensured for table schema update when the transaction level of the concurrent transactions is **REPEATABLE READ CLASS** with **READ UNCOMMITTED INSTANCES**.

session 1	**session 2**
;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 3 ;xr Isolation level set to: REPEATABLE READ SCHEMA, READ UNCOMMITTED INSTANCES.	;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 3 ;xr Isolation level set to: REPEATABLE READ SCHEMA, READ UNCOMMITTED INSTANCES.

--creating a table
 	
CREATE TABLE
isol3_tbl(host_year
integer, nation_code
char(3));
CREATE UNIQUE INDEX on
isol3_tbl(nation_code,
host_year);
INSERT INTO isol3_tbl
VALUES (2008, 'AUS');
 	
COMMIT;
;xr	

	--selecting records from the table SELECT * FROM isol3_tbl; ;xr === <Result of SELECT Command> === host_year nation_code =================================== 2008 'AUS' 1 rows selected.

```
INSERT INTO isol3_tbl
VALUES (2004, 'AUS');

INSERT INTO isol3_tbl
VALUES (2000, 'NED');
;xr

2 rows affected.


/* able to insert new rows
even if tran 2 uncommitted
*/
```

```
SELECT * FROM isol3_tbl;
;xr

=== <Result of SELECT Command> ===
    host_year  nation_code
===================================
         2008  'AUS'
         2004  'AUS'
         2000  'NED'

3 rows selected.

/* dirty read may occur so that
tran_2 can select new rows
uncommitted by tran_1 */
```

```
ROLLBACK;
;xr
```

```
SELECT * FROM isol3_tbl;
;xr

=== <Result of SELECT Command> ===
    host_year  nation_code
===================================
         2008  'AUS'

1 rows selected.

/* unrepeatable read may occur so
that selected results are different
*/
```

```
INSERT INTO isol3_tbl
VALUES (1994, 'FRA');
;xr

DELETE FROM isol3_tbl
WHERE nation_code = 'AUS'
and
host_year=2008;
;xr

1 rows affected.
1 rows affected.

/* able to delete rows even
if tran 2 uncommitted */
```

```
SELECT * FROM isol3_tbl;
;xr

=== <Result of SELECT Command> ===
    host_year  nation_code
===================================
         1994  'FRA'
```

	`1 rows selected.`

`ALTER TABLE isol3_tbl`
`ADD COLUMN gold INT;`
`;xr`
 	
`/* unable to alter the`
`table schema until tran 2`
`committed */`	

	`/* repeatable read is ensured while`
	`tran_1 is altering table schema */`
	
	`SELECT * FROM isol3_tbl;`
	`;xr`
	
	`=== <Result of SELECT Command> ===`
	`    host_year  nation_code`
	`===================================`
	`         1994  'FRA'`
	
	`1 rows selected.`

`1 command(s) successfully`
`processed.`
	`COMMIT;`
	`;xr`

	`SELECT * FROM isol3_tbl;`
	`;xr`

`COMMIT;`
`;xr`
	`=== <Result of SELECT Command > ===`
	`host_year  nation_code  gold`
	`===================================`
	`  1994  'FRA'           NULL`
	
	`1 rows selected.`

Note CUBRID flushes dirty data (or dirty instances) in the client buffers to the database (server) such as the following situations. For more information, see How to Handle Dirty Instances.

READ COMMITTED CLASS with READ COMMITTED INSTANCES

A relatively low isolation level (2). A dirty read does not occur, but non-repeatable or phantom read may occur. That is, this level is similar to **REPEATABLE READ CLASS** with **READ COMMITTED INSTANCES**(level 4) described above, but works differently for table schema. Non-repeatable read due to a table schema update may occur because another transaction T2 can change the schema of the table being viewed by the transaction T1.

The following are the rules of this isolation level:

- Transaction T1 cannot read the record being updated by another transaction T2.
- Transaction T1 can update/insert a record to the table being viewed by another transaction T2.
- Transaction T1 can change the schema of the table being viewed by another transaction T2.

This isolation level uses a two-phase locking protocol for an exclusive lock. However, non-repeatable read may occur because the shared lock on the tuple is released immediately after it is retrieved and the intention lock on the table is released immediately as well.

Example

The following is an example that shows that phantom or non-repeatable read for the record as well as for the table schema may occur because another transaction can add or update a new record while one transaction is performing the object read when the transaction level of the concurrent transactions is **READ COMMITTED CLASS** with **READ COMMITTED INSTANCES**.

session 1	**session 2**

`;autocommit off`
`AUTOCOMMIT IS OFF`
	`;autocommit off`
	`AUTOCOMMIT IS OFF`

```
SET TRANSACTION ISOLATION
LEVEL 2
;xr

Isolation level set to:
READ COMMITTED SCHEMA, READ
COMMITTED INSTANCES.
```

```
SET TRANSACTION ISOLATION LEVEL 2
;xr

Isolation level set to:
READ COMMITTED SCHEMA, READ
COMMITTED INSTANCES.
```

```
--creating a table

CREATE TABLE
isol2_tbl(host_year
integer, nation_code
char(3));
CREATE UNIQUE INDEX on
isol2_tbl(nation_code,
host_year);
INSERT INTO isol2_tbl
VALUES (2008, 'AUS');

COMMIT;
;xr
```

```
--selecting records from the table
SELECT * FROM isol2_tbl;
;xr

=== <Result of SELECT Command> ===
    host_year  nation_code
===================================
         2008  'AUS'

1 rows selected.
```

```
INSERT INTO isol2_tbl
VALUES (2004, 'AUS');

INSERT INTO isol2_tbl
VALUES (2000, 'NED');
;xr

2 rows affected.


/* able to insert new rows
even if tran 2 uncommitted
*/
```

```
SELECT * FROM isol2_tbl;
;xr

/* phantom read may occur when tran
1 committed */
```

```
COMMIT;
;xr
```

```
=== <Result of SELECT Command> ===
    host_year  nation_code
===================================
         2008  'AUS'
         2004  'AUS'
         2000  'NED'

3 rows selected.
```

```
INSERT INTO isol2_tbl
VALUES (1994, 'FRA');
;xr

1 rows affected.
```

```
SELECT * FROM isol2_tbl;
;xr
```

	/* unrepeatable read may occur when tran 1 committed */

DELETE FROM isol2_tbl
WHERE nation_code = 'AUS'
and
host_year=2008;
;xr
 	
1 rows affected.
 	
/* able to delete rows even
if tran 2 uncommitted */	

COMMIT;
;xr
	=== <Result of SELECT Command> ===
	    host_year nation_code
	===================================
	         2004 'AUS'
	         2000 'NED'
	         1994 'FRA'
	
	3 rows selected.

ALTER TABLE isol2_tbl ADD COLUMN gold INT; ;xr 1 command(s) successfully processed. /* able to alter the table schema even if tran 2 is uncommitted yet*/	
	/* unrepeatable read may occur so that result shows different schema */ SELECT * FROM isol2_tbl; ;xr
COMMIT; ;xr	=== <Result of SELECT Command > === host_year nation_code gold =================================== 2004 'AUS' NULL 2000 'NED' NULL 1994 'FRA' NULL 3 rows selected.

READ COMMITTED CLASS with READ UNCOMMITTED INSTANCES

The lowest isolation level (1). The concurrency level is the highest. A dirty, non-repeatable or phantom read may occur for the tuple and a non-repeatable read may occur for the table as well. Similar to **REPEATABLE READ CLASS** with **READ UNCOMMITTED INSTANCES**(level 3) described above, but works differently for the table schema. That is, non-repeatable read due to table schema update may occur because another transaction T2 can change the schema of the table being viewed by the transaction T1.

The following are the rules of this isolation level:

- Transaction T1 can read the record being updated by another transaction T2.
- Transaction T1 can update/insert record to the table being viewed by another transaction T2.
- Transaction T1 can change the schema of the table being viewed by another transaction T2.

This isolation level uses a two-phase locking protocol for an exclusive lock. However, the shared lock on the tuple is released immediately after it is retrieved, and the update lock on the tuple is released immediately after the update is executed. The intention lock on the table is released immediately after the retrieval as well.

Example

<table>
<tr><th>session 1</th><th>session 2</th></tr>
<tr><td>

```
;autocommit off
AUTOCOMMIT IS OFF

SET TRANSACTION ISOLATION
LEVEL 1
;xr

Isolation level set to:
READ COMMITTED SCHEMA,
READ UNCOMMITTED
INSTANCES.
```

</td><td>

```
;autocommit off
AUTOCOMMIT IS OFF

SET TRANSACTION ISOLATION LEVEL 1
;xr

Isolation level set to:
READ COMMITTED SCHEMA, READ
UNCOMMITTED INSTANCES.
```

</td></tr>
<tr><td>

```
--creating a table

CREATE TABLE
isol1_tbl(host_year
integer, nation_code
char(3));
CREATE UNIQUE INDEX on
isol1_tbl(nation_code,
host_year);
INSERT INTO isol1_tbl
VALUES (2008, 'AUS');

COMMIT;
;xr
```

</td><td></td></tr>
<tr><td></td><td>

```
--selecting records from the table
SELECT * FROM isol1_tbl;
;xr

=== <Result of SELECT Command> ===
    host_year  nation_code
===================================
         2008  'AUS'

1 rows selected.
```

</td></tr>
<tr><td>

```
INSERT INTO isol1_tbl
VALUES (2004, 'AUS');

INSERT INTO isol1_tbl
VALUES (2000, 'NED');
;xr

2 rows affected.


/* able to insert new rows
even if tran 2 uncommitted
*/
```

</td><td></td></tr>
<tr><td></td><td>

```
SELECT * FROM isol1_tbl;
;xr

=== <Result of SELECT Command> ===
    host_year  nation_code
===================================
         2008  'AUS'
         2004  'AUS'
         2000  'NED'

3 rows selected.

/* dirty read may occur so that
tran_2 can select new rows
uncommitted by tran_1 */
```

</td></tr>
</table>

```
ROLLBACK;
;xr
```

```
SELECT * FROM isol1_tbl;
;xr

=== <Result of SELECT Command> ===
    host_year  nation_code
===================================
         2008  'AUS'

1 rows selected.

/* unrepeatable read may occur so
that selected results are different
*/
```

```
INSERT INTO isol1_tbl
VALUES (1994, 'FRA');
;xr

DELETE FROM isol1_tbl
WHERE nation_code = 'AUS'
and
host_year=2008;
;xr

1 rows affected.
1 rows affected.

/* able to delete rows
while tran 2 is selecting
rows*/
```

```
SELECT * FROM isol1_tbl;
;xr

=== <Result of SELECT Command> ===
    host_year  nation_code
===================================
         1994  'FRA'

1 rows selected.
```

```
ALTER TABLE isol1_tbl
ADD COLUMN gold INT;
;xr

1 command(s) successfully
processed.

/* able to alter the table
schema even if tran 2 is
uncommitted yet*/
```

```
/* unrepeatable read may occur so
that result shows different schema
*/

SELECT * FROM isol1_tbl;
;xr
```

```
COMMIT;
;xr
```

```
=== <Result of SELECT Command > ===
host_year  nation_code  gold
====================================
  1994  'FRA'           NULL

1 rows selected.
```

UPDATE INCONSISTENCY

In this isolation level, uncommitted updates may be lost, which makes a transaction unrestorable (cannot be rolled back) because the data are committed before the end of the transaction. CUBRID does not support this isolation level because this can cause the updates made by the user to be lost. However, if this isolation level is specified, CUBRID provides an appropriate level to the user application.

The following are the rules of this isolation level:

- A transaction does not overwrite an object being modified by another transaction.

Note A transaction can be restored in all supported isolation levels because updates are not committed before the end of the transaction.

Combination of Unsupported Isolation Level

You can set customized isolation levels by using the **SET TRANSACTION ISOLATION LEVE** statement. However, combinations of isolation levels below are not supported. If they are used, a system error message is shown up and an isolation level closest to the one specified is chosen.

The following are unsupported isolation levels. If table schema is changed while data is selected, unrepeatable read occurs; therefore, the combinations below are not supported.

- **READ COMMITTED CLASS** with **REPEATABLE READ INSTANCES**
- **READ UNCOMMITTED CLASS** with **REPEATABLE READ INSTANCES**

Neither are isolation levels below supported because updating a row by a transaction is not allowed while table schema is changed by other transaction.

- **READ UNCOMMITTED CLASS** with **READ COMMITTED INSTANCES**
- **READ UNCOMMITTED CLASS** with **READ UNCOMMITTED INSTANCES**

How to Handle Dirty Instance

CUBRID flushes dirty data (or dirty instances) in the client buffers to the database (server) such as the following situations. In additions to those, there can be more situations where flushes can be performed.

- Dirty data can be flushed to server when a transaction is committed.
- Some of dirty data can be flushed to server when a lot of data is loaded into the client buffers.
- Dirty data of table A can be flushed to server when the schema of table A is updated.
- Dirty data of table A can be flushed to server when the table A is retrieved (**SELECT**)
- Some of dirty data can be flushed to server when a server function is called.

Transaction Termination and Restoration

Overview

The restore process in CUBRID makes it possible that the database is not affected even if a software or hardware error occurs. In CUBRID, all read and update commands that are made during a transaction must be atomic. This means that either all of the transaction's commands are committed to the database or none are. The concept of atomicity is extended to the set of operations that consists of a transaction. The transaction must either commit so that all effects are permanently applied to the database or roll back so that all effects are removed. To ensure transaction atomicity, CUBRID applies the effects of the committed transaction again every time an error occurs without the updates of the transaction being written to the disk. CUBRID also removes the effects of partially committed transactions in the database every time the site fails (some transactions may have not committed or applications may have requested to cancel transactions). This restore feature eases the burden for the applications of maintaining the database consistency depending on the system error. The restore process used in CUBRID is based on the undo/redo logging mechanism.

CUBRID provides an automatic restore method to maintain the transaction atomicity when a hardware or software error occurs. You do not have to take the responsibility for restore since CUBRID's restore feature always returns the database to a consistent state even when an application or computer system error occurs. For this purpose, CUBRID automatically rolls back part of committed transactions when the application fails or the user requests explicitly. For example, a system error that occurred during the execution of the **COMMIT WORK** statement must be stopped if the transaction has not committed yet (it cannot be confirmed that the user's operation has been committed). Automatic stop prevents errors causing undesired changes to the database by canceling uncommitted updates.

Restarting Database

CUBRID uses log volumes/files and database backups to restore committed or uncommitted transactions when a system or media (disk) error occurs. Logs are also used to support the user-specified rollback. A log consists of a collection of sequential files created by CUBRID. The most recent log is called the active log, and the rest are called archive logs. A log file refers to both the active log and archive logs.

All updates of the database are written to the log. Actually, two copies of the updates are logged. The first one is called a before image and used to restore data during execution of the user-specified **ROLLBACK WORK** statement or during media or system errors. The second copy is an after image and used to re-apply the updates when a media or system error occurs.

When the active log is full, CUBRID copies it to an archive log to store in the disk. The archive log is needed to restore the database when a system failure occurs. You don't need to maintain archive logs if there is no need for system failure restore. This configuration can be set by using the **media_failure_support** system parameter. For more information on this parameter, see Logging-Related Parameters.

Normal Termination or Error

CUBRID restores the database if it restarts due to a normal termination or a device error. The restore process re-applies the committed changes that have not been applied to the database and removes the uncommitted changes stored in the database. The general operation of the database resumes after the restore is completed. This restore process does not use any archive logs or database backup.

In a client/server environment, the database can restart by using server utilities.

Media Error

The user's intervention is somewhat needed to restart the database after a media error occurs. The first step is to restore the database by installing a backup of a known good state. In CUBRID, the most recent log file (the one after the last backup) must be installed. This specific log (archive or active) is applied to a backup copy of the database. As with normal termination, the database can restart after restoration is committed.

It is important to back up the database periodically. Backup periods differ depending on the frequency of database updates. Once a database backup is created, CUBRID uses the current database backup to specify the archive log that is not needed any more. However, CUBRID does not delete the archive log. The database administrator must take extra care when deleting the database backup or archive log. In some cases, the latest database backup may fail.

Note To minimize the possibility of losing database updates, it is recommended to create a snapshot of the archive log and backup the log to a disk before it is deleted from the disk. The DBA can backup and restore the database by using the **cubrid backupdb** and **cubrid restoredb** utilities. For more information on these utilities, see Database Backup.

Database User Authorization

Database User

CUBRID has two types of users by default: **DBA** and **PUBLIC**.

- All users have authorization granted to the **PUBLIC** user. All users of the database are automatically the members of **PUBLIC**. Granting authorization to the **PUBLIC** means granting it all users.
- The **DBA** user has the authorization of the database administrator. The **DBA** automatically becomes the member of all users and groups. That is, the **DBA** is granted the access for all tables. Therefore, there is no need to grant authorization explicitly to the **DBA** and **DBA** members. Each database user has a unique name. The database administrator can create multiple users simultaneously using the **cubrid createdb** utility (see How to Use the Database Management Utilities for details). A database user cannot have a member who already has the same authorization. If authorization is granted to a user, all members of the user is automatically granted the same authorization.

Managing User

Description

DBA and **DBA** members can create, drop and alter users by using SQL statements.

Syntax

```
CREATE USER user_name
[ PASSWORD password ]
[ GROUPS user_name [ {, user_name } ... ] ]
[ MEMBERS user_name [ {, user_name } ... ] ] ;
DROP USER user_name;
ALTER USER user_name PASSWORD password;
```

- *user_name* : Specifies the user name to create, delete or change.
- *password* : Specifies the user password to create or change.

Example 1

The following is an example in which the user Fred is created, the password is changed, and then the user Fred is deleted.

```
CREATE USER Fred;
ALTER USER Fred PASSWORD '1234';
DROP USER Fred;
```

Example 2

The following is an example in which a user is created and then members are added to the user. By the following statement, company becomes a group that has engineering, marketing and design as its members. marketing becomes a group with members smith and jones, design becomes a group with a member smith, and engineering becomes a group with a member brown.

```
CREATE USER company;
CREATE USER engineering GROUPS company;
CREATE USER marketing GROUPS company;
CREATE USER design GROUPS company;
CREATE USER smith GROUPS design, marketing;
CREATE USER jones GROUPS marketing;
CREATE USER brown GROUPS engineering;
```

Example 3

The following example creates the same groups as above, but uses the **MEMBERS** keyword instead of **GROUPS**.

```
CREATE USER smith;
CREATE USER brown;
CREATE USER jones;
CREATE USER engineering MEMBERS brown;
CREATE USER marketing MEMBERS smith, jones;
CREATE USER design MEMBERS smith;
CREATE USER company MEMBERS engineering, marketing, design;
```

Granting Authorization

Description

In CUBRID, the smallest grant unit of authorization is a table. You must grant appropriate authorization to other users (groups) before allowing them to access the table you created.

You don't need to grant authorization individually because the members of the granted group have the same authorization. The access to the (virtual) table created by a **PUBLIC** user is allowed to all other users. You can grant access authorization to a user by using the **GRANT** statement.

Syntax

```
GRANT operation [ { ,operation }_ ] ON table_name [ { ,table_name }_ ]
TO user [ { ,user }_ ] [ WITH GRANT OPTION ] [ ; ]
```

- *operation* : Indicates an operation that can be used when granting authorization. The following table shows the operations:
- **SELECT** : Allows to read the table definitions and retrieve records. The most general type of permissions.
- **INSERT** : Allows to create records in the table.
- **UPDATE** : Allows to modify the records already existing in the table.
- **DELETE** : Allows to delete records in the table.
- **ALTER** : Allows to modify the table definition, rename or delete the table.
- **INDEX** : Allows to call table methods or instance methods.
- **EXECUTE** : Allows to call table methods or instance methods.
- **ALL PRIVILEGES** : Includes all permissions described above.
- *table_name* : Specifies the name of the table or virtual table to be granted.
- *user* : Specifies the name of the user (group) to be granted. Enter the login name of the database user or **PUBLIC**, a system-defined user. If **PUBLIC** is specified, all database users are granted with the permission.
- **WITH GRANT OPTION** : **WITH GRANT OPTION** allows the grantee of authorization to grant that same privilege to another user.

Example 1

The following is an example in which the SELECT authorization for the olympic table is granted to Fred (all members of Fred).

```
GRANT SELECT ON olympic TO Fred;
```

Example 2

The following is an example in which **SELECT**, **INSERT**, **UPDATE** and **DELETE** authorization for the nation and athlete tables are granted to Jeniffer and Daniel (all members belonging to Jeniffer and Daniel).

```
GRANT SELECT, INSERT, UPDATE, DELETE ON nation, athlete TO  Jeniffer, Daniel;
```

Example 3

The following is an example in which all authorization for the game and event tables are granted to all users.

```
GRANT ALL PRIVILEGES ON game, event TO public;
```

Example 4

In the following example, the **GRANT** statement grants search authorization for the record and history tables to Ross, and **WITH GRANT OPTION** allows Ross to grant the same authorization to another user.

```
GRANT SELECT ON record, history TO Ross WITH GRANT OPTION;
```

Caution

- The grantor of authorization must be the owner of all tables listed before the grant operation or have **WITH GRANT OPTION** specified.
- Before granting **SELECT**, **UPDATE**, **DELETE** and **INSERT** authorization for a virtual table, the owner of the virtual table must have **SELECT** and GRANT authorization for all the tables included in the queries in the virtual table's query specification. The **DBA** user and the members of the **DBA** group are automatically granted all authorization for all tables.

Revoking Authorization

Description

You can revoke privileges using the **REVOKE** statement. The privileges granted to a user can be revoked anytime. If more than one privilege are granted to a user, all or part of the privileges can be revoked. In addition, if privileges on multiple tables are granted to more than one user using one **GRANT** statement, the privileges can be selectively revoked for specific users and tables.

If the privilege (**WITH GRANT OPTION**) is revoked from the grantor, the privilege granted to the grantee by that grantor is also revoked.

Syntax

```
REVOKE operation [ { , operation }_ ] ON table_name [ { , class_name }_ ]
FROM user [ { , user }_ ] [ ; ]
```

- *operation* : Indicates an operation that can be used when granting privileges. (See **Syntax** in Granting Privileges for details)
- *table_name* : Specifies the name of the table or virtual table to be granted.
- *user* : Specifies the name of the user (group) to be granted.

Example 1

The following is an example in which **SELECT**, **INSERT**, **UPDATE** and **DELETE** privileges for the nation and athlete tables are granted to Fred and John.

```
GRANT SELECT, INSERT, UPDATE, DELETE ON nation, athlete TO Fred, John;
```

Example 2

The following is an example in which the **REVOKE** statement is used to allow John only the SELECT privilege while maintaining all the privileges for Fred granted in Example 1. If John granted the privileges to another user, the grantee is also allowed to use the SELECT privilege only.

```
REVOKE INSERT, UPDATE, DELETE ON nation, athlete FROM John;
```

Example 3

The following is an example in which the **REVOKE** statement is used to revoke all privileges granted to Fred in Example 1. If the statement is executed, Fred is not be allowed to perform any operation on the nation and athlete tables.

```
REVOKE ALL PRIVILEGES ON nation, athlete FROM Fred;
```

User Authorization Management METHOD

Description

The database administrator (**DBA**) can check and modify user authorization by calling authorization-related methods defined in **db_user** where information about database user is stored, and **db_authorizations** (the system authorization class). The administrator can specify **db_user** or **db_authorization** depending on the method to be called, and save the return value of a method to a variable. In addition, some methods can be called only by **DBA** or members of **DBA** group.

Syntax

```
CALL method_definition ON CLASS auth_class [ TO variable ] [ ; ]
CALL method_definition ON variable [ ; ]
```

login( ) method

As a class method of **db_user** class, this method is used to change the users who are currently connected to the database. The name and password of a new user to connect are given as parameters, and they must be string type. If there is no password, a blank string ('') can be used as the parameter. **DBA** and **DBA** members can call the **login( )** method without a password.

```
-- Connect as DBA user who has no password
CALL login ('dba', '') ON CLASS db_user;
-- Connect as a user_1 whose password is cubrid
CALL login ('user_1', 'cubrid') ON CLASS db_user;
```

add_user( ) method

As a class method of **db_user** class, this method is used to add a new user. The name and password of a new user to add are given as parameters, and they must be string type. At this time, the new user name should not duplicate any user name already registered in a database. The **add_user( )** can be called only by **DBA** or members of **DBA** group.

```
-- Add user_2 who has no password
CALL add_user ('user_2', '') ON CLASS db_user;
-- Add user_3 who has no password, and save the return value of a method into an admin
variable
CALL add_user ('user_2', '') ON CLASS db_user to admin;
```

drop_user( ) method

As a class method of **db_user** class, this method is used to drop an existing user. Only the user name to be dropped is given as a parameter, and it must be a string type. However, the owner of a class cannot be dropped thus **DBA** needs to specify a new owner of the class before dropping the user. The **drop_user( )** method can be also called only by **DBA** or members of **DBA**.

```
-- Delete user_2
CALL drop_user ('user_2') ON CLASS db_user;
```

find_user( ) method

As a class method of **db_user** class, this method is used to find a user who is given as a parameter. The name of a user to be found is given as a parameter, and the return value of the method is stored into a variable that follows 'to'. The stored value can be used in a next query execution.

```
-- Find user_2 and save it into a variable called 'admin'
CALL find_user ('user_2') ON CLASS db_user to admin;
```

set_password( ) method

This method is an instance method that can call each user instance, and it is used to change a user's password. The new password of a specified user is given as a parameter. General users other than **DBA** and **DBA** group members can only change their own passwords.

```
-- Add user_4 and save it into a variable called user_common
CALL add_user ('user_4','') ON CLASS db_user to user_common;
```

```
-- Change the password of user_4 to 'abcdef'
CALL set_password('abcdef') on user_common;
```

change_owner() method

As a class method of **db_authorizations** class, this method is used to change the owner of a class. The name of a class for which you want to change the owner, and the name of a new owner are given as parameters. At this time, the class and owner that are specified as a parameter must exist in a database. Otherwise, an error occurs. **change_owner( )** can be called only by **DBA** or members of **DBA** group.

```
-- Change the owner of table_1 to user_4
CALL change_owner ('table_1', 'user_4') ON CLASS db_authorizations;
```

Example

The following is an example of a **CALL** statement that calls the find_user method defined in the system table **db_user**. It is called to determine whether the database user entered as the **find_user** exists. The first statement calls the table method defined in the **db_user** class. The name (**db_user** in this case) is stored in x if the user is registered in the database. Otherwise, **NULL** is stored.

The second statement outputs the value stored in the variable x. In this query statement, the **DB_ROOT** is a system class that can have only one record. It can be used to output the value of sys_date or other registered variables. For this purpose, the **DB_ROOT** can be replaced by another table having only one record.

```
CALL find_user('dba') ON CLASS db_user to x;
;xrun

=== <Result of CALL Command in Line 1> ===
Result
======================
db_user

SELECT x FROM db_root;
;xrun

=== <Result of SELECT Command in Line 1> ===
x
======================
db_user
```

With **find_user**, you can determine if the user exists in the database depending on whether the returned value is **NULL** or not.

Query Optimization

Updating Statistics

Description

With the **UPDATE STATISTICS ON** statement, you can generate internal statistics used by the query processor. Such statistics allow the database system to perform query optimization more efficiently.

Syntax

```
UPDATE STATISTICS ON { table_spec [ {, table_spec } ] | ALL CLASSES | CATALOG CLASSES }
[ ; ]
table_spec :
single_table_spec
( single_table_spec [ {, single_table_spec } ] )
single_table_spec :
[ ONLY ] table_name
| ALL table_name [ ( EXCEPT table_name ) ]
```

- **ALL CLASSES** : If the **ALL CLASSES** keyword is specified, the statistics on all the tables existing in the database are updated.

Using SQL Hint

Description

Using hints can affect the performance of query execution. you can allow the query optimizer to create more efficient execution plan by referring the SQL HINT. The SQL HINTs related tale join, index, and statistics information are provided by CUBRID.

Syntax

```
CREATE /*+ NO_STATS */ [TABLE | CLASS] ...;
ALTER /*+ NO_STATS */ [TABLE | CLASS] ...;

CREATE /*+ NO_STATS */ INDEX ...;
ALTER /*+ NO_STATS */ INDEX ...;
DROP /*+ NO_STATS */ INDEX ...;

SELECT /*+ hint [ { hint } ... ] */
SELECT --+ hint [ { hint } ... ]
SELECT //+ hint [ { hint } ... ]

hint :
USE_NL[(spec-name[{, spec-name}...])]
USE_IDX[(spec-name[{, spec-name}...])]
USE_MERGE[(spec-name[{, spec-name}...])]
ORDERED
```

SQL hints are specified by using plus signs and comments. CUBRID interprets this comment as a list of hints separated by blanks. The hint comment must appear after the **SELECT**, **CREATE**, or **ALTER** keyword, and the comment must begin with a plus sign (+), following the comment delimiter.

- *hint* : The following hints can be specified.
- **USE_NL** : Related to a table join, the query optimizer creates a nested loop join execution plan with this hint.
- **USE_MERGE** : Related to a table join, the query optimizer creates a sort merge join execution plan with this hint.
- **ORDERED** : Related to a table join, the query optimizer create a join execution plan with this hint, based on the order of tables specified in the FROM clause. The left table in the FROM clause becomes the outer table; the right one becomes the inner table.
- **USE_IDX** : Related to a index, the query optimizer creates a index join execution plan corresponding to a specified table with this hint.

- **NO_STATS** : Related to statistics information, the query optimizer does not update statistics information. Query performance for the corresponding queries can be improved; however, query plan is not optimized because the information is not updated.
- *spec_name* : If the *spec_name* is specified together with **USE_NL**, **USE_IDX** or **USE_MERGE**, the specified join method applies only to the *spec_name*. If **USE_NL** and **USE_MERGE** are specified together, the given hint is ignored. In some cases, the query optimizer cannot create a query execution plan based on the given hint. For example, if **USE_NL** is specified for a right outer join, the query is converted to a left outer join internally, and the join order may not be guaranteed.

Example 1

The following is an example of retrieving the years when Sim Kwon Ho won medals and the types of medals. Here, a nested loop join execution plan needs to be created which has the **athlete** table as an outer table and the **game** table as an inner table. It can be expressed by the following query. The query optimizer creates a nested loop join execution plan that has the **game** table as an outer table and the **athlete** table as an inner table.

```
SELECT /*+ USE_NL ORDERED  */ a.name, b.host_year, b.medal
FROM athlete a, game b WHERE a.name = 'Sim Kwon Ho' AND a.code = b.athlete_code;
=== <Result of SELECT Command in Line 1> ===
  name                    host_year  medal
=========================================================
  'Sim Kwon Ho'                2000  'G'
  'Sim Kwon Ho'                1996  'G'
2 rows selected.
```

Example 2

The following is an example of viewing query execution time with **NO_STAT** hint to improve the functionality of drop partitioned table (before_2008); any data is not stored in the table. Assuming that there are more than 1 million data in the **participant2** table. The execution time in the example can differ depending on system performance and database configuration.

```
-- Not using NO_STATS hint
ALTER TABLE participant2 DROP partition before_2008;
SQL statement execution time: 31.684550 sec
Current transaction has been committed.
1 command(s) successfully processed.

-- Using NO_STATS hint
ALTER /*+ NO_STATS */ TABLE participant2 DROP partition before_2008;
SQL statement execution time: 0.025773 sec
Current transaction has been committed.
1 command(s) successfully processed.
```

Viewing Query Plan

Description

To view a query plan for a CUBRID SQL query, change the value of the optimization level by using the **SET OPTIMIZATION** statement. You can get the current optimization level value by using the **GET OPTIMIZATION** statement.

The CUBRID query optimizer determines whether to perform query optimization and output the query plan by referencing the optimization level value set by the user. The query plan is displayed as standard output; the following explanations are based on the assumption that the plan is used in a terminal-based program such as the CSQL Interpreter. For information on how to view a query plan by using the CUBRID Manager, see the Query Execution Plan.

Syntax

```
SET OPTIMIZATION LEVEL opt-level [;]
GET OPTIMIZATION LEVEL [ { TO | INTO } variable ] [;]
```

- *opt-level*: A value that specifies the optimization level. It has the following meanings.

- 0 : Does not perform query optimization. The query is executed using the simplest query plan. This value is used only for debugging.
- 1 : Create a query plan by performing query optimization and executes the query. This is a default value used in CUBRID, and does not have to be changed in most cases.
- 2 : Creates a query plan by performing query optimization. However, the query itself is not executed. Generally, this value is not used; it is used together with the following values to be set for viewing query plans.
- 257 : Performs query optimization and outputs the created query plan. This value works for displaying the query plan by internally interpreting the value as 256+1 related with the value 1.
- 258 : Performs query optimization and outputs the created query plan. The difference from the value 257 is that the query is not executed. That is, this value works for displaying the query plan by internally interpreting the value as 256+2 related with the value 2. This setting is useful to examine the query plan but not to intend to see the query results.
- 513 : Performs query optimization and outputs the detailed query plan. This value works for displaying more detailed query plan than the value 257 by internally interpreting the value as 512+1.
- 514 : Performs query optimization and outputs the detailed query plan. However, the query is not executed. This value works for displaying more detailed query plan than the value 258 by internally interpreting the value as 512+2.

Example

The following example is to display the query plan but not execute a query itself by setting the optimization level to 258, the query is that retrieves the years when Sim Kwon Ho won medals and the types of medals.

```
GET OPTIMIZATION LEVEL

=== < Result of GET OPTIMIZATION Command in Line 1> ===
              Result
=============
                        1

SET OPTIMIZATION LEVEL 258

SELECT a.name, b.host_year, b.medal
FROM athlete a, game b WHERE a.name = 'Sim Kwon Ho' AND a.code = b.athlete_code
Query plan:
  Nested loops
        Sequential scan(game b)
        Index scan(athlete a, pk_athlete_code, a.code=b.athlete_code)

=== < Result of SELECT Command in Line 1> ===
There are no results.
0 rows selected.
```

TRIGGER

CREATE TRIGGER

Guideline for TRIGGER Definition

Trigger definition provides various and powerful functionalities. Before creating a trigger, you must consider the following:

- **Does the trigger condition expression cause unexpected results (side effects)?**

  You must use the SQL statements within an expectable range.

- **Does the trigger action change the table given as its event target?**

  While this type of design is not forbidden in the trigger definition, it must be carefully applied, because a trigger can be created that falls into an infinite loop. When the trigger action modifies the event target table, the same trigger can be called again. If a trigger occurs in a statement that contains a **WHERE** clause, there is no side effect in the table affected by the **WHERE** clause.

- **Does the trigger cause unnecessary overhead?**

  If the desired action can be expressed more effectively in the source, implement it directly in the source.

- **Is the trigger executed recursively?**

  If the trigger action calls a trigger and this trigger calls the previous trigger again, a recursive loop is created in the database. If a recursive loop is created, the trigger may not be executed correctly, or the current session must be forced to terminate to break the ongoing infinite loop.

- **Is the trigger definition unique?**

  A trigger defined in the same table or the one started in the same action becomes the cause of an unrecoverable error. A trigger in the same table must have a different trigger event. In addition, trigger priority must be explicitly and unambiguously defined.

TRIGGER Definition

Description

A trigger is created by defining a trigger target, condition and action to be performed in the **CREATE TRIGGER** statement. A trigger is a database object that performs a defined action when a specific event occurs in the target table.

Syntax

```
CREATE TRIGGER trigger_name
[ STATUS { ACTIVE | INACTIVE } ]
[ PRIORITYkey ]
event_time event_type[ event_target ]
[ IFcondition ]
EXECUTE [ AFTER | DEFERRED ] action [ ; ]

event_time:
   • BEFORE
   • AFTER
   • DEFERRED

event_type:
   • INSERT
   • STATEMENT INSERT
   • UPDATE
   • STATEMENT UPDATE
   • DELETE
   • STATEMENT DELETE
   • ROLLBACK
   • COMMIT

event_target:
   • ONtable_name
   • ONtable_name [ (column_name) ]

condition:
   • expression

action:
 • REJECT
 • INVALIDATE TRANSACTION
 •  PRINT message_string
 •  INSERT statement
 •  UPDATE statement
 •  DELETE statement
```

- *trigger_name* : Specifies the name of the trigger to be defined.
- [**STATUS** { **ACTIVE** | **INACTIVE** }] : Defines the state of the trigger (if not defined, the default value is **ACTIVE**).
  - If **ACTIVE** state is specified, the trigger is executed every time the corresponding event occurs.
  - If **INACTIVE** state is specified, the trigger is not executed even when the corresponding event occurs. The state of the trigger can be modified. For more information, see Altering TRIGGER Definition section.
- [**PRIORITY** *key*] : Specifies a trigger priority if multiple triggers are called for an event. *key* must be a floating point value that is not negative. If the priority is not defined, the lowest priority 0 is assigned. Triggers having the same priority are executed in a random order. The priority of triggers can be modified. For more information, see Altering TRIGGER Definition section.
- *event_time* : Specifies the point of time when the conditions and actions are executed. **BEFORE**, **AFTER** or **DEFERRED** can be specified. For more information, see the Event Time section.
- *event_type* : Trigger types are divided into a user trigger and a table trigger. For more information, see the TRIGGER Event Type section.
- *event_target* : An event target is used to specify the target for the trigger to be called. For more information, see the TRIGGER Event Target section.
- *condition* : Specifies the trigger condition. For more information, see the TRIGGER Condition section.
- *action* : Specifies the trigger action. For more information, see the TRIGGER Action section.

Example 1

The following is an example of creating a trigger that rejects the update if the number of medals won is smaller than 0 when an instance of the participant table is updated.

As shown below, the update is rejected if you try to change the number of gold medals that Korea won in the 2004 Olympic Games to a negative number.

```
CREATE TRIGGER medal_trigger
BEFORE UPDATE ON participant
IF new.gold < 0 OR new.silver < 0 OR new.bronze < 0
EXECUTE REJECT;

csql> UPDATE participant SET gold = -5 WHERE nation_code = 'KOR'
AND host_year = 2004;
csql> ;x
In line 1, column 1,
ERROR: The operation has been rejected by trigger "medal_trigger".
```

Event Time

Description

Specifies the point of time when trigger conditions and actions are executed. The types of event time are **BEFORE**, **AFTER** and **DEFERRED**.

- **BEFORE** : Checks the condition before the event is processed.
- **AFTER** : Checks the condition after the event is processed.
- **DEFERRED** : Checks the condition at the end of the transaction for the event. If you specify **DEFERRED**, you cannot use **COMMIT** or **ROLLBACK** as the event type.

Trigger Type

User Trigger

- A trigger relevant to a specific user of the database is called a user trigger.
- A user trigger has no event target and is executed only by the owner of the trigger (the user who created the trigger).
- Event types that define a user trigger are **COMMIT** and **ROLLBACK**.

Table Trigger

- A trigger that has a table as the event target is called a table trigger (class trigger).
- A table trigger can be seen by all users who have the **SELECT** privilege on the target table.
- Event types that define a table trigger are instance and statement events.

TRIGGER Event Type

Description

- Instance events : An event type whose unit of operation is an instance. The types of instance events are as follows:
    - **INSERT**
    - **UPDATE**
    - **DELETE**
- Statement events : If you define a statement event as an event type, the trigger is called only once when the trigger starts even when there are multiple objects (instances) affected by the given statement (event). The types of statement events are as follows:
    - **STATEMENT INSERT**
    - **STATEMENT UPDATE**
    - **STATEMENT DELETE**
- Other events : **COMMIT** and **ROLLBACK** cannot be applied to individual instances.
    - **COMMIT**
    - **ROLLBACK**

Example 1

The following is an example of using an instance event. The example trigger is called by each instance affected by the database update. For example, if the score values of five instances in the history table are modified, the trigger is called five times. If you want the trigger to be called only once, before the first instance of the score column is updated, use the **STATEMENT UPDATE** type as in example 2.

```
CREATE TRIGGER example
...
BEFORE UPDATE ON history(score)
...
```

Example 2

The following is an example of using a statement event. If you define a statement event, the trigger is called only once before the first instance gets updated even when there are multiple instances affected by the update.

```
CREATE TRIGGER example
...
BEFORE STATEMENT UPDATE ON history(score)
...
```

Caution

- You must specify the event target when you define an instance or statement event as the event type.
- **COMMIT** and **ROLLBACK** cannot have an event target.

TRIGGER Event Target

Description

An event target specifies the target for the trigger to be called. The target of a trigger event can be specified as a table or column name. If a column name is specified, the trigger is called only when the specified column is affected by the event. If a column is not specified, the trigger is called when any column of the table is affected. Only **UPDATE** and **STATEMENT UPDATE** events can specify a column as the event target.

Example

The following is an example of specifying the score column of the history table as the event target of the example trigger.

```
CREATE TRIGGER example
...
BEFORE UPDATE ON history(score)
...
```

Combination of Event Type and Target

Description

A database event calling triggers is identified by the trigger event type and event target in a trigger definition. The following table shows the trigger event type and target combinations, along with the meaning of the CUBRID database event that the trigger event represents.

Event Type	Event Target	Corresponding Database Activity
UPDATE	Table	Trigger is called whenever any column of the table is updated.
STATEMENT UPDATE	Table	Trigger is called whenever an **UPDATE** statement is executed on the table.
INSERT	Table	Trigger is called whenever an instance of the table is created.
STATEMENT INSERT	Table	Trigger is called whenever an **INSERT** statement is executed on the table.
DELETE	Table	Trigger is called whenever an instance of the table is deleted.
STATEMENT DELETE	Table	Trigger is called whenever a **DELETE** statement is executed on the table.
COMMIT	None	Trigger is called whenever a database transaction is committed. The **COMMIT WORK** statement initiates this trigger.
ROLLBACK	None	Trigger is called whenever the database transaction is rolled back. The **ROLLBACK WORK** statement initiates this trigger.

TRIGGER Condition

Description

You can specify whether a trigger action is to be performed by defining a condition when defining the trigger.

- If a trigger condition is specified, it can be written as an independent compound expression that evaluates to true or false. In this case, the expression can contain arithmetic and logical operators allowed in the **WHERE** clause of the **SELECT** statement. The trigger action is performed if the condition is true; if it is false, action is ignored.
- If a trigger condition is omitted, the trigger becomes an unconditional trigger, which refers to that the trigger action is performed whenever it is called.

Example 1

The following is an example of using a correlation name in an expression within a condition. If the event type is **INSERT**, **UPDATE** or **DELETE**, the expression in the condition can reference the correlation names **obj**, **new** or **old** to access a specific column. This example prefixes **obj** to the column name in the trigger condition to show that the example trigger tests the condition based on the current value of the record column.

```
CREATE TRIGGER example
........
IF obj.record * 1.20  < 500
.......
```

Example 2

The following is an example of using the **SELECT** statement in an expression within a condition. The trigger in this example uses the **SELECT** statement that contains an aggregate function **COUNT**(*) to compare the value with a constant. The **SELECT** statement must be enclosed in parentheses and must be placed at the end of the expression.

```
CREATE TRIGGER example
......
IF 1000 >  (SELECT COUNT( * ) FROM participant)
......
```

Caution

The expression given in the trigger condition may cause side effects on the database if a method is called while the condition is performed. A trigger condition must be constructed to avoid unexpected side effects in the database.

Correlation Name

You can access the column values defined in the target table by using a correlation name in the trigger definition. A correlation name is the instance that is actually affected by the database operation calling the trigger. A correlation name can also be specified in a trigger condition or action.

The types of correlation names are **new**, **old** and **obj**. These correlation names can be used only in instance triggers that have an **INSERT**, **UPDATE** or **DELETE** event.

As shown in the table below, the use of correlation names is further restricted by the event time defined for the trigger condition.

	BEFORE	AFTER or DERERRED
INSERT	**new**	**obj**
UPDATE	**obj** **new**	**obj** **old** (**AFTER**)
DELETE	**obj**	N/A

Correlation Name	Representative Attribute Value
obj	Refers to the current attribute value of an instance. This can be used to access attribute values before an instance is updated or deleted. It is also used to access attribute values after an instance has been updated or inserted.
new	Refers to the attribute value proposed by an insert or update operation. The new value can be accessed only before the instance is actually inserted or updated.
old	Refers to the attribute value that existed prior to the completion of an update operation. This value is maintained only while the trigger is being performed. Once the trigger is completed, the **old** values get lost.

TRIGGER Action

Description

A trigger action describes what to be performed if the trigger condition is true or omitted. If a specific point of time (**AFTER** or **DEFERRED**) is not given in the action clause, the action is executed at the same time as the trigger event.

The following is a list of actions that can be used for trigger definitions.

- **REJECT** : **REJECT** discards the operation that initiated the trigger and keeps the former state of the database, if the condition is not true. Once the operation is performed, **REJECT** is allowed only when the action time is **BEFORE** because the operation cannot be rejected. Therefore, you must not use **REJECT** if the action time is **AFTER** or **DERERRED**.
- **INVALIDATE TRANSACTION** : **INVALIDATE TRANSACTION** allows the event operation that called the trigger, but does not allow the transaction that contains the commit to be executed. You must cancel the transaction by using the **ROLLBACK** statement if it is not valid. Such action is used to protect the database from having invalid data after a data-changing event happens.
- **PRINT** : **PRINT** outputs trigger actions on the terminal screen in text messages, and can be used during developments or tests. The results of event operations are not rejected or discarded.
- **INSERT** : **INSERT** inserts one or more new instances to the table.
- **UPDATE** : **UPDATE** updates one or more column values in the table.
- **DELETE** : **DELETE** deletes one or more instances from the table.

Example

The following example shows how to define an action when a trigger is created. The medal_trig trigger defines **REJECT** in its action. **REJECT** can be specified only when the action time is **BEFORE**.

```
CREATE TRIGGER medal_trig
BEFORE UPDATE ON participant
IF new.gold < 0 OR new.silver < 0 OR new.bronze < 0
EXECUTE REJECT;
```

Caution

- Trigger may fall into an infinite loop when you use **INSERT** in an action of a trigger where an **INSERT** event is defined.
- If a trigger where an **UPDATE** event is defined runs on a partitioned table, you must be careful because the defined partition can be broken or unintended malfunction may occur. To prevent such situation, CUBRID outputs an error so that the **UPDATE** causing changes to the running partition is not executed. Trigger may fall into an infinite loop when you use **UPDATE** in an action of a trigger where an **UPDATE** event is defined.

ALTER TRIGGER

Description

In the trigger definition, **STATUS** and **PRIORITY** options can be changed by using the **ALTER** statement. If you need to alter other parts of the trigger (event targets or conditional expressions), you must delete and then re-create the trigger.

Syntax

```
ALTER TRIGGER trigger_name  trigger_option [ ; ]
trigger_option :
• STATUS { ACTIVE | INACTIVE }
• PRIORITY key
```

- *trigger_name* : Specifies the name of the trigger to be changed.
- *trigger_option* :
  - **STATUS** { **ACTIVE** | **INACTIVE** } : Changes the status of the trigger.
  - **PRIORITY** *key* : Changes the priority.

Example

The following is an example of creating the medal_trig trigger and then changing its state to **INACTIVE** and its priority to 0.7.

```
CREATE TRIGGER medal_trig
STATUS ACTIVE
BEFORE PDATE ON participant
IF new.gold < 0 OR new.silver < 0 OR new.bronze < 0
EXECUTE REJECT;
ALTER TRIGGER medal_trig STATUS INACTIVE;
ALTER TRIGGER medal_trig PRIORITY 0.7;
```

Caution

- Only one option can be specified in a single **ALTER TRIGGER** statement.
- To change a table trigger, you must be the trigger owner or granted the **ALTER** privilege on the table where the trigger belongs.
- A user trigger can only be changed by its owner. For more information on these options, see the CREATE TRIGGER (Syntax) section. The key specified together with the **PRIORITY** option must be a non-negative floating point value.

DROP TRIGGER

Description

You can drop a trigger by using the **DROP TRIGGER** statement.

Syntax

```
DROP TRIGGER trigger_name [ ; ]
```

- *trigger_name* : Specifies the name of the trigger to be dropped.

Example

The following is an example of dropping the medal_trig trigger.

```
DROP TRIGGER medal_trig;
```

Caution

- A user trigger (i.e. the trigger event is **COMMIT** or **ROLLBACK**) can be seen and dropped only by the owner.
- Only one trigger can be dropped by a single **DROP TRIGGER** statement. A table trigger can be dropped by a user who has an **ALTER** authorization on the table.

RENAME TRIGGER

Description

You can change a trigger name by using the **TRIGGER** reserved word in the **RENAME** statement.

Syntax

```
RENAME TRIGGER old_trigger_name AS new_trigger_name [ ; ]
```

- *old_trigger_name* : Specifies the current name of the trigger.
- *new_trigger_name* : Specifies the name of the trigger to be changed.

Example

```
RENAME TRIGGER medal_trigger AS medal_trig;
```

Caution

- A trigger name must be unique among all trigger names. The name of a trigger can be the same as the table name in the database.
- To rename a table trigger, you must be the trigger owner or granted the **ALTER** privilege on the table where the trigger belongs. A user trigger can only be renamed by its user.

Deferred Condition and Action

Definition

A deferred trigger action and condition can be executed later or canceled. These triggers include a **DEFERRED** time option in the event time or action clause. If the **DEFERRED** option is specified in the event time and the time is omitted before the action, the action is deferred automatically.

Executing Deferred Condition and Action

Description

Executes the deferred condition or action of a trigger immediately.

Syntax

```
EXECUTE DEFERRED TRIGGER trigger_identifier [ ; ]
trigger_identifier :
• trigger_name
• ALL TRIGGERS
```

- *trigger_identifier* :
  - *trigger_name* : Executes the deferred action of the trigger when a trigger name is specified.
  - **ALLTRIGGERS** : All currently deferred actions are executed.

Dropping Deferred Condition and Action

Description

Drops the deferred condition and action of a trigger.

Syntax

```
DROP DEFERRED TRIGGER trigger_identifier [ ; ]
trigger_option :
• trigger_name
• ALL TRIGGERS
```

- *trigger_option* :
  - *trigger_name* : Cancels the deferred action of the trigger when a trigger name is specified.
  - **ALLTRIGGERS** : All currently deferred actions are canceled.

Granting TRIGGER Authorization

Description

Trigger authorization is not granted explicitly. Authorization on the table trigger is automatically granted to the user if the authorization is granted on the event target table described in the trigger definition. In other words, triggers that have table targets (**INSERT**, **UPDATE**, etc.) are seen by all users. User triggers (**COMMIT** and **ROLLBACK**) are seen only by the user who defined the triggers. All authorizations are automatically granted to the trigger owner.

Caution

- To define a table trigger, you must have an **ALTER** authorization on the table.

- To define a user trigger, the database must be accessed by a valid user.

TRIGGER Debugging

Definition and Example

Description

Once a trigger is defined, it is recommended to check whether it is running as intended. Sometimes the trigger takes more time than expected in processing. This means that it is adding too much overhead to the system or has fallen into a recursive loop. This section explains several ways to debug the trigger.

Example

The following is an example of a trigger that was defined to fall into a recursive loop when it is called. A loop trigger is somewhat artificial in its purpose, but can be used as an example for debugging the trigger.

```
CREATE TRIGGER loop_tgr
BEFORE UPDATE ON participant(gold)
IF new.gold > 0
EXECUTE UPDATE participant
        SET gold = new.gold - 1
        WHERE nation_code = obj.nation_code AND host_year = obj.host_year;
```

Viewing TRIGGER Execution Log

Description

You can view the execution log of the trigger from a terminal by using the **SET TRIGGER TRACE** statement.

Syntax

```
SET TRIGGER TRACE switch [ ; ]
switch:
• ON
• OFF
```

- *switch* :
  - **ON** : Runs the **TRACE** until the switch is set to **OFF** or the current database session terminates.
  - **OFF** : Stops the **TRACE**.

Example

The following is an example of running the **TRACE** and executing the loop trigger to view the trigger execution logs. To identify the trace for each condition and action executed when the trigger is called, a message is displayed on the terminal. The following message appears 15 times because the loop trigger is executed until the gold value becomes 0.

```
SET TRIGGER TRACE ON;

UPDATE participant SET gold =15 WHERE nation_code = 'KOR' AND host_year = 1988;
TRACE: Evaluating condition for trigger "loop".
TRACE: Executing action for trigger "loop".
```

Limiting Nested TRIGGER

Description

With the **MAXIMUM DEPTH** keyword of the **SET TRIGGER** statement, you can limit the number of triggers to be initiated at each step. By doing so, you can prevent a recursively called trigger from falling into an infinite loop.

Syntax

```
SET TRIGGER [ MAXIMUM ] DEPTH count [ ; ]
```

```
count:
• unsigned_integer_literal
• INFINITE
```

- *count* :
  - *unsigned_integer_literal* : A positive integer value that specifies the number of times that a trigger can recursively start another trigger or itself. If the number of triggers reaches the maximum depth, the database request stops(aborts) and the transaction is marked as invalid. The specified **DEPTH** applies to all other triggers except for the current session.
  - **INFINITE** : Removes the limit to the number of times specified.

Example

The following is an example of setting the maximum number of times of recursive trigger calling to 10. This applies to all triggers that start subsequently. In this example, the gold column value is updated to 15, so the trigger is called 16 times in total. This exceeds the currently set maximum depth and the following error message occurs.

```
SET TRIGGER MAXIMUM DEPTH 10;

csql> UPDATE participant SET gold = 15 WHERE nation_code = 'KOR' AND host_year = 1988;
csql> ;x

In line 3, column 2,
ERROR: Maximum trigger depth 10 exceeded at trigger "loop_tgr".
```

TRIGGER Example

Description

This section covers trigger definitions in the demo database. The triggers created in the demodb database are not complex, but use most of the features available in CUBRID. If you want to maintain the original state of the demodb database when testing such triggers, you must perform a rollback after changes are made to the data.

Triggers created by the user in the own database can be as powerful as applications created by the user.

Example 1

The following trigger created in the participant table rejects an update to the medal column (gold, silver, bronze) if a given value is smaller than 0. The evaluation time must be **BEFORE** because a correlation name new is used in the trigger condition. Although not described, the action time of this trigger is also **BEFORE**.

```
CREATE TRIGGER medal_trigger
BEFORE UPDATE ON participant
IF new.gold < 0 OR new.silver < 0 OR new.bronze < 0
EXECUTE REJECT;
```

The medal_trigger trigger starts when the number of gold medals of the country whose nation code is 'BLA' is updated. Since a negative value is not permitted for the number of gold medals as shown above, this update is not allowed.

```
UPDATE participant
SET gold = -10
WHERE nation_code = 'BLA';
```

Example 2

The following trigger has the same condition as the one above except that **STATUS INACTIVE** is added. If the STATUS statement is omitted, the default value is **ACTIVE**. You can change the status to **INACTIVE** by using the **ALTER TRIGGER** statement.

You can specify whether or not to execute the trigger depending on the **STATUS** value.

```
CREATE TRIGGER medal_trig
STATUS ACTIVE
BEFORE UPDATE ON participant
IF new.gold < 0 OR new.silver < 0 OR new.bronze < 0
EXECUTE REJECT;
```

```
ALTER TRIGGER medal_trig
STATUS INACTIVE;
```

Example 3

The following trigger shows how integrity constraint is enforced when a transaction is committed. This example is different from the previous ones, in that one trigger can have specific conditions for multiple tables.

```
CREATE TRIGGER check_null_first
BEFORE COMMIT
IF 0 < (SELECT count(*) FROM athlete WHERE gender IS NULL)
OR 0 < (SELECT count(*) FROM game WHERE nation_code IS NULL)
EXECUTE REJECT;
```

Example 4

The following trigger delays the update integrity constraint check for the record table until the transaction is committed. Since the **DEFERRED** keyword is given as the event time, the trigger is not executed at the time.

```
CREATE TRIGGER deferred_check_on_record
DEFERRED UPDATE ON record
IF obj.score = '100'
EXECUTE INVALIDATE TRANSACTION;
```

Once completed, the update in the record table can be confirmed at the last point (commit or rollback) of the current transaction. The correlation name old cannot be used in the conditional clause of the trigger where **DEFERRED UPDATE** is used. Therefore, you cannot create a trigger as the following.

```
CREATE CLASS foo (n int);
CREATE TRIGGER foo_trigger
    DEFERRED UPDATE ON foo
    IF old.n = 100
    EXECUTE PRINT 'foo_trigger';
```

If you try to create a trigger as shown above, an error message is displayed and the trigger fails.

```
ERROR: Error compiling condition for 'foo_trigger' : old.n is not defined
```

The correlation name **old** can be used only with **AFTER**.

Java Stored Function/Procedure

Overview

Stored functions and procedures are used to implement complicated program logic that is not possible with SQL. They allow users to manipulate data more easily. Stored functions/procedures are blocks of code that have a flow of commands for data manipulation and are easy to manipulate and administer.

CUBRID supports to develop stored functions and procedures in Java. Java stored functions/procedures are executed on the JVM (Java Virtual Machine) hosted by CUBRID.

You can call Java stored functions/procedures from SQL statements or from Java applications using JDBC.

The advantages of using Java stored functions/procedures are as follows:

- **Productivity and usability** : Java stored functions/procedures, once created, can be reused anytime. They can be called from SQL statements or from Java applications using JDBC.
- **Excellent interoperability and portability** : Java stored functions/procedures use the Java Virtual Machine. Therefore, they can be used on any system where the Java Virtual Machine is available.

Environment Configuration for Java Stored Function/Procedure

To use Java-stored functions/procedures in CUBRID, you must have JRE (Java Runtime Environment) 1.6 or better installed in the environment where the CUBRID server is installed. You can download JRE from the Developer Resources for Java Technology (http://java.sun.com).

If the java_stored_procedure parameter in the CUBRID configuration file (cubrid.conf) is set to yes, CUBRID 64-bit needs a 64-bit Java Runtime Environment, and CUBRID 32-bit needs a 32-bit Java Runtime Environment. For example, when you run CUBRID 64-bit in the system in which a 32-bit JAVA Runtime Environment is installed, the following error may occur.

```
% cubrid server start demodb

This may take a long time depending on the amount of recovery works to do.
WARNING: Java VM library is not found :
/usr/java/jdk1.6.0_15/jre/lib/amd64/server/libjvm.so: cannot open shared object file: No
such file or directory.
Consequently, calling java stored procedure is not allowed
```

Execute the following command to check the JRE version if you have it already installed in the system.

```
% java -version Java(TM) SE Runtime Environment (build 1.6.0_05-b13)
Java HotSpot(TM) 64-Bit Server VM (build 10.0-b19, mixed mode)
```

Windows Environment

For Windows, CUBRID loads the **jvm.dll** file to run the Java Virtual Machine. CUBRID first locates the **jvm.dll** file from the **PATH** environment variable and then loads it. If it cannot find the file, it uses the Java runtime information registered in the system registry.

You can configure the **JAVA_HOME** environment variable and add the directory in which the Java executable file is located to **Path**, by executing the command as follows: For information on configuring environment variables using GUI, see Setting up the JDBC Environment.

- An example of installing 64 Bit JDK 1.6 and configuring the environment variables

```
% set JAVA_HOME=C:\jdk1.6.0
% set PATH=%PATH%;%JAVA_HOME%\jre\bin\server
```

- An example of installing 32 Bit JDK 1.6 and configuring the environment variables

```
% set JAVA_HOME=C:\jdk1.6.0
% set PATH=%PATH%;%JAVA_HOME%\jre\bin\client
```

To use other vendor's implementation instead of Sun's Java Virtual Machine, add the path of the **jvm.dll** file to the **PATH** variable during the installation.

Linux/UNIX Environment

For Linux/UNIX environment, CUBRID loads the **libjvm.so** file to run the Java Virtual Machine. CUBRID first locates the **libjvm.so** file from the **LD_LIBRARY_PATH** environment variable and then loads it. If it cannot find the file, it uses the **JAVA_HOME** environment variable. For Linux, glibc version 2.3.4 or higher is supported. The following is an example of configuring the Linux environment variable (e.g., **.profile**, **.cshrc**, **.bashrc**, **.bash_profile**, etc.).

- An example of installing 64 Bit JDK 1.6 and configuring the environment variables in a bash shell

```
% JAVA_HOME=/usr/java/jdk1.6.0_10
%
LD_LIBRARY_PATH=$JAVA_HOME/jre/lib/amd64:$JAVA_HOME/jre/lib/amd64/server:$LD_LIBRARY_PA
TH
% export JAVA_HOME
% export LD_LIBRARY_PATH
```

- An example of installing 32 Bit JDK 1.6 and configuring the environment variables in a bash shell

```
% JAVA_HOME=/usr/java/jdk1.6.0_10
%
LD_LIBRARY_PATH=$JAVA_HOME/jre/lib/i386/:$JAVA_HOME/jre/lib/i386/client:$LD_LIBRARY_PAT
H
% export JAVA_HOME
% export LD_LIBRARY_PATH
```

- An example of installing 64 Bit JDK 1.6 and configuring the environment variables in a csh

```
% setenv JAVA_HOME /usr/java/jdk1.6.0_10
% setenv LD_LIBRARY_PATH
$JAVA_HOME/jre/lib/amd64:$JAVA_HOME/jre/lib/amd64/server:$LD_LIBRARY_PATH
% set path=($path $JAVA_HOME/bin .)
```

- An example of installing 32 Bit JDK 1.6 and configuring the environment variables in a csh shell

```
% setenv JAVA_HOME /usr/java/jdk1.6.0_10
% setenv LD_LIBRARY_PATH
$JAVA_HOME/jre/lib/i386:$JAVA_HOME/jre/lib/i386/client:$LD_LIBRARY_PATH
% set path=($path $JAVA_HOME/bin .)
```

To use other vendor's implementation instead of Sun's Java Virtual Machine, add the path of the JVM (**libjvm.so**) to the library path during the installation.

The path of the **libjvm.so** file can be different depending on the platform. For example, the path is the **$JAVA_HOME/jre/lib/sparc** directory in a SUN Sparc machine.

How to Write Java Stored Function/Procedure

Steps to write a Java stored function/procedure are as follows:

- Check the cubrid.conf file
- Write and compile the Java source code
- Load the complied Java class into CUBRID
- Publish the loaded Java class
- Call the Java stored function/procedure

Check the cubrid.conf file

By default, the **java_stored_procedure** is set to **no** in the **cubrid.conf** file. To use a Java stored function/procedure, this value must be changed to **yes**. For more information on this value, see Other Parameters in Database Server Configuration.

Write and compile the Java source code

Compile the SpCubrid.java file as follows:

```
public class SpCubrid{
      public static String HelloCubrid() {
              return "Hello, Cubrid !!";
      }
      public static int SpInt(int i) {
              return i + 1;
      }
      public static void outTest(String[] o) {
              o[0] = "Hello, CUBRID";
      }
}

%javac SpCubrid.java
```

Here, the Java class method must be public static.

Load the compiled Java class into CUBRID

Load the compiled Java class into CUBRID.

```
% loadjava demodb SpCubrid.class
```

Publish the loaded Java class

Create a CUBRID stored function and publish the Java class as shown below.

```
csql> create function hello() return string
csql> as language java
csql> name 'SpCubrid.HelloCubrid() return java.lang.String';
csql> ;xrun
```

Call the Java stored function/procedure

Call the published Java stored function as follows:

```
csql> call hello() into :Hello;
csql> ;xrun

=== < Result of CALL Command in Line 1> ===


 Result
======================
'Hello, Cubrid !!'
```

Using Server-side Internal JDBC Driver

To access the database from a Java stored function/procedure, you must use the server-side JDBC driver. As Java stored functions/procedures are executed within the database, there is no need to make the connection to the server-side JDBC driver again. To acquire a connection to the database using the server-side JDBC driver, you can either use "**jdbc:default:connection:**" as the URL for JDBC connection, or call the **getDefaultConnection**() method of the **cubrid.jdbc.driver.CUBRIDDriver** class.

```
Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
Connection conn =    DriverManager.getConnection("jdbc:default:connection:");
```

or

```
cubrid.jdbc.driver.CUBRIDDriver.getDefaultConnection();
```

If you connect to the database using the JDBC driver as shown above, the transaction in the Java stored function/procedure is ignored. That is, database operations executed in the Java stored function/procedure belong to the transaction that called the Java stored function/procedure. In the following example, **conn.commit()** method of the **Athlete** class is ignored.

```
import java.sql.*;
public class Athlete{
    public static void Athlete(String name, String gender, String nation_code, String
event) throws SQLException{
        String sql="INSERT INTO ATHLETE(NAME, GENDER, NATION_CODE, EVENT)" + "VALUES
(?, ?, ?, ?)";
        try{
            Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
            Connection conn = DriverManager.getConnection("jdbc:default:connection:");
            PreparedStatement pstmt = conn.prepareStatement(sql);

            pstmt.setString(1, name);
            pstmt.setString(2, gender);
            pstmt.setString(3, nation_code);
            pstmt.setString(4, event);;
            pstmt.executeUpdate();

            pstmt.close();
            conn.commit();
            conn.close();
        } catch (Exception e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Connecting to Other Database

You can connect to another outside database instead of the currently connected one even when the server-side JDBC driver is being used. Acquiring a connection to an outside database is not different from a generic JDBC connection. For more information, see JDBC API.

If you connect to other databases, the connection to the CUBRID database does not terminate automatically even when the execution of the Java method ends. Therefore, the connection must be explicitly closed so that the result of transaction operations such as **COMMIT** or **ROLLBACK** will be reflected in the database. That is, a separate transaction will be performed because the database that called the Java stored function/procedure is different from the one where the actual connection is made.

```
import java.sql.*;
public class SelectData {
  public static void SearchSubway(String[] args) throws Exception {
Connection conn = null;
  Statement stmt = null;
  ResultSet rs = null;
  try {
    Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
    conn =
DriverManager.getConnection("jdbc:CUBRID:localhost:33000:demodb:::","","");
    String sql = "select line_id, line from line";
    stmt = conn.createStatement();
    rs = stmt.executeQuery(sql);
    while(rs.next()) {
      int host_year = rs.getString("host_year");
      String host_nation = rs.getString("host_nation");
      System.out.println("Host Year ==> " + host_year);
      System.out.println(" Host Nation==> " + host_nation);
      System.out.println("\n=========\n");
    }
    rs.close();
    stmt.close();
    conn.close();
    } catch ( SQLException e ) {
         System.err.println(e.getMessage());
    } catch ( Exception e ) {
         System.err.println(e.getMessage());
    } finally {
          if ( conn != null ) conn.close();
    }
  }
}
```

When the Java stored function/procedure being executed should run only on JVM located in the database server, you can check where it is running by calling System.getProperty ("cubrid.server.version") from the Java program source. The result value is the database version if it is called from the database; otherwise, it is **NULL**.

loadjava Utility

Description

To load a compiled Java or JAR (Java Archive) file into CUBRID, use the **loadjava** utility. If you load a Java *.class or *.jar file using the **loadjava** utility, the file is moved to the specified database path.

Syntax

```
loadjava <option> database-name java-class-file
```

- *database-name* : The name of the database where the Java file is to be loaded.
- *java-class-file* : The name of the Java class or jar file to be loaded.
- *<option>* :
  - **-y** : Automatically overwrites a class file with the same name, if any. The default value is **no**. If you load the file without specifying the **-y** option, you will be prompted to ask if you want to overwrite the class file with the same name (if any).

Loaded Java Class Publish

Overview

In CUBRID, it is required to publish Java classes to call Java methods from SQL statements or Java applications. You must publish Java classes by using call specifications because it is not known how a function in a class will be called by SQL statements or Java applications when Java classes are loaded.

Call Specifications

To use a Java stored function/procedure in CUBRID, you must write call specifications. With call specifications, Java function names, parameter types, return values and their types can be accessed by SQL statements or Java applications. To write call specifications, use **CREATE FUNCTION** or **CREATE PROCEDURE** statement. Java stored function/procedure names are not case sensitive. The maximum number of characters a Java stored function/procedure can have is 256. The maximum number of parameters a Java stored function/procedure can have is 64.

Syntax

```
CREATE {PROCEDURE procedure_name[(param[, param]...] | FUNCTION function_name[(param[,
param]...] RETURN sql_type }
{IS | AS} LANGUAGE JAVA
NAME 'method_fullname (java_type_fullname[,java_type_fullname]... [return
java_type_fullname]';

parameter_name [IN|OUT|IN OUT|INOUT] sql_type
   (default IN)
```

If the parameter of a Java stored function/procedure is set to **OUT**, it will be passed as a one-dimensional array whose length is 1. Therefore, a Java method must store its value to pass in the first space of the array.

Example

```
CREATE FUNCTION Hello() RETURN VARCHAR
AS LANGUAGE JAVA
NAME 'SpCubrid.HelloCubrid() return java.lang.String';
CREATE FUNCTION Sp_int(i int) RETURN int
AS LANGUAGE JAVA
NAME 'SpCubrid.SpInt(int) return int';

CREATE PROCEDURE Phone_Info(name varchar, phoneno varchar)
```

```
AS LANGUAGE JAVA
NAME 'PhoneNumber.Phone(java.lang.String, java.lang.String)';
```

When a Java stored function/procedure is published, it is not checked whether the return definition of the Java stored function/procedure coincides with the one in the declaration of the Java file. Therefore, the Java stored function/procedure follows the *sql_type* return definition provided at the time of registration. The return definition in the declaration is significant only as user-defined information.

Data Type Mapping

In call specifications, the data types SQL must correspond to the data types of Java parameter and return value. The following table shows SQL/Java data types allowed in CUBRID.

Data Type Mapping

SQL Type	Java Type
CHAR, VARCHAR	java.lang.String, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.lang.Byte, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.math.BigDecimal, byte, short, int, long, float, double
NUMERIC, SHORT, INT, FLOAT, DOUBEL, CURRENCY	java.lang.Byte, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.math.BigDecimal, java.lang.String, byte, short, int, long, float, double
DATE, TIME, TIMESTAMP	java.sql.Date, java.sql.Time, java.sql.Timestamp, java.lang.String
SET, MULTISET, SEQUENCE	java.lang.Object[], java primitive type array, java.lang.Integer[] ...
OBJECT	cubrid.sql.CUBRIDOID
CURSOR	cubrid.jdbc.driver.CUBRIDResultSet

Checking the Published Java Stored Function/Procedure Information

You can check the information on the published Java stored function/procedure The **db_stored_procedure** system virtual table provides virtual table and the **db_stored_procedure_args** system virtual table. The **db_stored_procedure** system virtual table provides the information on stored names and types, return types, number of parameters, Java class specifications, and the owner. The **db_stored_procedure_args** system virtual table provides the information on parameters used in the stored function/procedure.

```
csql> select * from db_stored_procedure
csql> ;xrun

=== <Result of SELECT Command in Line 2> ===

sp_name     sp_type   return_type    arg_count
sp_name               sp_type               return_type             arg_count  lang
target                owner
================================================================================
'hello'               'FUNCTION'            'STRING'                        0  'JAVA''SpCu
brid.HelloCubrid() return java.lang.String'  'DBA'

'sp_int'              'FUNCTION'            'INTEGER'                       1  'JAVA''SpCu
brid.SpInt(int) return int'  'DBA'

'athlete_add'         'PROCEDURE'           'void'                          4  'JAVA''Athl
ete.Athlete(java.lang.String, java.lang.String, java.lang.String,
java.lang.String)'  'DBA'

3 rows selected.

Current transaction has been committed.
1 command(s) successfully processed.
```

```
csql> select * from db_stored_procedure_args
csql> xrun

=== < Result of SELECT Command in Line 1> ===

sp_name   index_of  arg_name  data_type      mode

=================================================
 'sp_int'                        0  'i'                   'INTEGER'             'IN'
 'athlete_add'                   0  'name'                'STRING'              'IN'
 'athlete_add'                   1  'gender'              'STRING'              'IN'
 'athlete_add'                   2  'nation_code'         'STRING'              'IN'
 'athlete_add'                   3  'event'               'STRING'              'IN'

5 rows selected.

Current transaction has been committed.

1 command(s) successfully processed.
```

Deleting Java Stored Functions/Procedures

You can delete published Java stored functions/procedures in CUBRID. To delete a Java function/procedure, use the **DROP FUNCTION** *function_name* or **DROP PROCEDURE** *procedure_name* statement. Also, you can delete multiple Java stored functions/procedures at a time with several *function_names* or *procedure_names* separated by a comma (,).

A Java stored function/procedure can be deleted only by the user who published it or by DBA members. For example, if a **PUBLIC** user published the 'sp_int' Java stored function, only the **PUBLIC** or **DBA** members can delete it.

```
drop function hello[, sp_int]
drop procedure Athlete_Add
```

Java Stored Function/Procedure Call

Using CALL Statement

You can call the Java stored functions/procedures by using a **CALL** statement, from SQL statements or Java applications.

The following shows how to call them by using the **CALL** statement. The name of the Java stored function/procedure called from a **CALL** statement is not case sensitive.

Syntax

```
CALL {procedure_name ([param[, param]...) | function_name ([param[, param]...)
INTO :host_variable
param {literal | :host_variable}
```

Example

```
call Hello() into :HELLO;
call Sp_int(3) into :i;
call phone_info('Tom','016-111-1111');
```

In CUBRID, the Java functions/procedures are called by using the same **CALL** statement. Therefore, the **CALL** statement is processed as follows:

- It is processed as a method if there is a target class in the **CALL** statement.
- If there is no target class in the **CALL** statement, it is checked whether a Java stored function/procedure is executed or not; a Java stored function/procedure will be executed if one exists.
- If no Java stored function/procedure exists in step 2 above, it is checked whether a method is executed or not; a method will be executed if one with the same name exists.

The following error occurs if you call a Java stored function/procedure that does not exist.

```
csql> call deposit()
csql> ;xrun
```

```
In the command from line 1,
ERROR: Stored procedure/function 'deposit' is not exist.
0 command(s) successfully processed.
csql> call deposit('Tom', 3000000)
csql> ;xrun
In line 1, column 6,
ERROR: Methods require an object as their target.
0 command(s) successfully processed.
```

If there is no argument in the **CALL** statement, a message "ERROR: Stored procedure/function 'deposit' is not exist." appears because it can be distinguished from a method. However, if there is an argument in the **CALL** statement, a message "ERROR: Methods require an object as their target." appears because it cannot be distinguished from a method.

If the **CALL** statement is nested within another **CALL** statement calling a Java stored function/procedure, or if a subquery is used in calling the Java function/procedure, the **CALL** statement is not executed.

```
call phone_info('Tom', call sp_int(999));
call phone_info((select * from Phone where id='Tom'));
```

If an exception occurs during the execution of a Java stored function/procedure, the exception is logged and stored in the *dbname*_**java.log** file. To display the exception on the screen, change a handler value of the **$CUBRID/java/logging.properties** file to " java.lang.logging.ConsoleHandler." Then, the exception details are displayed on the screen.

Calling from SQL Statement

You can call a Java stored function from a SQL statement as shown below.

```
select Hello() from db_root;
select sp_int(99) from db_root;
```

You can use a host variable for the IN/OUT data type when you call a Java stored function/procedure as follows:

```
SELECT 'Hi' INTO :out_data FROM db_root;
CALL test_out(:out_data);
SELECT :out_data FROM db_root;
```

The first clause calls a Java stored procedure in out mode by using a parameter variable; the second is a query clause retrieving the assigned host variable out_data.

Calling from Java Application

To call a Java stored function/procedure from a Java application, use a **CallableStatement** object.

Create a phone class in the CUBRID database.

```
create class phone(
     name varchar(20),
     phoneno varchar(20)
)
```

Compile the following PhoneNumber.java file, load the Java class file into CUBRID, and publish it.

```
import java.sql.*;
import java.io.*;
public class PhoneNumber{    public static void Phone(String name, String phoneno) throws
Exception{
        String sql="INSERT INTO PHONE(NAME, PHONENO)"+ "VALUES (?, ?)";
        try{
            Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
            Connection conn = DriverManager.getConnection("jdbc:default:connection:");
            PreparedStatement pstmt = conn.prepareStatement(sql);

            pstmt.setString(1, name);
            pstmt.setString(2, phoneno);
            pstmt.executeUpdate();
            pstmt.close();
            conn.commit();
            conn.close();
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
```

```
    }
}
create PROCEDURE phone_info(name varchar, phoneno varchar)
as language java
name 'PhoneNumber.Phone(java.lang.String, java.lang.String)';
```

Create and run the following Java application.

```
import java.sql.*;
public class StoredJDBC{
    public static void main(){
        Connection conn = null;
        Statement stmt= null;
        int result;
        int i;
        try{
 Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
            conn =
DriverManager.getConnection("jdbc:CUBRID:localhost:33000:subway:::","","");
            CallableStatement cs;
            cs = conn.prepareCall("call PHONE_INFO(?, ?)");
            cs.setString(1, "Jane");
            cs.setString(2, "010-1111-1111");
            cs.executeUpdate();
            conn.commit();
            cs.close();
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }}
```

Retrieve the phone class after executing the program above; the following result would be displayed.

```
csql> select * from phone
csql> ;xrun
=== <Result of SELECT Command in Line 1>===
name                  phoneno
============================================
    'Jane'                '010-111-1111'
1 rows selected.
Current transaction has been committed.
1 command(s) successfully processed.
```

Note

Return Value of Java Stored Function/Procedure and Precision Type on IN/OUT

To limit the return value of Java stored function/procedure and precision type on IN/OUT, CUBRID processes as follows:

Checks the sql_type of the Java stored function/procedure.

Passes the value returned by Java to the database with only the type converted if necessary, ignoring the number of digits defined during creating the Java stored function/procedure. In principle, the user manipulates the passed data directly in the database.

Take a look at the following **typestring**() Java stored function.

```
public class JavaSP1{
    public static String typestring(){
        String temp = " ";
        for(int i=0 i< 1 i++)
            temp = temp + "1234567890";
        return temp;
}
}

create function typestring() return char(5)
as language java
name 'JavaSP1.typestring() return java.lang.String';
```

```
csql> call typestring()
csql> ;xrun

=== < Result of CALL Command in Line 1> ===

  Result
======================
  ' 1234567890'

Current transaction has been committed.

1 command(s) successfully processed.
```

Returning java.sql.ResultSet in Java Stored Procedure

In CUBRID, you must use **CURSOR** as the data type when you declare a Java stored function/procedure that returns a **java.sql.ResultSet**.

```
create function rset() return cursor
as language java
name 'JavaSP2.TResultSet() return java.sql.ResultSet'
```

Before the Java file returns **java.sql.ResultSet**, it is required to cast to the **CUBRIDResultSet** class and then to call the **setReturnable**() method.

```
public static class JavaSP2 {
public static ResultSet TResultSet(){
    try{
            Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
    Connection conn = DriverManager.getConnection("jdbc:default:connection:");
    ((CUBRIDConnection)con).setCharset("euc_kr");
    String sql = "select * from station";
    Statement stmt=con.createStatement();
    ResultSet rs = stmt.executeQuery(sql);
    ((CUBRIDResultSet)rs).setReturnable();
    return rs;
    } catch (Exception e) {
              e.printStackTrace();
    }
    return null;
  }
}
```

In the calling block, you must set the OUT argument with **Types.JAVA_OBJECT**, get the argument to the **getObject**() function, and then cast it to the **java.sql.ResultSet** type before you use it. In addition, the **java.sql.ResultSet** is only available to use in **CallableStatement** of JDBC.

```
import java.sql.*;
public class TestResultSet{
  public static void main(String[] args) {
    Connnection conn = null;
    Statement stmt= null;
    int result;
    int i;

    try{
           Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
    conn = DriverManager.getConnection("jdbc:CUBRID:localhost:33000:demodb:::","","");

    CallableStatement cstmt = con.prepareCall("?=CALL rset()");
    cstmt.registerOutParameter(1, Types.JAVA_OBJECT);
    cstmt.execute();
    ResultSet rs = (ResultSet) cstmt.getObject(1);
    while(rs.next()) {
      System.out.println(rs.getString(1));
    }
      rs.close();
     } catch (Exception e) {
             e.printStackTrace();
    }
}
```

You cannot use the **ResultSet** as an input argument. If you pass it to an IN argument, an error occurs. An error also occurs when calling a function that returns **ResultSet** in a non-Java environment.

IN/OUT of Set Type in Java Stored Function/Procedure

If the set type of the Java stored function/procedure in CUBRID is IN OUT, the value of the argument changed in Java must be applied to IN OUT. When the set type is passed to the OUT argument, it must be passed as a two-dimensional array.

```
Create procedure setoid(x in out set, z object)
as language java name
'SetOIDTest.SetOID(cubrid.sql.CUBRIDOID[][], cubrid.sql.CUBRIDOID';

public static void SetOID(cubrid.sql.CUBRID[][] set, cubrid.sql.CUBRIDOID aoid){
  Connection conn=null;
  Statement stmt=null;
  String ret="";
  Vector v = new Vector();
  cubrid.sql.CUBRIDOID[] set1 = set[0];
  try {
    if(set1!=null) {
      int len = set1.length;
      int i = 0;
      for (i=0 i< len i++)
        v.add(set1[i]);
    }
  v.add(aoid);
  set[0]=(cubrid.sql.CUBRIDOID[]) v.toArray(new cubrid.sql.CUBRIDOID[]{});
  } catch(Exception e) {
    e.printStackTrace();
    System.err.pirntln("SQLException:"+e.getMessage());
  }
}
```

Using OID in Java Stored Function/Procedure

In case of using the OID type value for IN/OUT in CUBRID, use the value passed from the server.

```
create procedure tOID(i inout object, q string)
as language java
name 'OIDtest.tOID(cubrid.sql.CUBRIDOID[], java.lang.String)';

public static void tOID(CUBRIDOID[] oid, String query)
{
  Connection conn=null;
  Statement stmt=null;
  String ret="";

  try {
    Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
    conn=DriverManager.getConnection("jdbc:default:connection:");

    conn.setAutoCommit(false);
    stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(query);
    System.out.println("query:"+ query);
    while(rs.next()) {
      oid[0]=(CUBRIDOID)rs.getObject(1);
      System.out.println("oid:"+oid[0].getTableName());
    }
    stmt.close();
    conn.close();
  } catch (SQLException e) {
    e.printStackTrace();
    System.err.println("SQLException:"+e.getMessage());
  } catch (Exception e) {
    e.printStackTrace();
    system.err.println("Exception:"+ e.getMessage());
  }
}
```

METHOD

Overview

This chapter describes methods (software routines) that extend or customize the features of the CUBRID database system.

The methods are written in C and called by the **CALL** or **EVALUATE** statement. A method program is loaded and linked with the application currently running by the dynamic loader when the method is called. The return value created as a result of the method execution is passed to the caller.

This chapter describes the following topics:

- Method Types
- Calling a Method

METHOD Type

The CSQL language supports the following two types of methods: class and instance methods.

- The **class method** is a method called by a class object. It is usually used to create a new class instance or to initialize it. It is also used to access or update class attributes.
- The **instance method** is a method called by a class instance. It is used more often than the class method because most operations are executed in the instance. For example, an instance method can be written to calculate or update the instance attribute. This method can be called from any instance of the class in which the method is defined or of the subclass that inherits the method.

The method inheritance rules are similar to those of the attribute inheritance. The subclass inherits classes and instance methods from the super class. The subclass has only the name of a class or instance method definition inherited from the super class.

The rules for resolving method name conflicts are same as those for attribute name conflicts. For more information about attribute/method inheritance conflicts, see Overview in Class Conflict Resolution.

Calling METHOD

Overview

Methods are executed by the **CALL** or **EVALUATE** statement, and their results are returned the same way as the query results.

These statements are also used to call a method from a query. (The **CALL** or **EVALUATE** keyword is omitted.)

CALL Statement

Description

In CUBRID, the **CALL** statement is used to call a method defined in the database. Both table and record methods can be called by the **CALL** statement.

Syntax

```
CALL method_call [ ; ]
method_call :
• method_name ( [ arg_value [ {, arg_value }_ ] ] ) ON call_target [ to_variable ]
• method_name ( call_target [, arg_value [ {, arg_value }_ ] ] ) [ to_variable ]
arg_value :
• any CSQL expression
call_target :
• an object-valued expression
```

```
to_variable :
• INTO variable
• TO variable
```

- The *method_name* is either the method name defined in the table or the system-defined method name provided with CUBRID. A method requires one or more parameters. If there is no parameter for the method, a set of blank parentheses must be used.
- *call_target* can use an object-valued expression that contains a class name, a variable, another method call (which returns an object). To call a class method for a class object, you must place the **CLASS** keyword before the *call_target*. In this case, the table name must be the name of the class where the table method is defined. To call a record method, you must specify the expression representing the record object. You can optionally store the value returned by the table or record method in the *to_variable*. This returned variable value can be used in the **CALL** statement just like the *call_target* or *arg_value* parameter.
- Calling nested methods is possible when other *method_call* is the *call_target* of the method or given as one of the *arg_value* parameters.

EVALUATE Statement

Description

The **EVALUATE** statement is also used to call a method defined in the database.

In the **EVALUATE** statement, a method call is a *term* in an expression. If the method returns a constant value, another constant (or a method returning a constant) can also be a term in an expression. Both class and instance methods can be called by the **EVALUATE** statement.

Syntax

```
EVALUATE expression [ ; ]

expression:
• [ + | - ] term [ { + | - | * | / } term ]

term:
• method_call

method_call :
• method_name ( call_target [, arg_value [ {, arg_value }_ ] ] ) [ to_variable ]
            method_name ( [ arg_value [ {, arg_value }_ ] ] )
            ON call_target [ to_variable ]
arg_value :
• literal
• variable
• expression

call_target :
• CLASS class_name
• variable
• expression
• method_call

to_variable :
• INTO variable
• TO variable
```

In the **EVALUATE** statement, the target argument for the specified method is represented in the parentheses following the *method_name*. The target can be the first field in the list, followed by method arguments. If the method executed is a class method, the **CLASS** keyword must precede the target class as the first field in the list. If only the method arguments are included in the parentheses, the *call_target* should be in the **ON** clause.

The **EVALUATE** statement also supports nested method calls by allowing one method call to be expressed as the target or the argument of another method. In these types of expressions, the result of the inner method is used to determine that of the outer method.

Partitioning

What is Partitioning?

Partitioning is a method by which a table is divided into multiple independent logical units. Each logical unit used in partitioning is called a partition. Partitioning can enhance manageability, performance and availability. Some advantages of partitioning are as follows:

- Improved management of large capacity tables
- Improved performance by narrowing the range of access when retrieving data
- Improved performance and decreased physical loads by distributing disk I/O
- Decreased possibility of data corruption and improved availability by partitioning a table into multiple chunks
- Optimized storage cost

Three types of partitioning methods are supported by CUBRID: range partitioning, hash partitioning, and list partitioning.

The maximum number of partitions cannot exceed 1,024. Each partition of a table is created as its subtable. The subtables created by the partitioning process cannot be altered or deleted by users. The name of the subtable is stored in the system table in a 'class_name__p__partition_name' format. Database users can check the partitioning information in the db_class and db_partition virtual tables. They can also check the information by using the ;sc <table name> command in the CUBRID Manager or the CSQL Interpreter.

Range Partitioning

Range Partitioning Definition

Description

You can define a range partition by using the **PARTITION BY RANGE** clause.

Syntax

```
CREATE TABLE(
...
)
PARTITION BY RANGE ( <partition_expression> ) (
PARTITION <partition_name> VALUES LESS THAN ( <range_value> ),
PARTITION <partition_name> VALUES LESS THAN ( <range_value> ) ),
... )
)
```

- *partition_expression* : Specifies the partition expression. The expression can be specified by the name of the column to be partitioned or by a function. For more information of the data types and functions available, see Data Types Available for Partition Expression.
- *partition_name* : Specifies the partition name.
- *range_value* : Specifies the partition-by value.

Example 1

The following is an example of creating the participant2 table with the participating countries, and inserting data that partitions the years into before and after the 2000 Olympic Games. When inserting data, the countries that participated in the 1988 and 1996 Olympic Games are stored in before_2000; the rest of them are stored in before_2008.

```
CREATE TABLE participant2 (host_year INT, nation CHAR(3), gold INT, silver INT, bronze INT)
PARTITION BY RANGE (host_year)
(PARTITION before_2000 VALUES LESS THAN (2000),
PARTITION before_2008 VALUES LESS THAN (2008) );

INSERT INTO participant2 VALUES (1988, 'NZL', 3, 2, 8);
INSERT INTO participant2 VALUES (1988, 'CAN', 3, 2, 5);
INSERT INTO participant2 VALUES (1996, 'KOR', 7, 15, 5);
INSERT INTO participant2 VALUES (2000, 'RUS', 32, 28, 28);
INSERT INTO participant2 VALUES (2004, 'JPN', 16, 9, 12);
```

Example 2

As shown below, the partition key value in a range partition is **NULL**, the data are stored in the first partition.

```
INSERT INTO participant2 VALUES(NULL, 'AAA', 0, 0, 0);
```

Caution

- The maximum number of partitions possible for a given table is 1024.
- If the partition key value is **NULL**, the data is stored in the first partition (see Example 2).

Range Partitioning Redefinition

Description

You can redefine a partition by using the **REORGANIZE PARTITION** clause of the **ALTER** statement. By redefining partitions, you can combine multiple partitions into one or divide one into multiple.

Syntax

```
ALTER {TABLE | CLASS} <table_name>
REORGANIZE PARTITION
<alter partition name comma list>
INTO ( <partition definition comma list> )

partition definition comma list:
PARTITION <partition_name> VALUES LESS THAN ( <range_value> ),....
```

- *table_name* : Specifies the name of the table to be redefined.
- *alter partition name comma list* : Specifies the partition to be redefined. Multiple partitions are separated by commas (,).
- *partition definition comma list* : Specifies the redefined partitions. Multiple partitions are separated by commas (,).

Example 1

The following is an example of repartitioning the before_2000 partition into the before_1996 and before_2000 partitions.

```
CREATE TABLE participant2 ( host_year INT, nation CHAR(3), gold INT, silver INT, bronze
INT)
PARTITION BY RANGE (host_year)
( PARTITION before_2000 VALUES LESS THAN (2000),
PARTITION before_2008 VALUES LESS THAN (2008) );

ALTER TABLE participant2 REORGANIZE PARTITION before_2000 INTO (
PARTITION before_1996 VALUES LESS THAN (1996),
PARTITION before_2000 VALUES LESS THAN (2000)
);
```

Example 2

The following is an example of combining two partitions redefined in Example 1 back into a single before_2000 partition.

```
ALTER TABLE participant2 REORGANIZE PARTITION before_1996, before_2000 INTO
(PARTITION before_2000 VALUES LESS THAN (2000) );
```

Caution

- When redefining a range or list partition, duplicate ranges or values are not allowed.
- The **REORGANIZE PARTITION** clause cannot be used to change the partition table type. For example, a range partition cannot be changed to a hash partition, or vice versa.

- The maximum number of partitions cannot exceed 1,024. There must be at least one partition remaining after deleting partitions. In a range-partitioned table, only adjacent partitions can be redefined.

Adding Range Partitioning

Description

You can add range partitions by using the **ADD PARTITION** clause of the **ALTER** statement.

Syntax

```
ALTER {TABLE | CLASS} <table_name>
ADD PARTITION <partition definitions comma list>
partition definition comma list:
PARTITION <partition_name> VALUES LESS THAN ( <range_value> ),...
```

- *table_name* : Specifies the name of the table to which partitions are added.
- *partition definition comma list* : Specifies the partitions to be added. Multiple partitions are separated by commas (,).

Example

Currently, the partition before the 2008 Olympic Games is defined in the participant2 table. The following is an example of adding the before_2012 and before_2016 partitions; the former will store the information about the 2012 Olympic Games and the latter will store the information about the 2016 Olympic Games.

```
ALTER TABLE participant2 ADD PARTITION (
PARTITION before_2012 VALUES LESS THAN (2012),
PARTITION before_2016 VALUES LESS THAN MAXVALUE );
```

Caution

- When a range partition is added, only the partition by value greater than the existing partition value can be added. Therefore, as shown in the above example, if the maximum value is specified by **MAXVALUE**, no more partitions can be added (you can add partitions by changing the **MAXVALUE** value by redefining the partition).
- To add the partition by value smaller than the existing partition value, use the redefining partitions (see Range Partitioning Redefinition).

Dropping Range Partitioning

Description

You can drop a partition by using the **DROP PARTITION** clause of the **ALTER** statement.

Syntax

```
ALTER {TABLE | CLASS} <table_name>
DROP PARTITION <partition_name>
```

- *table_name* : Specifies the name of the partitioned table.
- *partition_name* : Specifies the name of the partition to be dropped.

Example

The following is an example of dropping the before_2000 partition in the participant2 table.

```
ALTER TABLE participant2 DROP PARTITION before_2000;
```

Caution

- When dropping a partitioned table, all stored data in the partition are also dropped.
- If you want to change the partitioning of a table without losing data, use the **ALTER TABLE**...**REORGANIZE PARTITION** statement (see Range Partitioning Redefinition).
- The number of rows deleted is not returned when a partition is dropped. If you want to delete the data, but want to maintain the table and partitions, use the **DELETE** statement.

Hash Partitioning

Hash Partitioning Definition

Description

You can define a hash partition by using the **PARTITION BY HASH** clause.

Syntax

```
CREATE TABLE (
...
)
( PATITION BY HASH ( <partition_expression> )
 PATITIONS ( <number_of_partitions> )
)
```

- *partition_expression* : Specifies a partition expression. The expression can be specified by the name of the column to be partitioned or by a function.
- *number_of_partitions* : Specifies the number of partitions.

Example 1

The following is an example of creating the nation2 table with country codes and country names, and defining 4 hash partitions based on code values. Only the number of partitions, not the name, is defined in hash partitioning; names such as p0 and p1 are assigned automatically.

```
CREATE TABLE nation2
( code CHAR(3),
name VARCHAR(50) )
PARTITION BY HASH ( code) PARTITIONS 4;
```

Example 2

The following is an example of inserting data to the hash partition created in the example 1. When a value is inserted into a hash partition, the partition to store the data is determined by the hash value of the partition key. If the partition key value is **NULL**, the data is stored in the first partition.

```
INSERT INTO nation2 VALUES ('KOR','Korea');
INSERT INTO nation2 VALUES ('USA','USA United States of America');
INSERT INTO nation2 VALUES ('FRA','France');
INSERT INTO nation2 VALUES ('DEN','Denmark');
INSERT INTO nation2 VALUES ('CHN','China');
INSERT INTO nation2 VALUES (NULL,'AAA');
```

Caution

- The maximum number of partitions cannot exceed 1,024.

Hash Partitioning Redefinition

Description

You can redefine a partition by using the **COALESCE PARTITION** clause of the **ALTER** statement. Instances are preserved if the hash partition is redefined.

Syntax

```
ALTER {TABLE | CLASS} <table_name>
COALESCE PARTITION <unsigned integer>
```

- *table_name* : Specifies the name of the table to be redefined.
- *unsigned integer* : Specifies the number of partitions to be deleted.

Example

The following is an example of decreasing the number of partitions in the nation2 table from 4 to 2.

```
ALTER TABLE nation2 COALESCE PARTITION 2;
```

Caution

- Decreasing the number of partitions is only available.
- To increase the number of partitions, use the **ALTER TABLE** ... **ADD PARTITION** statement as in range partitioning (see Adding Range Partitioning for more information).
- There must be at least one partition remaining after redefining partitions.

List Partitioning

List Partitioning Definition

Description

You can define a list partition by using the **PARTITION BY LIST** statement.

Syntax

```
CREATE TABLE(
...
)
PARTITION BY LIST ( <partition_expression> ) (
PARTITION <partition_name> VALUES IN ( <partition_value_list> ),
PARTITION <partition_name> VALUES IN ( <partition_value_ list>
,
...
);
```

- *partition_expression* : Specifies a partition expression. The expression can be specified by the name of the column to be partitioned or by a function. For more information on the data types and functions available, see Data Types Available for Partition Expression.
- *partition_name* : Specifies the partition name.
- *partition_value_list* : Specifies the list of the partition by values.

Example 1

The following is an example of creating the athlete2 table with athlete names and sport events, and defining list partitions based on event values.

```
CREATE TABLE athlete2( name VARCHAR(40), event VARCHAR(30) )
PARTITION BY LIST (event) (
PARTITION event1 VALUES IN ('Swimming', 'Athletics ),
PARTITION event2 VALUES IN ('Judo', 'Taekwondo','Boxing'),
PARTITION event3 VALUES IN ('Football', 'Basketball', 'Baseball')
);
```

Example 2

The following is an example of inserting data to the list partition created in the example 1. In the last query of the example 2, if you insert an argument that has not been specified in the partition expression of the example 1, data inserting fails.

```
INSERT INTO athlete2 VALUES ('Hwang Young-Cho', 'Athletics');
INSERT INTO athlete2 VALUES ('Lee Seung-Yuop', 'Baseball');
INSERT INTO athlete2 VALUES ('Moon Dae-Sung','Taekwondo');
INSERT INTO athlete2 VALUES ('Cho In-Chul', 'Judo');
INSERT INTO athlete2 VALUES ('Hong Kil-Dong', 'Volleyball');
```

Example 3

The following is an example where an error occurs with no data inserted when the partition key value is **NULL**. To define a partition where a **NULL** value can be inserted, define one that has a list including a **NULL** value as in the event3 partition as below.

```
INSERT INTO athlete2 VALUES ('Hong Kil-Dong','NULL');

CREATE TABLE athlete2( name VARCHAR(40), event VARCHAR(30) )
PARTITION BY LIST (event) (
PARTITION event1 VALUES IN ('Swimming', 'Athletics ' ),
PARTITION event2 VALUES IN ('Judo', 'Taekwondo','Boxing'),
PARTITION event3 VALUES IN ('Football', 'Basketball', 'Baseball', NULL)
);
```

Caution

- The maximum number of partitions cannot exceed 1,024.

List Partitioning Redefinition

Description

You can redefine a partition by using the **REORGANIZE PARTITION** clause of the **ALTER** statement. By redefining partitions, you can combine multiple partitions into one or divide one into multiple.

Syntax

```
ALTER {TABLE | CLASS} <table_name>
REORGANIZEPARTITION
<alter partition name comma list>
INTO ( <partition definition comma list> )
partition definition comma list:
PARTITION <partition_name> VALUES IN ( <partition_value_list>),...
```

- *table_name* : Specifies the name of the table to be redefined.
- *alter partition name comma list* : Specifies the partition to be redefined. Multiple partitions are separated by commas (,).
- *partition definition comma list* : Specifies the redefined partitions. Multiple partitions are separated by commas (,).

Example 1

The following is an example of creating the athlete2 table partitioned by the list of sport events, and redefining the event2 partition to be divided into event2_1 (Judo) and event2_2 (Taekwondo, Boxing).

```
CREATE TABLE athlete2( name VARCHAR(40), event VARCHAR(30) )
PARTITION BY LIST (event) (
PARTITION event1 VALUES IN ('Swimming', 'Athletics ' ),
PARTITION event2 VALUES IN ('Judo', 'Taekwondo','Boxing'),
PARTITION event3 VALUES IN ('Football', 'Basketball', 'Baseball')
);

ALTER TABLE athlete2 REORGANIZE PARTITION event2 INTO
(PARTITION event2_1 VALUES IN ('Judo'),
PARTITION event2_2 VALUES IN ( 'Taekwondo','Boxing'));
```

Example 2

The following is an example that combining the event2_1 and event2_2 partitions divided in Example 1 back into a single event2 partition.

```
ALTER TABLE athlete2 REORGANIZE PARTITION event2_1, event2_2 INTO
(PARTITION event2 VALUES IN('Judo','Taekwondo','Boxing'));
```

Dropping List Partitioning

Description

You can drop a partition by using the **DROP PARTITION** clause of the **ALTER** statement.

Syntax

```
ALTER {TABLE | CLASS} <table_name>
DROP PARTITION <partition_name>
```

- *table_name* : Specifies the name of the partitioned table.
- *partition_name* : Specifies the name of the partition to be dropped.

Example

The following is an example of creating the athlete2 table partitioned by the list of sport events, and dropping the event3 partition.

```
CREATE TABLE athlete2( name VARCHAR(40), event VARCHAR(30) )
PARTITION BY LIST (event) (
PARTITION event1 VALUES IN ('Swimming', 'Athletics ' ),
PARTITION event2 VALUES IN ('Judo', 'Taekwondo','Boxing'),
PARTITION event3 VALUES IN ('Football', 'Basketball', 'Baseball')
);

ALTER TABLE athlete2 DROP PARTITION event3;
```

Partitioning Management

Retrieving and Manipulating Data in Partitioning

Description

When retrieving data, the **SELECT** statement can be used not only for partitioned tables but also for each partition.

Example

The following is an example of creating the athlete2 table to be partitioned by the list of sport events, inserting data, and retrieving the event1 and event2 partitions.

```
CREATE TABLE athlete2( name VARCHAR(40), event VARCHAR(30) )
PARTITION BY LIST (event) (
PARTITION event1 VALUES IN ('Swimming', 'Athletics ' ),
PARTITION event2 VALUES IN ('Judo', 'Taekwondo','Boxing'),
PARTITION event3 VALUES IN ('Football', 'Basketball', 'Baseball')
);

INSERT INTO athlete2 VALUES ('Hwang Young-Cho', 'Athletics');
INSERT INTO athlete2 VALUES ('Lee Seung-Yuop', 'Baseball');
INSERT INTO athlete2 VALUES ('Moon Dae-Sung','Taekwondo');
INSERT INTO athlete2 VALUES ('Cho In-Chul', 'Judo');
csql> select * from athlete2__p__event1;
csql> ;x
=== <Result of SELECT Command in Line 1> ===
  name                    event
============================================
'Hwang Young-Cho'     'Athletics'

1 rows selected.
csql> select * from athlete2__p__event2;
csql> ;x
=== <Result of SELECT Command in Line 1> ===
  name                    event
============================================
  'Moon Dae-Sung'         'Taekwondo'
  'Cho In-Chul'           'Judo'
```

```
2 rows selected.
```

Caution

- Data manipulation such as insert, update and delete for each partition of the partitioned table is not allowed.

Moving Data by Changing Partitioning Key Value

Description

If a partition key value is changed, the changed instance can be moved to another partition by the partition expression.

Example

The following is an example of moving the instance to another partition by changing the partition key value.

If you change the sport event information of Hwang Young-Cho in the event1 partition from Athletics to Football, the instance is moved to the event3 partition.

```
CREATE TABLE athlete2( name VARCHAR(40), event VARCHAR(30) )
PARTITION BY LIST (event) (
PARTITION event1 VALUES IN ('Swimming', 'Athletics ' ),
PARTITION event2 VALUES IN ('Judo', 'Taekwondo','Boxing'),
PARTITION event3 VALUES IN ('Football', 'Basketball', 'Baseball')
);

INSERT INTO athlete2 VALUES ('Hwang Young-Cho', 'Athletics');
INSERT INTO athlete2 VALUES ('Lee Seung-Yuop', 'Baseball');
csql> SELECT * FROM athlete2__p__event1;
csql> ;x
=== <Result of SELECT Command in Line 1> ===
  name                  event
============================================
  'Hwang Young-Cho'     'Athletics'

1 rows selected.
csql> UPDATE athlete2 SET event = 'Football' WHERE name = 'Hwang Young-Cho';
csql> ;x
1 rows affected.
csql> SELECT * FROM athlete2__p__event3;
csql> ;x
=== <Result of SELECT Command in Line 1> ===
  name                  event
============================================
  'Lee Seung-Yuop'      'Baseball'
  'Hwang Young-Cho'     'Football'

2 rows selected.
```

Caution

- Be aware that when moving data between partitions by changing a partition key value, it can cause performance degradation due to internal deletions and insertions.

Altering Regular Table into Partitioning Table

Description

To alter a regular table into a partitioned one, use the **ALTER TABLE** statement. Three partitioning methods can be used with the **ALTER TABLE** statement. The data in the existing table are moved to and stored in each partition according to the partition definition.

Syntax

```
ALTER {TABLE | CLASS} table_name
PARTITION BY {RANGE | HASH | LIST } ( <partition_expression> )
( PARTITION partition_name VALUES LESS THAN { MAXVALUE | ( <partition_value_option> ) }
| PARTITION partition_name VALUES IN ( <partition_value_option list) > ]
| PARTITION <UNSINGED_INTEGER> )

<partition_expression>
expression_
<partition_value_option>
literal_
```

- *table_name* : Specifies the name of the table to be altered.
- *partition_expression* : Specifies a partition expression. The expression can be specified by the name of the column to be partitioned or by a function. For more information on the data types and functions available, see Data Types Available for Partition Expressions.
- *partition_name* : Specifies the name of the partition.
- *partition_value_option* : Specifies the value or the value list on which the partition is based.

Example

The following are examples of altering the record table into a range, list and hash table respectively.

```
ALTER TABLE record PARTITION BY RANGE (host_year)
( PARTITION before_1996 VALUES LESS THAN (1996),
  PARTITION after_1996 VALUES LESS THAN MAXVALUE);

ALTER TABLE record PARTITION BY list (unit)
( PARTITION time_record VALUES IN ('Time'),
  PARTITION kg_record VALUES IN ('kg'),
  PARTITION meter_record VALUES IN ('Meter'),
  PARTITION score_record VALUES IN ('Score') );

ALTER TABLE record
PARTITION BY HASH (score) PARTITIONS 4;
```

Caution

- If there is data that does not satisfy the partition condition, partitions cannot be defined.

Altering Partitioning Table into Regular Table

Description

To alter an existing partitioned table into a regular one, use the **ALTER TABLE** statement.

Syntax

```
ALTER {TABLE | CLASS} <table_name>
REMOVE PARTITIONING
```

- *table_name* : Specifies the name of the table to be altered.

Example

The following is an example of altering the partitioned table of name "nation2" into a regular one.

```
ALTER TABLE nation2 REMOVE PARTITIONING;
```

Partition Pruning

Description

Partition pruning is an optimization, limiting the scope of your query according to the criteria you have specified. It is the skipping of unnecessary data partitions in a query. By doing this, you can greatly reduce the amount of data output from the disk and time spent on processing data as well as improve query performance and resource availability.

Example 1

The following is an example of creating the olympic2 table to be partitioned based on the year the Olympic Games were held, and retrieving the countries that participated in the Olympic Games since the 2000 Sydney Olympic Games.

In the **WHERE** clause, partition pruning takes place when equality or range comparison is performed between a partition key and a constant value. In this example, the before_1996 partition that has a smaller year value than 2000 is not scanned.

```
CREATE TABLE olympic2
( opening_date DATE, host_nation VARCHAR(40))
PARTITION BY RANGE ( EXTRACT (YEAR FROM opening_date) )
( PARTITION before_1996 VALUES LESS THAN (1996),
  PARTITION before_MAX VALUES LESS THAN MAXVALUE );

SELECT opening_date, host_nation FROM olympic2 WHERE EXTRACT ( YEAR FROM (opening_date))
>= 2000;
```

Example 2

The following is an example of showing the method of getting the effects of partition pruning by retrieving data with a specific partition when partition pruning does not occur. In the first query, partition pruning does not occur because the value compared is not in the same format as that of the partition expression.

Therefore, you can use the same effect of partition pruning by specifying the appropriate partition as shown in the second query.

```
SELECT host_nation FROM olympic2 WHERE opening_date >= '2000 - 01 - 01';

SELECT host_nation FROM olympic2__p__before_max WHERE opening_date >= '2000 - 01 - 01';
```

Example 3

The following is an example of specifying the search condition to make a partition pruning in the hash partitioned table, called the manager table.

For hash partitioning, partition pruning occurs only when equality comparison is performed between a partition key and a constant value in the **WHERE** clause.

```
CREATE TABLE manager (
code INT,
name VARCHAR(50))
PARTITION BY HASH ( code) PARTITIONS 4;

SELECT * FROM manager WHERE code = 10053;
```

Caution

- The partition expression and the value compared must be in the same format.

Data Types Available for Partitioning Expression

Description

The following table shows data types of the column that can or cannot be used as a partition key.

Data Types Available	Data Types Unavailable
CHAR	FLOAT
VARCHAR	REAL
NCHAR	DOUBLE
VARNCHAR	BIT
INTEGER	BIT VARYING
SMALLINT	NUMERIC OR DECIMAL
DATE	MONETARY
TIME	SET
TIMESTAMP	LIST OR SEQUENCE
	MULTISET
	OBJECT

The following operator functions can be used in partition expressions to be applied to partition keys.

- Number Operations

  +, -, *, /, MOD, STRCAT, FLOOR, CEIL, POWER, ROUND, ABS, TRUNC
- String Operations

  POSITION, SUBSTRING, OCTEC_LENGTH, BIT_LENGTH, CHAR_LENGTH, LOWER, UPPER, TRIM, LTRIM, RTRIM, LPAD, RPAD, REPLACE, TRANSLATE
- Date Operations

  ADD_MONTH, LAST_DAY, MONTH_BETWEEN, SYS_DATE, SYS_TIME, SYS_TIMESTAMP, TO_DATE, TO_NUMBER, TO_TIME, TO_TIMESTAMP, TO_CHAR
- Others

  EXTRACT, CAST

Creating VIEW with Partitioning Table

Description

You can define a virtual table by using each partition of a partitioned table. Retrieving data from the virtual table created is possible, but data insert, delete and update operations are not allowed.

Example

The following is an example of creating the participant2 table partitioned based on the participating year, and creating and retrieving a virtual table with the participant2__p__before_2000 partition.

```
CREATE TABLE participant2 (host_year INT, nation CHAR(3), gold INT, silver INT, bronze INT)
PARTITION BY RANGE (host_year)
( PARTITION before_2000 VALUES LESS THAN (2000),
 PARTITION before_2008 VALUES LESS THAN (2008) );

INSERT INTO participant2 VALUES (1988, 'NZL', 3, 2, 8);
INSERT INTO participant2 VALUES (1988, 'CAN', 3, 2, 5);
INSERT INTO participant2 VALUES (1996, 'KOR', 7, 15, 5);
INSERT INTO participant2 VALUES (2000, 'RUS', 32, 28, 28);
INSERT INTO participant2 VALUES (2004, 'JPN', 16, 9, 12);

CREATE VIEW v_2000 AS
SELECT * FROM participant2__p__before_2000
WHERE host_year = 1988;
csql> SELECT * FROM v_2000;
csql> ;x
=== <Result of SELECT Command in Line 1> ===
    host_year  nation                       gold       silver       bronze
==========================================================================
         1988  'NZL'                           3            2            8
         1988  'CAN'                           3            2            5

2 rows selected.
```

Updating Statistics on Partitioning Tables

You can update statistics on the database by using the **cubrid optimizedb** utility or the SQL statement called **UPDATE STATISTICS ON CLASSES**. You can also use the **ANALYZE PARTITION** statement for partitioned tables.

The following is an example of the **ANALYZE PARTITION** statement.

```
ALTER TABLE t1 ANALYZE PARTITION p3;
```

Class Inheritance

Overview

Description

Classes in CUBRID database can have class hierarchy. Attributes and methods can be inherited through such hierarchy.

As shown in the previous section, you can create a Manager class by inheriting attributes from an Employee class. The Manager class is called the **subclass** of the Employee class, and the Employee class is called the **super class** of the Manager class. Inheritance can simplify class creation by reusing the existing class hierarchy.

CUBRID allows multiple inheritance, which means that a class can inherit attributes and methods from more than one super class. However, inheritance can cause conflicts when an attribute or method of the super class is added or deleted.

Such conflict occurs in multiple inheritance if there are attributes or methods with the same name in different super classes. For example, if it is likely that a class inherits attributes of the same name and type from more than one super class, you must specify the attributes to be inherited. In such a case, if the inherited super class is deleted, a new attribute of the same name and type must be inherited from another super class. In most cases, the database system resolves such problems automatically. However, if you don't like the way that the system resolves a problem, you can resolve it manually by using the INHERIT clause.

When attributes are inherited from more than one super class, it is possible that their names are to be the same, while their domains are different. For example, two super classes may have the same attribute, whose domain is a class. In this case, a subclass automatically inherits attributes with more specialized (a lower in the class hierarchy) domains. If such conflict occurs between basic data types (e.g. STRING or INTEGER) provided by the system, inheritance fails. Conflicts during inheritance and their resolutions will be covered in the Resolving Class Conflicts section.

Caution

The following cautions must be observed during inheritance:

- The class name must be unique in the database. A class can be created as a subclass of one or more super class names in the database optionally. An error occurs if you create a class that inherits another class that does not exist.
- The name of a method/attribute must be unique within a class. The name cannot contain spaces, and cannot be a reserved keyword of CUBRID. Alphabets as well as '_', '#', '%' are allowed in the class name, but the first character cannot be '_'. A class name cannot exceed 255 English letters. Class names are not case-sensitive. A class name will be saved in the system after being converted to lowercase characters.

Note A super class name can begin with the user name so that the owner of the class can be easily identified.

Class Attribute and Method

You can create class attributes to store the aggregate property of all instances in the class. When you define a **CLASS** attribute or method, you must precede the attribute or method name with the keyword **CLASS**. Because a class attribute is associated with the class itself, not with an instances of the class, it has only one value. For example, a class attribute can be used to store the average value determined by a class method or the timestamp when the class was created. A class method is executed on the class object itself. It can be used to calculate the aggregate value for the instances of the class.

When a subclass inherits a super class, each class has a separate storage space for class attributes, so that two classes may have different values of class attribute. Therefore, the subclass does not change even when the attributes of the super class are changed.

The name of a class attribute can be the same as that of an instance attribute of the same class. Likewise, the name of a class method can be the same as that of an instance method of the same class.

Order Rule for Inheritance

The following rules apply to inheritance. The term class is generally used to describe the inheritance relationship between classes and virtual classes in the database.

- For an object without a super class, attributes are defined in the same order as in the **CREATE** statement (an ANSI standard).
- If there is one super class, locally created attributes are placed after the super class attributes. The order of the attributes inherited from the super class follows the one defined during the super class definition. For multiple inheritance, the order of the super class attributes is determined by the order of the super classes specified during the class definition.
- If more than one super class inherits the same class, the attribute that exists in both super classes is inherited to the subclass only once. At this time, if a conflict occurs, the attribute of the first super class is inherited.
- If a name conflict occurs in more than one super class, you can inherit only the ones you want from the super class attributes by using the **INHERIT** clause in order to resolve the conflict.
- If the name of the super class attribute is changed by the alias option of the **INHERIT** clause, its position is maintained.

INHERIT Clause

Description

When a class is created as a subclass, the class inherits all attributes and methods of the super class. A name conflict that occurs during inheritance can be handled by either a system or a user. To resolve the name conflict directly, add the **INHERIT** clause to the **CREATE CLASS** statement.

Syntax

```
CREATE CLASS
.
.
.
INHERIT resolution [ {, resolution }_ ]

resolution :
{ column_name | method_name } OF super class [ AS alias ]
```

For the *attr_mthd_name* in the **INHERIT** clause, specify the name of the attribute or method of the super class to inherit. With the **ALIAS** clause, you can resolve a name conflict that occurs in multiple inheritance statements by inheriting a new name.

ADD SUPERCLASS Clause

Description

To extend class inheritance, add a super class to a class. A relationship between two classes is created when a super class is added to an existing class. Adding a super class does not mean adding a new class.

Syntax

```
ALTER CLASS
.
.
.
ADD super class [ user_name.]class_name [ { , [ user_name.]class_name }_ ]
[ INHERIT resolution [ {, resolution }_ ] ] [ ]
resolution:
{ column_name | method_name } OF super class_name [ AS alias ]
```

For the first *class_name*, specify the name of the class where a super class is to be added. Attributes and methods of the super class can be inherited by using the syntax above.

Name conflicts can occur when adding a new super class. If a name conflict cannot be resolved by the database system, attributes or methods to inherit from the super class can be specified by using the **INHERIT** clause. You can use aliases to inherit all attributes or methods that cause the conflict. For more information on super class name conflicts, see the Resolving Class Conflict section.

Example

The following is an example of creating the female_event class by inheriting the event class included in demodb.

```
CREATE CLASS female_event UNDER event
```

DROP SUPERCLASS Clause

Description

Deleting a super class from a class means removing the relationship between two classes. If a super class is deleted from a class, it changes inheritance relationship of the classes as well as of all their subclasses.

Syntax

```
ALTER CLASS
.
.
.
DROP super class class_name [ { , class_name }_ ]
[ INHERIT resolution [ {, resolution }_ ] ] [ ]

resolution:
{ column_name | method_name } OF super class_name [ AS alias ]
```

For the first *class_name*, specify the name of the class to be modified. For the second *class_name*, specify the name of the super class to be deleted. If a name conflict occurs after deleting a super class, see the Resolving Class Conflict section for the resolution.

Example 1

In the following example, the female_event class inherits from the event class.

```
CREATE CLASS female_event UNDER event
```

Example 2

In the following example, the **ALTER** statement deletes the event super class from the female_event class. The attributes that the female_event class inherited from the event class do not exist any more.

```
ALTER CLASS female_event
DROP super class event
```

Class Conflict Resolution

Overview

If you modify the schema of the database, conflicts can occur between attributes or methods of inheritance classes. Most conflicts are resolved automatically by CUBRID otherwise, you must resolve the conflict manually. Therefore, you need to examine the possibility of conflicts before modifying the schema.

Two types of conflicts can cause damage to the database schema. One is conflict with a subclass when the subclass schema is modified. The other is conflict with a super class when the super class is modified. The following are operations that may cause conflicts between classes.

- Adding an attribute
- Deleting an attribute
- Adding a super class
- Deleting a super class
- Deleting a class

If a conflict occurs as the result of the above operations, CUBRID applies a basic resolution to the subclass where the conflict occurred. Therefore, the database schema can always maintain consistent state.

Resolution Specifier

Description

Conflicts between the existing classes or attributes, and inheritance conflicts can occur if the database schema is modified. If the system fails to resolve a conflict automatically or if you don't like the way the system resolved the problem, you can suggest how to resolve the conflict by using the **INHERIT** clause of the **ALTER** statement (often referred as resolution specifier).

When the system resolves the conflict automatically, basically, the existing inheritance is maintained (if any). If the previous resolution becomes invalid when the schema is modified, the system will arbitrarily select another one. Therefore, you must avoid excessive reuse of attributes or methods in the schema design stage because the way the system will resolve the conflict cannot always be predictable.

What will be discussed concerning conflicts is applied commonly to both attributes and methods.

Syntax

```
ALTER [ class_type ] class_name alter_clause
[ INHERIT resolution [ {, resolution }_ ] ] [ ]
resolution:
{ column_name | method_name } OF super class_name [ AS alias ]
```

Superclass Conflict

Adding a super class

The **INHERIT** clause of the **ALTER CLASS** statement is optional, but must be used when a conflict occurs due to class changes. You can specify more than one resolutions after the **INHERIT** clause.

super class_name specifies the name of the super class that has the new attribute or method to inherit when a conflict occurs. *attr_mthd_name* specifies the name of the attribute or method to inherit. You can use the **alias** clause when you need to change the name of the attribute or method to inherit.

The following example creates the soccer_stadium class by inheriting the event and stadium classes in the olympic database of demodb. Because both event and stadium classes have the name and code attributes, you must specify the attributes to inherit using the **INHERIT** clause.

```
CREATE CLASS soccer_stadium UNDER event, stadium
INHERIT name OF stadium, code OF stadium
```

When the two super classes (event and stadium) have the name attribute, if the soccer_stadium class needs to inherit both attributes, it can inherit the name unchanged from the stadium class and the name changed from the event class by using the **alias** clause of the **INHERIT**.

The following is an example in which the name attribute of the stadium class is inherited as it is, and that of the event class is inherited as the 'purpose' alias.

```
ALTER CLASS soccer_stadium
INHERIT name OF event AS purpose
```

Deleting a super class

A name conflict may occur again if a super class that explicitly inherited an attribute or method is dropped by using the **INHERIT**. In this case, you must specify the attribute or method to be explicitly inherited when dropping the super class.

The following is an example of creating the seoul_1988_soccer class by inheriting game, participant and stadium classes from demodb, and deleting the participant class from the super class. Because nation_code and host_year are explicitly inherited from the participant class, you must resolve their name conflicts before deleting it from the super class. However, host_year does not need to be specified explicitly because it exists only in the game class.

```
CREATE CLASS seoul_1988_soccer UNDER game, participant, stadium
INHERIT nation_code OF participant, host_year OF participant
ALTER CLASS seoul_1988_soccer
DROP super class participant
INHERIT nation_code OF stadium
```

Compatible Domains

When an attribute conflict occurs among two or more super classes, the statement resolving the conflict is not possible only if all attributes have compatible domains.

For example, the class that inherits a super class with the phone attribute of integer type cannot have another super class with the phone attribute of string type. If the types of the phone attributes of the two super classes are both String or Integer, you can add a new super class by resolving the conflict with the **INHERIT** clause.

Compatibility is checked when inheriting an attribute with the same name, but with the different domain. In this case, the attribute that has a lower class in the class inheritance hierarchy as the domain is automatically inherited. If the domains of the attributes to inherit are compatible, the conflict must be resolved in the class where an inheritance relationship is defined.

Subclass Conflict

Any changes in a class will be automatically propagated to all subclasses. If a problem occurs in the subclass due to the changes, CUBRID resolves the corresponding subclass conflict and then displays a message saying that the conflict has been resolved automatically by the system.

Subclass conflicts can occur due to operations such as adding a super class, or creating/deleting a method or an attribute. Any changes in a class will affect all subclasses. Since changes are automatically propagated, harmless changes can even cause side effects in subclasses.

Adding Attributes and Methods

The simplest subclass conflict occurs when an attribute is added. A subclass conflict occurs if an attribute added to a super class has the same name as one already inherited by another super class. In such cases, CUBRID will automatically resolve the problem. That is, the added attribute will not be inherited to all subclasses that have already inherited the attribute with the same name.

The following is an example of adding an attribute to the event class. The super classes of the soccer_stadium class are the event and the stadium classes, and the nation_code attribute already exists in the stadium class. Therefore, a conflict occurs in the soccer_stadium class if the nation_code attribute is added to the event class. However, CUBRID resolves this conflict automatically.

```
ALTER CLASS event
ADD ATTRIBUTE nation_code CHAR(3)
```

If the event class is dropped from the soccer_stadium super class, the cost attribute of the stadium class will be inherited automatically.

Dropping Attributes and Methods

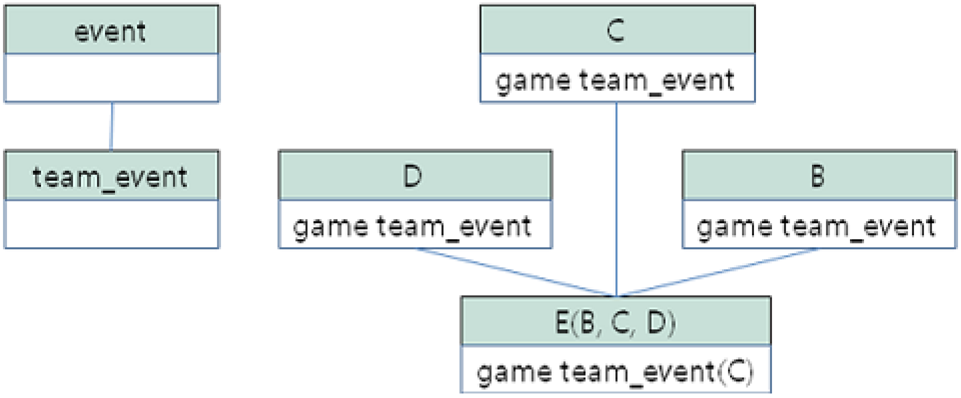
When an attribute is dropped from a class, any resolution specifiers which refer to the attribute by using the **INHERIT** clause are also removed. If a conflict occurs due to the deletion of an attribute, the system will determine a new inheritance hierarchy. If you don't like the inheritance hierarchy determined by the system, you can determine it by using the **INHERIT** clause of the **ALTER** statement. The following is an example of such conflict.

Suppose there is a subclass that inherits attributes from three different super classes. If a name conflict occurrs in all super classes and the explicitly inherited attribute is dropped, one of the remaining two attributes will be inherited automatically to resolve the problem.

The following is an example of a subclass conflict. Classes B, C and D are super classes of class E, and have an attribute whose name is team and the domain is team_event. Class E was created with the place attribute inherited from class C as follows:

```
create class E under B, C, D
inherit place of C
```

In this case, the inheritance hierarchy is as follows:



Suppose that you decide to delete class C from the super class. This drop will require changes to the inheritance hierarchy. Because the domains of the remaining classes B and D with the game attribute are at the same level, the system will randomly choose to inherit from one of the two classes. If you don't want the system to make a random selection, you can specify the class to inherit from by using the **INHERIT** clause when you change the class.

```
ALTER CLASS E
INHERIT game OF D
ALTER CLASS C
DROP game
```

Note If the domain of the game attribute of one super class is event and that of another super class is team_event, the attribute that has team_event as the domain will be inherited because team_event is more specific than event (as team_event exists lower in the inheritance hierarchy). In this case, you cannot force the attribute that has event as the domain to be inherited because the event class exists higher in the inheritance hierarchy than team_event.

Schema Invariant

Invariants of a database schema are a property of the schema that must be preserved consistently (before and after the schema change). There are four types of invariants: invariants of class hierarchy, name, inheritance and consistency.

- **Invariant of class hierarchy** has a single root and defines a class hierarchy as a Directed Acyclic Graph (DAG) where all connected classes have a single direction. That is, all classes except for the root have one or more super classes, and cannot become their own super classes. The root of DAG is "object," a system-defined class.
- **Invariant of name** means that all classes in the class hierarchy and all attributes in a class must have unique names. That is, attempts to create classes with the same name or to create attributes or methods with the same name in a single class are not allowed.

  Invariant of name is redefined by the 'rename' qualifier. The 'rename' qualifier allows the name of an attribute or method to be changed.

- **Invariant of inheritance** means that a class must inherit all attributes and methods from all super classes. This invariant can be distinguished with three qualifiers: source, conflict and domain. The names of inherited attributes and methods can be modified. For default or shared value attributes, the default or shared value can be modified. Invariant of inheritance means that such changes will be propagated to all classes that inherit these attributes and methods.
  - A **source qualifier** means that if class C inherits subclasses of class S, only one of the subclass attributes (methods) inherited from class S can be inherited to class C. That is, if an attribute (method) defined in class S is inherited by other classes, it is in effect a single attribute (method), even though it exists in many subclasses. Therefore, if a class multiply inherits from classes that have attributes (methods) of the same source, only one appearance of the attribute (method) is inherited.
  - A **conflict qualifier** means that if class C inherits from two or more classes that have attributes (methods) with the same name but of different sources, it can inherit more than one class. To inherit attributes (methods) with the same name, you must change their names so as not to violate the invariant of name.
  - A **domain qualifier** means that a domain of an inherited attribute can be converted to the domain's subclass.
- **Invariant of consistency** means that the database schema must always follow the invariants of a schema and all rules (Rules for Schema Changes) except when it is being changed.

Rule for Schema Changes

The Invariants of a Schema section has described the characteristics of schema that must be preserved all the time. There are some methods for changing schemas, and all these methods must be able to preserve the invariants of a schema. For example, suppose that in a class which has a single super class, the relationship with the super class is to be removed. If the relationship with the super class is removed, the class becomes a direct subclass of the object class, or the removal attempt will be rejected if the user specified that the class should have at least one super class. To have some rules for selecting one of the methods for changing schemas, even though such selection seems arbitrary, will be definitely useful to users and database designers.

The following three types of rules apply: conflict-resolution rules, domain-change rule and class-hierarchy rule.

Seven conflict-resolution rules reinforce the invariant of inheritance. Most schema change rules are needed because of name conflicts. A domain-change rule reinforces a domain resolution of the invariant of inheritance. A class-hierarchy rule reinforces the invariant of class hierarchy.

Conflict-Resolution Rules

- **Rule 1** : If an attribute (method) name of class C and an attribute name of the super class S conflict with each other (that is, their names are same), the attribute of class C is used. The attribute of S is not inherited.

  If a class has one or more super classes, three aspects of the attribute (method) of each super class must be considered to determine whether the attributes are semantically equal and which attribute to inherit. The three aspects of the attribute (method) are the name, domain and source. The following table shows eight combinations of these three aspects that can happen with two super classes. In Case 1 (two different super classes have attributes with the same name, domain and source), only one of the two subclasses should be inherited because two attributes are identical. In Case 8 (two different super classes have attributes with different names, domains and sources), both classes should be inherited because two attributes are totally different ones.

Case	Name	Domain	Source
1	Same	Same	Same
2	Same	Same	Different
3	Same	Different	Same

4	Same	Different	Different
5	Different	Same	Same
6	Different	Same	Different
7	Different	Different	Same
8	Different	Different	Different

Five cases (1, 5, 6, 7, 8) out of eight have clear meaning. Invariant of inheritance is a guideline for resolving conflicts in such cases. In other cases (2, 3, 4), it is very difficult to resolve conflicts automatically. Rules 2 and 3 can be resolutions for these conflicts.

- **Rule 2** : When two or more super classes have attributes (methods) with different sources but the same name and domain, one or more attributes (methods) can be inherited if the conflict-resolution statement is used. If the conflict-resolution statement is not used, the system will select and inherit one of the two attributes.

This rule is a guideline for resolving conflicts of Case 2 in the table above.

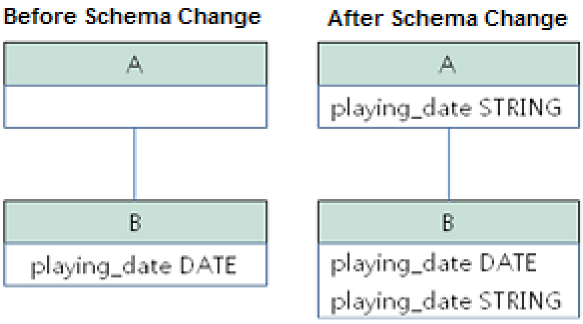
- **Rule 3** : If two or more super classes have attributes with different sources and domains but the same name, attributes (methods) with more detailed (lower in the inheritance hierarchy) domains are inherited. If there is no inheritance relationship between domains, schema change is not allowed.

This rule is a guideline for resolving conflicts of Case 3 and 4. If Case 3 and 4 conflict with each other, Case 3 has the priority.

- **Rule 4** : The user can make any changes except for the ones in Case 3 and 4. In addition, the resolution of subclass conflicts cannot cause changes in the super class.

The philosophy of Rule 4 is that "an inheritance is a privilege a subclass obtained from a super class, so changes in a subclass cannot affect the super class." Rule 4 means that the name of the attribute (method) included in the super class cannot be changed to resolve conflicts between class C and super classes. Rule 4 has an exception in cases where the schema change causes conflicts in Case 3 and 4.

- For example, suppose that class A is the super class of class B, and class B has the playing_date attribute of **DATE** type. If an attribute of **STRING** type named playing_date is added to class A, it conflicts with the playing_date attribute in class B. This is what happens in Case 4. The precise way to resolve such conflict is for the user to specify that class B must inherit the playing_date attribute of class A. If a method refers to the attribute, the user of class B needs to modify the method properly so that the appropriate playing_date attribute will be referenced. Schema change of class A is not allowed because the schema falls into an inconsistent state if the user of class B does not describe an explicit statement to resolve the conflict occurring from the schema change.



- **Rule 5** : If a conflict occurs due to a schema change of the super class, the original resolution is maintained as long as the change does not violate the rules. However, if the original resolution becomes invalid due to the schema change, the system will apply another resolution.

Rule 5 is for cases where a conflict is caused to a conflict-free class or where the original resolution becomes invalid.

This is the case where the name or domain of an attribute (method) is modified or a super class is deleted when the attribute (method) is added to the super class or the one inherited from the super class is deleted. The philosophy of Rule 5 coincides with that of Rule 4. That is, the user can change the class freely without considering what effects the subclass that inherits from the given class will have on the inherited attribute (method).

When you change the schema of class C, if you decide to inherit an attribute of the class due to an earlier conflict with another class, this may cause attribute (method) loss of class C. Instead, you must inherit one of the attributes (methods) that caused conflicts earlier.

The schema change of the super class can cause a conflict between the attribute (method) of the super class and the (locally declared or inherited) attribute (method) of class C. In this case, the system resolves the conflict automatically by applying Rule 2 or 3 and may inform the user.

Rule 5 cannot be applied to cases where a new conflict occurs due to the addition or deletion of the relationship with the super class. The addition/deletion of a super class must be limited to within the class. That is, the user must provide an explicit resolution.

- **Rule 6** : Changes of attributes or methods are propagated only to subclasses without conflicts.

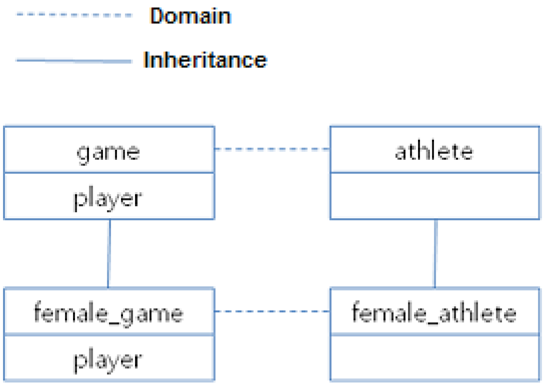
This rule limits the application of Rule 5 and the invariant of inheritance. Conflicts can be detected and resolved by applying Rule 2 and 3.

- **Rule 7** : Class C can be dropped even when an attribute of class R uses class C as a domain. In this case, the domain of the attribute that uses class C as a domain can be changed to object.

Domain-Change Rule

- **Rule 8** : If the domain of an attribute of class C is changed from D to a super class of D, the new domain is less generic than the corresponding domain in the super class from which class C inherited the attribute. The following example explains the principle of this rule.

Suppose that in the database there are the game class with the player attribute and the female_game class which inherits game. The domain of the player attribute of the game class is the athlete class, but the domain of the player attribute of the female_game class is changed to female_athlete which is a subclass of athlete. The following diagram shows such relationship. The domain of the player attribute of the female_game class can be changed back to athlete, which is the super class of female_athlete.



Class-Hierarchy Rule

- **Rule 9** : A class without a super class becomes a direct subclass of object. The class-hierarchy rule defines characteristics of classes without super classes. If you create a class without a super class, object becomes the super class. If you delete the super class S, which is a unique super class of class C, class C becomes a direct subclass of object.

CUBRID System Catalog

Overview

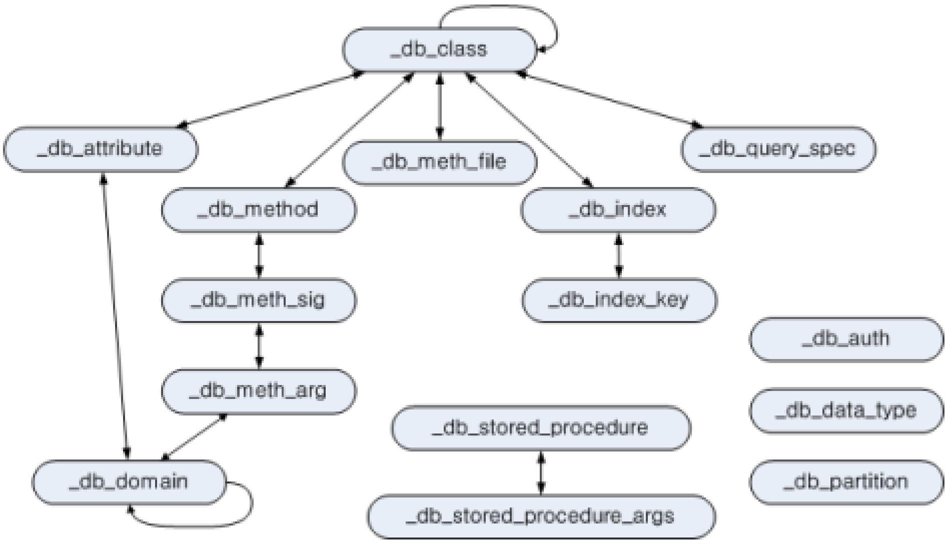
You can easily get various schema information from the SQL statement by using the system catalog virtual class (table). For example, you can get the following schema information by using the catalog virtual class.

```
-- Classes that refer to the 'b_user' class
SELECT class_name
FROM db_attribute
WHERE domain_class_name = 'db_user'

-- The number of classes that the current user can access
SELECT COUNT(*)
FROM db_class

-- Attribute of the 'db_user' class
SELECT attr_name, data_type
FROM db_attribute
WHERE class_name = 'db_user'
```

System Catalog Classes

System Catalog Classes

To define a catalog virtual class, define a catalog class first. The figure below shows catalog classes to be added and their relationships. The arrows represent the reference relationship between classes, and the classes that start with an underline (_) are catalog classes.



Added catalog classes represent information about all classes, attributes and methods in the database. Catalog classes are made up of class composition hierarchy and designed to have OIDs of catalog class instances for cross reference.

_db_class

Represents information about the class. An index for class_name is created.

Attribute Name	Data Type	Description
class_of	object	A class object. Represents a meta information object for the class saved in the system.
inst_attr_count	INTEGER	The number of instance attributes

shared_attr_count	INTEGER	The number of shared attributes
inst_meth_count	INTEGER	The number of instance methods
class_meth_count	INTEGER	The number of class methods
class_attr_count	INTEGER	The number of class attributes
is_system_class	INTEGER	0 for a user-defined class, and 1 for a system class.
class_type	INTEGER	0 for a class, and 1 for a virtual class.
owner	db_user	Class owner
class_name	VARCHAR(255)	Class name
sub_classes	SEQUENCE OF _db_class	Class one level down
super_classes	SEQUENCE OF _db_class	Class one level up
inst_attrs	SEQUENCE OF _db_attribute	Instance attribute
shared_attrs	SEQUENCE OF _db_attribute	Shared attribute
class_attrs	SEQUENCE OF _db_attribute	Class attribute
inst_meths	SEQUENCE OF _db_method	Instance method
class_meths	SEQUENCE OF _db_method	Class method
meth_files	SEQUENCE OF _db_methfile	File path in which the function for the method is located
query_specs	SEQUENCE OF _db_queryspec	SQL definition statement for a virtual class
indexes	SEQUENCE OF _db_index	Index created in the class

Example

The following is an example of retrieving all subclasses under the class owned by user 'PUBLIC' (for the child class female_event in the result, see the example in Adding a super class).

```
csql> select class_name, sequence(select class_name
csql>   from _db_class s
csql> where s in c.sub_classes)
csql> from _db_class c
csql> where c.owner.name = 'PUBLIC' and
csql> c.sub_classes is NOT NULL
csql> x

=== < Result of SELECT Command in Line 1> ===

  class_name                      sequence((select class_name from _db_class s where s in
c.sub_classes))
============================================
  'event'                            {'female_event'}
1 rows selected.
```

Note All examples of system catalog classes have been written in the csql utility. In this example, **--no-auto-commit** (inactive mode of auto-commit) and **-u** (specifying user DBA) options are used.

% csql --no-auto-commit -u dba demodb

_db_attribute

Represents information about attributes. Indexes for class_of and attr_name are created.

Attribute Name	Data Type	Description
class_of	_db_class	Class to which the attribute belongs
attr_type	INTEGER	Type defined for the attribute. 0 for an instance attribute, 1 for a class attribute, and 2 for a shared attribute.
data_type	INTEGER	Data type of the attribute. One of the values specified in the "Data Types Supported by CUBRID" table below.
def_order	INTEGER	Order of attributes in the class. Begins with 0. If the attribute is inherited, the order is the one defined in the super class. For example, if class y inherits attribute a from class x and a was first defined in x, def_order becomes 0.
from_class_of	_db_class	If the attribute is inherited, the super class in which the attribute is defined is used. Otherwise, **NULL**
from_attr_name	VARCHAR(255)	If the attribute is inherited and its name has been changed to resolve a name conflict, the original name defined in the super class is used. Otherwise, **NULL**
attr_name	VARCHAR(255)	Attribute name
default_value	VARCHAR(255)	Default value. Saved as a character string regardless of data types. If there is no default value, **NULL**. If the default value is **NULL**, 'NULL' is used. If the data type is an object, ʹvolume id | page id | slot idʹ is used. If the data type is a set, ʹ{element 1, element 2, ... is used.
domains	SEQUENCE OF _db_domain	Domain information of the data type
is_nullable	INTEGER	0 if a not null constraint is configured, and 1 otherwise.

Data Types Supported by CUBRID

Value	Meaning	Value	Meaning
1	INTEGER	13	MONETARY
2	FLOAT	18	SHORT
3	DOUBLE	20	OID
4	STRING	22	NUMERIC
5	OBJECT	23	BIT
6	SET	24	VARBIT
7	MULTISET	25	CHAR
8	SEQUENCE	26	NCHAR
9	ELO	27	VARNCHAR
10	TIME	31	BIGINT
11	TIMESTAMP	32	DATETIME
12	DATE		

Character Sets Supported by CUBRID

Value	Meaning
0	US English ? ASCII encoding

3	Latin 1 ? ISO 8859 encoding
4	KSC 5601 1990 ? EUC encoding

Example

The following is an example of retrieving user classes (from_class_of.is_system_class = 0) among the ones owned by user 'PUBLIC.'

```
csql> select class_of.class_name, attr_name
csql> from _db_attribute
csql> where class_of.owner.name = 'PUBLIC' and
csql>               from_class_of.is_system_class = 0
csql> order by 1, def_order
csql> xrun

class_of.class_name    attr_name
============================================
  'female_event'              'code'
  'female_event'              'sports'
  'female_event'              'name'
  'female_event'              'gender'
  'female_event'              'players'
```

_db_domain

Represents information about the domain. An index for object_of is created.

Attribute Name	Data Type	Description
object_of	object	Attribute that refers to the domain, which can be a method parameter or domain
data_type	INTEGER	Data type of the domain (a value in the "Value" column of the "Data Types Supported by CUBRID" table in _db_attribute)
prec	INTEGER	Precision of the data type. 0 is used if the precision is not specified.
scale	INTEGER	Scale of the data type. 0 is used if the scale is not specified.
class_of	_db_class	Domain class if the data type is an object, **NULL** otherwise.
code_set	INTEGER	Character set (value of table "character sets supported by CUBRID" in _db_attribute) if it is character data type. 0 otherwise.
set_domains	SEQUENCE OF _db_domain	Domain information about the data type of collection element if it is collection data type. **NULL** otherwise.

_db_method

Represents information about the method. Indexes for class_of and meth_name are created.

Attribute Name	Data Type	Description
class_of	_db_class	Class to which the method belongs
meth_type	INTEGER	Type of the method defined in the class. 0 for an instance method, and 1 for a class method.
from_class_of	_db_class	If the method is inherited, the super class in which it is defined is used otherwise **NULL**
from_meth_name	VARCHAR(255)	If the method is inherited and its name is changed to resolve a name conflict, the original name defined in the super class is used otherwise **NULL**
meth_name	VARCHAR(255)	Method name

signatures	SEQUENCE OF _db_meth_sig	C function executed when the method is called

Example

The following is an example of retrieving class methods of the class with a class method (c.class_meth_count > 0), among classes owned by user 'DBA.'

```
csql> select class_name, sequence(select meth_name
csql>                                                         from _db_method m
csql>                                                         where m in c.class_meths)
csql> from _db_class c
csql> where c.owner.name = 'DBA' and
csql>               c.class_meth_count > 0
csql> order by 1
csql> xrun

=== < Result of SELECT Command in Line 1> ===
  class name             sequence((select meth_name from _db_method m where in
c.class_meths))
============================================
 'db_serial'             {'change_serial_owner'}
 'db_authorizations'     {'add_user', 'drop_user', 'find_user', 'print_authorizations',
'info', 'change_owner', 'change_trigg r_owner', 'get_owner'}
 'db_authorization'      {'check_authorization'}
 'db_user'               {'add_user', 'drop_user', 'find_user', 'login'}
 'db_root'               {'add_user', 'drop_user', 'find_user', 'print_authorizations',
'info', 'change_owner', 'change_trigg r_owner', 'get_owner', 'change_sp_owner'}

5 rows selected.
```

_db_meth_sig

Represents information about the C function of the method. An index for meth_of is created.

Attribute Name	Data Type	Description
meth_of	_db_method	Method for the function information
arg_count	INTEGER	The number of input arguments of the function
func_name	VARCHAR(255)	Function name
return_value	SEQUENCE OF _db_meth_arg	Return value of the function
arguments	SEQUENCE OF _db_meth_arg	Input arguments of the function

_db_meth_arg

Represents information about the method argument. An index for meth_sig_of is created.

Attribute Name	Data Type	Description
meth_sig_of	_db_meth_sig	Information of the function to which the argument belongs
data_type	INTEGER	Data type of the argument (a value in the "Value" column of the "Data Types Supported by CUBRID" in _db_attribute)
index_of	INTEGER	Order of the argument listed in the function definition. Begins with 0 if it is a return value, and 1 if it is an input argument.
domains	SEQUENCE OF _db_domain	Domain of the argument

_db_meth_file

Represents information about the file in which the function is defined. An index for class_of is created.

Attribute Name	Data Type	Description

class_of	_db_class	Class to which the method file information belongs
from_class_of	_db_class	If the file information is inherited, the super class in which it is defined is used otherwise, **NULL**
path_name	VARCHAR(255)	File path in which the method is located

_db_query_spec

Represents the SQL definition statement of the virtual class. An index for class_of is created.

Attribute Name	Data Type	Description
class_of	_db_class	Class information of the virtual class
spec	VARCHAR(4096)	SQL definition statement of the virtual class

_db_index

Represents information about the index. An index for class_of is created.

Attribute Name	Data Type	Description
class_of	_db_class	Class to which to index belongs
index_name	varchar(255)	Index name
is_unique	INTEGER	1 if the index is unique, and 0 otherwise.
key_count	INTEGER	The number of attributes that comprise the key
key_attrs	SEQUENCE OF _db_index_key	Attributes that comprise the key
is_reverse	INTEGER	1 for a reverse index, and 0 otherwise.
is_primary_key	INTEGER	1 for a primary key, and 0 otherwise.
is_foreign_key	INTEGER	1 for a foreign key, and 0 otherwise.

Example

The following is an example of retrieving names of indexes that belong to the class.

```
csql> select class_of.class_name, index_name
csql> from _db_index
csql> order by 1;
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

  class_of.class_name   index_name
============================================
  '_db_attribute'       'i__db_attribute_class_of_attr_name'
  '_db_auth'            'i__db_auth_grantee'
  '_db_class'           'i__db_class_class_name'
  '_db_domain'          'i__db_domain_object_of'
  '_db_index'           'i__db_index_class_of'
  '_db_index_key'       'i__db_index_key_index_of'
  '_db_meth_arg'        'i__db_meth_arg_meth_sig_of'
  '_db_meth_file'       'i__db_meth_file_class_of'
  '_db_meth_sig'        'i__db_meth_sig_meth_of'
  '_db_method'          'i__db_method_class_of_meth_name'
  '_db_partition'       'i__db_partition_class_of_pname'
  '_db_query_spec'      'i__db_query_spec_class_of'
  '_db_stored_procedure'  'u__db_stored_procedure_sp_name'
  '_db_stored_procedure_args'  'i__db_stored_procedure_args_sp_name'
  'athlete'             'pk_athlete_code'
  'db_serial'           'pk_db_serial_name'
  'db_user'             'i_db_user_name'
  'event'               'pk_event_code'
  'game'                'pk_game_host_year_event_code_athlete_code'
  'game'                'fk_game_event_code'
  'game'                'fk_game_athlete_code'
  'history'             'pk_history_event_code_athlete'
  'nation'              'pk_nation_code'
  'olympic'             'pk_olympic_host_year'
  'participant'         'pk_participant_host_year_nation_code'
  'participant'         'fk_participant_host_year'
  'participant'         'fk_participant_nation_code'
  'record'              'pk_record_host_year_event_code_athlete_code_medal'
  'stadium'             'pk_stadium_code'
```

_db_index_key

Represents key information of the index. An index for index_of is created.

Attribute Name	Data Type	Description
index_of	_db_index	Index to which the key attribute belongs
key_attr_name	VARCHAR(255)	Name of the attribute that comprises the key
key_order	INTEGER	Order of the attribute in the key. Begins with 0.
asc_desc	INTEGER	1 if the order of attribute values is descending, and 0 otherwise.
key_prefix_length	INTEGER	Length of prefix to be used as a key

Example

The following is an example of retrieving names of indexes that belong to the class.

```
csql> select class_of.class_name, sequence(select key_attr_name
csql>                                      from _db_index_key k
csql>                                      where k in i.key_attrs)
csql> from _db_index i
csql> where key_count >= 2;
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

  class_of.class_name   sequence((select key_attr_name from _db_index_key k where k in
i.key_attrs))
============================================
  '_db_partition'       {'class_of', 'pname'}
  '_db_method'          {'class_of', 'meth_name'}
  '_db_attribute'       {'class_of', 'attr_name'}
  'participant'         {'host_year', 'nation_code'}
  'game'                {'host_year', 'event_code', 'athlete_code'}
  'record'              {'host_year', 'event_code', 'athlete_code', 'medal'}
  'history'             {'event_code', 'athlete'}
```

_db_auth

Represents user authorization information of the class. An index for the grantee is created.

Attribute Name	Data Type	Description
grantor	db_user	Authorization grantor
grantee	db_user	Authorization grantee
class_of	_db_class	Class object to which authorization is to be granted
auth_type	VARCHAR(7)	Type name of the authorization granted
is_grantable	INTEGER	1 if authorization for the class can be granted to other users, and 0 otherwise.

Authorization types supported by CUBRID are as follows:

- **SELECT**
- **INSERT**

- **UPDATE**
- **DELETE**
- **ALTER**
- **INDEX**
- **EXECUTE**

Example

The following is an example of retrieving the authorization information defined in the class 'db_trig'.

```
csql> select grantor.name, grantee.name, auth_type
csql> from _db_auth
csql> where class_of.class_name = 'db_trig';
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

  grantor.name          grantee.name          auth_type
==================================================================
  'DBA'                 'PUBLIC'              'SELECT'

1 rows selected.
```

_db_data_type

Represents the data type supported by CUBRID (see the "Data Types Supported by CUBRID" table in _db_attribute).

Attribute Name	Data Type	Description
type_id	INTEGER	Data type identifier. Corresponds to the "Value" column in the "Data Types Supported by CUBRID" table.
type_name	VARCHAR(9)	Data type name. Corresponds to the "Meaning" column in the "Data Types Supported by CUBRID" table.

Example

The following is an example of retrieving attributes and type names of the 'event' class.

```
csql> select a.attr_name, t.type_name
csql> from _db_attribute a join _db_data_type t
csql> on a.data_type = t.type_id
csql> where class_of.class_name = 'event'
csql> order by a.def_order;
csql> ;xrun


=== <Result of SELECT Command in Line 1> ===

  attr_name             type_name
============================================
  'code'                'INTEGER'
  'sports'              'STRING'
  'name'                'STRING'
  'gender'              'CHAR'
  'players'             'INTEGER'
```

_db_partition

Represents information about partitions. Indexes for class_of and pname are created.

Attribute Name	Data Type	Description
class_of	_db_class	OID of the parent class
pname	VARCHAR(255)	Parent - **NULL**
ptype	INTEGER	0 - HASH

		1 - RANGE 2 - LIST
pexpr	VARCHAR(255)	Parent only
pvalues	SEQUENCE OF	Parent - Column name, Hash size RANGE - MIN/MAX value : - Infinite MIN/MAX is saved as **NULL** LIST - value list

_db_stored_procedure

Represents information about Java stored procedures. An index for sp_name is created.

Attribute Name	Data Type	Description
sp_name	VARCHAR(255)	Stored procedure name
sp_type	INTEGER	Stored procedure type (function or procedure)
return_type	INTEGER	Return value type
arg_count	INTEGER	The number of arguments
args	SEQUENCE OF _db_stored_procedure_args	Argument list
lang	INTEGER	Implementation language (currently, Java)
target	VARCHAR(4096)	Name of the Java method to be executed
owner	db_user	Owner

_db_stored_procedure_args

Represents information about the Java stored procedure arguments. An index for sp_name is created.

Attribute Name	Data Type	Description
sp_name	VARCHAR(255)	Stored procedure name
index_of	INTEGER	Order of the arguments
arg_name	VARCHAR(255)	Argument name
data_type	INTEGER	Data type of the argument
mode	INTEGER	Mode (IN, OUT, INOUT)

db_user

Attribute Name	Data Type	Description
name	VARCHAR(1073741823)	User name
id	INTEGER	User identifier
password	db_password	User password. Not displayed to the user.
direct_groups	SET OF db_user	Groups to which the user belongs directly
groups	SET OF db_user	Groups to which the user belongs directly or indirectly
authorization	db_authorization	Information of the authorization owned by the user
triggers	SEQUENCE OF object	Triggers that occur due to user actions

Function Name

- **set_password**()
- **set_password_encoded**()
- **add_member**()
- **drop_member**()
- **print_authorizations**()
- **add_user**()
- **drop_user**()
- **find_user**()
- **login**()

db_authorization

Attribute Name	Data Type	Description
owner	db_user	User information
grants	SEQUENCE OF object	Sequence of {object for which the user has authorization, authorization grantor of the object, authorization type}

Method Name

- **check_authorization**(varchar(255), integer)

db_trigger

Attribute Name	Data Type	Description
owner	db_user	Trigger owner
name	VARCHAR(1073741823)	Trigger name
status	INTEGER	1 for INACTIVE, and 2 for ACTIVE. The default value is 2.
priority	DOUBLE	Execution priority between triggers. The default value is 0.
event	INTEGER	0 is set for UPDATE, 1 for UPDATE STATEMENT, 2 for DELETE, 3 for DELETE STATEMENT, 4 for INSERT, 5 for INSERT STATEMENT, 8 for COMMIT, and 9 for ROLLBACK.
target_class	object	Class object for the trigger target class
target_attribute	VARCHAR(1073741823)	Trigger target attribute name. If the target attribute is not specified, **NULL** is used.
target_class_attribute	INTEGER	If the target attribute is an instance attribute, 0 is used. If it is a class attribute, 1 is used. The default value is 0.
condition_type	INTEGER	1 for one of INSERT, UPDATE, DELETE, CALL and EVALUATE, 2 for REJECT, 3 for INVALIDATE_TRANSACTION, and 4 for PRINT
condition	VARCHAR(1073741823)	Action condition specified in the IF statement
condition_time	INTEGER	1 for BEFORE, 2 for AFTER, and 3 for DEFERRED.
action_type	INTEGER	1 for one of INSERT, UPDATE, DELETE, CALL and EVALUATE, 2 for REJECT, 3 for INVALIDATE_TRANSACTION, and 4 for PRINT.
action_definition	VARCHAR(1073741823)	Execution statement to be triggered

action_time	INTEGER	1 for BEFORE, 2 for AFTER, and 3 for DEFERRED.

db_ha_apply_info

A table that saves the progress status every time the **applylogdb** utility applies replication logs. This table is updated at every point the **applylogdb** utility commits, and the acculmative count of operations are stored in the *_counter column. The meaning of each column is as follows:

Column Name	Column Type	Meaning
db_name	VARCHAR(255)	Name of the database saved in the log
db_creation_time	DATETIME	Creation time of the source database for the log to be applied
copied_log_path	VARCHAR(4096)	Path to the log file to be applied
page_id	INTEGER	Page of the replication log committed in the slave database
offset	INTEGER	Offset of the replication log committed in the slave database
log_record_time	DATETIME	Timestamp included in replication log committed in the slave database, i.e. the creation time of the log
last_access_time	DATETIME	Time when applylogdb was committed in the slave database
insert_counter	BIGINT	Number of times that applylogdb was inserted
update_counter	BIGINT	Number of times that applylogdb was updated
delete_counter	BIGINT	Number of times that applylogdb was deleted
schema_counter	BIGINT	Number of times that applylogdb changed the schema
commit_counter	BIGINT	Number of times that applylogdb was committed
fail_counter	BIGINT	Number of times that applylogdb failed to be inserted/updated/deleted/committed and to change the schema
required_page_id	INTEGER	Minimum pageid that applylogdb can read
start_time	DATETIME	Time when the applylogdb process accessed the slave database
status	INTEGER	Progress status (0: IDLE, 1: BUSY)

System Catalog Virtual Class

System Catalog Virtual Class

General users can only see information of classes for which they have authorization through system catalog virtual classes.

This section explains which information each system catalog virtual class represents, and virtual class definition statements.

DB_CLASS

Represents information of the classes for which the current user has access authorization in the database.

Attribute Name	Data Type	Description
class_name	VARCHAR(255)	Class name
owner_name	VARCHAR(255)	Name of class owner
class_type	VARCHAR(6)	'CLASS' for a class, and 'VCLASS' for a virtual class

is_system_class	VARCHAR(3)	'YES' for a system class, and 'NO' otherwise.
partitioned	VARCHAR(3)	'YES' for a partitioned group class, and 'NO' otherwise.
is_reuse_oid_class	VARCHAR(3)	'YES' for a REUSE_OID class, and 'NO' otherwise.

Definition

```
CREATE VCLASS db_class (class_name, owner_name, class_type, is_system_class, partitioned,
is_reuse_oid_class)
AS

SELECT c.class_name, CAST(c.owner.name AS VARCHAR(255)),
       CASE c.class_type WHEN 0 THEN 'CLASS' WHEN 1 THEN 'VCLASS' ELSE 'UNKNOW' END,
       CASE WHEN MOD(c.is_system_class, 2) = 1 THEN 'YES' ELSE 'NO' END,
       CASE WHEN c.sub_classes IS NULL THEN 'NO' ELSE NVL((SELECT 'YES' FROM _db_partition
p WHERE p.class_of = c and p.pname IS NULL), 'NO') END,
       CASE WHEN MOD(c.is_system_class / 8, 2) = 1 THEN 'YES' ELSE 'NO' END FROM _db_class
c
WHERE CURRENT_USER = 'DBA' OR
                      {c.owner.name} SUBSETEQ (
                         SELECT SET{CURRENT_USER} + COALESCE(SUM(SET{t.g.name}), SET{})
                         FROM db_user u, TABLE(groups) AS t(g)
                         WHERE u.name = CURRENT_USER) OR
                       {c} SUBSETEQ (
                           SELECT SUM(SET{au.class_of})
                           FROM _db_auth au
                           WHERE {au.grantee.name} SUBSETEQ(
                                SELECT SET{CURRENT_USER} + COALESCE(SUM(SET{t.g.name}),
SET{})
                                FROM db_user u, TABLE(groups) AS t(g)
                                WHERE u.name = CURRENT_USER) AND au.auth_type =
'SELECT');
```

Example

The following is an example of retrieving classes owned by the current user.

```
csql> select class_name
csql> from db_class
csql> where owner_name = CURRENT_USER;
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

  class_name
======================
  'stadium'
  'code'
  'nation'
  'event'
  'athlete'
  'participant'
  'olympic'
  'game'
  'record'
  'history'
'female_event'
```

Note All examples of system catalog classes have been written in the csql utility. In this example, the user option is omitted (if omitted, the default user is **PUBLIC**). If not otherwise specified, **--no-auto-commit** (No auto-commit mode) and **-u** (Specify the user **dba**) options are used.

% csql --no-auto-commit -u dba demodb

The following is an example of retrieving virtual classes that can be accessed by the current user.

```
csql> select class_name
csql> from db_class
csql> where class_type = 'VCLASS';
csql> ;xrun
```

```
=== <Result of SELECT Command in Line 1> ===

  class_name
======================
  'db_stored_procedure_args'
  'db_stored_procedure'
  'db_partition'
  'db_trig'
  'db_auth'
  'db_index_key'
  'db_index'
  'db_meth_file'
  'db_meth_arg_setdomain_elm'
  'db_meth_arg'
  'db_method'
  'db_attr_setdomain_elm'
  'db_attribute'
  'db_vclass'
  'db_direct_super_class'
  'db_class'
```

The following is an example of retrieving system classes that can be accessed by the current user user (**PUBLIC** user).

```
csql> select class_name
csql> from db_class
csql> where is_system_class = 'YES' and
csql>       class_type = 'CLASS'
csql> order by 1;
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

  class_name
======================
  'db_authorization'
  'db_authorizations'
  'db_root'
  'db_serial'
  'db_user'
```

DB_DIRECT_SUPER_CLASS

Represents names of super classes (if any) of the class for which the current user has access authorization in the database.

Attribute Name	Data Type	Description
class_name	VARCHAR(255)	Class name
super_class_name	VARCHAR(255)	super class name

Definition

```
CREATE VCLASS db_direct_super_class (class_name, super_class_name)
AS
SELECT c.class_name, s.class_name
FROM _db_class c, TABLE(c.super_classes) AS t(s)
WHERE (CURRENT_USER = 'DBA' OR
              {c.owner.name} subseteq (
                              SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}),
set{})
                              from db_user u, table(groups) as t(g)
                              where u.name = CURRENT_USER ) OR
              {c} subseteq (
SELECT sum(set{au.class_of})
                              FROM _db_auth au
                              WHERE {au.grantee.name} subseteq (
                                                      SELECT set{CURRENT_USER} +
coalesce(sum(set{t.g.name}), set{})
                                                       from db_user u, table(groups) as t(g)
                                                      where u.name = CURRENT_USER ) AND
```

```
                                                                      au.auth_type =
'SELECT'))
```

Example

- The following is an example of retrieving super classes of the 'female_event' class (see ADD SUPERCLASS Clause).

```
csql> select super_class_name
csql> from db_direct_super_class
csql> where class_name = 'female_event'
csql> xrun

=== < Result of SELECT Command in Line 1> ===

  super_class_name
======================
  'event'
```

- The following is an example of retrieving super classes of the class owned by the current user (**PUBLIC** user).

```
csql> select c.class_name, s.super_class_name
csql> from db_class c, db_direct_super_class s
csql> where c.class_name = s.class_name and
csql>             c.owner_name = user
csql> order by 1
csql> xrun

=== < Result of SELECT Command in Line 1> ===

  class_name                      super_class_name
============================================
  'female_event'              'event'
```

DB_VCLASS

Represents SQL definition statements of virtual classes for which the current user has access authorization in the database.

Attribute Name	Data Type	Description
vclass_name	VARCHAR(255)	Virtual class name
vclass_def	VARCHAR(4096)	SQL definition statement of the virtual class

Definition

```
CREATE VCLASS db_vclass (vclass_name, vclass_def)
AS
SELECT q.class_of.class_name, q.spec
FROM _db_query_spec q
WHERE CURRENT_USER = 'DBA' OR
        {q.class_of.owner.name} subseteq (
                SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                from db_user u, table(groups) as t(g)
                where u.name = CURRENT_USER ) OR
        {q.class_of} subseteq (
SELECT sum(set{au.class_of})
                FROM _db_auth au
                WHERE {au.grantee.name} subseteq (
                            SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                            from db_user u, table(groups) as t(g)
                            where u.name = CURRENT_USER ) AND
                                    au.auth_type = 'SELECT');
```

Example

The following is an example of retrieving SQL definition statements of the 'db_class' virtual class.

```
csql> select vclass_def
csql> from db_vclass
csql> where vclass_name = 'db_class';
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

'SELECT c.class_name, CAST(c.owner.name AS VARCHAR(255)), CASE c.class_type WHEN 0 THEN
'CLASS' WHEN 1 THEN 'VCLASS' WHEN 2 THEN 'PROXY' ELSE 'UNKNOW' END, CASE WHEN
MOD(c.is_system_class, 2) = 1 THEN 'YES' ELSE 'NO' END, CASE WHEN c.sub_classes IS NULL
THEN 'NO' ELSE NVL((SELECT 'YES' FROM _db_partition p WHERE p.class_of = c and p.pname IS
NULL), 'NO') END FROM _db_class c WHERE CURRENT_USER = 'DBA' OR {c.owner.name} SUBSETEQ
(  SELECT SET{CURRENT_USER} + COALESCE(SUM(SET{t.g.name}), SET{})  FROM db_user u,
TABLE(groups) AS t(g)  WHERE u.name = CURRENT_USER) OR {c} SUBSETEQ (  SELECT
SUM(SET{au.class_of})  FROM _db_auth au  WHERE {au.grantee.name} SUBSETEQ (  SELECT
SET{CURRENT_USER} + COALESCE(SUM(SET{t.g.name}), SET{})  FROM db_user u, TABLE(groups) AS
t(g)  WHERE u.name = CURRENT_USER) AND  au.auth_type = 'SELECT')'
```

DB_ATTRIBUTE

Represents the attribute information of the class for which the current user has access authorization in the database.

Attribute Name	Data Type	Description
attr_name	VARCHAR(255)	Attribute name
class_name	VARCHAR(255)	Name of the class to which the attribute belongs
attr_type	VARCHAR(8)	'INSTANCE' for an instance attribute, 'CLASS' for a class attribute, and 'SHARED' for a shared attribute.
def_order	INTEGER	Order of attributes in the class. Begins with 0. If the attribute is inherited, the order is the one defined in the super class.
from_class_name	VARCHAR(255)	If the attribute is inherited, the super class in which it is defined is used. Otherwise, **NULL**
from_attr_name	VARCHAR(255)	If the attribute is inherited and its name is changed to resolve a name conflict, the original name defined in the super class is used. Otherwise, **NULL**
data_type	VARCHAR(9)	Data type of the attribute (one in the "Meaning" column of the "Data Types Supported by CUBRID" table in _db_attribute)
prec	INTEGER	Precision of the data type. 0 is used if the precision is not specified.
scale	INTEGER	Scale of the data type. 0 is used if the scale is not specified.
code_set	INTEGER	Character set (value of table "character sets supported by CUBRID" in _db_attribute) if it is string type. 0 otherwise.
domain_class_name	VARCHAR(255)	Domain class name if the data type is an object. **NULL** otherwise.
default_value	VARCHAR(255)	Saved as a character string by default, regardless of data types. If no default value is specified, **NULL** is saved if a default value is **NULL**, it is displayed as 'NULL'. An object data type is represented as 'volume id | page id | slot id' while a set data type is represented as '{element 1, element 2, ... '.
is_nullable	VARCHAR(3)	'NO' if a not null constraint is set, and 'YES' otherwise.

Definition

```
CREATE VCLASS db_attribute (
attr_name, class_name, attr_type, def_order, from_class_name, from_attr_name, data_type,
prec, scale, code_set, domain_class_name, default_value, is_nullable)
AS
SELECT a.attr_name, c.class_name,
            CASE WHEN a.attr_type = 0 THEN 'INSTANCE'
                      WHEN a.attr_type = 1 THEN 'CLASS'
                      ELSE 'SHARED' END,
            a.def_order, a.from_class_of.class_name, a.from_attr_name, t.type_name,
            d.prec, d.scale, d.code_set, d.class_of.class_name, a.default_value,
            CASE WHEN a.is_nullable = 0 THEN 'YES' ELSE 'NO' END
FROM _db_class c, _db_attribute a, _db_domain d, _db_data_type t
WHERE a.class_of = c AND d.object_of = a AND d.data_type = t.type_id AND
              (CURRENT_USER = 'DBA' OR
              {c.owner.name} subseteq (
                              SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}),
set{})
                              from db_user u, table(groups) as t(g)
                              where u.name = CURRENT_USER ) OR
              {c} subseteq (
SELECT sum(set{au.class_of})
                              FROM _db_auth au
                              WHERE {au.grantee.name} subseteq (
                                                      SELECT set{CURRENT_USER} +
coalesce(sum(set{t.g.name}), set{})
                                                       from db_user u, table(groups) as t(g)
                                                      where u.name = CURRENT_USER ) AND
                                                                      au.auth_type =
'SELECT'))
```

Example

- The following is an example of retrieving attributes and data types of the ‘event’ class.

```
csql> select attr_name, data_type, domain_class_name
csql> from db_attribute
csql> where class_name = 'event'
csql> order by def_order
csql> xrun

=== < Result of SELECT Command in Line 1> ===

  attr_name                          data_type                          domain_class_name
==================================================================
  'code'                                'INTEGER'                           NULL
  'sports'                            'STRING'                            NULL
  'name'                                'STRING'                            NULL
  'gender'                            'CHAR'                                NULL
  'players'                          'INTEGER'                         NULL
```

- The following is an example of retrieving attributes of the ‘female_event’ class and its super class.

```
csql> select attr_name, from_class_name
csql> from db_attribute
csql> where class_name = 'female_event'
csql> order by def_order
csql> xrun

=== < Result of SELECT Command in Line 1> ===

  attr_name                          from_class_name
============================================
  'code'                                'event'
  'sports'                            'event'
  'name'                                'event'
  'gender'                            'event'
  'players'                          'event'
```

- The following is an example of retrieving classes whose attribute names are similar to 'name,' among the ones owned by the current user. (The user is **PUBLIC**.)

```
csql> select a.class_name, a.attr_name
csql> from db_class c join db_attribute a
csql> on c.class_name = a.class_name
csql> where c.owner_name = CURRENT_USER and
csql>             attr_name like '%name%'
csql> order by 1
csql> xrun

=== < Result of SELECT Command in Line 1> ===

  class_name                        attr_name
```

```
=============================================
  'athlete'                           'name'
  'code'                                 'f_name'
  'code'                                 's_name'
  'event'                               'name'
  'female_event'                 'name'
  'nation'                             'name'
  'stadium'                           'name'
```

DB_ATTR_SETDOMAIN_ELM

Among attributes of the class to which the current user has access authorization in the database, if an attribute's data type is a set (set, multiset, sequence), this macro represents the data type of the element of the set.

Attribute Name	Data Type	Description
attr_name	VARCHAR(255)	Attribute name
class_name	VARCHAR(255)	Name of the class to which the attribute belongs
attr_type	VARCHAR(8)	ʹINSTANCEʹ for an instance attribute, 'CLASS' for a class attribute, and 'SHARED' for a shared attribute.
data_type	VARCHAR(9)	Data type of the element
prec	INTEGER	Precision of the data type of the element
scale	INTEGER	Scale of the data type of the element
code_set	INTEGER	Character set if the data type of the element is a character
domain_class_name	VARCHAR(255)	Domain class name if the data type of the element is an object

Definition

```
CREATE VCLASS db_attr_setdomain_elm (
attr_name, class_name, attr_type,data_type, prec, scale, code_set, domain_class_name)
AS
SELECT a.attr_name, c.class_name,
       CASE WHEN a.attr_type = 0 THEN 'INSTANCE'
            WHEN a.attr_type = 1 THEN 'CLASS'
            ELSE 'SHARED' END,
       et.type_name, e.prec, e.scale, e.code_set, e.class_of.class_name
FROM _db_class c, _db_attribute a, _db_domain d,
      TABLE(d.set_domains) AS t(e), _db_data_type et
WHERE a.class_of = c AND d.object_of = a AND e.data_type = et.type_id AND
        (CURRENT_USER = 'DBA' OR
        {c.owner.name} subseteq (
                SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                from db_user u, table(groups) as t(g)
                where u.name = CURRENT_USER ) OR
        {c} subseteq (
SELECT sum(set{au.class_of})
                FROM _db_auth au
                WHERE {au.grantee.name} subseteq (
                            SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                            from db_user u, table(groups) as t(g)
                            where u.name = CURRENT_USER ) AND
                                    au.auth_type = 'SELECT'));
```

If the set_attr attribute of class D is of a SET (A, B, C) type, the following three records exist.

Attr_name	Class_name	Attr_type	Data_type	prec	Scale	Code_set	Domain_class_name
ʹset_attrʹ	ʹDʹ	ʹINSTANCEʹ	ʹSETʹ	0	0	0	ʹAʹ
ʹset_attrʹ	ʹDʹ	ʹINSTANCEʹ	ʹSETʹ	0	0	0	ʹBʹ
ʹset_attrʹ	ʹDʹ	ʹINSTANCEʹ	ʹSETʹ	0	0	0	ʹCʹ

Example

The following is an example of retrieving set type attributes and data types of the 'city' class. (The city table defined in Containment Operators is created.)

```
csql> select attr_name, attr_type, data_type, domain_class_name
csql> from db_attr_setdomain_elm
csql> where class_name = 'city';
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

attr_name             attr_type             data_type             domain_class_name
==============================================================================

'sports'              'INSTANCE'            'STRING'              NULL
```

DB_METHOD

Represents the method information of the class for which the current user has access authorization in the database.

Attribute Name	Data Type	Description
meth_name	VARCHAR(255)	Method name
class_name	VARCHAR(255)	Name of the class to which the method belongs
meth_type	VARCHAR(8)	'INSTANCE' for an instance method, and 'CLASS' for a class method.
from_class_name	VARCHAR(255)	If the method is inherited, the super class in which it is defined is used otherwise **NULL**
from_meth_name	VARCHAR(255)	If the method is inherited and its name is changed to resolve a name conflict, the original name defined in the super class is used otherwise **NULL**
func_name	VARCHAR(255)	Name of the C function for the method

Definition

```
CREATE VCLASS db_method (
meth_name, class_name, meth_type, from_class_name, from_meth_name, func_name)
AS

SELECT m.meth_name, m.class_of.class_name,
            CASE WHEN m.meth_type = 0 THEN 'INSTANCE' ELSE 'CLASS' END,
            m.from_class_of.class_name, m.from_meth_name, s.func_name
FROM _db_method m, _db_meth_sig s
WHERE s.meth_of = m AND
              (CURRENT_USER = 'DBA' OR
              {m.class_of.owner.name} subseteq (
                              SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}),
set{})
                              from db_user u, table(groups) as t(g)
                              where u.name = CURRENT_USER ) OR
              {m.class_of} subseteq (
SELECT sum(set{au.class_of})
                              FROM _db_auth au
                              WHERE {au.grantee.name} subseteq (
                                                      SELECT set{CURRENT_USER} +
coalesce(sum(set{t.g.name}), set{})
                                                       from db_user u, table(groups) as t(g)
                                                      where u.name = CURRENT_USER ) AND
                                                                      au.auth_type =
'SELECT'))
```

Example

The following is an example of retrieving methods of the 'db_user' class.

```
csql> select meth_name, meth_type, func_name
csql> from db_method
csql> where class_name = 'db_user'
csql> order by meth_type, meth_name
csql> xrun

=== < Result of SELECT Command in Line 1> ===
  meth_name                    meth_type          func_name
==================================================================
  'add_user'                   'CLASS'            'au_add_user_method'
  'drop_user'                  'CLASS'            'au_drop_user_method'
  'find_user'                  'CLASS'            'au_find_user_method'
  'login'                      'CLASS'            'au_login_method'
  'add_member'                 'INSTANCE'         'au_add_member_method'
  'drop_member'                'INSTANCE'         'au_drop_member_method'
  'print_authorizations'       'INSTANCE'         'au_describe_user_method'
  'set_password'               'INSTANCE'         'au_set_password_method'
  'set_password_encoded'       'INSTANCE'         'au_set_password_encoded_method'
  'set_password_encoded_sha1'  'INSTANCE'         'au_set_password_encoded_sha1_method' 10
rows selected.
```

DB_METH_ARG

Represents the input/output argument information of the method of the class for which the current user has access authorization in the database.

Attribute Name	Data Type	Description
meth_name	VARCHAR(255)	Method name
class_name	VARCHAR(255)	Name of the class to which the method belongs
meth_type	VARCHAR(8)	'INSTANCE' for an instance method, and 'CLASS' for a class method.
index_of	INTEGER	Order in which arguments are listed in the function definition. Begins with 0 if it is a return value, and 1 if it is an input argument.
data_type	VARCHAR(9)	Data type of the argument
prec	INTEGER	Precision of the argument
scale	INTEGER	Scale of the argument
code_set	INTEGER	Character set if the data type of the argument is a character.
domain_class_name	VARCHAR(255)	Domain class name if the data type of the argument is an object.

Definition

```
CREATE VCLASS db_meth_arg (
meth_name, class_name, meth_type,
index_of, data_type, prec, scale, code_set, domain_class_name)
AS
SELECT s.meth_of.meth_name, s.meth_of.class_of.class_name,
       CASE WHEN s.meth_of.meth_type = 0 THEN 'INSTANCE' ELSE 'CLASS' END,
       a.index_of, t.type_name, d.prec, d.scale, d.code_set,
       d.class_of.class_name
FROM _db_meth_sig s, _db_meth_arg a, _db_domain d, _db_data_type t
WHERE a.meth_sig_of = s AND d.object_of = a AND d.data_type = t.type_id AND
        (CURRENT_USER = 'DBA' OR
        {s.meth_of.class_of.owner.name} subseteq (
                SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                from db_user u, table(groups) as t(g)
                where u.name = CURRENT_USER ) OR
        {s.meth_of.class_of} subseteq (
SELECT sum(set{au.class_of})
                FROM _db_auth au
                WHERE {au.grantee.name} subseteq (
                            SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                            from db_user u, table(groups) as t(g)
                            where u.name = CURRENT_USER ) AND
                                    au.auth_type = 'SELECT'));
```

Example

The following is an example of retrieving input arguments of the method of the 'db_user' class.

```
csql> select meth_name, data_type, prec
csql> from db_meth_arg
csql> where class_name = 'db_user;
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

  meth_name             data_type                    prec
=========================================================
  'append_data'         'STRING'               1073741823
```

DB_METH_ARG_SETDOMAIN_ELM

If the data type of the input/output argument of the method of the class is a set, for which the current user has access authorization in the database, this macro represents the data type of the element of the set.

Attribute Name	Data Type	Description
meth_name	VARCHAR(255)	Method name
class_name	VARCHAR(255)	Name of the class to which the method belongs
meth_type	VARCHAR(8)	'INSTANCE' for an instance method, and 'CLASS' for a class method.
index_of	INTEGER	Order of arguments listed in the function definition. Begins with 0 if it is a return value, and 1 if it is an input argument.
data_type	VARCHAR(9)	Data type of the element
prec	INTEGER	Precision of the element
scale	INTEGER	Scale of the element
code_set	INTEGER	Character set if the data type of the element is a character
domain_class_name	VARCHAR(255)	Domain class name if the data type of the element is an object

Definition

```
CREATE VCLASS db_meth_arg_setdomain_elm(
meth_name, class_name, meth_type,
index_of, data_type, prec, scale, code_set, domain_class_name)
AS
SELECT s.meth_of.meth_name, s.meth_of.class_of.class_name,
       CASE WHEN s.meth_of.meth_type = 0 THEN 'INSTANCE' ELSE 'CLASS' END,
       a.index_of, et.type_name, e.prec, e.scale, e.code_set,
       e.class_of.class_name
FROM _db_meth_sig s, _db_meth_arg a, _db_domain d,
      TABLE(d.set_domains) AS t(e), _db_data_type et
WHERE a.meth_sig_of = s AND d.object_of = a AND e.data_type = et.type_id AND
        (CURRENT_USER = 'DBA' OR
        {s.meth_of.class_of.owner.name} subseteq (
                SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                from db_user u, table(groups) as t(g)
                where u.name = CURRENT_USER ) OR
        {s.meth_of.class_of} subseteq (
SELECT sum(set{au.class_of})
                FROM _db_auth au
                WHERE {au.grantee.name} subseteq (
                            SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                            from db_user u, table(groups) as t(g)
                            where u.name = CURRENT_USER ) AND
                                    au.auth_type = 'SELECT'));
```

DB_METH_FILE

Represents information of the file where the method of the class for which the current user has access authorization in the database is defined.

Attribute Name	Data Type	Description
class_name	VARCHAR(255)	Name of the class to which the method file belongs
path_name	VARCHAR(255)	File path in which the C function is defined
from_class_name	VARCHAR(255)	Name of the super class in which the method file is defined if the method is inherited, and otherwise **NULL**

Definition

```
CREATE VCLASS db_meth_file (class_name, path_name, from_class_name)
AS
SELECT f.class_of.class_name, f.path_name, f.from_class_of.class_name
FROM _db_meth_file f
WHERE (CURRENT_USER = 'DBA' OR
              {f.class_of.owner.name} subseteq (
                              SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), 
set{})
                              from db_user u, table(groups) as t(g)
                              where u.name = CURRENT_USER ) OR
              {f.class_of} subseteq (
SELECT sum(set{au.class_of})
                              FROM _db_auth au
                              WHERE {au.grantee.name} subseteq (
                                                      SELECT set{CURRENT_USER} + 
coalesce(sum(set{t.g.name}), set{})
                                                       from db_user u, table(groups) as t(g)
                                                      where u.name = CURRENT_USER ) AND
                                                                      au.auth_type = 
'SELECT'))
```

DB_INDEX

Represents information of indexes created for the class for which the current user has access authorization in the database.

Attribute Name	Data Type	Description
index_name	VARCHAR(255)	Index name
is_unique	VARCHAR(3)	′YES′ for a unique index, and ′NO′ otherwise.
is_reverse	VARCHAR(3)	'YES′ for a reversed index, and ′NO′ otherwise.
class_name	VARCHAR(255)	Name of the class to which the index belongs
key_count	INTEGER	The number of attributes that comprise the key
is_primary_key	VARCHAR(3)	'YES' for a primary key, and ′NO′ otherwise.
is_foreign_key	VARCHAR(3)	'YES' for a foreign key, and ′NO′ otherwise.

Definition

```
CREATE VCLASS db_index (index_name, is_unique, is_reverse, class_name, key_count, 
is_primary_key, is_foreign_key)
AS
SELECT i.index_name, CASE WHEN i.is_unique = 0 THEN 'NO' ELSE 'YES' END,
CASE WHEN i.is_reverse = 0 THEN 'NO' ELSE 'YES' END, i.class_of.class_name, i.key_count,
CASE WHEN i.is_primary_key = 0 THEN 'NO' ELSE 'YES' END, CASE WHEN i.is_foreign_key = 0 
THEN 'NO' ELSE 'YES' END
FROM _db_index i
WHERE (CURRENT_USER = 'DBA' OR
        {i.class_of.owner.name} subseteq (
                SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                from db_user u, table(groups) as t(g)
                where u.name = CURRENT_USER ) OR
        {i.class_of} subseteq (
SELECT sum(set{au.class_of})
                FROM _db_auth au
                WHERE {au.grantee.name} subseteq (
                            SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                            from db_user u, table(groups) as t(g)
                            where u.name = CURRENT_USER ) AND
                                    au.auth_type = 'SELECT'));
```

Example

The following is an example of retrieving index information of the class.

```
csql> select class_name, index_name, is_unique
csql> from db_index
csql> order by 1;
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

  class_name            index_name            is_unique
==================================================================
  'athlete'             'pk_athlete_code'     'YES'
  'city'                'pk_city_city_name'   'YES'
  'db_serial'           'pk_db_serial_name'   'YES'
  'db_user'             'i_db_user_name'      'NO'
  'event'               'pk_event_code'       'YES'
  'female_event'        'pk_event_code'       'YES'
  'game'                'pk_game_host_year_event_code_athlete_code'  'YES'
  'game'                'fk_game_event_code'  'NO'
  'game'                'fk_game_athlete_code'  'NO'
  'history'             'pk_history_event_code_athlete'  'YES'
  'nation'              'pk_nation_code'      'YES'
  'olympic'             'pk_olympic_host_year'  'YES'
  'participant'         'pk_participant_host_year_nation_code'  'YES'
  'participant'         'fk_participant_host_year'  'NO'
  'participant'         'fk_participant_nation_code'  'NO'
  'record'              'pk_record_host_year_event_code_athlete_code_medal'  'YES'
  'stadium'             'pk_stadium_code'     'YES'
```

DB_INDEX_KEY

Represents the key information of indexes created for the class for which the current user has access authorization in the database.

Attribute Name	Data Type	Description
index_name	VARCHAR(255)	Index name
class_name	VARCHAR(255)	Name of the class to which the index belongs
key_attr_name	VARCHAR(255)	Name of attributes that comprise the key
key_order	INTEGER	Order of attributes in the key. Begins with 0.
asc_desc	VARCHAR(4)	'DESC' if the order of attribute values is descending, and 'ASC' otherwise.
key_prefix_length	INTEGER	Length of prefix to be used as a key

Definition

```
CREATE VCLASS db_index_key (index_name, class_name, key_attr_name, key_order,
key_prefix_length)
AS
SELECT k.index_of.index_name, k.index_of.class_of.class_name, k.key_attr_name, k.key_order
CASE k.asc_desc
WHEN 0 THEN 'ASC'
WHEN 1 THEN 'DESC' ELSE 'UNKN' END,
k.key_prefix_length
FROM _db_index_key k
WHERE (CURRENT_USER = 'DBA' OR
    {k.index_of.class_of.owner.name}
    subseteq (
        SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
        from db_user u, table(groups) as t(g)
        where u.name = CURRENT_USER ) OR {k.index_of.class_of}
        subseteq (
            SELECT sum(set{au.class_of})
            FROM _db_auth au
            WHERE {au.grantee.name} subseteq (
                SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                from db_user u, table(groups) as t(g)
                where u.name = CURRENT_USER ) AND
            au.auth_type = 'SELECT'));
```

Example

The following is an example of retrieving index key information of the class.

```
csql> select class_name, key_attr_name, index_name
csql> from db_index_key
csql> order by class_name, key_order;
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

  'athlete'             'code'                'pk_athlete_code'
  'city'                'city_name'           'pk_city_city_name'
  'db_serial'           'name'                'pk_db_serial_name'
  'db_user'             'name'                'i_db_user_name'
  'event'               'code'                'pk_event_code'
  'female_event'        'code'                'pk_event_code'
  'game'                'host_year'           'pk_game_host_year_event_code_athlete_code'
  'game'                'event_code'          'fk_game_event_code'
  'game'                'athlete_code'        'fk_game_athlete_code'
```

DB_AUTH

Represents authorization information of the classes for which the current user has authorization in the database.

Attribute Name	Data Type	Description
grantor_name	VARCHAR(255)	Name of the user who grants authorization
grantee_name	VARCHAR(255)	Name of the user who is granted authorization
class_name	VARCHAR(255)	Name of the class for which authorization is to be granted
auth_type	VARCHAR(7)	Name of the authorization type granted
is_grantable	VARCHAR(3)	'YES' if authorization for the class can be granted to other users, and 'NO' otherwise.

Definition

```
CREATE VCLASS db_auth (grantor_name, grantee_name, class_name, auth_type, is_grantable )
AS
SELECT CAST(a.grantor.name AS VARCHAR(255)),
        CAST(a.grantee.name AS VARCHAR(255)),
        a.class_of.class_name, a.auth_type,
        CASE WHEN a.is_grantable = 0 THEN 'NO' ELSE 'YES' END
FROM _db_auth a
WHERE (CURRENT_USER = 'DBA' OR
        {a.class_of.owner.name} subseteq (
                SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                from db_user u, table(groups) as t(g)
                where u.name = CURRENT_USER ) OR
        {a.class_of} subseteq (
SELECT sum(set{au.class_of})
```

```
                FROM _db_auth au
                WHERE {au.grantee.name} subseteq (
                            SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                            from db_user u, table(groups) as t(g)
                            where u.name = CURRENT_USER ) AND
                                    au.auth_type = 'SELECT'));
```

Example

The following is an example of retrieving authorization information of the classes whose names begin with 'db_a'.

```
csql> select class_name, auth_type, grantor_name
csql> from db_auth
csql> where class_name like 'db_a%'
csql> order by 1;
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

  class_name            auth_type             grantor_name
==================================================================
  'db_attr_setdomain_elm'  'SELECT'             'DBA'
  'db_attribute'           'SELECT'             'DBA'
  'db_auth'                'SELECT'             'DBA'
  'db_authorization'       'EXECUTE'            'DBA'
  'db_authorization'       'SELECT'             'DBA'
  'db_authorizations'      'EXECUTE'            'DBA'
  'db_authorizations'      'SELECT'             'DBA'
```

DB_TRIG

Represents information of the trigger that has the class for which the current user has access authorization in the database, or its attribute as the target.

Attribute Name	Data Type	Description
trigger_name	VARCHAR(255)	Trigger name
target_class_name	VARCHAR(255)	Target class
target_attr_name	VARCHAR(255)	Target attribute. If not specified in the trigger, **NULL**
target_attr_type	VARCHAR(8)	Target attribute type. If specified, 'INSTANCE' is used for an instance attribute, and 'CLASS' is used for a class attribute.
action_type	INTEGER	1 for one of INSERT, UPDATE, DELETE, CALL and EVALUATE, 2 for REJECT, 3 for INVALIDATE_TRANSACTION, and 4 for PRINT.
action_time	INTEGER	1 for BEFORE, 2 for AFTER, and 3 for DEFERRED.

Example

- The following is an example of showing information of the trigger that has the class for which the current user has access authorization, or its attribute as the target.

```
CREATE VCLASS db_trig (
trigger_name, target_class_name, target_attr_name, target_attr_type, action_type,
action_time)
AS
SELECT CAST(t.name AS VARCHAR(255)), c.class_name,
        CAST(t.target_attribute AS VARCHAR(255)),
        CASE WHEN t.target_class_attribute = 0 THEN 'INSTANCE' ELSE 'CLASS' END,
        t.action_type, t.action_time
FROM _db_class c, db_trigger t
WHERE t.target_class = c.class_of AND
        (CURRENT_USER = 'DBA' OR
        {c.owner.name} subseteq (
                SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                from db_user u, table(groups) as t(g)
                where u.name = CURRENT_USER ) OR
        {c} subseteq (
SELECT sum(set{au.class_of})
                FROM _db_auth au
                WHERE {au.grantee.name} subseteq (
                            SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
                            from db_user u, table(groups) as t(g)
                            where u.name = CURRENT_USER ) AND
                                    au.auth_type = 'SELECT'));
```

DB_PARTITION

Represents information of partitioned classes for which the current user has access authorization in the database.

Attribute Name	Data Type	Description
class_name	VARCHAR(255)	Class name
partition_name	VARCHAR(255)	Partition name
partition_class_name	VARCHAR(255)	Partitioned class name
partition_type	VARCHAR(32)	Partition type (HASH, RANGE, LIST)
partition_expr	VARCHAR(255)	Partition expression
partition_values	SEQUENCE OF	RANGE ? MIN/MAX value - For infinite MIN/MAX, **NULL** LIST - value list

Definition

```
CREATE VCLASS db_partition
(sp_name, sp_type, return_type, arg_count, lang, target, owner)
AS
SELECT p.class_of.class_name AS class_name, p.pname AS partition_name,
            p.class_of.class_name || '__p__' || p.pname AS partition_class_name,
            CASE WHEN p.ptype = 0 THEN 'HASH'
                 WHEN p.ptype = 1 THEN 'RANGE'
            ELSE 'LIST' ENDASpartition_type,
            TRIM(SUBSTRING( pi.pexpr FROM 8 FOR (POSITION(' FROM ' IN pi.pexpr)-8))) AS
               partition_expression,
            p.pvalues AS partition_values
FROM _db_partition p,
     ( select * from _db_partition sp
where sp.class_of =  p.class_of AND sp.pname is null) pi
WHERE p.pname is not null AND
      ( CURRENT_USER = 'DBA'
        OR
        {p.class_of.owner.name} SUBSETEQ
         ( SELECT SET{CURRENT_USER} + COALESCE(SUM(SET{t.g.name}), SET{})
           FROM db_user u, TABLE(groups) AS t(g)
           WHERE u.name = CURRENT_USER
         )
        OR
        {p.class_of} SUBSETEQ
         ( SELECT SUM(SET{au.class_of})
           FROM _db_auth au
           WHERE {au.grantee.name} SUBSETEQ
                 ( SELECT SET{CURRENT_USER} + COALESCE(SUM(SET{t.g.name}), SET{})
                   FROM db_user u, TABLE(groups) AS t(g)
                   WHERE u.name = CURRENT_USER) AND
                   au.auth_type = 'SELECT'
         )
      )
```

Example

The following is an example of retrieving the partition information currently configured for the participant2 class (see examples in Defining Range Partitions).

```
csql> select * from db_partition where class_name = 'participant2';
csql> ;x
```

```
=== <Result of SELECT Command in Line 2> ===

  class_name            partition_name        partition_class_name         partition_type
  partition_expr        partition_values
==========================================================================================
==========================================
  'participant2'        'before_2000'         'participant2__p__before_2000'  'RANGE'
  'host_year'           {NULL, 2000}
  'participant2'        'before_2008'         'participant2__p__before_2008'  'RANGE'
  'host_year'           {2000, 2008}
```

DB_STORED_PROCEDURE

Represents information of Java stored procedures for which the current user has access authorization in the database.

Attribute Name	Data Type	Description
sp_name	VARCHAR(255)	Stored procedure name
sp_type	VARCHAR(16)	Stored procedure type (function or procedure)
return_type	VARCHAR(16)	Return value type
arg_count	INTEGER	The number of arguments
lang	VARCHAR(16)	Implementing language (currently, Java)
target	VARCHAR(4096)	Name of the Java method to be executed
owner	VARCHAR(256)	Owner

Definition

```
CREATE VCLASS db_stored_procedure
(sp_name, sp_type, return_type, arg_count, lang, target, owner)
AS
SELECT sp.sp_name,
            CASE sp.sp_type   WHEN 1 THEN 'PROCEDURE'
            ELSE 'FUNCTION' END,
            CASE WHEN sp.return_type = 0 THEN 'void'
                 WHEN sp.return_type = 28 THEN 'CURSOR'
            ELSE ( SELECT dt.type_name
                   FROM _db_data_type dt
                   WHERE sp.return_type = dt.type_id) END,
           sp.arg_count,
           CASE sp.lang   WHEN 1 THEN 'JAVA'
           ELSE '' END, sp.target, sp.owner.name
FROM _db_stored_procedure sp
```

Example

The following is an example of retrieving Java stored procedures owned by the current user.

```
csql> select sp_name, target from db_stored_procedure
csql> where sp_type = 'FUNCTION'
csql> and owner = CURRENT_USER
csql> ;x

=== <Result of SELECT Command in Line 3> ===

  sp_name               target
============================================
  'hello'               'SpCubrid.HelloCubrid() return java.lang.String'
  'sp_int'              'SpCubrid.SpInt(int) return int'
```

DB_STORED_PROCEDURE_ARGS

Represents the argument information of Java stored procedures for which the current user has access authorization in the database.

Attribute Name	Data Type	Description
sp_name	VARCHAR(255)	Stored procedure name
index_of	INTEGER	Order of the arguments
arg_name	VARCHAR(256)	Argument name
data_type	VARCHAR(16)	Data type of the argument
mode	VARCHAR(6)	Mode (IN, OUT, INOUT)

Definition

```
CREATE VCLASS db_stored_procedure_args (sp_name, index_of, arg_name, data_type, mode)
AS
SELECT sp.sp_name, sp.index_of, sp.arg_name,
            CASE sp.data_type   WHEN 28 THEN 'CURSOR'
            ELSE ( SELECT dt.type_name FROM _db_data_type dt
                   WHERE sp.data_type = dt.type_id) END,
            CASE WHEN sp.mode = 1 THEN 'IN' WHEN sp.mode = 2 THEN 'OUT'
            ELSE 'INOUT' END
FROM _db_stored_procedure_args sp
ORDER BY sp.sp_name, sp.index_of ;
```

Example

The following is an example of retrieving arguments the 'phone_info' Java stored procedure in the order of the arguments.

```
csql> select index_of, arg_name, data_type, mode
csql> from  db_stored_procedure_args
csql> where sp_name = 'phone_info'
csql> order by index_of
csql> ;x

=== <Result of SELECT Command in Line 3> ===

     index_of  arg_name              data_type             mode
===============================================================
            0  'name'                'STRING'              'IN'
            1  'phoneno'             'STRING'              'IN'
```

Catalog Class/Virtual Class Authorization

Catalog classes are created to be owned by **DBA**. However, **DBA** can only execute **SELECT** operations. If **DBA** executes operations such as **UPDATE**/**DELETE**, an authorization failure error occurs. General users cannot execute queries on system catalog classes.

Although catalog virtual classes are created to be owned by **DBA**, all users can perform the **SELECT** statement on catalog virtual classes. Of course, **UPDATE**/**DELETE** operations on catalog virtual classes are not allowed.

Updating catalog classes/virtual classes is automatically performed by the system when users execute a DDL statement that creates/modifies/deletes a class/attribute/index/user/authorization.

Consistency of Catalog Information

Catalog information is represented by the instance of a catalog class/virtual class. If such information is accessed at the **READ UNCOMMITTED INSTANCES** (**TRAN_REP_CLASS_UNCOMMIT_INSTANCE** or **TRAN_COMMIT_CLASS_UNCOMMIT_INSTANCE**) isolation level, incorrect values (values being changed) can be read. Therefore, to get correct catalog information, you must use the **SELECT** query on the catalog class/virtual class at the **READ COMMITTED INSTANCES** isolation level or higher.

Querying on Catalog

To query on catalog classes, you must convert identifiers such as class, virtual class, attribute, trigger, method and index names to lowercases, and create them. Therefore, you must use lowercases when querying on catalog classes.

```
CREATE TABLE Foo(name varchar(255));
SELECT class_name, partitioned FROM db_class WHERE class_name = 'Foo';

=== <Result of SELECT Command in Line 1> ===

There are no results.

0 rows selected.

SELECT class_name, partitioned FROM db_class WHERE class_name = 'foo';

=== <Result of SELECT Command in Line 1> ===

  class_name   partitioned
============================
  'foo'       'NO'

1 rows selected.
```

Administrator's Guide

The "Administrator's Guide" provides the database administrators (**DBA**) with details on how to operate the CUBRID system. The guide includes instructions on the following: database management tasks (creating and deleting databases, adding volume, etc.), migration tasks (moving database to a different location or making changes so that it fits the system's version), and making back-ups and rollbacks of the database in case of failures.

It also includes instructions on how to use the CUBRID utilities, which starts and stops various processes of the CUBRID server, the broker and manager server.

This chapter contains the following:

- How to use CUBRID utilities
- How to control the CUBRID (service, database server, broker, manager server)
- How to use the database administrative utilities
- Database migration
- Database backup and restore
- Database replication

CUBRID Utilities

The CUBRID utilities provide features that can be used to comprehensively manage the CUBRID service. CUBRID utilities are divided into the service management utility, which is used to manage the CUBRID service process, and the database management utility, which is used to manage the database.

The service management utility is as follows:

- Service Utilities : Operates and manages the master process.
  - cubrid service
- Server Utility : Operates and manages the server process.
  - cubrid server
- Broker Utility : Operates and manages the broker process and application server (CAS) process.
  - cubrid broker
- Manager Utility : Operates and manages the manager server process.
  - cubrid manager
- Replication Utility : Operates and manages the replication server process.
  - cubrid repl_server
  - cubrid repl_agent

See Registering Services for details.

The database management utility is as follows:

- Database create/add volume/delete utility
  - cubrid createdb
  - cubrid addvoldb
  - cubrid deletedb
- Database rename/alter host/copy/install utility
  - cubrid renamedb
  - cubrid alterdbhost
  - cubrid copydb
  - cubrid installdb
- Database space check/space compaction utility
  - cubrid spacedb
  - cubrid compactdb
- Database query plan check/optimization utility
  - cubrid plandump
  - cubrid optimizedb
  - cubrid statdump
- Database lock check/transaction kill/consistency check utility
  - cubrid lockdb
  - cubrid killtran
  - cubrid checkdb
- Database diagnostics/log re-generation utility
  - cubrid diagdb
  - cubrid emergency_patchlog

    - cubrid paramdump
- Database loading Utilities
    - cubrid loaddb
    - cubrid unloaddb
- Database backup/restore utility
    - cubrid backupdb
    - cubrid restoredb
- HA utilities
    - cubrid changemode
    - cubrid copylogdb
    - cubrid applylogdb

See How to Use the CUBRID Management Utilities (Syntax) for details.

The following information will be displayed upon entering **cubrid** in the prompt.

```
% cubrid

cubrid utility, version R3.0
usage: cubrid <utility-name> [args]
Type 'cubrid <utility-name>' for help on a specific utility.

Available service's utilities:
    service
    server
    broker
    manager
    repl_server
    repl_agent
    heartbeat

Available administrator's utilities:
    addvoldb
    alterdbhost
    backupdb
    checkdb
    compactdb
    copydb
    createdb
    deletedb
    diagdb
    emergency_patchlog
    installdb
    killtran
    loaddb
    lockdb
    optimizedb
    plandump
    renamedb
    restoredb
    spacedb
    unloaddb
    paramdump
    statdump
    changemode
    copylogdb
    applylogdb

cubrid is a tool for DBMS.
For additional information, see http://www.cubrid.com
```

CUBRID Controls

How to Use CUBRID Utilities (Syntax)

The following provides descriptions on how to use CUBRID utilities (syntaxes).

CUBRID Service Control

The following is the **cubrid** utility syntax used to control services registered in the CUBRID configuration file. The following can be used as *command*; **start** to start the service, **stop** to stop the service, **restart** to restart the service, **status** to verify the status. It is not required to enter additional options or arguments.

```
cubrid service command
command : { start | stop | restart | status }
```

Database Server Control

The following is the **cubrid** utility syntax used to control the database server process. The following can be used as *command*; **start** to start the service, **stop** to stop the process, **restart** to restart the process, and **status** to verify the status. In all commands, except **status**, the database name must be assigned as a argument.

```
cubrid server command [<database_name>]
command : { start | stop | restart | status  }
```

Broker Control

The following is the **cubrid** utility syntax used to control the CUBRID broker process. The following can be used as *command*; **start** to start the broker process, **stop** to stop the process, **restart** to restart the process, **status** to verify the status, **on** to start a specific broker and **off** to stop it.

```
cubrid broker command
command : { start | stop | restart | status [<broker_name>] | on <broker_name>| off
<broker_name> }
```

CUBRID Manager Server Control

The manager server must be executed where the database server is executed to use the CUBRID Manager. The following is the **cubrid** utility syntax used to control the CUBRID Manager process. The following can be used as *command*; **start** to start the manager server process, **stop** to stop the process, **status** to verify the status.

```
cubrid manager command
command : { start | stop | status }
```

Replication Process Control

The following is the **cubrid** utility syntax used to control the CUBRID replication process.

```
cubrid repl_server command
command : { start <master_database_name> <server_network_port> [-a <max_agents>] [-e
<error_file_name>] | stop <master_database_name> | status }

cubrid repl_agent command
command : { start <dist_database_name> [<dba_password>] | stop <dist_database_name> |
status }
```

CUBRID Services

Registering Services

You can register one or more of database server, CUBRID Broker or CUBRID Manager as CUBRID services in the database environment configuration file (cubrid.conf). Only a master process is registered by default if you have not registered a specific service by yourself. You can conveniently run, stop or check the status of all related processes at once by using the **cubrid service** utility if they are registered as CUBRID services. The following is an example of registering the database Server and Broker as services in the database environment configuration file, and configuring the **demodb** and **testdb** databases to be started automatically when the CUBRID service starts.

```
%vi cubrid.conf
#
# Copyright (C) 2008 Search Solution Corporation. All rights reserved by Search Solution.
#
# $Id$
#
# cubrid.conf

#....

[service]

# The list of processes to be started automatically by 'cubrid service start' command
# Any combinations are available with server, broker and manager.
service=server,broker

# The list of database servers in all by 'cubrid service start' command.
# This property is effective only when the above 'service' property contains 'server'
keyword.
server=demodb,testdb
```

Starting and Stopping Services

Starting Services

On Linux, after installing CUBRID, enter the following to start a CUBRID service. If no services are registered in the database environment configuration file, only a master process is stopped by default.

On Windows, the following command can be normally executed by a 'SYSTEM' user only. An administrator or general user can run or stop the CUBRID Server by clicking the CUBRID Service tray icon that appears after installing the CUBRID Manager.

```
% cubrid service start
@ cubrid master start
++ cubrid master start: success
```

The following message appears if the master process is already running:

```
% cubrid service start
@ cubrid master start
++ cubrid master is running.
```

The following message appears if the master process fails to start: The following is an example that the service fails to start due to the conflict between the **cubrid_port_id** parameters, which is set in the database environment configuration file (cubrid.conf). In a such case, you can change the port to prevent conflicts. If it fails starting even if no port is occupied by the processes, you should restart it after deleting a /tmp/CUBRID1523 file.

```
% cubrid service start
@ cubrid master start
cub_master: '/tmp/CUBRID1523' file for UNIX domain socket exist.... Operation not
permitted
++ cubrid master start: fail
```

After registering a service as explained in Registering Services, enter the following to start the service. You can see that the master process, database server process, Broker and registered demodb, and testdb all start at the same time.

```
% cubrid service start
@ cubrid master start
++ cubrid master start: success
@ cubrid server start: demodb

This may take a long time depending on the amount of restore works to do.

CUBRID 2008 R3.0

++ cubrid server start: success
```

```
@ cubrid server start: testdb

This may take a long time depending on the amount of recovery works to do.

CUBRID 2008 R3.0……

++ cubrid server start: success
@ cubrid broker start
++ cubrid broker start: success
```

Stopping Services

Enter the following to stop a CUBRID service. If no services are registered by the user, only the master process is stopped.

```
% cubrid service stop
@ cubrid master stop
++ cubrid master stop: success
```

Enter the following to stop the registered CUBRID service. You can see that the server process, Broker process and master process as well as demodb and testdb are all stopped.

```
% cubrid service stop
@ cubrid server stop: demodb
Server demodb notified of shutdown.
This may take several minutes. Please wait.
++ cubrid server stop: success
@ cubrid server stop: testdb
Server testdb notified of shutdown.
This may take several minutes. Please wait.
++ cubrid server stop: success
@ cubrid broker stop
++ cubrid broker stop: success
@ cubrid master stop
++ cubrid master stop: success
```

Restarting Services

Enter the following to restart a CUBRID service. If no services are registered by the user, only the master process is stopped and then restarted.

```
% cubrid service restart
@ cubrid master stop
++ cubrid master stop: success
@ cubrid master start
++ cubrid master start: success
```

Enter the registered CUBRID service as shown below. You can see that the server process, Broker process and master process as well as demodb and testdb are all stopped and then restarted.

```
% cubrid service restart
@ cubrid server stop: demodb
Server demodb notified of shutdown.
This may take several minutes. Please wait.
++ cubrid server stop: success
@ cubrid server stop: testdb
Server testdb notified of shutdown.
This may take several minutes. Please wait.
++ cubrid server stop: success
@ cubrid broker stop
++ cubrid broker stop: success
@ cubrid master stop
++ cubrid master stop: success
@ cubrid master start
++ cubrid master start: success
@ cubrid server start: demodb

This may take a long time depending on the amount of recovery works to do.

CUBRID 2008 R3.0……

++ cubrid server start: success
```

```
@ cubrid server start: testdb

This may take a long time depending on the amount of recovery works to do.

CUBRID 2008 R3.0……

++ cubrid server start: success
@ cubrid broker start
++ cubrid broker start: success
```

Checking Service Status

Enter the following to check the status of the registered master process, database server and replication server.

```
% $ cubrid service status
@ cubrid master status
++ cubrid master is running.
@ cubrid server status

 Server testdb (rel 8.2, pid 31059)
 Server demodb (rel 8.2, pid 30950)

@ cubrid broker status
% query_editor  - cub_cas [15464,40000]
/home1/cubrid22/CUBRID/log/broker//query_editor.access
/home1/cubrid22/CUBRID/log/broker//query_editor.err
 JOB QUEUE:0, AUTO_ADD_APPL_SERVER:ON, SQL_LOG_MODE:ALL:100000
 LONG_TRANSACTION_TIME:60.00, LONG_QUERY_TIME:60.00, SESSION_TIMEOUT:300
 KEEP_CONNECTION:AUTO, ACCESS_MODE:RW
----------------------------------------
ID   PID   QPS   LQS PSIZE STATUS
----------------------------------------
 1 15465     0     0 48032 IDLE
 2 15466     0     0 48036 IDLE
 3 15467     0     0 48036 IDLE
 4 15468     0     0 48036 IDLE
 5 15469     0     0 48032 IDLE

@ cubrid manager server status
++ cubrid manager server is not running.
@ cubrid replication status
```

The following message appears if the master process has been stopped.

```
% cubrid service status
@ cubrid master status
++ cubrid master is not running.
```

Database Server

Starting and Stopping Database Server

Starting the Database Server

Enter the following to run the demodb server.

```
% cubrid server start demodb
@ cubrid server start: demodb

This may take a long time depending on the amount of recovery works to do.

CUBRID 2008 R3.0

++ cubrid server start: success
```

If you start the demodb server when the master process stops, the master process runs and then the specified database starts automatically.

```
% cubrid server start demodb
@ cubrid master start
++ cubrid master start: success
```

```
@ cubrid server start: demodb

This may take a long time depending on the amount of recovery works to do.

CUBRID 2008 R3.0

++ cubrid server start: success
```

The following message appears if the demodb server is already running.

```
% cubrid server start demodb
@ cubrid server start: demodb
++ cubrid server 'demodb' is running.
```

Stopping the Database Server

Enter the following to stop the demodb server.

```
% cubrid server stop demodb
@ cubrid server stop: demodb
Server demodb notified of shutdown.
This may take several minutes. Please wait.
++ cubrid server stop: success
```

The following message appears if the demodb server has already been stopped.

```
% cubrid server stop demodb
@ cubrid server stop: demodb
++ cubrid server 'demodb' is not running.
```

Restarting the Database Server

Enter the following to restart the demodb server. You can see that the currently running demodb server is stopped and then restarted.

```
% cubrid server restart demodb
@ cubrid server stop: demodb
Server demodb notified of shutdown.
This may take several minutes. Please wait.
++ cubrid server stop: success
@ cubrid server start: demodb

This may take a long time depending on the amount of recovery works to do.

CUBRID 2008 R3.0

++ cubrid server start: success
```

Checking Database Server Status

Enter the following to check the status of the database server. Names of all currently running database servers are displayed.

```
% cubrid server status
@ cubrid server status
 Server testdb (rel 8.2, pid 24465)
 Server demodb (rel 8.2, pid 24342)
```

The following message appears if the master process has been stopped.

```
% cubrid server status
@ cubrid server status
++ cubrid master is not running.
```

Broker

Starting and Stopping Broker

Enter the following to start the Broker.

```
% cubrid broker start
@ cubrid broker start
```

```
++ cubrid broker start: success
```

The following message appears if the Broker is already running.

```
% cubrid broker start
@ cubrid broker start
++ cubrid broker is running.
```

Enter the following to stop the Broker.

```
% cubrid broker stop
@ cubrid broker stop
++ cubrid broker stop: success
```

The following message appears if the Broker has been stopped.

```
% cubrid broker stop
@ cubrid broker stop
++ cubrid broker is not running.
```

Checking Broker Status

Description

By providing various options, the **cubrid broker status** utility allows you to check the status of the Broker, such as the number of completed jobs by each Broker and the number of standby jobs. Take a look at the syntax and its examples.

Syntax

The following is the syntax for monitoring the status of the CUBRID Broker. If *args* is specified, monitoring of the status of the specified Broker is performed; if omitted, all Brokers registered in the CUBRID Broker environment configuration file (**cubrid_broker.conf**) are monitored.

```
cubrid broker status options [<expr>]
options : [ -b | -f[-l secs] | -q|-t|-s secs]
```

Options

The following table shows options that can be used together with cubrid broker status.

Options	Description
expr	Displays the status of the Broker of which name contains <*expr*>. If this is not specified, the status of all Brokers is displayed.
-b	Displays the status of the Broker only, excluding the information about the application server (CAS).
-f[-l secs]	Displays DB and host information accessed by Broker. If it is used with the -b option, information on CAS is displayed as well. The -l secs option is used to specify accumulation period (unit: sec.) when displaying the number of CASs of which status is Waiting or Busy. If the -l secs option is omitted, one sec. is specified by default.
-q	Displays standby jobs in the job queue.
-t	Displays on screen in tty mode. The output contents can be redirected so that it can be used as a file.
-s	Displays the status of the Broker regularly according to the specified time period. Returns to the command prompt if q is entered.
-f	Displays DB and host information where the Broker is connected.

Example

If you do not specify any option and argument to check the status of all Brokers, you will get the following output:

```
% cubrid broker status
@ cubrid broker status
% query_editor - cub_cas [28433,40820] /home/CUBRID/log/broker/query_editor.access
/home/CUBRID/

 JOB QUEUE:0, AUTO_ADD_APPL_SERVER:ON, SQL_LOG_MODE:ALL:100000
 LONG_TRANSACTION_TIME:60, LONG_QUERY_TIME:60, SESSION_TIMEOUT:300
 KEEP_CONNECTION:AUTO, ACCESS_MODE:RW
---------------------------------------------------------------
ID PID  QPS  LQS  PSIZE STATUS
---------------------------------------------------------------
1 28434  0    0    50144 IDLE
2 28435  0    0    50144 IDLE
3 28436  0    0    50144 IDLE
4 28437  0    0    50144 IDLE
5 28438  0    0    50144 IDLE

% broker1  - cub_cas [28443,40821] /home/CUBRID/log/broker/broker1.access /home/CUBRID/
 JOB QUEUE:0, AUTO_ADD_APPL_SERVER:ON, SQL_LOG_MODE:ALL:100000
 LONG_TRANSACTION_TIME:60, LONG_QUERY_TIME:60, SESSION_TIMEOUT:300
 KEEP_CONNECTION:AUTO, ACCESS_MODE:RW
----------------------------------------
ID   PID     QPS   LQS  PSIZE STATUS
----------------------------------------
 1  28444     0     0   50144 IDLE
 2  28445     0     0   50140 IDLE
 3  28446     0     0   50144 IDLE
 4  28447     0     0   50144 IDLE
 5  28448     0     0   50144 IDLE
```

- % query_editor : Broker name
- cub_cas : Type of the CUBRID application server
- [28433, 40820] : PID and connection port number of the Broker
- /home/CUBRID/log/broker/query_editor.access : Path of the access log file of query_editor
- JOB QUEUE : The number of standby jobs in the job queue
- AUTO_ADD_APPL_SERVER : The value of the AUTO_ADD_APPL_SERVER parameter in **cubrid_broker.conf** is ON, which allows the application server to be added automatically.
- SQL_LOG_MODE : The value of the SQL_LOG parameter in the **cubrid_broker.conf** file is ALL, which allows the SQL log to be recorded.
- LONG_TRANSACTION_TIME : Transaction execution time which determines long-duration transaction. Exceeding 60 seconds is regarded as long-duration transaction.
- LONG_QUERY_TIME : Query execution time which determines long-duration query. Exceeding 60 seconds is regarded as long-duration query.
- SESSION_TIMEOUT : Session timeout value; the value of SESSION_TIMEOUT parameter in the **cubrid_broker.conf** file is 300.
- KEEP_CONNECTION : The value of KEEP_CONNECTION parameter in the **cubrid_broker.conf** file is AUTO, which allows client applications is automatically connected to their server.
- ACCESS_MODE: The Broker action mode; Database manipulation as well as retrieval is allowed in the RW mode.
- ID : Broker ID or the standby job ID in the job queue
- PID : Process ID of the Broker
- QPS : The number of queries processed per second
- LQS : The number of long-duration queries processed per second
- PSIZE : Size of the application server process
- STATUS : The current status of the application server (BUSY, IDLE, CLIENT_WAIT, CLOSE_WAIT)

To check the status of the Broker, enter as follows:

```
% cubrid broker status -b
@ cubrid broker status
  NAME          PID  PORT   AS  JQ      REQ  TPS  QPS  LONG-T  LONG-Q ERR-Q
===========================================================================
* query_editor  4094 30000   5   0       0    0    0    0/60    0/60    0
* broker1       4104 33000   5   0       0    0    0    0/60    0/60    0
```

- NAME : Broker name

- PID : Process ID of the Broker
- PORT : Port number of the Broker
- AS : The number of application servers
- JQ : The number of standby jobs in the job queue
- REQ : The number of client requests processed by the Broker
- TPS : The number of transactions processed per second (calculated only when the option is configured to "-b -s <sec>")
- QPS : The number of queries processed per second (calculated only when the option is configured to "-b -s <sec>")
- LONG-T : The number of transactions which exceed LONG_TRANSACTION_TIME; the value of the LONG_TRANSACTION_TIME parameter
- LONG-Q : The number of queries which exceed LONG_QUERY_TIME; the value of the LONG_QUERY_TIME parameter
- ERR-Q : The number of queries with errors found

Check the status of the Broker whose name contains **broker1** by using the **-q** option, and then enter the following to check the status of the standby jobs in the job queue of the specified Broker. If **broker1** is not entered as an argument, the list of all standby jobs in the job queue of all Brokers is outputted.

```
% cubrid broker status -q broker1
@ cubrid broker status
% broker1  - cub_cas [28443,40821] /home/CUBRID/log/broker/broker1.access /home/CUBRID/
 JOB QUEUE:0, AUTO_ADD_APPL_SERVER:ON, SQL_LOG_MODE:ALL:100000
 LONG_TRANSACTION_TIME:60, LONG_QUERY_TIME:60, SESSION_TIMEOUT:300
 KEEP_CONNECTION:AUTO, ACCESS_MODE:RW
----------------------------------------
ID   PID   QPS   LQS PSIZE STATUS
----------------------------------------
 1 28444     0     0 50144 IDLE
 2 28445     0     0 50140 IDLE
 3 28446     0     0 50144 IDLE
 4 28447     0     0 50144 IDLE
 5 28448     0     0 50144 IDLE
```

Enter the monitoring interval of the Broker whose name contains **broker1** by using the **-s** option, and then enter the following to monitor the status of the Broker regularly. If **broker1** is not entered as an argument, monitoring of the status of all Brokers is performed regularly. If you enter **q**, the monitoring screen returns to the command prompt.

```
% cubrid broker status -s 5 broker1
% broker1  - cub_cas [28443,40821] /home/CUBRID/log/broker/broker1.access /home/CUBRID/
 JOB QUEUE:0, AUTO_ADD_APPL_SERVER:ON, SQL_LOG_MODE:ALL:100000
 LONG_TRANSACTION_TIME:60, LONG_QUERY_TIME:60, SESSION_TIMEOUT:300
 KEEP_CONNECTION:AUTO, ACCESS_MODE:RW
----------------------------------------
ID   PID   QPS   LQS PSIZE STATUS
----------------------------------------
 1 28444     0     0 50144 IDLE
 2 28445     0     0 50140 IDLE
 3 28446     0     0 50144 IDLE
 4 28447     0     0 50144 IDLE
 5 28448     0     0 50144 IDLE
```

Output TPS and QPS information to a file by using the **-t** option. To cancel the output process, press <CTRL+C> to stop the program.

```
% cubrid broker status -b -t -s 1 > log_file
```

Enter the following to monitor the status of all Brokers (including TPS and QPS) regularly by using the **-b** and **-s** options.

```
% cubrid broker status -b -s 1
NAME           PID  PORT  AS  JQ      REQ  TPS  QPS  LONG-T  LONG-Q ERR-Q
===========================================================================
* query_editor 28433 40820   5   0        0    0    0    0/60    0/60    0
* broker1      28443 40821   5   0        0    0    0    0/60    0/60    0
```

Enter the following to view information of a server/database connected to the Broker, its access time, and the IP addresses connected to CAS by using the **-f** option.

```
$ cubrid broker status -f broker1
@ cubrid broker status
% broker1  - cub_cas [28443,40821] /home/CUBRID/log/broker/broker1.access /home/CUBRID/
 JOB QUEUE:0, AUTO_ADD_APPL_SERVER:ON, SQL_LOG_MODE:ALL:100000
 LONG_TRANSACTION_TIME:60, LONG_QUERY_TIME:60, SESSION_TIMEOUT:300
 KEEP_CONNECTION:AUTO, ACCESS_MODE:RW
------------------------------------------------------------------------------------------
------------------------
ID   PID   QPS   LQS  PSIZE STATUS     LAST ACCESS TIME        DB      HOST   LAST CONNECT
TIME      CLIENT IP
------------------------------------------------------------------------------------------
------------------------
 1  26946   0    0    51168  IDLE     2009/11/06 16:06:41      -         -            -
             10.0.1.101
 2  26947   0    0    51172  IDLE     2009/11/06 16:06:41      -         -            -
             10.0.1.101
 3  26948   0    0    51172  IDLE     2009/11/06 16:06:41      -         -            -
             10.0.1.101
 4  26949   0    0    51172  IDLE     2009/11/06 16:06:41      -         -            -
             10.0.1.101
 5  26950   0    0    51172  IDLE     2009/11/06 16:06:41      -         -            -
             10.0.1.101
```

The -b and the -f options are used to display information on AS(T W B Ns-W Ns-B) and CANCELED. The description of each information are as follows:

- T : Total number of CASs being executed
- W : The number of CASs in the state of Waiting
- B : The number of CASs in the state of Busy
- Ns-W : The number of CASs that the client belongs to has been waited for N secs.
- Ns-B : The number of CASs that the client belongs to has been Busy for N secs.
- CANCELED: The number of queries have canceled by user interruption since Broker is started (if it is used with the -l N option, it specifies the number of accumulations for N secs).

```
// Adding the -f option upon the execution of Broker state information. The -l option is
used to specify the N value (unit: sec) so that Ns-W and Ns-B can be displayed for
specified N secs.
% cubrid broker status -b -f -l 2
@ cubrid broker status
NAME          PID    PSIZE PORT  AS(T W B 2s-W 2s-B) JQ REQ TPS QPS LONG-T LONG-Q ERR-Q
CANCELED ACCESS_MODE SQL_LOG
==========================================================================================
========================
query_editor 16784 56700 38000    5 0 0 0     0      0  0    0    0   0/60.0 0/60.0
0      0     RW         ALL
```

Managing Specific Broker

Enter the following to start *broker1* only. Here, *broker1* is a broker that has been already configured in the shared memory.

```
% cubrid broker on broker1
```

The following message appears if *broker1* is not configured in the shared memory.

```
% cubrid broker on broker1
Cannot open shared memory
```

Enter the following to stop *broker1* only. Here, you can also remove the service pool of *broker1*.

```
% cubrid broker off broker1
```

Enter the following to restart *broker1*.

```
% cubrid broker restart broker1
```

Dynamically Changing Broker Parameters

Description

You can configure the parameters related to running the Broker in the broker environment configuration file (cubrid_broker.conf). For more information, see Parameter by Broker in the "Performance Management Guide." You can also modify some broker parameters temporarily while the Broker is running by using the broker_changer utility. The following broker parameters can be modified dynamically.

- ACCESS_MODE
- ACCESS_LOG
- APPL_SERVER_MAX_SIZE
- KEEP_CONNECTION
- LOG_BACKUP
- SQL_LOG
- SQL_LOG_MAX_SIZE
- STATEMENT_POOLING
- TIME_TO_KILL

Syntax

The syntax for the **broker_changer** utility, which is used to change broker parameters while the Broker is running, is as follows. Enter the name of the currently running Broker for the *broker_name*. The *parameter*s can be used only for dynamically modifiable parameters. The *value* must be specified based on the parameter to be modified.

```
broker_changer  broker_name  parameters  value
```

Example 1

Enter the following to configure the SQL_LOG parameter to ON so that SQL logs can be written to the currently running Broker. Such dynamic parameter change is effective only while the Broker is running.

```
% broker_changer query_editor sql_log on
OK
```

Example 2

Enter the following to change Broker's **ACCESS_MODE** to **Read Only** and automatically reset the Broker in HA environment.

```
% broker_changer broker_m access_mode ro
OK
```

Broker Logs

There are three types of logs that relate to starting the Broker: access, error and SQL logs. Each log can be found in the log directory under the installation directory. You can change the directory where these logs are to be saved through LOG_DIR and ERROR_LOG_DIR parameters of the broker environment configuration file (cubrid_broker.conf).

Checking the Access Log

The access log file records information about the application client and is saved with the name of *broker_name.access*. If the **LOG_BACKUP** parameter is configured to **ON** in the Broker environment configuration file, when the Broker stops properly, the access log file is saved with the date and time that the Broker has stopped. For example, if broker1 stopped at 12:27 P.M. on June 17, 2008, an access file named broker1.access.20080617.1227 is generated in the **log/broker** directory. The following is an example of an access log.

The following is an example and description of an access log file created in the log directory:

```
1 192.168.1.203 - - 972523031.298 972523032.058 2008/06/17 12:27:46~2008/06/17 12:27:47
7118 - -1
```

```
2 192.168.1.203 - - 972523052.778 972523052.815 2008/06/17 12:27:47~2008/06/17 12:27:47
7119 ERR 1025
1 192.168.1.203 - - 972523052.778 972523052.815 2008/06/17 12:27:49~2008/06/17 12:27:49
7118 - -1
```

- 1 : ID assigned to the application server of the Broker
- 192.168.1.203 : IP address of the application client
- 972523031.298 : UNIX timestamp value when the client's request processing started
- 2008/06/17 12:27:46 : Time when the client's request processing started
- 972523032.058 : Unix timestamp value when the client's request processing finished
- 2008/06/17 12:27:47 : Time when the client's request processing finished
- 7118 : Process ID of the application server
- -1 : No error occurred during the request processing
- ERR 1025 : Error occurred during the request processing. Error information exists in offset=1025 of the error log file

Checking the Error Log

The error log file records information about errors that occurred during the client's request processing and is stored with the name of *broker_name_app_server_num.err*.

The following is an example and description of an error log:

```
Time: 02/04/09 13:45:17.687 - SYNTAX ERROR *** ERROR CODE = -493, Tran = 1, EID = 38
Syntax: Unknown class "unknown_tbl". select * from unknown_tbl
```

- Time : 02/04/09 13:45:17.687 : Time when the error occurred
- - SYNTAX ERROR : Type of error (e.g. SYNTAX ERROR, ERROR, etc.)
- *** ERROR CODE = -493 : Error code
- Tran = 1 : Transaction ID. -1 indicates that no transaction ID is assigned.
- EID = 38 : Error ID. This ID is used to find the SQL log related to the server or client logs when an error occurs during SQL statement processing.
- Syntax... : Error message (An ellipsis (...) indicates omission.)

Managing the SQL Log

The SQL log file records SQL statements requested by the application client and is stored with the name of *broker_name_app_server_num.sql.log*. The SQL log is generated in the log/broker/sql_log directory when the SQL_LOG parameter is set to ON. Note that the size of the SQL log file to be generated cannot exceed the value set for the SQL_LOG_MAX_SIZE parameter. CUBRID offers the **broker_log_top**, **broker_log_converter**, and **broker_log_runner** utilities to manage SQL logs. Each utility should be executed in a directory where the corresponding SQL log exists.

The following are examples and descriptions of SQL log files:

```
02/04 13:45:17.687 (38) prepare 0 insert into unique_tbl values (1)
02/04 13:45:17.687 (38) prepare srv_h_id 1
02/04 13:45:17.687 (38) execute srv_h_id 1 insert into unique_tbl values (1)
02/04 13:45:17.687 (38) execute error:-670 tuple 0 time 0.000, EID = 39
02/04 13:45:17.687 (0) auto_rollback
02/04 13:45:17.687 (0) auto_rollback 0
*** 0.000
02/04 13:45:17.687 (39) prepare 0 select * from unique_tbl
02/04 13:45:17.687 (39) prepare srv_h_id 1 (PC)
02/04 13:45:17.687 (39) execute srv_h_id 1 select * from unique_tbl
02/04 13:45:17.687 (39) execute 0 tuple 1 time 0.000
02/04 13:45:17.687 (0) auto_commit
02/04 13:45:17.687 (0) auto_commit 0
*** 0.000
```

- 02/04 13:45:17.687 : Time when the application sent the request
- (39) : Sequence number of the SQL statement group. If prepared statement pooling is used, it is uniquely assigned to each SQL statement in the file.
- prepare 0 : Whether or not it is a prepared statement

- prepare srv_h_id 1 : Prepares the SQL statement as srv_h_id 1.
- (PC) : It is outputted if the data in the plan cache is used.
- SELECT... : SQL statement to be executed. (An ellipsis (...) indicates omission.) For statement pooling, the binding variable of the WHERE clause is represented as a question mark (?).
- Execute 0 tuple 1 time 0.000 : One row is executed. The time spent is 0.000 second.
- auto_commit/auto_rollback : Automatically committed or rolled back. The second auto_commit/auto_rollback is an error code. 0 indicates that the transaction has been completed without an error.

The **broker_log_top** utility analyses the SQL logs which are generated for a specific period. As a result, the information of SQL statements and time execution are outputted in files by order of the longest execution time; the results of SQL statements are stored in **log.top.q** and those of execution time are stored in **log.top.res**, respectively.

The **broker_log_top** utility is useful to analyse the long query. The syntax is as follows:

```
broker_log_top [options] sql_log_file_list
options : {-t | -F  from_date | -T  to_date}
```

The results are outputted in transaction unit if the **-t** option is specified. SQL statements which are used for a specific period time can be analyzed by using the **-F** and **-T** options. All logs are outputted by SQL statement if any option is not specified.

The following logs are the results of executing the broker_log_top utility; logs are generated from Nov. 11th to Nov. 12th, and it is displayed in the order of the longest execution of SQL statements. Each month and day are separated by a whitespace when specifying peroid. Note that "*.sql.log" is not recognized so the SQL logs should separated by a white space on Windows.

```
--Execution broker_log_top on Linux
% broker_log_top -F "11 11" -T "11 12" -t *.sql.log
query_editor_1.sql.log
query_editor_2.sql.log
query_editor_3.sql.log
query_editor_4.sql.log
query_editor_5.sql.log

--Executing broker_log_top on Windows
% broker_log_top -F "11 11" -T "11 12" -t query_editor_1.sql.log query_editor_2.sql.log
query_editor_3.sql.log query_editor_4.sql.log query_editor_5.sql.log
```

The log.top.q and log.top.res files are generated in the same directory where the analyzed logs are stored when executing the example above; In the log.top.q file, you can view each SQL statement, and its line number. In the log.top.res, you can the minimum, maximum and avg. time, and the number of execution queries for each SQL statement.

```
--log.top.q file
[Q1]-------------------------------------------
broker1_6.sql.log:137734
11/11 18:17:59.396 (27754) execute_all srv_h_id 34 select a.int_col, b.var_col from
dml_v_view_6 a, dml_v_view_6 b, dml_v_view_6 c , dml_v_view_6 d, dml_v_view_6 e where
a.int_col=b.int_col and b.int_col=c.int_col and c.int_col=d.int_col and
d.int_col=e.int_col order by 1,2;
11/11 18:18:58.378 (27754) execute_all 0 tuple 497664 time 58.982
.
.
[Q4]-------------------------------------------
broker1_100.sql.log:142068
11/11 18:12:38.387 (27268) execute_all srv_h_id 798 drop table list_test;
11/11 18:13:08.856 (27268) execute_all 0 tuple 0 time 30.469

-- log.top.res file
max           min      avg     cnt(err)
-----------------------------------------------------
[Q1]        58.982    30.371    44.676    2 (0)
[Q2]        49.556    24.023    32.688    6 (0)
[Q3]        35.548    25.650    30.599    2 (0)
[Q4]        30.469    0.001     0.103  1050 (0)
```

To store SQL logs created in log/broker/sql_log under the installation directory to a separate file, the **broker_log_converter** utility is executed. The syntax of the **broker_log_converter** utility is as follows: This example saves queries stored in the query_editor_1.sql.log file to the query_convert.in file.

```
broker_log_converter  SQL_log_file  output_file
```

The following example shows that the query in the query_editor_1.sql.log file is converted into the query_convert.in file.

```
% broker_log_converter query_editor_1.sql.log query_convert.in
```

To re-execute queries saved in the query file which has been created by the **broker_log_converter** utility, the **broker_log_runner** utility is executed. The syntax of the **broker_log_runner** utility is as follows: This example re-executes queries saved in the query_convert.in in demodb. It is assumed that the IP address of the Broker is 192.168.1.10 and its port number is 30,000.

```
broker_log_runner options input_file
options : -I cas_ip -P cas_port  -d dbname  [-u dbuser [-p dbpasswd ]]  [-t num_thread] [-
r repeat_count] [ -o result_file]
```

broker_log_runner Utility Options

Option	Description
-I *broker_ip*	IP address or host name of the CUBRID Broker
-P *broker_port*	Port number of the CUBRID Broker
-d *dbname*	Name of the database against which queries are to be executed
-u *dbuser*	Database user name (default value : public)
-p *dbpasswd*	Database password
-t *numthread*	The number of threads (default value : 1)
-r *repeat_count*	The number of times that the query is to be executed (default value : 1)
-o *result_file*	Name of the file where execution results are to be stored

```
% broker_log_runner -I 192.168.1.10  -P 30000 -d demodb -t 2 query_convert.in
cas_ip = 192.168.1.10
cas_port = 30000
num_thread = 2
repeat = 1
dbname = demodb
dbuser = public
dbpasswd =
exec_time : 0.001
exec_time : 0.000
0.000500 0.000500 -
```

CUBRID Manager Server

Starting and Stopping CUBRID Manager

Starting the CUBRID Manager

Enter the following to run the CUBRID Manager Server.

```
% cubrid manager start
```

The following message appears if the CUBRID Manager server is already running.

```
% cubrid manager start
@ cubrid manager server start
++ cubrid manager server is running.
```

Stopping the CUBRID Manager

Enter the following to stop the CUBRID Manager server.

```
% cubrid manager stop
```

```
@ cubrid manager server stop
++ cubrid manager server stop: success
```

CUBRID Manager Server Log

CUBRID Manager Server-related logs are stored in log/manager directory under the installation directory. They are stored as one of the following four types of files depending on the process of the Manager Server.

- cub_auto.access.log : Access log of a client that logged into and out of the Manager Server successfully
- cub_auto.error.log : Access log of a client that failed to log into or out of the Manager Server
- cub_js.access.log : Log of the jobs processed by the Manager Server
- cub_js.error.log : Error log that occurred while the Manager Server is processing jobs

Database Administration

How to Use the CUBRID Administration Utilities (Syntax)

The following shows how to use the CUBRID management utilities.

```
cubrid utility_name
utility_name :
  createdb [option] <database_name>   --- Creating a database
  deletedb [option] <database_name>   --- Deleting a database
  installdb [option] <database-name>   --- Installing a database
  renamedb [option] <source-database-name> <target-database-name>  --- Renaming a database
  copydb [option] <source-database-name> <target-database-name>  --- Copying a database
  backupdb [option] <database-name>  --- Backing up a database
  restoredb [option] <database-name>  --- Restoring a database
  addvoldb [option] <database-name> number-of-pages  --- Adding a database volume file
  spacedb [option] <database-name>  --- Displaying details of database space
  lockdb [option] <database-name>  --- Displaying details of database lock
  killtran [option] <database-name>  --- Removing transactions
  optimizedb [option] <database-name>  --- Updating database statistics
  statdump [option] <database-name>  --- Outputting statistic information of database
server execution
  compactdb [option] <database-name>  --- Optimizing space by freeing unused space
  diagdb [option] <database-name>  --- Displaying internal information
  emergency_patchlog [option] <database-name>  --- Database log patch for emergency
situations
  checkdb [option] <database-name>  --- Checking database consistency
  alterdbhost [option] <database-name>  --- Altering database host
  plandump [option] <database-name>  --- Displaying details of the query plan
  loaddb [option] <database-name>  --- Loading data and schema
  unloaddb [option] <database-name>  --- Unloading data and schema
paramdump [option] <database-name>  --- Checking out the parameter values configured in a
database
  changemode [option] <database-name>  --- Displaying or changing the server HA mode
  copylogdb [option] <database-name>  --- Multiplating transaction logs to configure HA
  applylogdb [option] <database-name>  --- Reading and applying replication logs from
transaction logs to configure HA
```

Database Users

A CUBRID database user can have members with the same authorization. If authorization **A** is granted to a user, the same authorization is also granted to all members belonging to the user. A database user and its members are called a "group."

CUBRID provides **DBA** and **PUBLIC** users by default.

- **DBA** can access every object in the database, that is, it has authorization at the highest level. Only **DBA** has sufficient authorization to add, alter and delete the database users.
- All users including **DBA** are members of **PUBLIC**. Therefore, all database users have the authorization granted to **PUBLIC**. For example, if authorization **B** is added to **PUBLIC** group, all database members will automatically have the **B** authorization.

databases.txt File

Description

CUBRID saves information about the locations of all existing databases in the **databases.txt** file. This file is called the "database location file." A database location file is used when CUBRID executes utilities for creating, renaming, deleting or replicating databases; it is also used when CUBRID runs each database. By default, this file is located in the **databases** directory under the installation directory. The directory is located through the environment variable **CUBRID_DATABASES**.

Syntax

```
db_name db_directory server_host logfile_directory
```

The format of each line of a database location file is the same as defined by the above syntax; it contains information about the database name, database path, server host and the path to the log files. The following is an example of checking the contents of a database location file.

```
% more databases.txt
dist_testdb /home1/user/CUBRID/bin d85007 /home1/user/CUBRID/bin
dist_demodb /home1/user/CUBRID/bin d85007 /home1/user/CUBRID/bin
testdb /home1/user/CUBRID/databases/testdb d85007 /home1/user/CUBRID/databases/testdb
demodb /home1/user/CUBRID/databases/demodb d85007 /home1/user/CUBRID/databases/demodb
```

By default, the database location file is stored in the **databases** directory under the installation directory. You can change the default directory by modifying the value of the **CUBRID_DATABASES** environment variable. The path to the database location file must be valid so that the **cubrid** utility for database management can access the file properly. You must enter the directory path correctly and check if you have write permission on the file. The following is an example of checking the value configured in the **CUBRID_DATABASES** parameter.

```
% set | grep CUBRID_DATABASES
CUBRID_DATABASES=/home1/user/CUBRID/databases
```

An error occurs if an invalid directory path is set in the **CUBRID_DATABASES** environment variable. If the directory path is valid but the database location file does not exist, a new location information file is created. If the **CUBRID_DATABASES** environment variable has not been configured at all, CUBRID retrieves the location information file in the current working directory.

Creating Database

Descripton

The **cubrid createdb** utility creates databases and initializes them with the built-in CUBRID system tables. It can also define initial users to be authorized in the database and specify the locations of the logs and databases. Generally, the **cubrid createdb** utility is used only by DBA.

Syntax

```
cubrid createdb options database_name
options :
[{-p|--pages=}number]
[--comment]
[{-F|--file-path=}path]
[{-L|--log-path=}path]
[{-B|--lob-base-path=}path]
[--server-name=host ]
[{-r | --replace} ]
[--more-volume-file=file]
[--user-definition-file=file]
[ --csql-initialization-file=file]
[{-o|--output-file=} file]
[{-v | --verbose}]
[{-l | --log-page-count=} number]
[--log-page-size=size]
[ {-s | --page-size=} size]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **createdb** : A command used to create a new database.
- *options* : A short option starts with a single dash (-) while a full name option starts with a double dash (--).
- *database_name* : Specifies a unique name for the database to be created, without including the path name to the directory where the database will be created. If the specified database name is the same as that of an existing database name, CUBRID halts creation of the database to protect existing files.

Option

The following table shows options that can be used with **cubrid createdb**. Options are case sensitive.

Option	Description
-p --pages	Specifies the number of pages of the database volume (**generic**) to be created. Default value : 5000 pages
-F --file-path	Specifies the directory path where the database will be created. Default value : Current working directory
-L --log-path	Specifies the directory path where log files will be stored. Default value : A directory path specified with the **-F** option
-B --lob-base-path	Specifies the directory path where LOB data files will be stored. Default value : <location of database volumns created>/lob directory
-r --replace	Allows overwriting if the name of the database to be created is the same as that of an existing database. Default value : Deactivated
-o --output-file	Specifies the file where output messages concerning database creation are stored.
-v --verbose	Displays detailed messages to the screen concerning database creation. Default value : Deactivated
-l --log-page-count	Specifies the number of pages of the log volume. Default value : The number of pages of the generic volume specified by the **-p** option
--log-page-size	Specifies the page size of log volume in unit of byte. Default value : Database size
-s --page-size	Specifies the database page size in bytes. Default value : 4096(4 KB)
--comment	Adds information about the database to be created in the form of a comment.
--server-name	Specifies the name of the server host to connect to. Default value : localhost
--more-volume-file	Specifies the file that includes the specifications for creating an additional volume of the database.
--user-definition-file	Specifies the file that includes user definitions.
--csql-initialization-file	Specifies the file for csql initialization.

Number of pages (-p)

The following example shows creating a database named testdb to which 10,000 pages are assigned. The **-p** option is used to specify the number of pages for a database application. The default value is 5,000.

```
cubrid createdb -p 10000 testdb
```

Database directory path (-F)

The following example shows creating a database named testdb in the directory /dbtemp/new_db.

The **-F** option is used to specify the absolute path to a directory where the new database will be created. If the **-F** option is not specified, the new database is created in the current working directory.

```
cubrid createdb -F "/dbtemp/new_db/" testdb
```

Log file directory path (-L)

The following example shows creating a database named testdb in the directory /dbtemp/newdb and log files in the directory /dbtemp/db_log.

The **-L** option is used to specify the absolute path to the directory where database log files are created. If the **-L** option is not specified, log files are created in the directory specified by the **-F** option. If neither **-F** nor **-L** option is specified, database log files are created in the current working directory.

```
cubrid createdb -F "/dbtemp/new_db/" -L "/dbtemp/db_log/" testdb
```

LOB data file directory (--lob-base-path)

The following example shows creating a database called testdb in the working directory and specifying "/home/data1" of local file system as a location of LOB data files.

The **--lob-base-path** option is used to specify a directory where LOB data files are stored when BLOB/CLOB data is used. If the **--lob-base-path** option is not specified, LOB data files are store in "<location of database volumns created>/lob" directory.

```
cubrid createdb --lob-base-path "file:/home1/data1" testdb
```

Overwrite (-r)

The following example shows creating a new testdb database which overwrites the existing database with the same name.

The **-r** option is used to create a new database and overwrite an existing database if one with the same name exists. If the **-r** option is not specified, database creation is halted when this occurs.

```
cubrid createdb -r testdb
```

Saving output messages to a file (-o)

The following example shows creating a database named testdb and saving the output of the utility to the db_output file instead of displaying it on the console screen.

The **-o** option is used to save messages related to the database creation to the file given as a parameter. The file is created in the same directory where the database was created. If the **-o** option is not specified, messages are displayed on the console screen. The **-o** option allows you to use information about the creation of a certain database by saving messages, generated during the database creation, to a specified file.

```
cubrid createdb -o db_output testdb
```

Verbose output (-v)

The following example shows creating a database named testdb and outputting detailed information about the operation onto the screen.
The **-v** option is used to output all information about the database creation operation onto the screen. Like the **-o** option, this option is useful in checking information related to the creation of a specific database. Therefore, if you specify the **-v** option together with the **-o** option, you can save the output messages in the file given as a parameter; the messages contain the operation information about the **cubrid createdb** utility and database creation process.

```
cubrid createdb -v testdb
```

Log page (-l)

The following example shows creating a database named testdb and setting the number of pages of the log volume to 1,000.

The **-l** option is used to specify the number of pages of the database log volume. The default value is the number of pages of the **generic** volume specified by the **-p** option. The number of pages of the log volume varies depending on the data modification throughput and transaction duration, but an appropriate value is at least equal to or twice greater than that of the database volume.

```
cubrid createdb -l 1000 testdb
```

Log page size (--log-page-size)

This statement creates a database named testdb and sets the size of the log volume page to 4 Kbytes.

The **--log-page-size** option specifies the size of the log volume page of the created database. Its default value is the same as the page size of the generic volume specified by the **-pages** option. The log page size should be specified by considering the transaction interval, log capacity and I/O page size of the OS. If this value is set too high in an environment in which the log capacity is flushed to a disk due to a short transaction commit interval, a disk I/O cost may unnecessarily increase.

```
cubrid createdb
-?log-page-size 8192 testdb
```

Data page size (-s)

The following example shows setting the page size of the volume of the database named testdb to 8192 bytes.

The **-s** option is used to specify the size of the database page to be one of 1024, 2048, 4096, 8192 and 16384 bytes. The default value is 4096. If any number besides these is specified, the system configures the page size as the number of ceils. Note that if you change the page size of a database volume by using the **-s** option, related database parameters such as data_buffer_pages, sort_buffer_pages and log_buffer_pages are also affected.

```
cubrid createdb -s 8192 testdb
```

Comment (--comment)

The following example shows creating a database named testdb and adding a related comment to the database volume.

The **--comment** option is used to specify a comment to be included in the database volume header. If the character string contains spaces, the comment must be enclosed in double quotes.

```
cubrid createdb --comment "a new database for study" testdb
```

Server host name (--server-name)

The following example shows creating and registering a database named testdb on the *aa_host* host.

The **--server-name** option is used to specify that the server for a certain database will be running on a specified host when a client / server version of CUBRID is used. The information about the server host specified with this option is written in the database location file (**databases.txt**). If this option is not specified, the default value is the current localhost.

```
cubrid createdb --server-name aa_host testdb
```

Adding a database volume (--more-volume-file)

The following example shows creating a database named testdb as well as an additional volume based on the specification stored in the *vol_info.txt* file.

The **--more-volume-file** option creates an additional volume based on the specification contained in the file specified by the option. The volume is created in the same directory where the database is created. Instead of using this option, you can add a volume by using the **cubrid addvoldb** utility.

```
cubrid createdb --more-volume-file vol_info.txt testdb
```

The following is a specification of the additional volume contained in the *vol_info.txt* file. The specification of each volume must be written on a single line.

```
#xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# NAME volname COMMENTS volcmnts PURPOSE volpurp NPAGES volnpgs
NAME data_v1 COMMENTS "Data information volume" PURPOSE data NPAGES 1000
NAME data_v2 COMMENTS "Data information volume" PURPOSE data NPAGES 1000
NAME data_v3 PURPOSE data NPAGES 1000
NAME index_v1 COMMENTS "Index information volume" PURPOSE index NPAGES 500
NAME temp_v1 COMMENTS "Temporary information volume" PURPOSE temp NPAGES 500
NAME generic_v1 COMMENTS "Generic information volume" PURPOSE generic NPAGES 500
#xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

As shown in the example, the specification of each volume is composed of followings.

```
NAME volname COMMENTS volcmnts PURPOSE volpurp NPAGES volnpgs
```

- **NAME** *volname* : *volname* is the name of the volume to be created. It must follow the UNIX file name conventions and be a simple name not including the directory path. The specification of a volume name can be omitted. If it is, the "database name to be created by the system_volume identifier" becomes the volume name.
- **COMMENTS** *volcmnts* : *volcmnts* is a comment to be written in the volume header and contains information about the additional volume to be created. The specification of the comment on a volume can also be omitted.

- **PURPOSE** *volpurp* : *volpurp* must be one of the types: **data**, **index**, **temp**, and **generic**, with the purpose of saving volumes. The specification of the purpose of a volume can be omitted in which case the default value is **generic**.
- **NPAGES** *volnpgs* : *volnpgs* is the number of pages of the additional volume to be created. The specification of the number of pages of the volume cannot be omitted; it must be specified.

File containing CSQL statements (--csql-initialization-file)

The following example shows creating a database named testdb and executing the SQL statement defined in table_schema.sql through the CSQL Interpreter.

The **--csql-initialization-file** option executes an SQL statement on the database to be created by using the CSQL Interpreter. A schema can be created based on the SQL statement contained in the file specified by the parameter.

```
cubrid createdb --csql-initialization-file table_schema.sql testdb
```

User information file (--user-definition-file)

The following example shows creating a database named testdb and adding users to testdb based on the user information defined in the user_info.txt file.

The **--user-definition-file** option is used to add users who have access to the database to be created. It adds a user based on the specification contained in the user information file specified by the parameter. Instead of using the **--user-definition-file** option, you can add a user by using the Managing USER statement.

```
cubrid createdb --user-definition-file user_info.txt testdb
```

The syntax of a user information file is as follows:

```
USER user_name [ groups_clause | members_clause ]|
groups_clause:
 [ GROUPS group_name [ { group_name }... ] ]
members_clause:
 [ MEMBERS member_name [ { member_name... } ] ]
```

- The *user_name* is the name of the user who has access to the database. It must not include spaces.
- The **GROUPS** clause is optional. The *group_name* is the upper level group that contains the *user_name*. Here, the *group_name* can be multiply specified and must be defined as **USER** in advance.
- The **MEMBERS** clause is optional. The *member_name* is the name of the lower level member that belongs to the *user_name*. Here, the *member_name* can be multiply specified and must be defined as **USER** in advance.

Comments can be used in a user information file. A comment line must begin with a hyphen (-). Blank lines are ignored.

The following example is a user information file that defines the group sedan to include grandeur and sonata, the group suv to include tuscan, and the group hatchback to include i30. The name of the user information file is user_info.txt.

```
--
-- Example 1 of a user information file
--
USER sedan
USER suv
USER hatchback
USER grandeur GROUPS sedan
USER sonata GROUPS sedan
USER tuscan GROUPS suv
USER i30 GROUPS hatchback
```

The following file defines the same user relationship as the one above, except that it uses the **MEMBERS** clause.

```
--
-- Example 2 of a user information file
--
USER grandeur
USER sonata
USER tuscan
USER i30
USER sedan MEMBERS sonata grandeur
USER suv MEMBERS tuscan
USER hatchback MEMBERS i30
```

Adding Database Volume

Descripton

For how to add new volumes to a database by using the CUBRID Manager, see Database Space.

Syntax

```
cubrid addvoldb  options   [args]  database_name    number_of_pages
options :   [{-n | --volume_name= } name]   [{-F | --file-path=} path] [ {-p | --
page=}number]
[-S | -C | --SA-mode | --CS-mode] [--comment]
```

- **cubrid** : An integrated utility for CUBRID service and database management.
- **addvoldb** : A command that adds a specified number of pages of the new volume to a specified database.
- *options* : A short option starts with a single dash (-) while a full name option starts with a double dash (--).
- *database_name* : Specifies the name of the database to which a volume is to be added without including the path name to the directory where the database is to be created.
- *number_of_pages* : The number of pages which is to be added to the specified database volume. It is recommended to configure a sufficiently large number of pages to be added depending on the purpose and store each volume in a separate disk according to its usages in terms of performance.

Option

The following table shows options that can be used with **cubrid addvoldb** utility.

Option	Description
-n **--volume-name**	Specifies the name of the database volume to be added. Default value : A value in the format of *dbname_number*, configured by the system
-F **--file-path**	Specifies the directory path where the database volume to be added will be created. Default value : A value of **volume_extension_path**, the database parameter
-p **--purpose**	Specifies the purpose of the database volume to be added. Default value : Generic volume
-S **--SA-mode**	Adds the database volume in standalone mode.
-C **--CS-mode**	Adds the database volume in client/server mode.
--comment	Inserts a comment about the database volume to be added.

Name of the extended volume (-n)

The following example shows adding a volume for which 1000 pages are assigned to the testdb database in standalone mode. The volume name testdb_v1 will be created. **-n** is an option that specifies the name of the volume to be added to a specified database. The volume name must follow the file name protocol of the operating system and be a simple one without including the directory path or spaces. If the **-n** option is omitted, the name of the volume to be added is configured by the system automatically as "database name_volume identifier." For example, if the database name is testdb, the volume name *testdb_x001* is automatically configured.

```
cubrid addvoldb -S -n testdb_v1 testdb 1000
```

Path of the extended volume (-F)

The following example shows adding a volume for which 1000 pages are assigned to the *testdb* database in standalone mode. The added volume is created in the /dbtemp/addvol directory. Because the **-n** option is not specified for the volume name, the volume name *testdb_x001* will be created. The **-F** option is used to specify the directory path where the volume to be added will be stored. If the **-F** option is omitted, the value of the database parameter **volume_extension_path** is used by default.

```
cubrid addvoldb -S -F /dbtemp/addvol/ testdb 1000
```

Purpose of the volume (-p)

The following example shows adding a volume for which 1000 pages are assigned to the *testdb* database in standalone mode. The **-p** option is used to specify the purpose of the volume to be added. The reason for specifying the purpose of the volume is to improve the I/O performance by storing volumes separately on different disk drives according to their purpose. Parameter values that can be used for the **-p** option are **data**, **index**, **temp** and **generic**. The default value is **generic**. For the purpose of each volume, see "Database Volume Structure."

```
cubrid addvoldb -S -p index testdb 1000
```

Standalone mode (-S)

The **-S** option is used to access the database in standalone mode without running the server process. This option has no parameter. If the **-S** option is not specified, the system assumes to be in client/server mode.

```
cubrid addvoldb -S testdb 1000
```

Client/server mode (-C)

The **-C** option is used to access the database in client/server mode by running the server and the client separately. There is no parameter. Even when the **-C** option is not specified, the system assumes to be in client/server mode by default. If the **-S** or **-C** option is not specified and the environment variable **CUBRID_MODE** is not defined, the system assumes to be in client/server mode.

```
cubrid addvoldb -C -testdb 1000
```

Comment about the added volume (--comment)

The following example shows adding a volume for which 1000 pages are assigned to the *testdb* database in standalone mode and inserts a comment about the volume. The **--comment** option is used to facilitate to retrieve information about the added volume by adding such information in the form of comments. It is recommended that the contents of a comment include the name of **DBA** who adds the volume, or the purpose of adding the volume. The comment must be enclosed in double quotes.

```
cubrid addvoldb -S --comment "data volume added_cheolsoo kim" testdb 1000
```

Deleting Database

Description

The **cubrid deletedb** utility is used to delete a database. You must use the **cubrid deletedb** utility to delete a database, instead of using the file deletion commands of the operating system; a database consists of a few interdependent files. The **cubrid deletedb** utility also deletes the information about the database from the database location file (**databases.txt**). The **cubrid deletedb** utility must be run offline, that is, in standalone mode when nobody is using the database.

Syntax

```
cubrid deletedb options database_name
options : [{-o|--output-file=} file]  [-d|--delete-backup]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **deletedb** : A command to delete a database, its related data, logs and all backup files. It can be executed successfully only when the database is in a stopped state.
- *options* : **-o** and **-d** options are provided.
- *database_name* : Specifies the name of the database to be deleted without including the path name.

Option

Saving output messages (-o **or** --output-file**)**

The following example shows deleting testdb and writes output messages to the file specified by using the **-o** option.

```
cubrid deletedb -o deleted_db.out testdb
```

The **cubrid deletedb** utility also deletes the database information contained in the database location file (**databases.txt**). The following message appears if you enter a utility that tries to delete a non-existing database.

```
cubrid deletedb testdb
Database "testdb" is unknown, or the file "databases.txt" cannot be accessed.
```

Deleting backup files simultaneously (-d **or** --delete-backup**)**

The following example shows deleting testdb and its backup volumes and backup information files simultaneously by using the **-d** option. If the **-d** option is not specified, backup volume and backup information files are not deleted.

```
cubrid deletedb -d testdb
```

Renaming Database

Description

The **cubrid renamedb** utility renames a database. The names of information volumes, log volumes and control files are also renamed to conform to the new database one.

The **cubrid alterdbhost** utility configures or changes the host name of the specified database. It changes the host name configuration in the **databases.txt** file.

Syntax

```
cubrid renamedb options src_database_name dest_database_name
options : [{-E | --extended-volumn-path=}path ] [ {-i | --control-file=} file ] [-d | --
delete-backup]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **renamedb** : A command that changes the existing name of a database to a new one. It executes successfully only when the database is in a stopped state. The names of related information volumes, log volumes and control files are also changed to new ones accordingly.
- *options* : The **-E**, **-i** and **-d** options are supported. For details about each option, see its description and the examples.
- *src_database_name* : The name of the existing database to be renamed. The path name to the directory where the database is to be created must not be included.
- *dest_database_name* : The new name of the database. It must not be the same as that of an existing database. The path name to the directory where the database is to be created must not be included.

Option

Saving the renamed extended volume to a new directory (-E or --extended-volume-path)

The following example shows renaming an extended volume created in a specific directory path (e.g. /dbtemp/addvol/) with a **-E** option, and then moves the volume to a new directory. The **-E** option is used to specify a new directory path (e.g. /dbtemp/newaddvols/) where the renamed extended volume will be moved. If the **-E** option is not specified, the extended volume is only renamed in the existing path without being moved. If a directory path outside the disk partition of the existing database volume or an invalid one is specified, the rename operation is not executed. This option cannot be used together with the **-i** option.

```
cubrid renamedb -E /dbtemp/newaddvols/ testdb testdb_1
```

Specifying the input file where the directory information is stored (-i or --control-file)

The following example shows specifying an input file which saves directory information with an **-i** option, to assign different directories as well as to change database names for each volume and file at once. The **-i** option cannot be used together with the **-E** option.

```
cubrid renamedb -i rename_path testdb testdb_1
```

The followings are the syntax and example of a file that contains the name of each volume, the current directory path and the directory path where renamed volumes will be saved.

```
volid   source_fullvolname   dest_fullvolname
```

- *volid* : An integer that is used to identify each volume. It can be checked in the database volume control file (database_name_vinf).
- *source_fullvolname* : The current directory path to each volume.
- *dest_fullvolname* : The target directory path where renamed volumes will be moved. If the target directory path is invalid, the database rename operation is not executed.

```
  -5  /home1/user/testdb_vinf       /home1/CUBRID/databases/testdb_1_vinf
  -4  /home1/user/testdb_lginf      /home1/CUBRID/databases/testdb_1_lginf
  -3  /home1/user/testdb_bkvinf     /home1/CUBRID/databases/testdb_1_bkvinf
  -2  /home1/user/testdb_lgat       /home1/CUBRID/databases/testdb_1_lgat
   0  /home1/user/testdb            /home1/CUBRID/databases/testdb_1
   1  /home1/user/backup/testdb_x001/home1/CUBRID/databases/backup/testdb_1_x001
```

Deleting and renaming backup files simultaneously (-d or --delete-backup)

By using the **-d** option, the following example shows renaming the testdb database and at the same time forcefully deletes all backup volumes and backup information files that are in the same location as testdb. Note that you cannot use the backup files with the old names once the database is renamed. If the **-d** option is not specified, backup volumes and backup information files are not deleted.

```
cubrid renamedb -d testdb testdb_1
```

Copying/Moving Database

Description

The **cubrid copydb** utility copy or move a database to another location. As arguments, source and target name of database must be given. A target database name must be different from a source database name. Wh the target name argument is specified, the location of target database name is registered in the **databases.txt** file. The **cubrid copydb** utility can be executed only offline (that is, state of a source database stop).

Syntax

```
cubrid copydb [OPTION] src-database-name dest-database-name

options : [{--server-name=}host] [{-F | --file-path=} database_path ] [ {-L | --log-path=}
log_path ] [{-B | --lob-base-path=} lob_file_path] [{-E | --extended-volume-path=}
path ][{-i, --control-file=}FILE ] [ -r | --replace ] [ -d | --delete-source ] [ --copy-
lob-path ]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **copyd** : A command that copy or move a database from one to another location.
- *options* : For details about each option, see its description and the examples. If options are omitted, a target database is copied into the same directory of a source database.
- *src_database_name* : The names of source and target databases to be copied or moved.
- *dest_database_name* : A new (target) database name.

Option

Registering the host name (--server-name)

The following example shows specifying a host name of new database. The host name is registered in the **databases.txt** file. If this option is omitted, a local host is registered.

```
cubrid copydb --server-name=cub_server1 demodb new_demodb
```

Storing a new database volume in a specific directory (-F or --file-path)

The following example shows specifying a specific directory path where a new database volume is stored with an **-F** option. It represents specifying an absolute path. If the specified directory does not exist, an error is outputted. If this option is omitted, a new database volume is created in the current working directory. And this information is specified in **vol-path** of the **databases.txt** file.

```
cubrid copydb -F /home/usr/CUBRID/databases demodb new_demodb
```

Storing a new database log volume in a specific directory (-L or --log-path)

The following example shows specifying a specific directory path where a new database volume is stored with an **-L** option. It represents specifying an absolute path. If the specified directory does not exist, an error is outputted. If this option is omitted, a new database volume is created in the current working directory. And this information is specified in **log-path** of the **databases.txt** file.

```
cubrid copydb -L /home/usr/CUBRID/databases/logs demodb new_demodb
```

Storing a new database extended volume in a specific directory (-E or --extended-volume-path)

The following example shows specifying a specific directory path where a new database extended volume is stored with an **-E**. If this option is omitted, a new database extended volume is created in the location of a new database volume or in the registered path of controlling file. The **-i** option cannot be used with this option.

```
cubrid copydb -E home/usr/CUBRID/databases/extvols demodb new_demodb
```

Specifying an input file where directory path information is stored (-i or --control file)

The following example shows specifying an input file where a new directory path information and a source volume are stored to copy or move multiple volumes into a different directory, respectively. The **-i** option cannot be used with the **-E** option. An input file named copy_path is specified in the example below.

```
cubrid copydb -i copy_path demodb new_demodb
```

The following is an exmaple of input file that contains each volume name, current directory path, and new directory and volume names.

```
# volid source_fullvolname dest_fullvolname
0 /usr/databases/demodb /drive1/usr/databases/new_demodb
1 /usr/databases/demodb_data1 /drive1/usr/databases/new_demodb new_data1
2 /usr/databases/ext/demodb index1 /drive2//usr/databases/new_demodb new_index1
3 /usr/ databases/ext/demodb index2 /drive2/usr/databases/new_demodb new_index2
```

- *volid* : An integer that is used to identify each volume. It can be checked in the database volume control file (**database_name_vinf**).
- *source_fullvolname* : The current directory path to each source database volume.
- *dest_fullvolname* : The target directory path where new volumes will be stored. You should specify a vaild path.

Overwriting if same database exists (-r or --replace)

If the **-r** option is specified, a new database name overwrites the existing database name if it is identical, insteading outputting an error.

```
cubrid copydb -r -F /home/usr/CUBRID/databases demodb new_demodb
```

Deleting a source database if is is copied (-d or --delete-source)

If the **-d** option is specified, a source database is deleted after the database is copied. This execution brings the same the result as executing **cubrid deletedb** utility after copying a database. Note that if a source database contains LOB data, LOB file directory path of a source database is copied into a new database and it is registered in the **lob-base-path** of the **databases.txt** file.

```
cubrid copydb -d -copyhome/usr/CUBRID/databases demodb new_demodb
```

Copying LOB file directory (--copy-lob-path)

If the **--copy-lob-path** option is specified, a new directory path for LOB files is created and a source database is copied into a new directory path. If this option is omitted, the directory path is not created. Therefore, the **lob-base-path** of the **databases.txt** file should be modified separately. This option cannot be used with the **-B** option.

```
cubrid copydb --copy-lob-path demodb new_demodb
```

Copying LOB file directory simultaneously with specifying it (-B or --delete-backup)

If the **-B** option is specified, a specified directory is specified as for LOB files of a new database and a source database is copied. This option cannot be used with the **--copy-lob-path** option.

```
cubrid copydb -B /home/usr/CUBRID/databases/new_lob demodb new_demodb
```

Installing Database

Description

The **cubrid installdb** utility is used to register the information of a newly installed database to **databases.txt**, which stores database location information. The execution of this utility does not affect the operation of the database to be registered.

Syntax

```
cubrid installdb options database_name
options : [{--server-name=} host] [{-F|--file-path=} database_path] [-L| --log-path=}
log_path]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **installdb** : A command that registers the information of a moved or copied database to **databases.txt**.
- *options* : **--server-name**, **-F**, **-L** options are available. For more information about each option, see the option description and example. If no option is used with a command, the command must be executed in the directory where the corresponding database exists.
- *database_name* : The name of database to be registered to **databases.txt**.

Option

Registering the host name (--server-name)

The following example shows registering the server host information of a database to **databases.txt** with a specific host name. If this option is not specified, the current host information is registered.

```
cubrid installdb --server-name=cub_server1 testdb
```

Registering the directory path of a database volume (-F or ?file-path)

The following example shows registering the directory path of a database volume to **databases.txt** with an **-F** option. If this option is not specified, the path of a current directory is registered as default.

```
cubrid installdb ?F /home/cubrid/CUBRID/databases/testdb testdb
```

Registering the directory path of a database log volume (-L or ?log-path)

The following example shows registering the directory path of a database log volume to **databases.txt** with an **-L** option. If this option is not specified, the directory path of a volume is registered.

```
cubrid installdb ?L /home/cubrid/CUBRID/databases/logs/testdb testdb
```

Checking Used Space

Description

The **cubrid spacedb** utility is used to check how much space of database volumes is being used. It shows a brief description of all permanent data volumes in the database. Information returned by the **cubrid spacedb** utility includes the ID, name, purpose and total/free space of each volume. You can also check the total number of volumes and used/unused database pages.

Syntax

```
cubrid spacedb options database_name
options : [{-o|--output-file=}file] [--size_unit=PAGE|M|G|T|H] [-S|--SA-mode | -C|--CS-
mode]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **spacedb** : A command that checks the space in the database. It executes successfully only when the database is in a stopped state.
- *options* : The **-o**, **-S**, **-C** and **--size_unit** options are supported. For details about each option, refer to its description and the examples.
- *database_name* : The name of the database whose space is to be checked. The path-name to the directory where the database is to be created must not be included.

Option

Saving output messages to a file (-o)

The above example shows saving the result of checking the space information of testdb to a file named *db_output*.

```
cubrid spacedb -o db_output testdb
```

Executing in stand-alone mode (-S or --SA-mode)

The **-S** option is used to access a database in standalone, which means it works without processing server; it does not have an argument. If **-S** is not specified, the system recognizes that a database is running in client/server mode.

```
cubrid spacedb --SA-mode testdb
```

Executing in client/server mode (-C or --CS-mode)

The **-C** option is used to access a database in client/server mode, which means it works in client/server process respectively; it does not have an argument. If **-C** is not specified, the system recognize that a database is running in client/server mode by default.

```
cubrid spacedb --CS-mode testdb
```

Outputing in megabytes (--size_unit=M)

```
cubrid spacedb --size_unit=M testdb
```

Outputing in print-friendly version (--size_unit=H)

The unit is automatically determined as follows: M if 1MB = DB size < 1024MB, G if 1GB = DB size < 1024GB.

```
cubrid spacedb --size_unit=H testdb
```

Compacting Used Space

Description

The **cubrid compactdb** utility is used to secure unused space of the database volume. In case the database server is not running (offline), you can perform the job in stand-alone mode. In case the database server is running, you can perform it in client-server mode.

The **cubrid compactdb** utility secures the space being taken by OIDs of deleted objects and by class changes. When an object is deleted, the space taken by its OID is not immediately freed because there might be other objects that refer to the deleted one. Reference to the object deleted during compacting is displayed as **NULL**, which means this can be reused by OIDs.

Syntax

```
cubrid compactdb options database_name [class_name], class_name2,...]
options : [-v | --verbose] [-S|--SA-mode | -C| --CS-mode]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **compactdb** : A command that compacts the space of the database so that OIDs assigned to deleted data can be reused.
- *options* : The **-v** , **-S**, and **-C** options are supported. Options (**-I**, **-i**, **-c**, **-d**, **-p**) that is applied in client/server mode only.

- *database_name* : The name of the database whose space is to be compacted. The path name to the directory where the database is to be created must not be included.
- *class_name_list* : You can specify the list of tables names that you want to compact space after a database name; the -i option cannot be used together. It is used in client/server mode only.

Option

Outputting detailed messages during execution (-v)

You can output messages that shows which class is currently being compacted and how many instances have been processed for the class by using the **-v** option.

```
cubrid compactdb -v testdb
```

Executing in stand-alone mode (-S or --SA mode)

The **-S** option is specified to compact used space in stand-alone mode while database server is not running; no arugment is specified. If the **-S** option is not specified, system recognizes that the job is executed in client/server mode.

```
cubrid compactdb --SA-mode testdb
```

Executing in client/server mode (C or --CS mode)

The **-C** option is specified to compact used space in client/server mode while database server is running; no argument is specified. Even though this option is omitted, system recognizes that the job is executed in client/server mode. The following options can be used in client/server mode only.

- - i, --input-class-file=FILE

  You can specify an input file name that contains the table table name with this option. Write one table name in a single line; invalid table name is ignored. Note that you cannot specify the list of the table names after a database name in case of you use this option.
- -p, --page-commited-once = NUMBER

  You can specify the number of maximum pages that can be commited once with this option. The default value is 10, the minimum value is 1, and the maximum value is 10. The less option value is specified, the more concurrency is enhanced because the value for class/instance lock is small; however, it causes slowdown on operation, and vice versa.
- -d, --delete-old-repr

  You can delete an existing table representation from catalog with this option.
- -I, --Instance-lock-timeout

  You can specify a value of instance lock timeout with this option. The default value is 2(second), the minimum value is 1, and the maximum value is 10. The less option value is specified, the more operation speeds up. However, the number of instances that can be processed becomes smaller, and vice versa.
- -c, --class-lock-timeout

  You can specify a value of instance lock timeout with this option. The default value is 10(second), the minimum value is 1, and the maximum value is 10. The less option value is specified, the more operation speeds up. However, the number of tables that can be processed becomes smaller, and vice versa.

```
cubrid compactdb --CS-mode -p 10 testdb tbl1, tbl2, tbl5
```

Updating Statistics

Description

Updates statistical information such as the number of objects, the number of pages to access, and the distribution of attribute values.

Syntax

```
cubrid optimizedb options database_name
options : [{-n|--class-name=} name]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **optimizedb** : Updates the statistics information, which is used for cost-based query optimization of the database. If the option is specified, only the information of the specified class is updated.
- *options* : The **-n** option is supported.
- *database_name* : The name of the database whose cost-based query optimization statistics are to be updated.

Option

Updating the query statistics of the target database

The following example shows updating the query statistics information of all classes in the database.

```
cubrid optimizedb testdb
```

Updating the query statistics of a specific class in the database (-n or --class-name)

The following example shows updating the query statistics information of the given class by using the **-n** option.

```
cubrid optimizedb -n event_table testdb
```

Outputting Statistics Information of Server

Description

The cubrid statdump utility checks statistics information processed by the CUBRID database server. The statistics information mainly consists of the followings: File I/O, Page buffer, Logs, Transactions, Concurrency/Lock, Index, and Network request

Note that you must specify the parameter **communication_histogram** to **yes** in the **cubrid.conf** before executing the utility. You can also check statistics information of server with session commands (**;.h on** or **;.h all**) in the CSQL.

Syntax

```
cubrid statdump options database_name
options : [{-o|--output-file=} file_name]] [{-i|--interval=} secs] [-c|--cumulative]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **installdb** : A command that dumps the statistics information about the database server execution.
- *options* : **--o**, **-i**, and **-c** options are available.
- *database_name* : The name of database which has the statistics data to be dumped.

Option

Outputting statistics information periodically (-i or --interval)

```
cubrid statdump -i 5 testdb

Wed March 31 11:23:56 KST 2010

*** SERVER EXECUTION GLOBAL STATISTICS ***
Num_file_creates = 0
Num_file_removes = 0
Num_file_ioreads = 0
Num_file_iowrites = 0
Num_file_iosynches = 0
Num_data_page_fetches = 0
Num_data_page_dirties = 0
Num_data_page_ioreads = 0
Num_data_page_iowrites = 0
Num_data_page_victims = 0
Num_data_page_iowrites_for_replacement = 0
Num_log_page_ioreads = 0
Num_log_page_iowrites = 0
Num_log_append_records = 0
Num_log_archives = 0
Num_log_checkpoints = 0
```

```
Num_log_wals = 0
Num_page_locks_acquired = 0
Num_object_locks_acquired = 0
Num_page_locks_converted = 0
Num_object_locks_converted = 0
Num_page_locks_re-requested = 0
Num_object_locks_re-requested = 0
Num_page_locks_waits = 0
Num_object_locks_waits = 0
Num_tran_commits = 0
Num_tran_rollbacks = 0
Num_tran_savepoints = 0
Num_tran_start_topops = 0
Num_tran_end_topops = 0
Num_tran_interrupts = 0
Num_btree_inserts = 0
Num_btree_deletes = 0
Num_btree_updates = 0
Num_query_selects = 0
Num_query_inserts = 0
Num_query_deletes = 0
Num_query_updates = 0
Num_query_sscans = 0
Num_query_iscans = 0
Num_query_lscans = 0
Num_query_setscans = 0
Num_query_methscans = 0
Num_query_nljoins = 0
Num_query_mjoins = 0
Num_query_objfetches = 0
Num_network_requests = 2
Num_adaptive_flush_pages = 0
Num_adaptive_flush_log_pages = 0
Num_adaptive_flush_max_pages = 90

*** OTHER STATISTICS ***
Data_page_buffer_hit_ratio = 0.00
```

Category of Statistics Information

Category	Item	Description
File I/O	Num_file_removes	The number of files removed
	Num_file_creates	The number of files created
	Num_file_ioreads	The number of files read
	Num_file_iowrites	The number of files saved
	Num_file_iosynches	The number of file synchronization
Page buffer	Num_data_page_fetches	The number of pages fetched
	Num_data_page_dirties	The number of duty pages
	Num_data_page_ioreads	The number of pages read
	Num_data_page_iowrites	The number of pages saved
Logs	Num_log_page_ioreads	The number of log pages read
	Num_log_page_iowrites	The number of log pages saved
	Num_log_append_records	The number of log records appended
	Num_log_archives	The number of logs archived
	Num_log_checkpoints	The number of checkpoints
Transactions	Num_tran_commits	The number of commits
	Num_tran_rollbacks	The number of rollbacks
	Num_tran_savepoints	The number of savepoints

	Num_tran_start_topops	The number of top operations started
	Num_tran_end_topops	The number of top perations stopped
	Num_tran_interrupts	The number of interruptions
Concurrency/lock	Num_page_locks_acquired	The number of locked pages acquired
	Num_object_locks_acquired	The number of locked objects acquired
	Num_page_locks_converted	The number of locked pages converted
	Num_object_locks_converted	The number of locked objects converted
	Num_page_locks_re-requested	The number of locked pages requested
	Num_object_locks_re-requested	The number of locked objects requested
	Num_page_locks_waits	The number of locked pages waited
	Num_object_locks_waits	The number of locked objects waited
Index	Num_btree_inserts	The number of nodes inserted
	Num_btree_deletes	The number of nodes deleted
	Num_btree_updates	The number of nodes updated
Query (Servuce Workload)	Num_query_selects	The number of SELECT requested
	Num_query_inserts	The number of INSERT queries
	Num_query_deletes	The number of DELETE queries
	Num_query_updates	The number of UPDATE queries
	Num_query_sscans	The number of sequential scans (full scan)
	Num_query_iscans	The number of index scans
	Num_query_lscans	The number of LIST scans
	Num_query_setscans	The number of SET scans
	Num_query_methscans	The number of METHOD scans
	Num_query_nljoins	The number of nested loop joins
	Num_query_mjoins	The number of parallel joins
	Num_query_objfetches	The number of fetch objects
Network request	Num_network_requests	The number of networks requested
	Data_page_buffer_hit_ratio	Hit Ratio of page buffers(Num_data_page_fetches - Num_data_page_ioreads)*100 / Num_data_page_fetches

Saving statistics information to a file (-o or --output-file)

The **-o** options is used to save statistics information of server processing for the database to a specified file.

```
cubrid statdump -o statdump.log testdb
```

Displays the accumulated operation statistics information (-c or --cumulative)

You can display the accumulated operation statistics information of the target database server by using the -c option. By combining this with the ?i option, you can check the operation statistics information that is not reset at a specified interval.

```
cubrid statdump ?i 5 ?c testdb
```

Checking Lock Status

Description

The **cubrid lockdb** utility is used to check the information about the lock being used by the current transaction in the database.

Syntax

```
cubrid lockdb options database_name
options : [{-o|--output-file=} file ]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **lockdb** : A command used to check the information about the lock being used by the current transaction in the database.
- *options* : The **-o** option is supported.
- *database_name* : The name of the database where lock information of the current transaction is to be checked.

Option

Displaying the lock information on the screen

The following example shows displaying lock information of the testdb database on a screen without any option.

```
cubrid lockdb testdb
```

Displaying the lock information to the specified file (-o)

The following example shows displaying lock information of the testdb database as a output.txt by using the **-o** option.

```
cubrid lockdb -o output.txt testdb
```

Checking Database Consistency

Description

The **cubrid checkdb** utility is used to check the consistency of a database. You can use **cubrid checkdb** to identify data structures that are different from indexes by checking the internal physical consistency of the data and log volumes. If the **cubrid checkdb** utility reveals any inconsistencies, you must try automatic repair by using the **--repair** option.

Syntax

```
cubrid checkdb options database_name
options : [-S|--SA-mode | -C|--CS-mode]  [-r | --repair]
```

- **cubrid** : An integrated utility for CUBRID service and database management.
- **checkdb** : A utility that checks the data consistency of a specific database.
- *options* : **-S**, **-C** and **-r** options are supported.
- *database_name* : The name of the database whose consistency status will be either checked or repaired.

Option

Checking the database consistency in standalone mode (-S or --SA-mode)

The **-S** option is used to access a database in standalone, which means it works without processing server; it does not have an argument. If **-S** is not specified, the system recognizes that a database is running in client/server mode.

```
cubrid checkdb -S testdb
```

Checking the database consistency in client/server mode (-C or --CS-mode)

The **-C** option is used to access a database in client/server mode, which means it works in client/server process respectively; it does not have an argument. If **-C** is not specified, the system recognize that a database is running in client/server mode by default.

```
cubrid checkdb -C testdb
```

Repairing in case of a database consistency problem (-r or --repair)

The **-r** option is used to repair an issue if a consistency error occurs in a database.

```
cubrid checkdb -r testdb
```

Killing Database Transactions

Description

The **cubrid killtran** is used to check transactions or abort specific transaction. Only a DBA can execute this utility.

Syntax

```
cubrid killtran options database_name
options :
[{-i|--kill-transaction-index=}index] [--kill-user-name=id] [--kill-host-name=host] [--
kill-program-name=program_name] [{-p|--dba-password=}password] [-d|--display-information]
[-f|--force]
```

- **cubrid** : An integrated utility for the CUBRID service and database management
- **killtran** : A utility that manages transactions for a specified database
- *options* : Some options refer to killing specified transactions; others refer to outputting active transactions. If no option is specified, **-d** is specified by default so all transactions are outputted on the screen. **-p** A value followed by the -p option is a password of the **DBA**, and should be entered in the prompt.
- *database_name* : The name of database whose transactions are to be killed

Option

Outputting all transactions (no option)

```
cubrid killtran testdb

Tran index      User name      Host name      Process id      Program name
-------------------------------------------------------------------------------
      1(+)            dba      myhost             664           cub_cas
      2(+)            dba      myhost            6700              csql
      3(+)            dba      myhost            2188           cub_cas
      4(+)            dba      myhost             696              csql
      5(+)         public      myhost            6944              csql
-------------------------------------------------------------------------------
```

Killing transactions in a specified index (-i or --kill-transation-index)

```
cubrid killtran -i 1 testdb

Ready to kill the following transactions:

Tran index      User name      Host name      Process id      Program name
-------------------------------------------------------------------------------
      1(+)            dba      myhost            4760              csql
-------------------------------------------------------------------------------
Do you wish to proceed ? (Y/N)y
Killing transaction associated with transaction index 1
```

Outputting all transactions (-d or --display)

```
cubrid killtran -d testdb

Tran index      User name      Host name      Process id      Program name
-------------------------------------------------------------------------------
      2(+)            dba      myhost            6700              csql
      3(+)            dba      myhost            2188           cub_cas
      4(+)            dba      myhost             696              csql
      5(+)         public      myhost            6944              csql
-------------------------------------------------------------------------------
```

Killing transactions for a specified OS user ID (--kill-user-name)

```
cubrid killtran --kill-user-name=os_user_id testdb
```

Killing transactions for a specified client host (--kill- host-name)

```
cubrid killtran --kill-host-name=myhost testdb
```

Killing transactions for a specified program (--kill-program-name)

```
cubrid killtran --kill-program-name=cub_cas testdb
```

Omitting a prompt to check transactions to be stopped (-f or --force)

```
cubrid killtran -f -i 1 testdb
```

Checking the Query Plan Cache

Description

The **cubrid plandump** utility is used to display information about the query plans saved (cached) on the server.

Syntax

```
cubrid plandump options database_name
options : [-d|--drop]  [{-o|--output-file=} file]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **plandump** : A utility that displays the query plans saved in the current cache of a specific database.
- *options* : The **-d** and **-o** options are supported.
- *database_name* : The name of the database where the query plans are to be checked or dropped from its sever cache.

Option

Checking the query plans saved in the cache

```
cubrid plandump testdb
```

Dropping the query plans saved in the cache (-d or --drop)

```
cubrid plandump -d testdb
```

Saving the results of the query plans saved in the cache to a file (-o or --output)

```
cubrid plandump -o output.txt testdb
```

Restoring Emergency Database Logs

Description

It is used to restore the damaged log file when the database has been aborted due to damage to the log file caused by a system conflict and cannot be restarted. To modify the damaged database log, execute the utility without options; to create a new log file, execute the utility with the **-r** option.

Syntax

```
cubrid emergency_patchlog options database_name
options : [ -r | --recreate-log ]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.

- **emergency_patchlog** : It is used to restore the log file. It normally executes only when the database is in a stopped state. At first, you must execute the utility without the **-r** option to restore corrupted database.
- *options* : You should use the **-r** option only when necessary logs do not exist or you cannot restore the corrupted logs. The size of the newly created log file is the size of the generic log volume.
- *database_name* : The name of the database to be restored.

Option

Restoring with the existing log (1st execution)

```
cubrid emergency_patchlog testdb
```

Discarding the existing log and creating a new empty log (-r) (2nd execution)

```
cubrid emergency_patchlog -r testdb
```

Outputting Internal Database Information

Description

You can check various pieces of internal information about the database with the **cubrid diagdb** utility. Information provided by **cubrid diagdb** is helpful in diagnosing the current status of the database or figuring out a problem.

Syntax

```
cubrid diagdb options database_name
options : [{-d | --dump-type=} type]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **diagdb** : A command that is used to check the current storage state of the database by outputting the information contained in the binary file managed by CUBRID in text format. It normally executes only when the database is in a stopped state. You can check the whole database or the file table, file size, heap size, class name or disk bitmap selectively by using the provided option.
- *options* : The **-d** option is provided.
- *database_name* : The name of the database to be diagnosed.

Option

Specifying the output range (-d or --dump-type)

The following example displays the information of all files in the testdb database. If any option is not specified, the default value of 1 is used.

```
cubrid diagdb -d 1 myhost testdb
```

The utility has 9 types of -d options as follows:

Type	Description
-1	Outputs all database information.
1	Outputs file table information.
2	Outputs file capacity information.
3	Outputs heap capacity information.
4	Outputs index capacity information.
5	Outputs class name information.
6	Outputs disk bitmap information.
7	Outputs catalog information.
8	Outputs log information.

9	Outputs hip information.

Backup and Restore

DBA must perform regular backups of the database so that it can be restored successfully to a state at a certain point in time in case of system failure. For more information, see Database Backup.

Export and Import

To use a newer version of CUBRID database, the existing version must be migrated to a new one. For this purpose, you can use "Export to a ASCII text file" and "Import from a ASCII text file" features provided by CUBRID. For more information on export and import, see Migrating Database.

Outputting Parameters Used in Server/Client

Description

The **cubrid paramdump** utility outputs parameter information used in the server/client process.

Syntax

```
cubrid paramdump options database_name
options : [{-o|--output-file=}filename]  [{-b|--both}]  [{-S|--SA-mode}]  [{-C|--CS-mode}]
```

- **cubrid** : An integrated utility for the CUBRID service and database management
- **paramdump** : A utility that outputs parameter information used in the server/client process
- *options* : A short name option starts with a single dash (-) while a full name option starts with a double dash (--). **-o**, **-b**, **-S** and **-C** options are provided.
- *database_name* : The name of the database in which parameter information is to be outputted

Option

Saving the output information to a file (-o)

The **-o** option is used to save information of the parameters used in the server/client process of the database into a specified file. The file is created in the current directory. If the **-o** option is not specified, messages are displayed on the console screen.

```
cubrid paramdump -o db_output testdb
```

Outputting information of the server/client parameters (-b)

The **-b** option is used to output parameter information used in server/client process into a console screen. If the **-b** option is not specified, only server-side information is outputted.

```
cubrid paramdump -b testdb
```

Outputting parameter information of the server process in standalone mode (-S or --SA-mode)

```
cubrid paramdump -S testdb
```

Outputting parameter information of the server process in client/server mode (-C or --CS-mode)

```
cubrid paramdump -C testdb
```

Database Migration

Migrating Database

To use a newer version of CUBRID database, you might migrate an existing data to a new one. For this purpose, you can use the "Export to a ASCII text file" and "Import from a ASCII text file" features provided by CUBRID. The following section explains migration steps using the **cubrid unloaddb** and **cubrid loaddb** utilities.

Recommended scenario and procedures

The following is an explanation of a migration scenario that can be applied while the existing version of CUBRID is running. For database migration, the **cubrid unloaddb** and **cubrid loaddb** utilities are used. For more information, see Unloading Database and Loading Database.

1. Back up the existing database

   Back up the existing version of the database by using the **cubrid backupdb** utility. The purpose of this step is to safeguard against failures that might occur during the database unload/load operations. For more information about the database backup, see Database Backup.

2. Unload the existing database

   Unload the database created for the existing version of CUBRID by using the **cubrid unloaddb** utility. For more information about the database unload, see Unloading Database.

3. Storing the existing CUBRIDG configuration files

   Save configurations files such as **cubrid.conf**, **cubrid_broker.conf** and **cm.conf** located in the **CUBRID/conf** directory. The purpose of this step is to conveniently apply parameter values for the existing CUBRID database environment to the new one.

4. Install a new version of CUBRID

   Once backing up and unloading of the data created by the existing version of CUBRID have been completed, delete the existing version of CUBRID and its databases and then install the new version of CUBRID. For more information about installing CUBRID, see Installing and Running on Linux in "Getting Started."

5. Configure the new CUBRID

   You can configure the new version of CUBRID by referring to configuration files of the existing database saved in the step 3, "**Save configuration files of the existing version of CUBRID**." For more information about configuration, see Installing and Running on Windows in "Getting Started."

6. Load the new database

   Create a database by using the **cubrid createdb** utility and then use the **cubrid loaddb** utility to load into the new database the data which had previously been unloaded. For more information about creating a database, see Creating Database in "Administrator's Guide." For more information about database loading, see Loading Database.

7. Back up the new database

   Once the data has been successfully loaded into the new database, back up the database created for the new version of CUBRID by using the **cubrid backupdb** utility. The reason for this step is because you cannot restore the data backed up in the existing version of CUBRID when using the new version. For more information about backing up the database, see Database Backup.

Unloading Database

Description

The purposes of unloading/loading a database are as follows:

- To reconstruct the database by rebuilding the database volume
- To perform migration to a different system environment
- To perform migration to a different version of the DBMS

Syntax

```
cubrid unloaddb [ options ] database_name
[ options ]
-i | -O | -s | -d | -v | -S | -C |
--input-class-file | --output-path | --schema-only | --data-only | --verbose | --SA-mode |
--CS-mode | --include-reference | --input-class-only | --lo-count | --estimated-size | --
cached-pages | --output-prefix | --hash-file | --use-delimiter
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **unloaddb** : A utility that creates ASCII files from a database. It is used together with the **cubrid loaddb** utility for replacing system, upgrading product version or reorganizing database volumes. It can be used both in standalone and client/server modes. Data can be unloaded even when the database is running.
- *options* : A short option starts with a single dash (-) while a full name option starts with a double dash (--). Note that options are case sensitive.
- *database_name* : Specifies the name of the database to be unloaded.

Return value

Return values of **cubrid unloaddb** utility are as follows:

- 0 : Success
- Non-zero : Failure

Generated Files

- Schema file (*database-name*_**schema**) : A file that contains information about the schema defined in the database.
- Object file (*database-name*_**objects**) : A file that contains information about the records in the database.
- Index file (*database-name*_**indexes**) : A file that contains information about the indexes defined in the database.
- Trigger file (*database-name*_**trigger**) : A file that contains information about the triggers defined in the database. If you don't want triggers to be running while loading the data, load the trigger definitions after the data loading has completed.

Schema, object, index and trigger files are created in the same directory.

Option

The following table shows options that can be used with **cubrid unloaddb** utility. Options are case sensitive.

Option	Description
-i --input-class-file	Unloads the database class into the input file specified in an argument.
-O --output-path	Specifies the directory in which to create schema and object files. If the option is not specified, files are created in the current directory.
-s --schema-only	Creates only the schema file, not the data file.
-d --data-only	Creates only the data file, not the schema file.
-v --verbose	Displays detailed information about the database being unloaded.
-S --SA-mode	Unloads the database in standalone mode.
-C --CS-mode	Unloads the database in client/server mode.
--include-reference	Unloads the object reference as well when the specified database class is unloaded with the **-i** option.
--input-class-only	Is used with the **-i** option. Creates only the schema files which are related to tables included in the input file.

--lo-count	Specifies the number of large object (LO) data files to be created in a single directory. Default value : 0
--estimated-size	Specifies the number of records expected.
--cached-pages	Configures the number of object tables to be cached in the memory. Default value : 100
--output-prefix	Specifies the prefix for schema and object file names.
--hash-file	Specifies the name of the hash file.
--use-delimiter	Outputs the attribute name enclosed in quotes.

Input file with the list of tables to be unloaded (-i or --input-class-file)

The following is an example of a input file (table_list.txt).

```
table_1
table_2
..
table_n
```

The **-i** option specifies the input file where the list of tables to be unloaded is stored so that only specified part of the database can be unloaded.

```
cubrid unloaddb -i table_list.txt demodb
```

The **-i** option can be used together with the **--input-class-only** option that creates the schema file related to only those tables included in the input file.

```
cubrid unloaddb --input-class-only -i table_list.txt demodb
```

The **-i** option can be used together with the **--include-reference** option that creates the object reference as well.

```
cubrid unloaddb --include-reference -i table_list.txt demodb
```

Specifying the directory where files created will be saved (-O or --output-path)

The **-O** option specifies the directory where the output files generated by the unload operation is saved. If the **-O** option is not specified, output files are created in the current working directory.

```
cubrid unloaddb -O ./CUBRID/Databases/demodb demodb
```

If the specified directory does not exist, the following error message will be displayed.

```
unloaddb: No such file or directory.
```

Creating the schema file only (-s or --schema-only)

The **-s** option specifies that only the schema file will be created from amongst all the output files which can be created by the unload operation.

```
cubrid unloaddb -s demodb
```

Creating the data file only (-d or -data-only)

The **-d** option specifies that only the data file will be created from amongst all of the output files which can be created by the unload operation.

```
cubrid unloaddb -d demodb
```

Displaying the unload status information (-v or --verbose)

The **-v** option displays detailed information about the database tables and records being unloaded while the unload operation is under way.

```
cubrid unloaddb -v demodb
```

Standalone mode (-S or --SA-mode)

The **-S** option performs the unload operation by accessing the database in standalone mode.

```
cubrid unloaddb -S demodb
```

Client/server mode (-C or --CS-mode)

The **-C** option performs the unload operation by accessing the database in client/server mode.

```
cubrid unloaddb -C demodb
```

Number of estimated records (--estimated-size)

The **--estimated-size** option allows you to assign hash memory to save records of the database to be unloaded. If the **--estimated-size** option is not specified, the number of records of the database is determined based on recent statistics information. This option can be used if the recent statistics information has not been updated or if a large amount of hash memory needs to be assigned. Therefore, if the number given as the argument for the option is too small, the unload performance deteriorates due to hash conflicts.

```
cubrid unloaddb --estimated-size 1000 demodb
```

Number of pages to be cached (--cached-pages)

The **--cached-pages** option specifies the number of pages of tables to be cached in the memory. Each page is 4,096 bytes. The administrator can configure the number of pages taking into account the memory size and speed. If this option is not specified, the default value is 100 pages.

```
cubrid unloaddb --cached-pages 500 demodb
```

Specifying the prefix for the name of the file to be created (--output-prefix)

The **--output-prefix** option specifies the prefix for the names of schema and object files created by the unload operation. Once the example is executed, the schema file name becomes abcd_schema and the object file name becomes abcd_objects. If the **--output-prefix** option is not specified, the name of the database to be unloaded is used as the prefix.

```
cubrid unloaddb --output-prefix abcd demodb
```

Loading Database

Description

You can load a database by using the **cubrid loaddb** utility in the following scenarios:

- When migrating a previous CUBRID database version to a new version
- When migrating a database of third-party DBMS to a CUBRID database
- When entering mass data faster than executing the **INSERT** statement

Generally, the **cubrid loaddb** utility uses files created by the **cubrid unloaddb** utility (schema definition file, object input file and index definition file).

Syntax

```
cubrid loaddb [ options ] database_name
[ options ]
-u | -p | -l | -v | -c | -s | -i | -d |
--user | --password | --load-only | --verbose | --periodic-commit |  --schema-file | --
index-file | --data-file | --data-file-check-only | --estimated-size | --no-oid | --no-
statistics | --ignore-class-file |--error-control-file |
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **loaddb** : A utility loads files which is generated by the unload operation and then creates a new database. It is also used to enter mass data into a database faster than ever by loading the input file written by a user. Database loading is performed in standalone mode with **DBA** authorization.
- *options* : A short name option starts with a single dash (-) while a full name option starts with a double dash (--). The options are case sensitive.

- *database_name* : Specifies the name of the database to be created.

Return value

Return values of **cubrid loaddb** utility are as follows:

- 0 : Success
- Non-zero : Failure

Input file

- Schema file (*database-name*_**schema**) : A file generated by the unload operation; it contains schema information defined in the database.
- Object file (*database-name*_**objects**) : A file created by an unload operation. It contains information about the records in the database.
- Index file (*database-name*_**indexes**) : A file created by an unload operation. It contains information about the indexes defined in the database.
- Trigger file (*database-name*_**trigger**) : A file created by an unload operation. It contains information about the triggers defined in the database.
- User-defined object file (*user_defined_object_file*) : A file in table format written by the user to enter mass data.

Option

The following table shows options that can be used with **cubrid loaddb** utility. The options are case sensitive.

Option	Description
-u **--user**	Enters the database user's account. The default value is **PUBLIC**.
-p **--password**	Enters the database user's password.
-l **--load-only**	Skips checking statements and types included in the object file and loads records.
-v **--verbose**	Displays detailed information about the data loading status on the screen.
-c **--periodic-commit**	Commits the transaction whenever a specified number of records has been entered.
-s **--schema-file**	Specifies the schema file created by the unload operation and performs schema loading.
-i **--index-file**	Specifies the index file created by the unload operation and loads indexes.
-d **--data-file**	Specifies the data file created by the unload operation and loads records.
--data-file-check-only	Performs checking only for statements and types included in the data file, but does not load records.
--estimated-size	Specifies the number of records expected.
--no-oid	Ignores the OID reference relationship included in the data file and loads records.
--no-statistics	Loads records without updating database statistics information.
--ignore-class-file	Specifies the ignoring classes.
--error-control-file	Specifies the file that describes how to handle specific errors occurring during data loading.

Entering a user account (-u or --user)

The **-u** option specifies the user account of a database where records are loaded. If the option is not specified, the default value is **PUBLIC**.

```
cubrid loaddb -u admin -d demodb_objects newdb
```

Entering the password (-p or --password)

The **-p** option specifies the password of a database user who will load records. If the option is not specified, you will be prompted to enter the password.

```
cubrid loaddb -p admin -d demodb_objects newdb
```

Loading records without checking syntax (-l or --load-only)

The **-l** option loads data directly without checking the syntax for the data to be loaded. The following example is a statement that loads data included in demodb_objects to newdb.
If the **-l** option is used, loading speed increases because data is loaded without checking the syntax included in demodb_objects, but an error might occur.

```
cubrid loaddb -l -d demodb_objects newdb
```

Displaying the loading status information (-v or --verbose)

The following is a statement that outputs detailed information about the tables and records of the database being loaded while the database loading operation is performed. You can check the detailed information such as the progress level, the class being loaded and the number of records entered by using the **-v** option.

```
cubrid loaddb -v -d demodb_objects newdb
```

Configuring the commit interval (-c or --periodic-commit)

The following command performs commit regularly every time 100 records are entered into the newdb by using the **-c** option. If the **-c** option is not specified, all records included in demodb_objects are loaded to newdb before the transaction is committed. If the **-c** option is used together with the **-s** or **-i** option, commit is performed regularly every time 100 DDL statements are loaded. The recommended commit interval varies depending on the data to be loaded. It is recommended that the parameter of the **-c** option be configured to 50 for schema loading, 1,000 for record loading, and 1 for index loading.

```
cubrid loaddb -c 100 -d demodb_objects newdb
```

Schema loading (-s or --schema-file)

The following statement loads the schema information defined in demodb into the newly created newdb database. demodb_schema is a file created by the unload operation and contains the schema information of the unloaded database. You can load the actual records after loading the schema information first by using the **-s** option.

```
cubrid loaddb -u dba -s demodb_schema newdb

Start schema loading.
Total 86 statements executed.
Schema loading from demodb_schema finished.
Statistics for Catalog classes have been updated.
```

Index loading (-i or --index-file)

The following command loads the index information defined in demodb into the newly created newdb database. demo_indexes is a file created by the unload operation and contains the index information of the unloaded database. You can create indexes after loading records by using the **-i** option together with the **-d** option.

```
cubrid loaddb -u dba -i demodb_indexes newdb
```

Data loading (-d or -data-file)

The following command loads the record information into newdb by specifying the data file or the user-defined object file with the **-d** option. demodb_objects is either an object file created by the unload operation or a user-defined object file written by the user for mass data loading.

```
cubrid loaddb -u dba -d demodb_objects newdb
```

Checking the syntax for the data to be loaded only (--data-file-check-only)

The following is a command that checks the statements for the data contained in demodb_objects by using the **--data-file-check-only** option. Therefore, the execution of the command below does not load records.

```
cubrid loaddb --data-file-check-only -d demodb_objects newdb
```

Number of expected records (--estimated-size)

The **--estimated-size** option can be used to improve loading performance when the number of records to be unloaded exceeds the default value of 5,000. That is, you can improve the load performance by assigning large hash memory for record storage with this option.

```
cubrid loaddb --estimated-size 8000 -d demodb_objects newdb
```

Loading records while ignoring the reference relationship (--no-oid**)**

The following is a command that loads records into newdb ignoring the OIDs in demodb_objects.

```
cubrid loaddb --no-oid -d demodb_objects newdb
```

Loading records without updating statistics information (--no-statistics)

The following is a command that does not update the statistics information of newdb after loading demodb_objects. It is useful especially when small data is loaded to a relatively big database; you can improve the load performance by using this command.

```
cubrid loaddb --no-statistics -d demodb_objects newdb
```

Specifying the ignoring classes (--ignore-class-file**)**

You can specify a file that lists classes to be ignored during loading records. All records of classes except ones specified in the file will be loaded.

```
cubrid loaddb --ignore-class-file=skip_class_list -d demodb_objects newdb
```

Specifying the error information file (--error-control-file)

This option specifies the file describing how to handle specific errors occurring during database loading.

```
cubrid loaddb --error-control-file=error_test -d demodb_objects newdb
```

How to Write Files to Load Database

You can add mass data to the database more rapidly by writing the object input file used in the **cubrid loaddb** utility. An object input file is a text file in simple table form that consists of comments and command/data lines.

Comment

In CUBRID, a comment is represented by two hyphens (--).

```
-- This is a comment!
```

Command Line

A command line begins with a percent character (%) and consists of **%class** and **%id** commands; the former defines classes, and the latter defines aliases and identifiers used for class identification.

Assigning an identifier to a class

You can assign an identifier to class reference relationships by using the **%id** command.

Syntax

```
%id class_name class_id
class_name:
    identifier
class_id:
    integer
```

The *class_name* specified by the **%id** command is the class name defined in the database, and *class_id* is the numeric identifier which is assigned for object reference.

Example 1

```
%id employee 2
%id office 22
%id project 23
%id phone 24
```

Specifying the class and attribute

You can specify the classes (tables) and attributes (columns) upon loading data by using the **%class** command. The data line should be written based on the order of attributes specified.

Syntax

```
%class class_name ( attr_name [ { attr_name }_ ]
```

The schema must be pre-defined in the database to be loaded.

The *class_name* specified by the **%class** command is the class name defined in the database and the *attr_name* is the name of the attribute defined.

Example 2

The following is an example that specifies a class and three attributes by using the **%class** command to enter data into a class named employee. Three pieces of data should be entered on the data lines after the **%class** command. For this, see Example 3 in the "Configuring a reference relationship" section.

```
%class employee (name age department)
```

Data Line

A data line comes after the **%class** command line. Data loaded must have the same type as the class attributes specified by the **%class** command. The data loading operation stops if these two types are different.

Data for each attribute must be separated by at least one space and be basically written as a single line. However, if the data to be loaded takes more than one line, you should specify the plus sign (+) at the end of the first data line to enter data continuously on the following line. Note that no space is allowed between the last character of the data and the plus sign.

Loading an instance

As shown below, you can load an instance that has the same type as the specified class attribute. Each piece of data is separated by at least one space.

Example 1

```
%class employee (name)
'jordan'
'james'
'garnett'
'malone'
```

Assigning an instance number

You can assign a number to a given instance at the beginning of the data line. An instance number is a unique positive number in the specified class. Spaces are not allowed between the number and the colon (:). Assigning an instance number is used to configure the reference relationship for later.

Example 2

```
%class employee (name)
1: 'jordan'
2: 'james'
```

```
3: 'garnett'
4: 'malone'
```

Configuring a reference relationship

You can configure the object reference relationship by specifying the reference class after an "at sign (**@**)" and the instance number after the "vertical line (|)."

Syntax

```
@class_ref | instance_no
class_ref:
     class_name
     class_id
```

Specify a class name or a class id after the **@** sign, and an instance number after a vertical line (|). Spaces are not allowed before and after a vertical line (|).

Example 3

The following is an example that loads class instances into the paycheck class. The name attribute references an instance of the employee class. As in the last line, data is loaded as **NULL** if you configure the reference relationship by using an instance number not specified earlier.

```
%class paycheck(name department salary)
@employee|1   'planning'   8000000
@employee|2   'planning'   6000000
@employee|3   'sales'   5000000
@employee|4   'development'   4000000
@employee|5   'development'   5000000
```

Example 4

Since the id 21 was assigned to the employee class by using the **%id** command in the Assigning an identifier to a class section, Example 3 can be written as follows:

```
%class paycheck(name department salary)
@21|1   'planning'   8000000
@21|2   'planning'   6000000
@21|3   'sales'   5000000
@21|4   'development'   4000000
@21|5   'development'   5000000
```

Database Backup and Restore

Database Backup

A database backup is the procedure of storing CUBRID database volumes, control files and log files, and it is executed by using the **cubrid backupdb** utility or the CUBRID Manager. **DBA** must regularly back up the database so that the database can be properly restored in the case of storage media or file errors. The restore environment must have the same operating system and the same version of CUBRID as the backup environment. For such a reason, you must perform a backup in a new environment immediately after migrating a database to a new version.

To recover all database pages, control files and the database to the state at the time of backup, the **cubrid backupdb** utility copies all necessary log records.

Syntax

```
cubrid backupdb [ options ] database_name
[ options ]
-D | -r | -l | -o | -S | -C | -t | -z | -e |
--destination-path | --remove-archive | --level | --output-file | --SA-mode | --CS-mode |
--thread-count | --compress | --except-active-log | --no-check
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **backupdb** : A utility for database backup. Performs an online, offline, compressed or parallel backup depending on the option used. This utility can only be executed by a user who has the backup authorization (e.g. **DBA**).
- *options* : A short option starts with a single dash (-) while a full name option starts with a double dash (--). Options are case sensitive.
- *database_name* : Specifies the name of the database to be backed up.

Return Value

- 0 : Success
- Non-zero : Failure

Option

The following table shows options that can be used with **cubrid backupdb** utility. Note that options are case sensitive.

Option	Description
-D --destination-path	Specifies the directory path name or device name where backup volumes are to be created. The default value is the location of **log_path** specified in the database location file (**databases.txt**) which was generated upon database creation.
-r --remove-archive	Removes unnecessary archive logs after the backup is complete.
-l --level	Configures the backup level to 0, 1 or 2. The default value is a full backup (0).
-o --output-file	Specifies the name of the file where progress information is to be outputted.
-S --SA-mode	Performs a backup in standalone mode. The default value is the one specified by the system parameter **CUBRID_MODE**.
-C --CS-mode	Performs a backup in client/server mode. The default value is the one specified by the system parameter **CUBRID_MODE**.
-t	Specifies the maximum number of threads allowed for a parallel

--thread-count	backup. The default value is the number of CPUs in the system.
-z --compress	Performs a compressed backup.
-e --except-active-log	Configures that active log volumes are not included in the backup.
--no-check	Does not perform a consistency check on a database before making a backup.
-sp --safe-page-id	This option is used with the **-r** option where replication is configured. The **-r** option must be used with the **-sp** option so that unnecessary archive logs can be cleared, while the necessary information for replication is stored.

Performing a backup by specifying the directory in which backup files are to be stored (-D or --destination-path)

The following is an example that uses the **-D** option to store backup files in the specified directory. The backup file directory must be specified before performing this job. If the **-D** option is not specified, backup files are stored in the directory specified in the **databases.txt** file which stores database location information.

```
cubrid backupdb -D /home/cubrid/backup demodb
```

The following example stores backup files in the current directory by using the **-D** option. If you enter a period (.) following the **-D** option as an argument, the current directory is specified.

```
cubrid backupdb -D . demodb
```

Removing archive logs after a backup (-r or --remove-archive)

If the database parameter **media_failure_support** is configured to 1, when the active logs are full, they are written to a new archive log file. If a backup is performed in such a situation and backup volumes are created, backup logs created before the backup will not be used in subsequent backups. The **-r** option is used to remove archive log files that will not be used any more in subsequent backups after the current one is complete.

```
cubrid backupdb -r demodb
```

The **-r** option does not affect the restore because it removes only unnecessary archive logs before the backup, but full restore may not be possible if the administrator removes archive logs created after the backup as well; when you remove archive logs, you must check if those logs would be required in any subsequent restore.

If you perform an incremental backup (backup level 1 or 2) with the **-r** option, there is the risk that normal recovery of the database will be impossible later on. Therefore, it is recommended that the **-r** option only be used when a full backup is performed.

Storing page information necessary for replication (-sp or --safe-page-id)

The **-sp** option is necessarily used when the **-r** option is used where replication is configured. Only archive logs that have smaller IDs than the ID of the latest log page are safely cleared while not the latest ID. That is, the **-sp** option makes the **-r** option limitedly performed in order to reserve the log page information necessary for replication. You should specify 'repl_safe_page *database name*', which is one of replication scripts, as an argument.

```
cubrid backupdb -r -sp 'repl_safe_page demodb' demodb
```

Performing a backup with the backup level specified (-l or --level)

The following example performs an incremental backup of the level specified by using the **-l** option. If the **-l** option is not specified, a full backup is performed. For more information about backup levels, see Incremental Backup.

```
cubrid backupdb -l 1 demodb
```

Saving backup progress information in the specified file (-o or --output-file)

The following example writes the progress of the database backup to the info_backup file by using the **-o** option.

```
cubrid backupdb -o info_backup demodb
```

The following is an example of showing the contents of the info_backup file. You can check the information about the number of threads, compression method, backup start time, the number of permanent volumes, backup progress and backup end time.

```
[ Database(demodb) Full Backup start ]
- num-threads: 1
- compression method: NONE
- backup start time: Mon Jul 21 16:51:51 2008
- number of permanent volumes: 1
- backup progress status
-------------------------------------------------------------------------------
 volume name                   | # of pages | backup progress status     | done
-------------------------------------------------------------------------------
 demodb_vinf                   |          1 | ########################## | done
 demodb                        |      25000 | ########################## | done
 demodb_lginf                  |          1 | ########################## | done
 demodb_lgat                   |      25000 | ########################## | done
-------------------------------------------------------------------------------
# backup end time: Mon Jul 21 16:51:53 2008
[Database(demodb) Full Backup end]
```

Performing a backup in standalone mode (-S or --SA-mode)

The following example performs a backup in standalone by using the **-S** option. The demodb database is backed up offline. If the **-S** option is not specified, the backup is performed in the mode specified by the **CUBRID_MODE** environment variable.

```
cubrid backupdb -S demodb
```

Performing a backup in client/server mode (-C or --CS-mode)

The following example performs a backup in client/server mode by using the **-C** option. The demodb database is backed up online. If the **-C** option is not specified, a backup is performed in the mode specified by the **CUBRID_MODE** environment variable.

```
cubrid backupdb -C demodb
```

Parallel backup (-t or --thread-count)

The following example performs a parallel backup with the number of threads specified by the administrator by using the **-t** option. Even when the argument of the **-t** option is not specified, a parallel backup is performed by automatically assigning as many threads as CPUs in the system.

```
cubrid backupdb -t 4 demodb
```

Compressed backup (-z or --compress)

The following example compresses the database and stores it in the backup file by using the **-z** option. The size of the backup file and the time required for backup can be reduced by using the **-z** option.

```
cubrid backupdb -z demodb
```

Enabling to exclude active log volumes (-e or --except-active-log)

The following example performs a backup, excluding active logs of the database by using the **-e** option. You can reduce the time required for backup by using the **-e** option. However, extra caution is required because active logs needed for completing a restore to the state of a certain point from the backup point are not included in the backup file, which may lead to an unsuccessful restore.

```
cubrid backupdb -e demodb
```

Disabling a database consistency check (--no-check)

The following example performs a backup without checking the consistency of the database by using the **--no-check** option.

```
cubrid backupdb --no-check demodb
```

Backup Strategy and Method

The following must be considered before performing a backup:

- **Selecting the data to be backed up**
  - Determine whether it is valid data worth being preserved.
  - Determine whether to back up the entire database or only part of it.
  - Check whether there are other files to be backed up along with the database.
- **Choosing a backup method**
  - Choose the backup method from one of incremental and online backups. Also, specify whether to use compression backup, parallel backup, and mode.
  - Prepare backup tools and devices available.
- **Determining backup time**
  - Identify the time when the least usage in the database occur.
  - Check the size of the archive logs.
  - Check the number of clients using the database to be backed up.

Online Backup

An online backup (or a hot backup) is a method of backing up a currently running database. It provides a snapshot of the database image at a certain point in time. Because the backup target is a currently running database, it is likely that uncommitted data will be saved and the backup may affect the operation of other databases.

To perform an online backup, use the **cubrid backupdb -C** command.

Offline Backup

An offline backup (or a cold backup) is a method of backing up a stopped database. It provides a snapshot of the database image at a certain point in time.

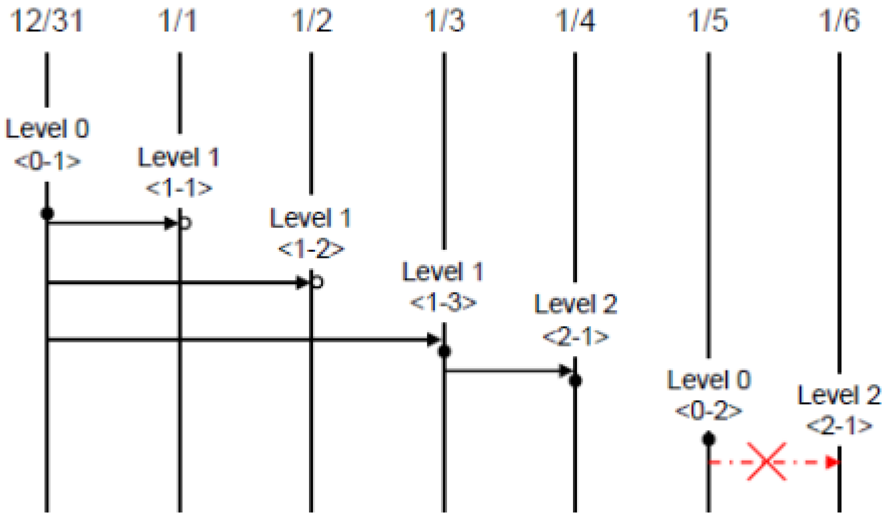
To perform an offline backup, use the **cubrid backupdb -S** command.

Incremental Backup

An incremental backup, which is dependent upon a full backup, is a method of only backing up data that have changed since the last backup. This type of backup has an advantage of requiring less volume and time than a full backup. CUBRID supports backup levels 0, 1 and 2. A higher level backup can be performed sequentially only after a lower lever backup is complete.

To perform an incremental backup, use the **cubrid backupdb -l** <*level*> command.

The following is an example of an incremental backup. With this example, we will examine backup levels in detail.



- **Full backup (backup level 0)** : Backup level 0 is a full backup that includes all database pages.

  The level of a backup which is attempted first on the database naturally becomes a 0 level. **DBA** must perform full backups regularly to prepare for restore situations. In the example, full backups were performed on December 31st and January 5th.

- **First incremental backup (backup level 1)** : Backup level 1 is an incremental backup that only saves changes since the level 0 full backup, and is called a "first incremental backup."

  Note that the first incremental backups are attempted sequentially such as <1-1>, <1-2> and <1-3> in the example, but they are always performed based on the level 0 full backup.

  Suppose that backup files are created in the same directory. If the first incremental backup <1-1> is performed on January 1st and then the first incremental backup <1-2> is attempted again on January 2nd, the incremental backup file created in <1-1> is overwritten. The final incremental backup file is created on January 3rd because the first incremental backup is performed again on that day.

  Since there can be a possibility that the database needs to be restored the state of January 1st or January 2nd, it is recommended for **DBA** to save the incremental backup files <1-1> and <1-2> separately in storage media before overwriting with the final incremental file.

- **Second incremental backup (backup level 2)** : Backup level 2 is an incremental backup that only saves data that have changed since the first incremental backup, and is called a "second incremental backup."

  A second incremental backup can be performed only after the first incremental backup. Therefore, the second incremental backup attempted on January fourth succeeds; the one attempted on January sixth fails.

  Backup files created for backup levels 0, 1 and 2 may all be required for database restore. To restore the database to its state on January fourth, for example, you need the second incremental backup generated at <2-1>, the first incremental backup file generated at <1-3>, and the full backup file generated at <0-1>. That is, for a full restore, backup files from the most recent incremental backup file to the earliest created full backup file are required.

Compress Backup

A compress backup is a method of backing up the database by compressing it. This type of backup reduces disk I/O costs and saves disk space because it requires less backup volume.

To perform a compress backup, use the **cubrid backupdb -z|--compress** command.

Parallel Backup Mode

A parallel or multi-thread backup is a method of performing as many backups as the number of threads specified. In this way, it reduces backup time significantly. Basically, threads are given as many as the number of CPUs in the system.

To perform a parallel backup, use the **cubrid backupdb -t|--thread-count** command.

Managing Backup Files

One or more backup files can be created in sequence based on the size of the database to be backed up. A unit number is given sequentially (000, 001-0xx) to the extension of each backup file based in the order of creation.

Managing Disk Capacity during the Backup

During the backup process, if there is not enough space on the disk to save the backup files, a message saying that the backup cannot continue appears on the screen. This message contains the name and path of the database to be backed up, the backup file name, the unit number of backup files and the backup level. To continue the backup process, the administrator can choose one of the following options:

- Option 0 : An administrator enters 0 to discontinue the backup.
- Option 1 : An administrator inserts a new disk into the current device and enters 1 to continue the backup.
- Option 2 : An administrator changes the device or the path to the directory where backup files are saved and enters 2 to continue the backup.

```
******************************************************************
Backup destination is full, a new destination is required to continue:
Database Name: /local1/testing/demodb
     Volume Name: /dev/rst1
        Unit Num: 1
    Backup Level: 0 (FULL LEVEL)
Enter one of the following options:
Type
   -  0 to quit.
   -  1 to continue after the volume is mounted/loaded. (retry)
   -  2 to continue after changing the volume's directory or device.
******************************************************************
```

Database Restore

A database restore is the procedure of restoring the database to its state at a certain point in time by using the backup files, active logs and archive logs which have been created in an environment of the same CUBRID version. To perform a database restore, use the **cubrid restoredb** utility or the CUBRID Manager.

The **cubrid restoredb** utility (restordb.exe in Windows) recovers the database from the database backup by using the information written to all the active and archive logs since the execution of the last backup.

Syntax

```
cubrid restoredb [ options ] database_name
[ options ]
-d | -B | -l | -p | -o | -u |
--up-to-date | --backup-file-path | --level | --partial-recovery | --output-file | --
replication-mode | --use-database-location-path | --list
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **restoredb** : A command for recovery of the specified database. For a successful recovery, you must prepare backup files, active log files and archive log files. This command can be performed only in standalone mode.
- *options* : A short name option starts with a single dash (-) while a full name option starts with a double dash (--). This option is case sensitive.
- *database_name* : Specifies the name of the database to be recovered.

Return Value

- 0 : Success
- Non-zero : Failure

Option

The following table shows options that can be used with **cubrid restoredb**. Options are case sensitive.

Option	Description

-d **--up-to-date**	Directly sets the time to backup the database or specifies the **backuptime** keyword.
-B **--backup-file-path**	Specifies the directory pathname or device name where backup files are to be located.
-l **--level**	Sets the recovery level to 0, 1 or 2. The default value is a full recovery (0).
-p **--partial-recovery**	Performs a partial recovery.
-o **--output-file**	Specifies the name of the file where recovery information is to be displayed.
-u **--use-database-location-path**	Recovers the database to the path specified in the database location file (**databases.txt**).
--list	Displays information about backup volumes of the database on the screen.

Performing a recovery by specifying a recovery point (-d or --up-to-date)

The following command recovers demodb. If no option is specified, demodb is recovered to the point of the last commit by default. If no active/archive log files are required to recover to the point of the last commit, the database is recovered only to the point of the last backup.

```
cubrid restoredb demodb
```

demodb can be recovered to the given point by using the **-d** option and the syntax which specifies the date and time of the recovery. The user can specify the recovery point manually in the dd-mm-yyyy:hh:mm:ss (e.g. 14-10-2008:14:10:00) format. If no active log/archive log files are required to recover to the point specified, the database is recovered only to the point of the last backup.

```
cubrid restoredb -d 14-10-2008:14:10:00 demodb
```

The following syntax specifies the recovery point by using the **-d** option and the **backuptime** keyword and recovers demodb to the point of the last backup.

```
cubrid restoredb -d backuptime demodb
```

Performing a recovery by specifying the directory path to the backup files (-B or --backup-file-path)

You can specify the directory where backup files are to be located by using the **-B** option. If this option is not specified, the system retrieves the backup information file (**dbname_bkvinf**) generated upon a database backup; the backup information file in located in the **log_path** directory specified in the database location information file (**databases.txt**). And then it searches the backup files in the directory path specified in the backup information file. However, if the backup information file has been damaged or the location information of the backup files has been deleted, the system will not be able to find the backup files. Therefore, the administrator must manually specify the directory where the backup files are located by using the **-B** option.

```
cubrid restoredb -B /home/cubrid/backup demodb
```

If the backup files of demodb is in the current directory, the administrator can specify the directory where the backup files are located by using the **-B** option.

```
cubrid restoredb -B . demodb
```

Performing a recovery by specifying the backup level (-l or --level)

You can perform a restoration by specifying the backup level of the database to 0, 1, or 2. For more information about backup levels, see Increment Backup.

```
cubrid restoredb -l 1 demodb
```

Performing a partial recovery (-p or --partial-recovery)

The following command performs a partial recovery without requesting for the user's response by using the **-p** option. If active or archive logs written after the backup point are not complete, by default the system displays a request message informing that log files are needed and prompting the user to enter an execution option. A partial recovery can be performed directly without such a request message by using the **-p** option. Therefore, if the **-p** option is used when performing a recovery, data is always recovered to the point of the last backup.

```
cubrid restoredb -p demodb
```

When the **-p** option is not specified, the message requesting the user to select the execution option is as follows:

```
***********************************************************
Log Archive /home/cubrid/test/log/demodb_lgar002
 is needed to continue normal execution.
   Type
   -  0 to quit.
   -  1 to continue without present archive. (Partial recovery)
   -  2 to continue after the archive is mounted/loaded.
   -  3 to continue after changing location/name of archive.
***********************************************************
```

- Option 0 : An administrator enters 0 to stop the recovery.
- Option 1 : An administrator enters 1 to perform a partial recovery without log files.
- Option 2 : An administrator enters 2 to perform a recovery after moving archive logs to the current device.
- Option 3 : An administrator enters 3 after changing a log location to resume a restoration.

Storing recovery progress information in the specified file (-o or --output-file)

The following command writes the recovery progress of the database to the info_restore file by using the **-o** option.

```
cubrid restoredb -o info_restore demodb
```

Recovering data to the directory specified in the database location file (-u or --use-database-location-path)

The following syntax recovers the database to the path specified in the database location file (**databases.txt**) by using the **-u** option. The **-u** option is useful when you perform a backup on server A and recover the backup files on server B.

```
cubrid restoredb -u demodb
```

Checking the backup information of the database (--list)

The following syntax displays the information about backup files of the database by using the **--list** option; it does not perform recovery.

```
cubrid restoredb --list demodb
```

The following is an example of backup information displayed as a result of using the **--list** option. You can identify the path to which backup files of the database are originally stored as well as backup levels.

```
*** BACKUP HEADER INFORMATION ***
Database Name: /local1/testing/demodb
 DB Creation Time: Mon Oct 1 17:27:40 2008
         Pagesize: 4096
Backup Level: 1 (INCREMENTAL LEVEL 1)
        Start_lsa: 513|3688
         Last_lsa: 513|3688
Backup Time: Mon Oct 1 17:32:50 2008
 Backup Unit Num: 0
Release: 8.1.0
     Disk Version: 8
Backup Pagesize: 4096
Zip Method: 0 (NONE)
        Zip Level: 0 (NONE)
Previous Backup level: 0 Time: Mon Oct 1 17:31:40 2008
(start_lsa was -1|-1)
Database Volume name: /local1/testing/demodb_vinf
     Volume Identifier: -5, Size: 308 bytes (1 pages)
Database Volume name: /local1/testing/demodb
     Volume Identifier: 0, Size: 2048000 bytes (500 pages)
Database Volume name: /local1/testing/demodb_lginf
     Volume Identifier: -4, Size: 165 bytes (1 pages)
Database Volume name: /local1/testing/demodb_bkvinf
```

```
     Volume Identifier: -3, Size: 132 bytes (1 pages)
```

With the backup information displayed by using the **--list** option, you can check that backup files have been created at the backup level 1 as well as the point where the full backup of backup level 0 has been performed. Therefore, to recover the database in the example, you must prepare backup files for backup levels 0 and 1.

Restore Strategy and Procedure

The following must be considered before performing a database restore:

- **Preparing a backup file**
  - Identify the directory where the backup and log files are to be stored.
  - If the database has been incrementally backed up, check whether an appropriate backup file for each backup level exists.
  - Check whether the backed-up CUBRID database and the CUBRID database to be backed up are the same version.
- **Choosing a restore method**
  - Determine whether to perform a partial or full restore.
  - Determine whether or not to perform a restore using incremental backup files.
  - Prepare restore tools and devices available.
- **Determining restore time**
  - Identify the point in time when the database server was terminated.
  - Identify the point in time when the last backup was performed before database failure.
  - Identify the point in time when the last commit was made before database failure.

Database Restore Procedure

The following is an example of a backup and restore process described in the order of time.

- Performs a full backup of demodb which stopped running at 2008/8/14 04:30.
- Performs the first incremental backup of demodb running at 2008/8/14 10:00.
- Performs the first incremental backup of demodb running at 2008/8/14 15:00. Overwrites the first incremental backup file in step 2.
- A system failure occurs at 2008/8/14 15:30, and the system administrator prepares the restore of demodb. Sets the restore time as 15:25, which is the time when the last commit was made before database failure
- The system administrator prepares the full backup file created in Step 1 and the first incremental backup file created in Step 3, restores the demodb database up to the point of 15:00, and then prepares the active and archive logs to restore the database up to the point of 15:25.

Time	Command	Description
2008/8/14 04:25	cubrid server stop demodb	Shuts down demodb.
2008/8/14 04:30	cubrid backupdb -S -D /home/backup -l 0 demodb	Performs a full backup of demodb in offline mode and creates backup files in the specified directory.
2008/8/14 05:00	cubrid server start demodb	Starts demodb.
2008/8/14 10:00	cubrid backupdb -C -D /home/backup -l 1 demodb	Performs the first incremental backup of demodb online and creates backup files in the specified directory.
2008/8/14 15:00	cubrid backupdb -C -D /home/backup -l 1 demodb	Performs the first incremental backup of demodb online and creates backup files in the specified directory. Overwrites the first incremental backup file created at 10:00.
2008/8/14		A system failure occurs.

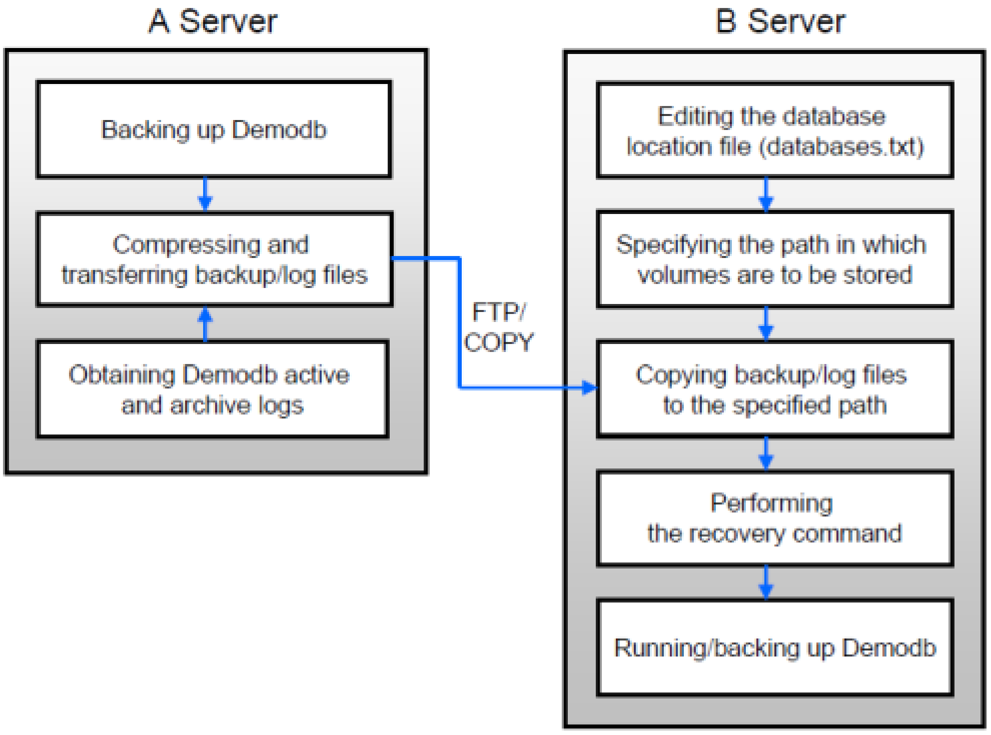
15:30		
2008/8/14 15:40	cubrid restoredb -l 1 -d 08/14/2008:15:25:00 demodb	Restores demodb based on the full backup file, first incremental backup file, active logs and archive logs. The database is restored to the point of 15:25 by the full and first incremental backup files, the active and archive logs.

Restoring Database to Different Server

The following shows how to back up demodb on server A and restore it on server B with the backed up files.

Backup and Restore Environments

Suppose that demodb is backed up in the /home/cubrid/db/demodb directory on server A and restored into /home/cubrid/data/demodb on server B.


A Server
Backing up Demodb
Compressing and transferring backup/log files
Obtaining Demodb active and archive logs
FTP/ COPY
B Server
Editing the database location file (databases.txt)
Specifying the path in which volumes are to be stored
Copying backup/log files to the specified path
Performing the recovery command
Running/backing up Demodb

1. Backing up on server A

   Back up demodb on server A. If a backup has been performed earlier, you can perform an incremental backup for data only that have changed since the last backup. The directory where the backup files are created, if not specified in the **-D** option, is created by default in the location where the log volume is stored. The following is a backup command with recommended options. For more information on the options, see Database Backup.

   ```
   cubrid backupdb -z -t demodb
   ```

2. Editing the database location file on Server B

   Unlike a general scenario where a backup and restore are performed on the same server, in a scenario where backup files are restored using a different server, you need to add the location information on database restore in the database location file (**databases.txt**) on server B. In the diagram above, it is supposed that **demodb** is restored in the **/home/cubrid/data/demodb** directory on server B (hostname: pmlinux); edit the location information file accordingly and create the directory on server B.

   Put the database location information in one single line. Separate each item with a space. The line should be written in [database name]. [data volume path] [host name] [log volume path] format; that is, write the location information of **demodb** as follows:

   ```
   demodb /home/cubrid/data/demodb pmlinux /home/cubrid/data/demodb
   ```

3. Transferring backup/log files to server B

   For a restore, you must prepare a backup file (e.g. demodb_bk0v000) and a backup information file (e.g. demodb_bkvinf) of the database to be backed up. To restore the entire data up to the point of the last commit, you must prepare an active log (e.g. demodb_lgat) and an archive log (e.g. demodb_lgar000). Then, transfer the backup information, active log, and archive log files created on server A to server B. That is, the backup information, active log and archive log files must be located in a directory (e.g. /home/cubrid/temp) on server B.

4. Restoring the database on server B

   Perform database restore by calling the **cubrid restoredb** utility from the directory into which the backup, backup information, active log and archive log files which were transferred to server B had been stored. With the **-u** option, demodb is restored in the directory path from the **databases.txt** file.

   ```
   cubrid restoredb -u demodb
   ```

   To call the **cubrid restoredb** utility from a different path, specify the directory path to the backup file by using the **-B** option as follows:

   ```
   cubrid restoredb -u -B /home/cubrid/temp demodb
   ```

5. Backing up the restored database on server B

   Once the restore of the target database is complete, run the database to check if it has been properly restored. For stable management of the restored database, it is recommended to restore the database again on the server B environment.

Database Replication

Concept of Database Replication

Overview

Database replication is one of the distributed database techniques that make objects available to more than two database servers by physically copying objects stored in one database to other separate databases. Replication techniques can be used for the following purposes:

- To improve performance by distributing access of applications using the same object to multiple database servers.
- To satisfy different operational requirements by using the replicated database server for other purposes.
- To urgently cope with failure by switching to the replicated database server when a failure occurs in the currently running database server.

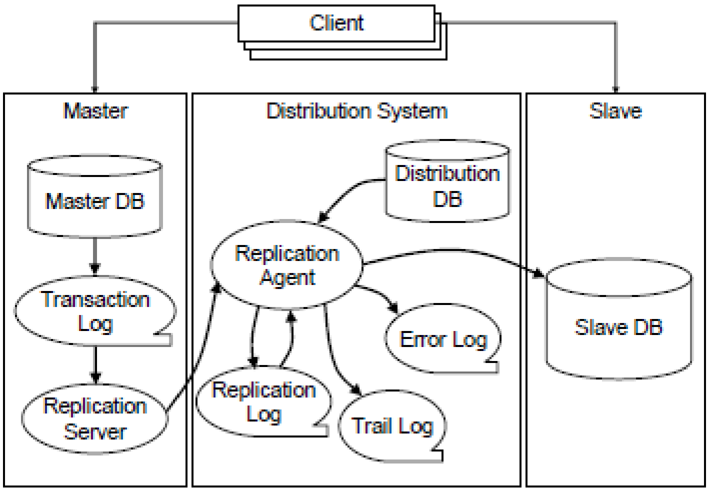
Note on Outdated & Deprecated Feature The replication feature is no longer supported in a later version of CUBRID 2008 R3.0. Therefore, it is recommended to configure duplex environment by using HA feature.

Note Currently, replication feature is supported only on Linux series; it is not supported on Windows.

Architecture and Terms of Replication System

The following diagram shows the architecture of the CUBRID replication system.

The original database to be replicated is called a master database, and the database to be filled with the data replicated from the master is called a slave database. The master database is a generic database that allows all operations such as write and read; the slave database allows read operations only.



- Once replication is configured, the replication server on the master reads the transaction log of the master database.
- The replication server creates a replication log containing information related to replication only, out of the transaction logs. And then, it transfers the log to the replication agent on the distribution system (the distributor). The distribution system can be configured on the master or slave host, or it can be configured on a separate host to ensure maximum availability.
- The replication agent reads the distributor database and then determines how to replicate the replication log from the master on which slave. The distributor database is one of the CUBRID databases and manages all metadata concerning replication configuration.
- The replication agent first saves the replication log received from the replication server on disk.
- The replication log is distributed to the slave database specified in the distribution database, reflecting changes in the master database on the slave. Trail logs are managed to track changes reflected on each slave database in this process. All errors that occur during the distribution are saved in error logs.

The following is a summary of each component of the CUBRID replication system.

Master Database

This is a generic CUBRID database and the original database which is to be replicated. Because CUBRID replication is asynchronous, database operations are not affected by the replication system (details about asynchronous and synchronous replications are explained later). Unlike the slave database, the master database allows all database operations including read and write.

Transaction Log

The transaction log is used to ensure the transaction integrity of the master database (independent of the replication system) and the database Availability (in case of failure). The replication system is informed of the information about all changes that have occurred in the master database through the transaction log.

Replication Server

Once replication is configured, the replication server creates the replication log necessary for the replication distribution by analyzing the transaction log of the master database, and then transfers it to the replication agent on the distribution server host. The CUBRID replication server is supported by the **cubrid repl_server** command. Details about how to use this command are explained later in this chapter.

Distribution System

The replication system replicates to a slave the replication log received from a master; it consists of the replication agent, replication database and related log files. The distribution server can be configured on the master or slave host, or it can be configured on a separate host to increase availability in case of failure.

Replication Agent

The replication agent is a process that performs the actual replication by using the replication log transferred from the replication server. Because multiple slaves can be configured for a single master, the replication agent first saves the replication log received from the replication server to a file. Then it reflects the replication log on all slave databases configured in the distribution database. The CUBRID replication agent is supported by the **cubrid repl_agent** utility. Details about how to use this command are explained later in this chapter.

Distribution Database

The distribution database makes the replication agent inform you where a master and slave databases are allocated in the distribution system; it is one of the CUBRID databases. Detailed information about the distribution database including its schema is explained later in this chapter.

Replication Log

The replication log saves log files from a master replication server in the format of <master_dbname>.copy. You must have enough disk space to store the replication log because the size of the replication log varies depending on the amount of changes in the master.

Trail Log

This is file that records the progress of the distribution such as the sequential log number created for each slave database while the replication agent is performing the actual replication. The trail log is managed as a single file for each replication agent and its name has the format of <dist_db_name>.trail.

Error Log

The error log is a file that records all errors that occur when the replication agent is performing the replication. Its name is in the format of <dist_db_name>.err.

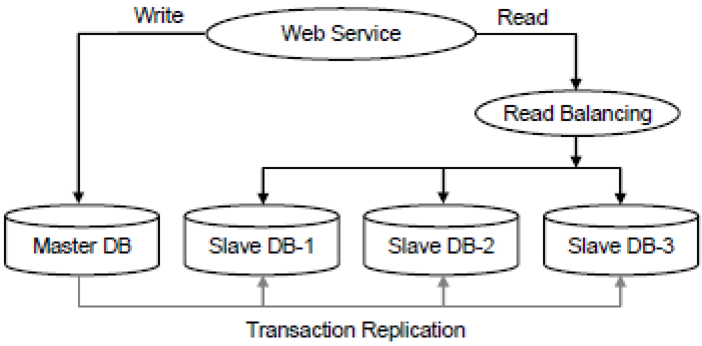
Slave Database

The slave database is the destination of replication. It is a database where changes in the master are automatically reflected to by the replication system. Unlike the master database, the slave database can be used for read operations only.

Replication Scenario

Improving Performance through Load Balancing

Replication is used to improve the system performance by distributing especially read operations across multiple database servers when a single database server cannot meet the performance requirements. Replication in CUBRID allows both read and write operations in the master, but only read operations in the slave. Therefore, it provides excellent performance improvement in many applications where write operations are limited but read operations are abundant. Many applications such as blogs, bulletin boards and news applications used in internet services belong to this category. The following diagram shows a Web service architecture designed to improve the performance through load balancing.



You can build a replication environment as above by configuring a distribution database and then adding a slave database.

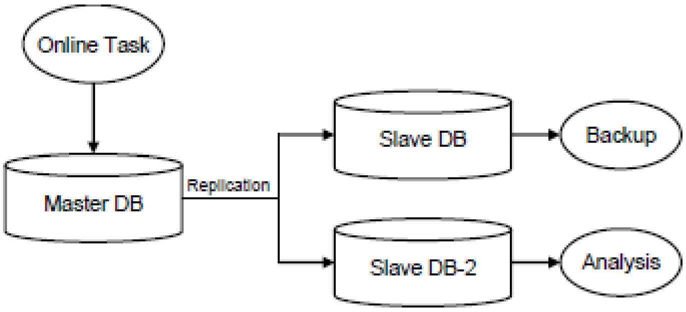
Improving Availability against Failure

If you set a slave using database replication, you can respond effectively by replacing the master with the slave when a failure occurs in the master. Recovery functions using the master database backup might be enough depending on the type of the failure. However, if a failure occurs on the master host, you can respond through a replication or High-Availability (HA) of the master host. For example, in the example above of improving the performance, you can configure the system so that one of the slave databases replaces the master and write operations in applications are sent to the new master when a failure occurs in the host on which the master database is running.

Replacing the master with a slave is not a simple operation. Especially, depending on the replication configuration of the master and the slave, making the slave perform all the functions of the master might require a very complicated process. Details about replacing the master with a slave are explained later in this chapter.

Improving Flexibility through Separation

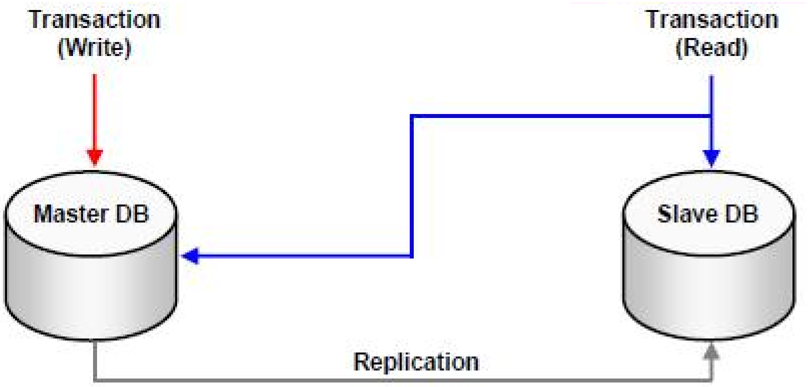
By using the CUBRID replication, you can have a database with the same contents as the master on a separate host. This way you can improve the flexibility in using the database by allowing multiple additional tasks to be done without affecting the environment where the master database is running. For example, if the master database is being used all day long for online tasks and unable to perform batch operations that might affect the online tasks (e.g. backup, analysis, etc.) in any time period, you can perform batch operations not affecting the master by creating a slave as shown below.



Replication Features

One-way replication

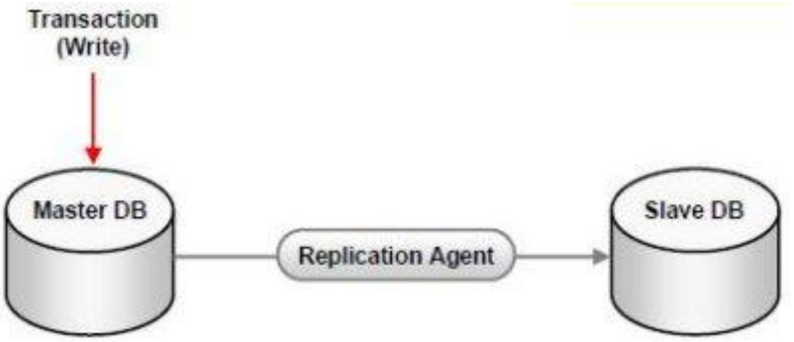
CUBRID supports only one-way replication. That is, a master database allows read and write operations while a slave database allows read operations only, as shown below. You can build up a 1:N replication system with this one-way replication.



Synchronous/asynchronous

There are the two methods to replicate data: synchronous replication in which data update operations of the master and slave databases are processed as a single transaction so that the data consistency will be ensured, and asynchronous replication in which two operations are separately performed.

- **Synchronous** : In the synchronous model, errors in one system can spread out other system, which reduces availability. For example, errors in a slave system can cause errors in a master system.
- **Asynchronous** : In the asynchronous replication, an error in the slave system does not affect the master system because the slave database can be fixed by using the transaction log in the master system, reflecting it asynchronously on the slave database. Therefore, CUBRID ensures data consistency as well as maximum availability by providing asynchronous replication functionality by using transaction logs.



Transaction-level replication

CUBRID replicates based on transaction logs. Transaction logs can restore the system by logging every write operation in the database when the system failure occurs. CUBRID analyzes transaction logs, extracts updated items from the master database, and reflects them on the slave database in the order in which they were modified. Therefore, transaction consistency of replication can always be ensured.

Online replication

CUBRID can perform online replication by synchronizing the slave databases without suspend of the master database.

Schema independence

For flexibility of the slave database, you can define its schema independent from the master database. You can create separate classes, indexes, user accounts and triggers aside from the ones being replicated from the master database. You can also perform write operations by defining a class in the slave database. For example, you can replicate class1 and class2 from the master database while defining class3 and class4 in the slave database.

To provide the maximum flexibility of the slave database, all indexes except the primary key index are excluded from the object to be replicated. The administrator must manage indexes of the slave database separately depending on the purpose of the slave system. For example, if a slave system is to be built for data analysis, its index design must be different from that of a master database built for online transaction processing.

User accounts of the master database are not replicated, either. User accounts must be managed independently in the slave system, and extra caution is required especially in managing write accounts. The **repl_make_slavedb** utility that is used to configure an initial slave database changes all owner accounts in the object to be replicated into replication accounts. Therefore, it is recommended that accounts to be added later be managed as read-only ones.

Because data changes due to a trigger defined in the master database are already reflected on its transaction logs, an error occurs during the replication if the same trigger is defined in the slave database. Therefore, trigger in the master database are excluded from the object to be replicated, and all triggers are deleted while the initial slave database is configured. However, depending on the purpose of the slave database, you can define and use separate triggers independent from those of the master database as long as its condition does not include the object to be replicated. For example, if class1 and class2 are replicated from the master database and class3 and class4 are managed independently by the slave, you can define and use an independent trigger in the slave that inserts the same data into class3 as well when data are inserted into class1. However, if data is deleted from class2, but there is a trigger that inserts the data into class1, the trigger may cause an error during the replication.

Object to be replicated

Object	Replicated
Data	O
Index	O
Trigger	X
User account	X
Schema	O

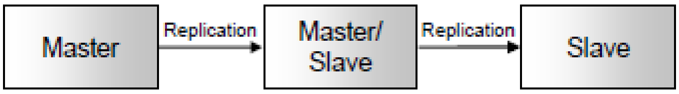
Using the Primary Key

Replication in CUBRID is performed based on the primary key. That is, the primary key is used to identify data items to be reflected by the slave database. Therefore, classes without the primary keys are excluded from the source.

Currently, replication in CUBRID is performed for each database, or only the class that has a primary key can be replicated to the slave database. For configuring primary keys, see Constraints in "Creating Tables."

Chained Replication

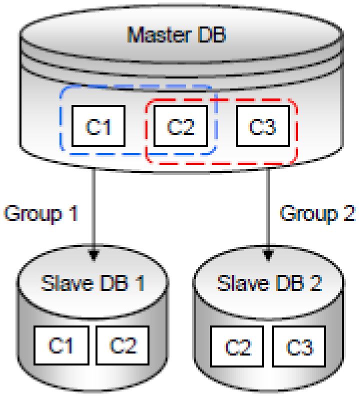
Replication performed in a hierarchical manner is called chained replication. It means that a slave database can become a master database at the same time, so it can transfer changed replication data to the other slave.



CUBRID supports such a replication architecture. In the case of figure above, the second master/slave databases are read-only.

Replication Group

CUBRID supports building up several slave databases from a single master database. At this time, the administrator can replicate some of the master database to the slave database by specifying them as a replication group.

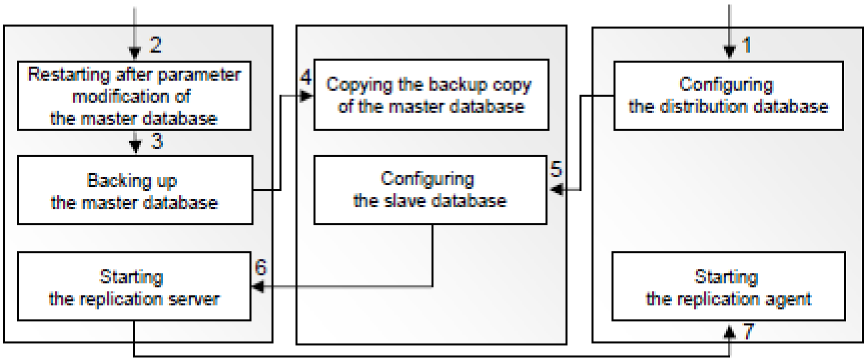


Such method is useful when you want to configure the slave server for a specific purpose. For example, if you want to build up the slave server by replicating only statistics data, you can do such a work by specifying a replication group.

Configuration

Replication Setup Procedure

You can make the replication as follows:


2
Restarting after parameter modification of the master database
3
Backing up the master database
6
Starting the replication server
4
Copying the backup copy of the master database
5
Configuring the slave database
1
Configuring the distribution database
Starting the replication agent
7

1. Configure the distribution database

   Execute the **repl_make_distdb** utility to create the distribution database and to configure parameters for the replication.

2. Modify the parameter value of master database and then restart it. If the parameter value has already been configured, this step is not necessary.

   If **replication** is set to **no** in the **cubrid.conf**, modify it to **yes**. And then restart the server of master database.

3. Back up the master database

   Back up the master database to synchronize a master and a slave.

4. Move backup of the master database in which the slave database will be located

   Copy backup volumes and an information file of the master database to the slave system by using FTP or file copying.

5. Configure (add) the slave database

   Execute the **repl_make_slavedb** utility to synchronize the slave database to the mast database and them set up replication.

6. Set up replication server

   Run replication server in the master system.

7. Run the replication agent

   Run replication agent in the slave system. When it is finished, configuring environment for replication is completed and all changes after the master is backed up will be synchronized to slave.

Configuring Distribution Database

Description

The distribution database is a database that manages metadata for replication. It must be configured as the first step of replication configuration. The **repl_make_distdb** utility is used to configure the distribution database.

The distribution database may exist on any host. However, it is recommended not to configure it on the host where the master database is running for performance.

You must always run the distribution database with the **DBA** account to ensure security. The **repl_make_distdb** utility requires the name of the distribution database and the password for its **DBA** account.

Syntax

```
repl_make_distdb dist_db_name [ -p password ]
```

Make sure to execute the **repl_make_distdb** utility in the directory where the distribution database is located. If you don't configure a password, you are not required to enter it; however, it is recommended to configure the password to ensure security.

The following is a help screen displayed when you execute **repl_make_distdb**.

```
###########################################################################
#   Configuring the CUBRID replication environment : Configuring the replication agent#
#   You must perform tasks as the following steps to configure the replication environment.
#      1. Configuring the replication agent (Run the repl_make_distdb utility)
#      2. Performing a full backup of the master database (Run the cubrid backupdb utility)
#      3. Copying the backup copy of the master database (from the master database host to
the slave database host)
#      4. Building the slave database (Run the repl_make_slavedb utility)
#      5. Running the replication server (Run the cubrid repl_server utility)
#      6. Running the replication agent (Run the cubrid repl_agent utility)
#
#    NOTE: The master database can be backed up at any time before the slave database is
built.
#          However, if it is an online backup, you can save time spent on
#          initial replication by performing it before you build the
#          slave database.
#          Necessary backup files are as follows:
#                          - master_db_name.bk_vinf
#                          - master_db_name.bk0v???
#
#    Create a distribution database that is needed for the replication agent to perform
tasks.
#    You must have the DBA account to run the distribution database. Please specify the
DBA
#    account.
#    In CUBRID, one distribution database is created for one slave database.
#    The distribution database and the replication agent (repl_agent) must run on the host
#    where the slave database is located.
#    If the script was stopped abnormally (such as with Ctrl-C),
#    delete the distribution database and restart by using a utility such as
#    cubrid server stop/cubrid deletedb.
#########################################################################
```

The **repl_make_distdb** utility consists of six steps. You are required to enter necessary information in step 5 and 6.

```
repl_make_distdb sample_dist -p dba1
```

When you execute the **repl_make_distdb** utility, the progress status of each step is displayed, and a message is prompted asking you to enter an input value if necessary.

```
 STEP 1 : The distribution database is being created. Please wait for a moment.
 STEP 2 : The distribution database server is being started. Please wait for a moment.
 STEP 3 : Configure a DBA account for the distribution database.
 STEP 4 : Create tables needed for replication.
 STEP 5 : Enter the information of the target master database.
      1. Please enter the name of the master database. >>
      2. Please enter the IP address of the host where the master database is located.
         - Replication cannot be performed successfully unless the IP address is entered
correctly.
               master database IP >>
      3. Please enter the TCP/IP port number used by the replication server (cubrid
repl_server).>>
      4. Please enter the directory where replication logs needed for replication are to
be saved.
         If it is /home1/cubrid/CUBRID, press the Enter key. >>
 STEP 6 : Configure replication environment variables.
      1. Please enter the directory where trail logs will be saved.
         If it is /home1/cubrid/CUBRID, press the Enter key. >>
      2. Please enter the directory where error logs will be saved.
         If it is /home1/cubrid/CUBRID, press the enter key. >>
      3. Please enter the TCP/IP port number for the replication agent (repl_agent) status
to be displayed.>>
      4. Please enter the size of the replication delay time log file (number of lines) >>
      5. Please specify whether or not replication will restart in case of network error
(y/n). >>
```

- **(Step 1) Creating the distribution database** : Create the distribution database.
- **(Step 2) Starting the distribution database server** : Start the distribution database server.
- **(Step 3) Configuring the DBA account of the distribution database** : Configure the DBA account using the password specified by the input value.
- **(Step 4) Creating the schema of the distribution database** : Create various classes of the distribution database needed for the replication.
- **(Step 5) Entering the information of the master database used in replication** : Enter the name of the master database, the IP address of the host where the master database is located, and the TCP port number used for communication with the replication server. The replication server is a process that reads transaction logs of the master and transfers them to the replication agent. It uses the TCP port to communicate with the replication agent. You must enter the path where distribution logs will be saved as well. The distribution log is a file that is saved temporarily to replicate transaction logs of the master database and is created in the specified path with the name <dist_db_name>.copy once replication starts.
- **(Step 6) Configuring replication environment variables** : This is the last step in configuring the distribution database. Enter the paths where trail and error logs will be saved. The trail log is a file that saves the location of the last log processed by the replication agent. When a failure occurs, the restart time is determined by the information recorded in this file. Therefore, you must always make sure that this file does not get damaged. The error log is a file that records errors occurring during the replication.

Additionally, you need the following three pieces of information to manage the status of the replication agent and replication delay time as well as to maintain the connection:

- **TCP/IP port number for the replication agent (repl_agent) status** : This connection port is used when **repl_check_sync** connects to repl_agent to check the current status of repl_agent. The default port number is 33333.
- **The size of the replication delay time log file (number of lines)** : Replication delay time logs are written to monitor the performance of the replication. The size of the file to be written must be defined.
- **Whether or not to restart the replication** : Whether or not the replication agent will try to reconnect when the replication server and the replication agent get disconnected. When the network connected is dropped, the

replication agent is stopped. If you specify 'y,' the replication agent restarts and resumes the replication; if you specify 'n,' the replication stops immediately.

Configuring and Backing up Master Database

As a next step, you must change the configuration of the master database and create a backup volume for synchronization with the slave database. For the master database to create transaction logs required for replication, change the value of the **replication** parameter to **yes** by modifying the **$CUBRID/conf/cubrid.conf** file and then restart the master database server. You don't need to restart the master database if this parameter has already been configured to **yes**. The following output is displayed if you configure **replication** to **yes** in the **$CUBRID/conf/cubrid.conf** file:

```
[@sample_master]
replication=yes
```

In the above example, the **replication** parameter applies only to the sample_master database. As shown above, if you specify a parameter under **[@**<*dbname*>**]**, it applies only to the specified <*dbname*>.

Restart the master database server to reflect the modified parameter on the server.

```
cubrid server stop sample_master  -- Stop the server
cubrid server start sample_master -- Start the server
```

Now back up the master database to synchronize the master and slave databases in the current state.

```
cubrid backupdb sample_master
```

Here, replication may not be possible if you use the **-r** option to delete archive logs. Since CUBRID supports online replication, there may be cases where a mass write takes place in the master database during the synchronization of the slave database, causing changes since the backup to be already passed to the archive log file. In these cases, the replication system reads the archive log file and reflects it on the slave. Therefore, you must perform the backup in a mode where archive logs are not deleted in cases where logs not reflected on the slave database are included in transaction or archive logs.

After the backup, check the backup volume and information files with the following command:

```
ls sample_master_bk*
```

For the synchronization with the slave database, transfer the backup volume and backup volume information files to the slave host through ftp or file copying.

Adding Slave Database

The **repl_make_slavedb** utility is used to restore the slave database from copied backup volumes of the master.

The **repl_make_slavedb** utility requires the master database name, the slave database name, a user name, and a password for the replication account.

Syntax

```
repl_make_slavedb master_db_name slave_db_name -u userid -p passwd
```

The following is a help screen displayed when you execute **repl_make_slavedb**.

```
#######################################################################
#
# Configuring the CUBRID replication environment : Configuring the slave database
#
# You must perform tasks as the following steps to configure the replication environment.
#     1. Configuring the replication agent (Run the repl_make_slavedb script)
#     2. Performing a full backup of the master database (Run the cubrid backupdb utility)
#     3. Copying the backup copy of the master database (from the master database host to
the slave database host)
#     4. Building the slave database (Run the repl_make_slavedb utility)
#     5. Running the replication server (Run the cubrid repl_server utility)
#     6. Running the replication agent (Run the cubrid repl_agent utility)
#
#    NOTE1: Make sure to create the distribution DB by using the repl_make_distdb script
#           before you configure the slave database.
#    NOTE2: You must perform a full backup of the master database.
#           Backup can be performed at any time before the slave database is configured. 
However,
#           if it is an online backup, you can save time spent on.
#           Initial replication by performing it before you build the
#           lave database.
#           Necessary backup files are as follows:
#             - master_db_name.bk_vinf
#             - master_db_name.bk0v???
#    NOTE3: Currently, this script must be executed in the directory where
#           received backup files are located.
#
# If the script was stopped abnormally (such as with Ctrl-C),
# delete the slave database and restart by using a utility such as
# cubrid server stop/cubrid deletedb.
#########################################################################
```

The **repl_make_slavedb** utility consists of five steps. You are required to enter necessary information in step 1, 3 and 5.

```
     STEP 1 : Perform a preliminary task to restore the backup copy of the master database.
          a. Enter the directory path where volumes of the slave database to be built will 
be saved.
             If it's the current directory, press the Enter key. >>
          b. Enter the directory path where log volumes of the slave database to be built 
will be saved.
             If it is the current directory, press the Enter key. >>
     STEP 2 : Restore the backup copy of the slave database.
             - Backup files and backup volume information files must exist in the current 
directory.
             - The slave database is being restored. Please wait...
             - The slave database has been restored.
     STEP 3 : Record the number of the last log restored and the slave database 
information in the distribution DB.
          a. Enter the name of the distribution DB. >>
             - The distribution DB does not exist on the same host.
             - Enter the IP address of the host where the distribution DB is located. >>
          b. Enter the password for the DBA account to connect to the distribution DB. >>
     STEP 4 : Create and run the slave database.
     STEP 5 : Perform post-process after building the slave database
             - You need the DBA account of the master database for this task.
             - Enter the DBA account of the master database. >>
```

- **(Step 1) Performing preliminary tasks** : Enter the directory path where data volumes and log volumes of the slave database will be located. Both paths must be canonical paths.
- **(Step 2) Synchronizing with the slave database** : Synchronize master and slave databases by restoring the backup volume of the master database to a slave database.
- **(Step 3) Recording the information of the synchronized slave database in the distribution database** : Record the number of the last log reflected on the slave database in the distribution DB so that it can be used by the replication agent.
- **(Step 4) Running the slave database** : Run the slave database server.
- **(Step 5) Post-processing** : Change owners of all classes in the slave database to replication accounts entered by the user so that unnecessary write operations do not take place. Remove all user-defined triggers so that errors do not occur during replication. You need the DBA account of the master database to perform this task.

You can skip step 6 and 7 below if you don't need to configure replication parameters and groups.

```
      STEP 6 : Configure parameters
           1. perf_poll_interval      - Unit in which replication delay time is measured 
(in seconds)
           2. size_of_log_buffer      - Log buffer size of the replication agent (page)
           3. size_of_cache_buffer    - Replication log buffer size of the replication 
agent (page)
           4. size_of_copylog         - Number of pages of the replication log
           5. index_replication       - Whether to replicate indexes
           6. for_recovery            - Whether to replicate the master database for 
recovery
           7. log_apply_interval      - Replication interval (in seconds)
           8. restart_interval        - Slave reconnection interval (in seconds)
     -- The number of the parameter to be modified (q - stop) q
```

- **(Step 6) Configuring parameters** : Each parameter for the configuration of the replication environment of the slave database can be user-defined. If the parameter is not modified, the default value is used. For more information, see Configuring Replication Parameters.

```
      STEP 7 : Configure replication groups
            - Select replication target classes.
MASTER_HOST :
-------------------------------------------------------------------------
     All classes in $master_db_name are specified as the replication target.
     Will you reconfigure replication groups? (y or n) >>
     1. Initialize replication groups and add new classes
     2. Specify classes that will be excluded from the replication
       => Enter the job number (q - quit) >>
```

- **(Step 7) Configuring replication groups** : By default, the slave database is configured the same way as the master database. Additional methods are provided to change the default configuration by adding or excluding classes to be replicated. For more information, see Configuring Replication Groups.

Caution

If you create a new slave database by using the **repl_make_slavedb** utility, you must check whether the previously used replication and trail logs are remained and delete them for successful replication. For example, if you configure trail log files in the **/home/replication/log** directory when you run the **repl_make_distdb** utility, you must check whether files such as <*master_db_name*>**.copy**, <*master_db_name*>**.copy.ar0** and <*dist_db_name*>**.tail** exist in the directory.

Starting Replication Server and Agent

Description

To start the replication, you must start the replication server and agent. The replication server reads transaction logs from the host where the master database is running and transfers them to the replication agent.

Syntax 1

```
cubrid repl_server command
command:
start     master-database-name server-network-port [-a max-agent-num] [-e error-file]
stop      master-database-name
status
```

The name of the master database and the TCP port number used for communicating with the replication agent are parameters that must be entered while the replication server is running. The TCP port number must be identical to that of the replication server entered during the execution of the **repl_make_distdb** utility. You can optionally enter the number of replication agents, which are available in the service.

Use the **cubrid repl_server status** to check if the replication server is running normally. For more information, see How to Use CUBRID Utilities (Syntax).

Configuring replication is completed and the replication service is started when the replication agent is performed on the slave host after the replication server is performed on the master host.

Syntax 2

```
cubrid repl_agent command
command:
start     dist-database-name [dba-password]
stop      dist-database-name
status
```

The replication agent requires you to enter the name of the distribution database and the password of its **DBA** account. You are not required to enter the password if it does not exist.

Once the replication agent is performed, it generates the replication, error logs, trail logs under the corresponding path configured by **repl_make_distdb** respectively.

The replication log is generated as <*master_db_name*>**.copy**; the trail log is generated as <*dist_db_name*>**.trail**; the error log is generated as <*dist_db_name*>**.err**. Extra caution is required not to corrupt files; the replication does not proceed further.

You can check the status of the replication agent with **cubrid repl_agent status**.

Deleting Slave Database

To delete the slave database completely, you must stop the replication agent and then delete the distribution database.

- **(Step 1) Stopping the replication agent** : Use the **cubrid repl_agent stop** utility to specify the name of the distribution database as a parameter.

```
cubrid repl_agent stop sample_dist
```

- **(Step 2) Deleting the distribution database** : Stop and delete the distribution database server.

```
cubrid server stop sample
cubrid deletedb sample_dist
```

- **(Step 3) Deleting replication-related logs** : Delete files such as <*master_db_name*>.**copy**, <*master_db_name*>.**copy**.**ar0**, <*dist_db_name*>.**trail** and <*dist_db_name*>.**err** that were generated during the replication.

If you want to stop replication temporarily, stop the replication agent and server only. Start the replication server before the replication agent when you restart replication later on.

Configuring Replication Groups

Description

If you want to build a slave database by replicating only certain classes from the master database, you can configure a replication group by using the **repl_make_group** utility. By default, the replication configuration is targeted to all classes where primary keys are defined in the master database, and the slave database must be created.

Syntax

```
repl_make_group master_db_name dist_db_name [option]
```

The following table shows the options for **repl_make_group** .

Option	Description
-p *passwd*	The **DBA** password for the distribution database.
-f *file_name*	The file containing a list of classes to be replicated. Class names are separated by space or comma.
-a *class_name_list*	Adding classes to be replicated as a group. The classes being separated by space or comma.
-d *class_name_list*	Deleting classes to be replicated as a group. Classes to be deleted are separated by space or comma.
-i	Determining whether or not to initialize group replication. Can be configured as either **Y** or **N**; the default value is **N**.

Follow the steps below to specify whether or not to replicate each class by using the **repl_make_group** utility:

- **(Step 1) If all classes are specified as the replication target** : The two options below are provided when configuring a replication group with the **repl_make_group** utility just after setting up slave databases by default.

```
MASTER_HOST :
-------------------------------------------------------------------------
All classes in $master_db_name are specified as the replication target.
    Will you reconfigure the replication group? (y or n) >>
    1. Initialize the replication group and add new classes
    2. Specify classes that will be excluded from the replication
==> Enter the job number (q - quit) >>
```

- **(Step 2) If partial classes are specified as the replication target** : If partial classes make up the configuration of the slave database, the following five options are provided.

```
MASTER_HOST : 192.168.2.77
-------------------------------------------------------------------------
All classes in $master_db_name are specified as the replication target.
Will you reconfigure the replication group? (y or n) >>
    1. Initialize the replication group and add new classes
    2. Specify classes that will be excluded from replication
    3. Add new classes in the current state
    4. Specify classes to be excluded from replication in the current state
    5. Specify all classes as the replication target
```

Caution

- If replication targets are added by using the replication group utility during replication, you must synchronize the data for the class of the master database by using the snapshot synchronization (**repl_make_snapshot**) utility.
- If the replication target class references another class, make sure to specify the reference class as a replication target as well.
- Input classes can be multiply specified, and must be separated by space or comma.

Configuring Replication Parameters

Description

Parameters already set can be changed for each slave database. There are eight modifiable parameters. Modification is supported by the **repl_change_param** utility.

Syntax

```
repl_change_param master_db_name slave_db_name dist_db_name [option]
```

Options for **repl_change_param** are as follows:

Options	Description
-p *passwd*	The **DBA** password for the distribution database.
-n *parameter_name*	A valid parameter name. Must be in lowercase.
-v *parameter_value*	The value of the parameter defined by the **-n** option
--help	Help message

```
repl_change_param  masterdb  slavedb distdb

               1.  perf_poll_interval      - unit in which replication delay is measured
(in seconds)
               2.  size_of_log_buffer      - size of the log buffer of the replication
agent (in pages)
               3.  size_of_cache_buffer    - size of the replication log buffer of the
replication agent (in pages)
               4.  size_of_copylog         - the number of replication log pages
               5.  index_replication       - whether or not to replicate the index
               6.  for_recovery            - whether or not to replicate the master for
replacement
               7.  log_apply_interval      - replication execution interval (in seconds)
               8.  restart_interval        - slave reconnection interval (in seconds)
```

Each parameter for the configuration of the replication environment of the slave database can be user-defined. If the parameter is not modified, the default value is used.

1. perf_poll_interval
   - Allowable range : 10 - 60
   - A parameter that measures the replication delay time (in seconds). The default value is **10**.
2. size_of_log_buffer
   - Allowable range : 100 - 1000
   - Specifies the number of buffers where repl_agent saves transaction logs temporarily. The size of a buffer is equal to that of a database page. The default value is **500**.
3. size_of_cache_buffer
   - Allowable range : 100 - 500

- Specifies the number of buffers where repl_agent saves replication logs temporarily. The size of a buffer is equal to that of a database page. The default value is **100**.

4. size_of_copylog
   - Allowable range : 1000 - 10000
   - Specifies the number of pages of the replication log. If the number of pages exceeds the specified number, pages already processed are truncated. The default value is **5000**.
5. index_replication
   - Allowable value : Y/N
   - Configure this parameter if you want the indexes to be replicated automatically. Y indicates "Set," and N indicates "Do not set." Y/N values are not case sensitive. The default value is **N**.
6. for_recovery
   - Allowable value : Y/N
   - Set when the replication is configured with the purpose of replacing with the master database if a failure occurs in the master. If this value is configured to Y, the value of **index_replication** is also configured to Y. That is, all classes in the master are replicated. Y indicates "Set", and N indicates "Do not set." Y/N values are not case sensitive.
   - You can set a server to replace the master database (for_recovery=Y) and then release it from this purpose by changing the value of the **for_recovery** parameter to N. However, you must reconfigure the replication if you want to configure a server again to replace the master database after it was released from such purpose. The **for_recovery** parameter for a slave database can be configured only for a single master database. The default value is **N**.
7. log_apply_interval
   - Allowable range : 0 - 600
   - A parameter that configures the interval within which changes in the master database will be reflected to the slave. The unit is second. If it is configured to 0, changes are reflected in real time. The default value is **0**.
8. restart_interval
   - Allowable range : 1 - 60
   - A parameter that configures the interval in which to reconnect if the connection to the slave database fails. The unit is second. The default value is **100**.

Example

```
$ repl_change_param masterdb slavedb distdb admin
-- The number of parameters to be changed (q - quit) > > 1
    > > perf_poll_interval value (10 - 60) > > 10
-- The number of parameters to be changed (q - quit) > > q
$ repl_change_param masterdb slavedb distdb admin -n size_of_cache_buffer -v 500
        The parameter was changed successfully.
```

Synchronizing Snapshots

Snapshot synchronization (**repl_make_snapshot**) is a utility used to add to the replication classes of the master database that were not included in the replication even though they were replication targets to the current replication. This utility can be used in the case of slave databases that replicate only part of the master database via the **repl_make_group** utility.

Syntax

```
repl_make_snapshot -m masterdb_name -s slavedb_name -d distdb_name -cf classes_file_name
[-p dist_db_password]
```

- **-m** *masterdb_name* : The name of the master database
- **-s** *slavedb_name* : The name of the slave database
- **-d** *distdb_name* : The name of the distribution database

- **-cf** *classes_file_name* : The name of the file containing the list of classes to be added. Class names are separated by space or comma.
- **-p** *dist_db_password* : The **DBA** password for the distribution database

Replacing Master Database (Manual)

If the master database cannot be restored, you can replace the master database with a slave. To replace the master database with the slave database, the following conditions should be met.

- All classes in the master database must be replicated to the slave database with primary key.
- The final schema information of the master database must have been backed up so that information about user accounts, indexes and triggers of the master database can be obtained.
- Deleting replication accounts of the slave database must be done manually after all tasks are completed.
- Serials defined in the master database are not restored. Therefore, if serials are used, you must adjust their maximum values manually after all tasks are completed.

You can replace the master database with a slave by using the following steps.

Preliminary step: Back up the master database schema

The schema of the master database must be backed up by using the **cubrid unloaddb** utility every time one of its user accounts, classes, indexes or triggers changes.

```
cubrid unloaddb -s -C master_db_name
```

master_db_name_schema and master_db_name_trigger files are created as a result of the execution of the **cubrid unloaddb** utility. These files are stored and then used later when the master database is replaced with a slave.

Step 1: Back up and copy the slave database

Perform a full backup with the **cubrid backupdb** utility and then copy the backup volume and all active log and archive log files to the host where the master database will be located.

```
cubrid backupdb slave_db_name
```

Make sure to back up the whole slave database to replace the master with it. You must copy all active and archive log files as well as the backup volume to restore the whole database.

Step 2: Restore and rename the slave database

Restore the backup volume on the master host and rename it.

```
cubrid restoredb slave_db_name
cubrid renamedb slave_db_name master_db_name
```

If the database volume path of the slave host and that of the master host are different, specify the volume path as the same one of the previous master database by using the **-n** option to **cubrid restoredb**. For more information about the **cubrid restoredb** utility, see Database Restore.

Step 3: Apply user account and schema changes

Create the same user accounts and indexes as were on the master database before failure by using the schema information file saved in the "Preliminary step: Back up the master database schema."

There is a syntax that recreates the password for the **DBA** account and that recreates a user account at the beginning of the master_db_name_schema file. As the account already exists in the slave database, change the syntax creating a user account to a syntax which finds one:

```
call add_user('REPL', '') on class db_root to auser;
```

Find and replace every occurrence of the above syntax with the following.

```
call find_user('REPL') on class db_user to auser;
```

An error occurs because a DDL statement creating the schema of the master database is stored in the master_db_name_schema file. Ignore this error.

```
csql -S -u dba -p dba_passwd master_db_name -e -i master_db_name_schema
```

Step 4: Define triggers

The next step is to reflect again the trigger information defined in the master in the same way.

```
csql -S -u dba -p dba_passwd master_db_name -e -i master_db_name_trigger
```

Post-step 1: Adjust serials

Adjust the current values of serials because they are not to be restored.

```
csql -S -u dba -p dba_passwd master_db_name
drop serial a;
call find_user('REPL') on class db_user to auser;
create serial a
     start with 1743715
     increment by 1
     minvalue 1
     maxvalue 10000000000000000000000000000000000000
     nocycle;
update db_serial set owner = :auser, started=1 where name= 'a';
```

Post-step 2: Delete the replication account

Delete the replication account.

```
csql -S -u dba -p dba_passwd master_db_name -c "drop user repl_user"
```

Replacing Master Database (Automatic)

The automatic method provides a faster and more convenient way of replacing the master database with a slave. The replacement is done by using the **repl_check_sync** and **repl_change_master** utilities. Only slave databases whose replication parameter option **for_recovery** was set to 'Y' can be used for automatic replacement.

When a slave database is created, the ownership of every class is changed to that of the replication user. Therefore, authorization changes are not replicated even in **for_recovery**. Only user and group additions/deletions are replicated. After the master database is replaced with the slave, authorization for each user must be defined by **DBA**. Also, after the slave database has replaced the master, triggers must be activated by **DBA** because all of them are inactivated in the slave database.

Check the synchronization status of the slave database (repl_check_sync)

If a failure occurs in the master database, you must use one of the slave databases to replace the master. In such an event, the first thing that should be done is to check the synchronization status among all slave databases and whether or not it has already been completed. The **repl_check_sync** utility checks the status of the synchronization among currently configured slave databases.

Syntax

```
repl_check_sync dist_db_name.config [-p passwd]
```

- *dist_db_name.config* : Distribution database configuration log file
- **-p** *passwd* : The **DBA** password for the distribution database.
- dist_db_name.config : Writes the **cubrid repl_agent** host line by line (hostname port_id)

Example

```
 192.168.2.100 5627
 192.168.2.200 5627
```

As the result of the execution, the numbers of the last logs to be reflected on each slave database are displayed.

Result :

agent_ip status   log number   last_updated_time

1.1.1.1	A	1234/110	15:30:30
1.1.1.2	A	1234/123	15:29:30
1.1.1.3	A	1234/110	15:28:30
1.1.1.4	A	1234/123	15:27:30
1.1.1.5	A	1234/123	15:26:30

In the output, status consists of the following information.

A : Active (Replication being performed)

F : First (Start the initial replication)

I : Idle (Idle - No replication contents)

Replace the master database with the slave (repl_change_master)

After checking the synchronization status with the **repl_check_sync** utility, you must change the information about triggers, accounts, and so on to be the same as that of the original. The **repl_change_master** utility transforms the specified slave database to a master database based on the information about the master contained in the distribution database.

Syntax

```
repl_change_master servers_info_file
```

- *servers_info_file* : Writes information of the distribution database line by line (dist_name dist_host dist_passwd)

Example

```
distdb1 192.168.2.100 admin
distdb2 192.168.2.200 admin
```

Managing Replication Status

Checking the Replication Status

Use **cubrid repl_server status** or **cubrid repl_agent status** to check if replication is being performed.

- Check the status of the replication server or agent only.

```
$ cubrid repl_server status
```

or

```
$ cubrid repl_agent status
@ cubrid replication status
repl_agent distdb (rel 8.1, pid 22203)
repl_server masterdb (rel 8.1, pid 12341)
```

- Check the status of all CUBRID processes.

```
$ cubrid service status
@ cubrid master status
++ cubrid master is running.
@ cubrid server status
Server slavedb (rel 8.1, pid 11325)
Server distdb (rel 8.1, pid 31440)
Server masterdb (rel 8.1, pid 29191)
@ cubrid broker status
    NAME                     PID   PORT   AS   JQ           REQ TPS AUTO   SES SQLL CONN
===================================================================
* query_editor 12149 30300     5     0               0 ---   ON     OFF ON:A AUTO
* broker1           12161 33300     5     0               0 ---   ON     OFF ON:A AUTO
@ cubrid manager server status
++ cubrid manager server is running.
@ cubrid replication status
  repl_agent distdb (rel 8.1, pid 22203)
  repl_server masterdb (rel 8.1, pid 12341)
```

Monitoring Replication Performance

Replication performance monitoring is not supported by a separate utility, but as a performance log file to be created in the directory defined when the distribution database was created and the one in which trail logs are created.

The performance log records the lapse of time between when a transaction was committed in the master database and when it was reflected on the slave database. This holds true only when the system time between the master server and the slave server is synchronized. If the time between the two servers is not synchronized, a distortion occurs in the delay time log. The delay time is determined by the perf_poll_interval parameter specified for each slave database.

The following is an example of checking the performance log by using the tail command when the database name is assumed to be distdb.

```
$ tail ?f distdb.perf
--------------------------------------------------------------------------------
No.
master_db_name   tran_index      master_time                  slave_time              del
ay
--------------------------------------------------------------------------------
001    mdb                     8                     2007/10/17  10:03:24   2007/10/17
10:03:24   0
002    mdb                    -1                     ----/--/-- --:--:--   2007/10/17
10:04:07   0
003     mdb                   10                     2007/10/17 10:04:07   2007/10/17
10:04:07   0
004      mdb                    11                     2007/10/17 10:04:07   2007/10/17
10:04:07   0
005    mdb                   -1                     ----/--/-- --:--:--   2007/10/17
10:04:07   0
```

The part in the middle where tran_index is displayed as -1 indicates that replication has been synchornized at the point.

Replication Parameters

The following are replication-related parameters.

Replication parameters

Parameter	Default	Description	How to Configure	Related Process
replication	no	If it is configured to yes, creates the replication log	By modifying the cubrid.conf file	$CUBRID/bin/cub_server
error_log	NULL	The directory path where error logs of the replication agent will be saved	By specifying it as an input to the **repl_make_distdb** utility	$CUBRID/bin/repl_agent
trail_log	NULL	The directory path where trail logs of the replication agent will be saved	By specifying it as an input to the **repl_make_distdb** utility	$CUBRID/bin/repl_agent
copylog_path	NULL	The directory path where the replication log of the replication agent will be saved	By specifying it as an input to the **repl_make_distdb** utility	$CUBRID/bin/repl_agent
agent_port	NULL	The socket port number to be used to obtain the information of the repl_agent with repl_check_sync	By specifying it as an input to the **repl_make_distdb** utility	$CUBRID/bin/repl_agent
perf_log_size	10000	The size of .perf (a file that records the replication delay time) in number of lines.	By specifying it as an input to the **repl_make_distdb** utility	$CUBRID/bin/repl_agent

commit_interval_msecs	500	The interval within which the repl_agent commits replication information to the slave in mseconds. The repl_agent reflects information about each transaction when it replicates the slave database, but does not commit. Commits are performed within the interval specified by this option.	By specifying it as an input to the **repl_make_distdb** utility	$CUBRID/bin/repl_agent
retry_connect	n	Whether or not to restart the repl_agent when it stops due to a network failure	By specifying it as an input to the **repl_make_distdb** utility	$CUBRID/bin/repl_agent

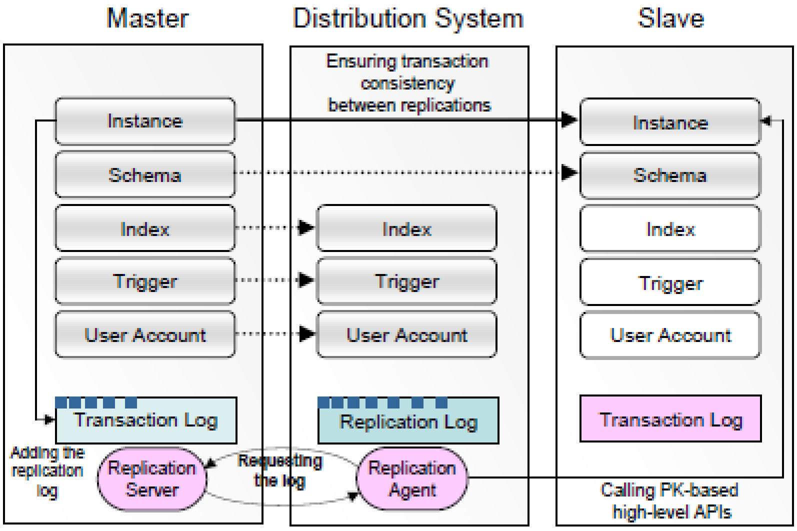
Implementing Replication

Rules for Replication Implementation and Architecture

The rules in designing and implementing the CUBRID replication system are as follows:

- Transaction consistency must be ensured.
- Schema independence of the slave database must be ensured to utilize its various functions. That is, it must be possible to build separate tables or indexes in the slave database.
- Effects on the master database as a result of replication must be minimized.
- Real-time replication must be ensured.

Therefore, the CUBRID replication system runs based on transaction logs, and it is designed in such a way that the replication server, independent of the master server, transfers them to the replication agent, which in turn analyzes them and passes the results to the slave database.



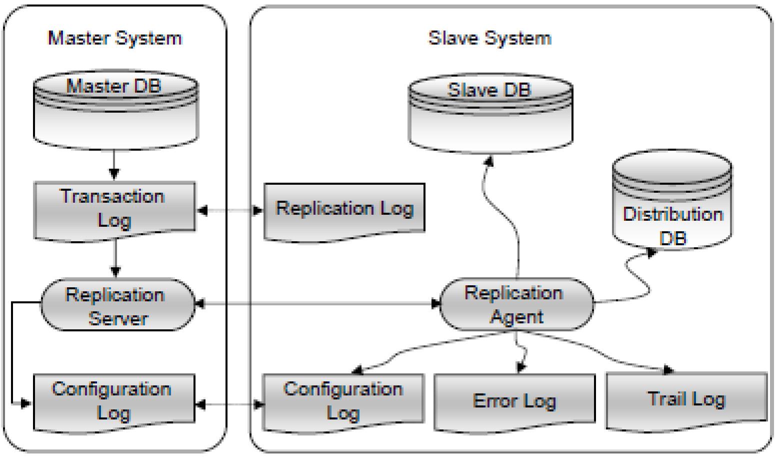
Changes in instance- and schema-related information are reflected on the slave system in real time. In the case of indexes, triggers and user accounts, changes are recorded in the distribution database only without being reflected on the slave database; in this way, independence of its schema is ensured. The final data of the master database, schema and indexes recorded in the distribution database are automatically synchronized if the master database is replaced by the slave database due to a system failure in the master; however, triggers and user accounts are not synchronized automatically.

Architecture of Replication Server and Agent

The replication server receives many requests from and transfers the transaction logs to multiple replication agents. The replication server consists of threads that processes the requests and a threads that reads transaction logs from the disk.

The replication agent saves transaction logs transferred from the replication server to local disk, analyzes and reflects them on the slave database. The replication agent consists of the threads that save the received transaction logs in a buffer, the threads that write the log data stored in the buffer to the replication log, and the threads that reflect the changes to the slave database.

A single master database can be replicated to multiple slaves at the same time.



Distribution Database Schema

The distribution database is a database for managing metadata required for replication. It consists of information about the master and slave databases, trail, replication parameters, and others for automatic replication.

Class Name : db_info

db_info is the super class of master_info and slave_info, and manages commonly required items.

Attribute Name	Type	Description	Remarks
dbid	integer	Unique identifier of the database	Primary key
dbname	varchar(126)	Database name	
master_ip	varchar(50)	IP address of the database server	
portnum	integer	TCP port of the replication server	

Class Name : master_info

The master_info class manages information such as the database name of the original database to be replicated, the IP address and the TCP port number used to connect to the replication server, and the directory path to the replication log file. You can configure the value by executing the **repl_make_distdb** utility.

Attribute Name	Type	Description	Remarks
dbid	integer	Unique identifier of the database	Inherits from db_info
dbname	varchar(126)	Database name	Inherits from

			db_info
master_ip	varchar(50)	IP address of the replication server	Inherits from db_info
portnum	integer	Uses cubrid_port_id of cubrid.conf	Inherits from db_info
copylog_path	varchar(256)	Path to the file where replication logs will be saved	
size_of_log_buffer	integer	Specifies the number of buffers where repl_agent will store transaction logs temporarily. The size of a buffer equals that of a database page.	Default value : 500
size_of_cache_buffer	integer	Specifies the number of buffers where repl_agent will store transaction logs temporarily. The size of a buffer equals that of a database page.	Default value : 100
size_of_copylog	integer	Specifies the number of pages of the replication log. If the number of pages exceeds the specified number, pages already processed are truncated.	Default value : 5000
start_pageid	integer	The first page number of the replication log	Not used
first_pageid	integer	The first page number of the active replication log	
last_pageid	integer	The last page number of the replication log	

Class Name : slave_info

The slave_info class manages the database name of the replication target slave database, replication accounts and passwords. You can configure the value by executing the **repl_make_slavedb** utility.

Attribute Name	Type	Description	Remarks
dbid	integer	Unique identifier of the database	Inherits from db_info
dbname	varchar(126)	Database name	Inherits from db_info
master_ip	varchar(50)	IP address of the database server	Inherits from db_info
portnum	integer	TCP port of the replication server	Inherited from db_info
userid	varchar(32)	Replication account of the slave database	
passwd	varchar(32)	Replication account password of the slave database	
trails	set_of(trail_info)	The number of the last log reflected on the slave database	Initialized when the **repl_make_slavedb** utility is running

Class Name : trail_info

The trail_info class manages information about the master to be replicated and slaves as well as the first log number to be used when the replication starts. You can configure the value by executing the **repl_make_slavedb** utility.

Attribute name	Type	Description	Remarks
master_dbid	integer	Identifier of the master database	
slave_dbid	integer	Identifier of the slave database	
final_pageid	integer	The number of the last log reflected on the slave database	

final_offset	integer	The location of the last log reflected on the slave database	
all_repl	char(1)	Whether or not to replicate all class. Y or y : Yes N or n : No	Currently has the same effect as for_recovery. Will run differently in the future.
repl_count	number(15,0)	The number of transactions replicated	You can check the updated information by using the **repl_check_sync** utility.
status	char(1)	Current status information of repl_agent. F : Agent has never run since the replication was configured. A : Replication is being performed. I : Idle state	You can check the updated information by using the **repl_check_sync** utility.
error_msg	string	Not used	
stop_time	timestamp	Not used	
perf_poll_interval	integer	A parameter that configures the measurement of the replication delay time (in seconds). Allowable range : 10 - 60 seconds	Default value : 10
log_apply_interval	integer	Not used	
for_recovery	integer	Set when the replication is configured with the purpose of replacing the master database if a failure occurs in the master. If the value is y, the value of index_replication is also configured to y and all classes in the master are replicated. y : Configured n : Not configured	Default value : n
index_replication	char(1)	Configuring to replicate indexes automatically. y : Configured n : Not configured	Default value : n
restart_interval	integer	Configures the interval in which to reconnect when the connection to the slave database fails (in seconds). Also is the interval between repl_agent connections made to the distribution database and reflecting the information.	Default value : 100

Constraints on Replication

- Replication features are supported only on UNIX-family platforms.
- Master and slave databases must be configured on the same platform. For example, you cannot configure the master database on Linux and the slave on Solaris.
- Only tables with primary key can be replicated.
- Tables with attributes of object type can cause errors during replication. You can use the foreign key ON CACHE OBJECT to replicate an object type. For more information, see Foreign Key Constraint.
- The master database can be replaced by the slave database only when primary keys are configured in all tables in the master database.
- You cannot execute the **UPDATE** statement which meets the condition 1 and 2 at the same time. In this case, errors are as follows:
  - Condition 1 : Executing **UPDATE** with the attribute with unique constraints

- Condition 2 : Executing **UPDATE** several records

```
// English
Current version of replication does not allow changing multiple rows with a single UPDATE
statement which can violate the UNIQUE constraint.

// Korean
한글 버전의 복제는 하나의 UPDATE 질의로 여러 레코드를 변경시키는 것이 UNIQUE 제약을 위반하는 경우
허용되지 않습니다.
```

Direction for Replication

- In most cases, changes in the master database are reflected on the slave database in real time. However, replication can be delayed due to long transactions which update a large amount of data at a time.
- It is recommended, if possible, not to delete at least 5 - 10 transaction archive logs so that changed data can be read when replication is delayed.
- If you replace a new slave database by using the **repl_make_slavedb** utility, you must delete log files such as the replication log and trail logs that were used for the previous replication configuration before you run the replication agent.
- **TIMESTAMP** does not mean the time when replication is applied to the slave database, but is replicated with the same value as one of the master database.
- If the master database stops due to a failure, the replication server and the replication agent replicates data changed prior to the failure to the slave system without stopping. However, if the slave database stops due to a failure, the replication agent also stops operation. Therefore, the replication agent will start manually after the slave database server restarts.
- Be careful not to damage the replication log created during the replication, the first replication archive log and trail logs. If you don't specify the **-ar** option when you run the replication agent, it creates only the first replication archive log **<master_db_name>.copy.ar0** without creating more. Replication archive logs are files that store transaction logs already reflected; they do not need to be kept because the replication agent does not use them any more. However, you must not to delete the first replication archive log because it will be needed for recovery in case of failure.
- It is recommended not to modify data items in the distribution database connected manually.

CUBRID HA

Overview

High Availability (HA) refers to an ability to minimize system down time while continuing normal operation of service in the event of hardware, software, or network failure. This ability is a critical element in the network computing area where services should be provided 24/7. A HA system consists of more than two server systems, each of which provides uninterrupted services even when a failure occurs in the other system component.

The CUBRID HA is a high-availability feature applied to CUBRID. The CUBRID HA feature provides services, keeping the database synchronized for multiple server systems. In addition, if a failure occurs in a system where a service is being performed, this feature minimizes the service down time by allowing another system to perform the service automatically.

The CUBRID HA feature has a shared-nothing architecture and performs the following two steps for the data synchronization from the master to the slave database server.

- A transaction log multiplication step where the transaction log created in the database server is replicated in real time to another node
- A transaction log reflection step where data is applied to the slave database server through the analysis of the transaction log being replicated in real time

The CUBRID HA feature performs these two steps so that synchronized data is maintained all the time between the master and slave databases. Therefore, if an unexpected failure occurs in the master database where a service is being provided, the slave database server can provide the service without interruption in place of the master database server.

The CUBRID HA feature monitors the state of the system and CUBRID in real time and uses the Heartbeat solution of the Linux-HA project to automatically perform failover in case of system failure.

Note In CUBRID 2008 R2.0, R2.1, Linux Heartbeat solution-based HA features are supported and from CUBRID 2008 R2.2, self-developed CUBRID Heartbeat-based HA features are supported.

Glossary

Master Database

The source database that becomes the target of the replication. All operations including a read and write operations are performed in this database.

Slave Database

A replicated database (replica) with the same contents as the master database. Changes in the master database are automatically reflected in the slave database. Unlike the master database, the slave database can be used for read operations only.

Active Server

A server (also called a "primary server") that provides users with services. An active server provides all services including read and write by using the master database.

Standby Server

A server (called called a "secondary," "passive" or "failover" server) that provides services in place of an active server when it cannot provide services due to failure. A standby server provides a read service by using a slave database.

Failover

A feature that allows a standby server to perform the failover and continue to provide services when the failure of an active server or the system running the active server is detected.

Failback

A feature that allows the restored active server to resume services when it is restored to the original state.

Role Change

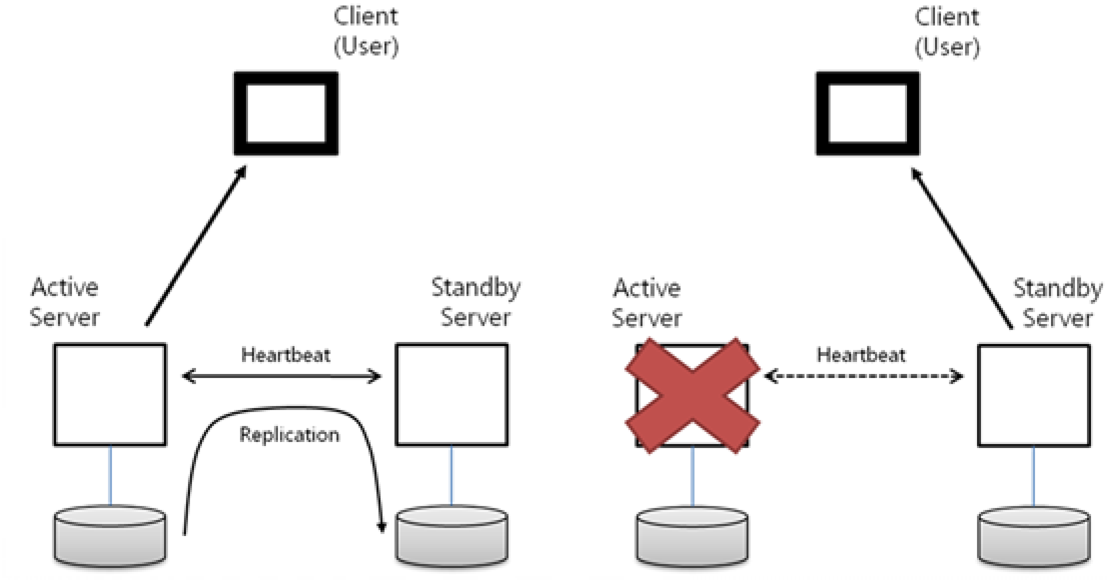
A feature that allows services to be provided continuously even when the failure in the previous active server is restored.

Heartbeat

An essential element for providing HA features. The CUBRID Heartbeat feature is included in the cub_master process. It exchanges heartbeat messages with cub_master processes of other nodes and executes failover on the standby server when a failure is detected. It also monitors the availability of the HA related processes (**cub_server**, **copylogdb**, **applylogdb**) on a regular basis.

Server Duplication

Building the system with duplicate server hardware to provide HA functionalities. Two methods are used: to allow the standby server to perform the functionality of the active server upon failure (Active-Standby, see the figure below) and to build a duplicate system that provides services while additionally performing the roles of the server upon failure (Active-Active).



Database Server Multiplication

An architecture with multiple database servers so that the service will be provided without interruption even when a database failure occurs. If a failure occurs in an active database providing a service, a standby database with the same data can provide the service.

Broker Multiplication Architecture

An architecture built with broker multiplication so that a service can be provided without interruption by another broker when a failure occurs in a certain broker. In addition, each broker can have different characteristics as described below.

- **Read-only broker** : A broker that performs read operations only. It provides services through the connection with a standby server. If a standby server does not exist, it can send a read request to an active server. That is, the order of attempting to connect to a database server is as follows: first it attempts to connect to a standby server; if it fails, it can be connected to an active server.

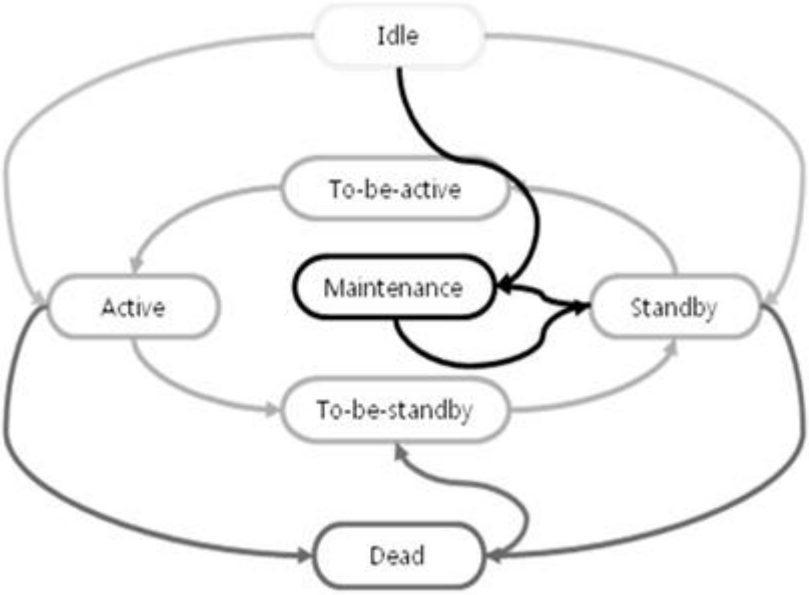
- **Slave-only broker** : Unlike the read-only broker, a slave-only broker can send a request only to a standby server. It does not attempt to connect to an active server even when a standby server does not exist.

Transaction Log Multiplication

A feature that allows the transaction log created in an active server to be sent in real time to one or more standby servers so that the same log will be recorded in all the servers.

HA Mode of the CUBRID Database Server

- **active** : A state that provides a service for common read and write requests and creates transaction logs required for replication.
- **to-be active** : When the state of the server changes from standby to active, it goes through the to-be-active stage before it becomes active. In a to-be-active state, all incoming requests are suspended, and the server changes to an active state after reflecting the unapplied replication log.
- **standby** : A state that provides a service for read requests only, but denies write requests.
- **to-be standby** : When the state of the server changes to standby, it goes through the to-be-standby stage before it becomes standby. In a to-be-standby state, incoming requests are denied, and the server changes to a standby state when the transaction being performed is complete.
- **maintenance** : A mode for database maintenance operations (schema change, configuration change, etc.). You can perform necessary maintenance operations by temporarily excluding the given database from the HA configuration and then running the database in maintenance mode. This mode behaves as follows:
    - Only clients of the local host can connect; copylogdb and applylogdb utilities which replicate or apply the transaction log cannot connect.
    - The database can be modified with write operations, but the replication logs for the changes are not created.



- State change of the database server during failover
    - active server: active -> dead
    - standby server: standby -> to-be-active -> active

HA Constraints

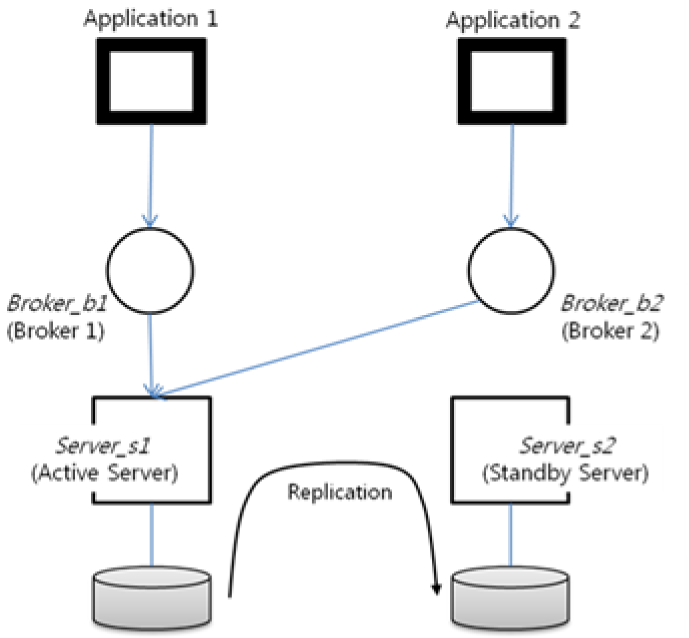
- You can use CUBRID HA features only in Linux-based systems.
- Active and standby servers must be configured on the same platform.
- Only tables for which primary keys are defined can be replicated.

- In the following cases, data between a master database and a slave database may not match.
- If a table trigger or Java Stored Procedure have been configured : It is executed in duplicate with a slave database.
- If a table method is used : When replication is performed, a replication logs are not generated.
- If the **NOT NULL** option is configured for a column, using CUBRID Manager : The replication log are not generated.
- If an operation is performed in standalone mode in HA configuration: If server processing mode is inactive in standalone mode, the DB operations can be performed. Therefore, the logs generated in standalone mode is not reflected in a slave database.

HA Configuration

To use the HA feature, the following configuration is required.

For HA configuration, the CUBRID heartbeat feature must be activated in both active and standby servers. The broker and the database server can be configured either on a single device or in separate systems. The following figure describes how to configure the environment and run.



Create a Node Group ID

Create the same user account (e.g. ha_user1) for Active and Standby servers and start CUBRID with this account. The UNIX user account that is created becomes the node group ID of the server that will run in HA mode.

Prepare to Run HA Start Script

The HA script is located in the **CUBRID/share/init.d/cubrid-ha**. Prepare it in an active and a standby **(etc/init.d/**) server, respectively. If you configure HA with a different account in one host and then register this configuration to the system server, you should changer the HA start script name into **cubrid-ha**-{**unix user id**}. And then you should copy each script to **/etc/init.d** directory. Note that a root authorization is required to register the system service; otherwise, you can omit the script name change and copy the script.

```
[root@server_s1 init.d]# cd /home/ha_user1/CUBRID/share/init.d/
[root@server_s1 init.d]# cp cubrid-ha /etc/init.d/
```

Copy the Database

Create a new DB (e.g. tdb01) in the active server and copy it to the standby server. You can use one of the following methods - data volume copy, backup/recovery and unloaddb/loaddb.

Configure the Database Server (cubrid.conf)

You must add the HA related parameter to the **cubrid.conf** configuration file. The HA related parameter must be configured in [common] section. For databases that will not run in HA mode, set the value of the **ha_mode** to no in [@<*database*>] section. An error will occur when the value of the **ha_mode** is **no** in [common] section and the value of the **ha_mode** is **yes** in [@<*database*>] section.

- **ha_mode** : A parameter used to configure the HA feature. Its default value is off. Change the value of this parameter to **on**.
- **ha_port_id** : A parameter used to set the UDP port. Through this port, each **cub_master** process exchanges **heartbeat** messages and detects node failures.
- **ha_node_list** : A parameter used to specify the ID for the node group running in HA mode and the host names of the member nodes that belong to the group. Host names of the member nodes and of the current node must be registered in the **/etc/hosts**. The following example specifies the host names of the active server (server_s1) and of the standby server (server_s2) as members of the node group called ha_user1.

```
#cubrid.conf
[common]
...
ha_mode = on
ha_port_id = 41523
ha_node_list = ha_user1@server_s1:server_s2
```

Configure the HA Script

Add the following configuration to the HA start script files prepared in active and standby database servers.

- **CUBRID_USER** : Specifies the node group ID, which is the UNIX user account used to install and start CUBRID.
- **DB_LIST** : Specifies the name of the database that will run in HA mode. If you specify more than one name, separate them with spaces.

```
"#cubrid-ha

CUBRID_USER = ha_user1
DB_LIST = ’tdb01’

# DB_LIST = ’tdb01 tdb02 tdb03’
```

Configuring the Broker

The default operation mode of the broker is one that requires read and write operations. If necessary, you can set it as a Read only or Slave only broker by using the **ACCESS_MODE** parameter in the **cubrid_broker.conf** file.

```
#cubrid_broker.conf
ACCESS_MODE = RW|RO|SO

RW := Read-Write broker (default value)
RO := Read-Only broker
SO := Slave-Only broker
```

You can check the operation mode of a running broker with the **-f** parameter of the **cubrid broker status** utility. For more information about the broker status, see Checking Broker Status.

```
% broker1  - cub_cas [4430,40821] /home/CUBRID/log/broker/broker1.access /home/CUBRID/
 JOB QUEUE:0, AUTO_ADD_APPL_SERVER:ON, SQL_LOG_MODE:ALL:100000
 LONG_TRANSACTION_TIME:60, LONG_QUERY_TIME:60, SESSION_TIMEOUT:300
 KEEP_CONNECTION:AUTO, ACCESS_MODE:RW
------------------------------------------------------------------------------------------
-----------------------------
ID   PID   QPS   LQS PSIZE STATUS          LAST ACCESS TIME     DB       HOST   LAST
CONNECT TIME       CLIENT IP
------------------------------------------------------------------------------------------
-----------------------------
1 26946     0     0 51168 IDLE        2009/11/06 16:06:41      -          -
                  -       10.0.1.101
2 26947     0     0 51172 IDLE        2009/11/06 16:06:41      -          -
                  -       10.0.1.101
3 26948     0     0 51172 IDLE        2009/11/06 16:06:41      -          -
                  -       10.0.1.101
```

```
4 26949     0     0 51172 IDLE         2009/11/06 16:06:41      -          -
                   -       10.0.1.101
5 26950     0     0 51172 IDLE         2009/11/06 16:06:41      -          -
                   -       10.0.1.101
```

Setting Database Host Information

You need to add the host information of the Active and the Standby servers to be used in the HA configuration to the database location file (**databases.txt**) in the broker/server system. The Active and the Standby servers are separated by a colon (:). More than one Standby servers can be added with each host being separated by a colon (:). As described above, in the environment where multiple hosts are specified for the same database, you must specify a host name (@host) next to a target database name when executing utilities such as **csql**, **backupdb** or **applylogdb** in client/server (CS) mode. On the other hand, in case of executing utilities in stand-alone (SA) mode, you cannot specify a host name next to a target database.

```
#databases.txt
#db-name         vol-path         db-host                   log-path
tbl01            /home/db/db2      server_s1:server_s2       /home/db/db2
tbl02            /home/db/db1      server_s1:server_s2       /home/db/db1
```

HA Start Mode

cubrid-ha start

cubrid-ha start is a command that starts the database server in HA mode. This command starts **cub_master**, **cub_server**, **copylogdb** and **applylogdb** processes in that order, and creates the path to which the replication log (**copylog**) is saved. Active and standby servers are determined depending on the order in which the **cubrid-ha** start command is executed.

The database of the server node (server_s1) that starts first becomes the master database.

```
[root@server_s1 ~]# service cubrid-ha start
Starting cubrid-ha: [ OK ]
```

The database of the server node (server_s2) that starts later becomes the slave database.

```
[root@server_s2 ~]# service cubrid-ha start
Starting cubrid-ha: [ OK ]
```

cubrid-ha status

cubrid-ha status is a command that checks the status information of the database server running in HA mode. The status information of the HA node where this command was executed and of the HA-related processes is displayed.

```
[root@server_s2 ~]# service cubrid-ha status

HA-Node Info (current server_s2, state slave)
   Node server_s2 (priority 2, state slave)
   Node server_s1 (priority 1, state master)

HA-Process Info (master 30519, state slave)
   Applylogdb tdb01@localhost:/home1/cubrid1/DB/tdb01_server_s1 (pid 30796, state
registered)
   Copylogdb tdb01@server_s1:/home1/cubrid1/DB/tdb01_server_s1 (pid 30788, state
registered)
Server tdb01 (pid 30551, state registered)

++ cubrid heartbeat list: success
Status cubrid-ha: [ OK ]
```

cubrid-ha stop

cubrid-ha stop is a command that stops the database server running in HA mode. This command stops the **applylogdb**, **copylogdb**, **cub_server** and **cub_master** processes in that order. Note that all related processes are also terminated. If the command is executed on only one node (server_s1), failover is performed on the other node (server_s1): if it is executed on both nodes (server_s1, server_s2), the service will stop.

```
[root@server_s1 ~]# service cubrid-ha start
Starting cubrid-ha: [ OK ]
```

To stop the service, execute this command on the other node (server_s2).

```
[root@server_s2 ~]# service cubrid-ha start
Starting cubrid-ha: [ OK ]
```

cubrid-ha deact

cubrid-ha deact is a command used to failover a service node to a different node to perform the batch operation at the node in service mode. This command changes the status of the node in service mode (server_s1) to unknown, and automatically failovers to a standby server node (server_s2).

```
[root @server_s1 ~]# service cubrid-ha deact
deactivate cubarid-ha                                        [   OK   ]
[root @server_s1 ~]# service cubrid-ha status

HA-Node Info (current server_s1, state unknown)

HA-Process Info (master 13396, state unknown)

Status cubrid-ha: [ OK ]
```

cubrid-ha act

cubrid-ha act is a command that is used to recover the node to a service node after a batch operation is completed on the node in which **cubrid-ha deact** is executed. This command changes the status of the node (server_s1) from "unknown" to "active."

```
root@server_s1 ~]# service cubrid-ha act
activate cubrid-ha: [ OK ]
```

cubrid heartbeat deregister

cubrid heartbeat deregister is a command that is used to stop a certain process running on the current node and delete it from the list of processes to manage. The process ID must be specified. Even if the process stops, failover is not performed on the standby server.

The following is an example that stops the tdb01 server process running on the current node and deletes it from the process list when tdb01 and tdb02 are included in the database list and running in HA mode.

```
[ha_user1@server_s1 ~]$ cubrid heartbeat deregister 4087
@ cubrid heartbeat deregister: 4087

 HA-Process Info (master 4072, state master)
   Applylogdb tdb01@localhost:/home1/cubrid1/DB/tdb01_server_s2 (pid 4307, state 
registered)
   Applylogdb tdb02@localhost:/home1/cubrid1/DB/tdb02_server_s2 (pid 4313, state 
registered)
   Copylogdb tdb02@server_s2:/home1/cubrid1/DB/tdb01_server_s2 (pid 4311, state registered)
   Copylogdb tdb01@server_s2:/home1/cubrid1/DB/tdb01_server_s2 (pid 4305, state registered)
   Server tdb02 (pid 4195, state registered_and_active)
   Server tdb01 (pid 4087, state deregistered)
```

HA Utilities

CUBRID utilities used for the HA feature are as follows. The following utilities can be used only in servers where the **ha_mode** parameter is configured to **on**.

cubrid changemode

cubrid copylogdb

cubrid applylogdb

Outputting/Configuring the Operation Mode of the Database Server

Description

The syntax for the **cubrid changemode** utility, which is used to output or change the state of the database server, is as follows. The history of the server error log is output every time the operation mode of the server changes. The current operation mode is output if the **-m** option is not specified.

Syntax

```
cubrid changemode [option] <database_name>@<hostname>
option:
-m, --mode=<MODE> : Specifies the mode to change. Available values are active, standby and maintenance.
```

Example 1

```
[ha_user1@server_s1 ~]$ cubrid changemode tdb01@server_s1
The server 'tdb01@server_s1''s current HA running mode is active.

[ha_user1@server_s1 ~]$ cubrid changemode tdb01@server_s2
The server 'tdb01@server_s2''s current HA running mode is standby.
```

Example 2

You can check the HA mode of database currently connected by entering **;database** command in the CSQL session.

```
csql> ;database
     demodb@localhost (active)
```

Saving the Transaction Log of the Database Server

Description

The syntax for the **cubrid copylogdb** utility, which is used to copy the transaction log created by the remote database server to a specified path, is as follows:

There are three ways of sending the transaction log as follows: You should choose one that meets your operation policy.

- Synchronous : A database server sends all transaction logs to the copylog process; it does not perform commit until the logs are written to a disk. That is, this method guarantees both to send and write logs. **sync** is specified for the **cubrid copylogdb -m** value.
- Semi-Synchronous : A database server sends all transaction logs to the **copylogdb** process; it performs commit when it gets response.
  That is, this method guarantees only to send logs; it does not guarantee to write logs. **semisync** is specified for the **cubrid copylogdb -m** value.
- Asynchronous : A database server performs commit right after it sending transaction logs to the copylog log. That is, this method even does not guarantee to send logs. **async** is specified for the **cubrid copylogdb -m** value.

Syntax

```
cubrid copylogdb [option] <database_name>@<hostname>
option:
-L, --log-path=<PATH>: A file path to which the copied transaction log is to be stored.
-m, --mode=<MODE>: Specifies the method by which the transaction log page is to be copied.
The one of following options are available: sync, semisync, async. Guaranteeing sending
and writing logs jobs is determined by selected mode.
```

Reflecting the Stored Transaction Log to the Database

Description

The syntax for the **cubrid applylogdb** utility, which reads the copied transaction log file from the specified path, analyzes it, and then reflects it to the local database server, is as follows:

Syntax

```
cubrid applylogdb [option] <database_name>@<hostname>
option:
-L, --log-path=<PATH> : The path to the transaction log file to be read.
--max-mem-size=<SIZE> : The maximum memory size available for process. The memory unit is
MB; up to 1,000 MB is allowed.
```

Note The restart condition of the applylogdb process can be configured with the **--max-mem-size** option. For information about configuring the restart condition of the CAS process according to current memory usage, see the **APPL_SERVER_MAX_SIZE** parameter in Parameter by Broker. The applylogdb process restarts when the current memory size reaches the size specified in the **--max-mem-size**, or reaches twice the size of the memory at the start of the process, whichever is larger.

HA-Related JDBC Configuration

Default Configuration

- JDK 1.6 or higher
- CUBRID 2008 R2.1 or higher
- CUBRID JDBC Driver 2008 R2.1 or higher

JDBC Connection String in HA Environment

Description

To use the HA feature in JDBC, you must additionally specify the connection information of the broker, on which failover will be performed when a broker failure occurs, to the URL string. Attributes that are specified for HA are the host information (althosts) of one or more standby brokers that will be connected when there is a failure or the interval (rctime) at which the connection (failback) to the active broker will be attempted when recovering from the failure of the active server. For details about connection configuration in JDBC, see Connection Configuration.

Syntax

```
jdbc:cubrid:<host>:<port>:<db-name>:[user-id]:[password]:?[<property> [& <property>]]

host :
hostname | ip_address

property :
althosts= <alternative_hosts> | rctime= <second> | charset= <character_set>

alternative_hosts :
<standby_broker1_host>:<port> [,<standby_broker2_host>:<port>]
```

- **althosts** : Specifies the information of a broker to connect subsequently (failover) if the connection to the default broker fails. Multiple brokers to failover can be specified, and a connection is attempted according to the order listed by **alhosts**.
- **rctime** : The interval for attempting failback to the active broker where a failure occurs. This value is used when a failover is performed to the standby broker due to the failure in the active broker. After a failure occurs, the system connects to the broker specified by **althosts** (failover), end the transaction and then attempts to connect to the existing active broker at every **rctime** (failback). If **rctime** is not specified, it is configured to 600 seconds.

Example

```
--connection URL string when user name and password omitted

URL=jdbc:CUBRID:127.0.0.1:31000:db1:::

--connection URL string when charset property specified

URL=jdbc:CUBRID:127.0.0.1:31000:db1:::?charset=utf-8

--connection URL string when a property(althosts) specified for HA
URL=jdbc:CUBRID:127.0.0.1:31000:db1:::?althosts=127.0.0.2:31000,127.0.0.3:31000
```

```
--connection URL string when properties(althosts,rctime) specified for HA
URL=jdbc:CUBRID:127.0.0.1:31000:db1:::?althosts=127.0.0.2:31000,127.0.0.3:31000&rctime=600

--connection URL string when properties(althosts,rctime, charset) specified for HA
URL=jdbc:CUBRID:127.0.0.1:31000:db1:::?althosts=127.0.0.2:31000,127.0.0.3:31000&rctime=600
&charset=utf-8
```

HA-Related System Catalog

db_ha_apply_info

A table that saves the progress status every time the **applylogdb** utility applies replication logs. This table is updated at every point the **applylogdb** utility commits, and the acculmative count of operations are stored in the *_counter column. The meaning of each column is as follows:

Column Name	Column Type	Meaning
db_name	VARCHAR(255)	Name of the database saved in the log
db_creation_time	DATETIME	Creation time of the source database for the log to be applied
copied_log_path	VARCHAR(4096)	Path to the log file to be applied
page_id	INTEGER	Page of the replication log committed in the slave database
offset	INTEGER	Offset of the replication log committed in the slave database
log_record_time	DATETIME	Timestamp included in replication log committed in the slave database, i.e. the creation time of the log
last_access_time	DATETIME	Time when applylogdb was committed in the slave database
insert_counter	BIGINT	Number of times that applylogdb was inserted
update_counter	BIGINT	Number of times that applylogdb was updated
delete_counter	BIGINT	Number of times that applylogdb was deleted
schema_counter	BIGINT	Number of times that applylogdb changed the schema
commit_counter	BIGINT	Number of times that applylogdb was committed
fail_counter	BIGINT	Number of times that applylogdb failed to be inserted/updated/deleted/committed and to change the schema
required_page_id	INTEGER	Minimum pageid that applylogdb can read
start_time	DATETIME	Time when the applylogdb process accessed the slave database
status	INTEGER	Progress status (0: IDLE, 1: BUSY)

Performance Tuning

This chapter provides information about configuring system parameters that can affect the system performance. System parameters determine overall performance and operation of the system. This chapter explains how to use configuration files for the CUBRID Manager server as well as a description of each parameter.

This chapter covers the following topics:

- Configuring the Database server
- Configuring the Broker
- Configuring the CUBRID Manager server

Database Server Configuration

Scope of Database Server Configuration

CUBRID consists of the Database Server, the Broker and the CUBRID Manager. Each component has its configuration file. The system parameter configuration file for the Database Server is **cubrid.conf** located in the **$CUBRID/conf** directory. System parameters configured in **cubrid.conf** affect overall performance and operation of the database system. Therefore, it is very important to understand the Database Server configuration.

The CUBRID Database Server has a client/server architecture. To be more specific, it is divided into a Database Server process linked to the server library and a Broker process linked to the client library. The server process manages the database storage structure and provides concurrency and transaction functionalities. The client process prepares for query execution and manages object/schema.

System parameters for the database server, which can be set in the **cubrid.conf file**, are classified into a client parameter, a server parameter and a client/server parameter according to the range to which they are applied. A client parameter is only applied to client processes such as the broker. A server parameter affects the behaviors of the server processes. A client/server parameter must be applied to both the server and the client.

Location of cubrid.conf File and How It Works

- A Database Server process refers only to the **$CUBRID/conf/cubrid.conf** file. Database-specific configurations are distinguished by sections in the **cubrid.conf** file.
- A client process (i) refers to the **$CUBRID/conf/cubrid.conf** file and then (ii) additionally refers to the **cubrid.conf** file in the current directory (**$PWD**). The configuration of the file in the current directory (**$PWD/cubrid.conf**) overwrites that of the **$CUBRID/conf/cubrid.conf** file. That is, if the same parameter configuration exists in **$PWD/cubrid.conf** and in **$CUBRID/conf/cubrid.conf**, the configuration in **$PWD/cubrid.conf** has the priority.

cubrid.conf Configuration File and Default Parameters

CUBRID consists of the Database Server, the Broker and the CUBRID Manager. The name of the configuration file for each component is as follows. These files are all located in the **$CUBRID/conf** directory.

- Database Server configuration file : **cubrid.conf**
- Broker configuration file : **cubrid_broker.conf**
- CUBRID Manager server configuration file : **cm.conf**

cubrid.conf is a configuration file that sets system parameters for the CUBRID Database Server and determines overall performance and operation of the database system. In the **cubrid.conf** file, some important parameters needed for system installation are provided, having their default values.

Database Server System Parameters

The following are Database Server system parameters that can be used in the **cubrid.conf** configuration file. For the scope of **client** and **server parameters**, see Scope of Database Server Configuration.

Parameter Name	Scope	Type	Default Value
cubrid_port_id	client parameter	int	1523
communication_histogram	client parameter	bool	no
db_hosts	client parameter	string	NULL
max_clients	server parameter	int	50
ansi_quotes	client parameter	bool	yes
block_ddl_statement	client parameter	bool	no

block_nowhere_statement	client parameter	bool	no
compat_numeric_division_scale	client/server	bool	no
intl_mbs_support	client parameter	bool	no
oracle_style_empty_string	client parameter	bool	no
only_full_group_by	client parameter	bool	no
pipes_as_concat	client parameter	bool	yes
data_buffer_pages	server parameter	int	25000
dont_reuse_heap_file	server parameter	bool	no
index_scan_oid_buffer_pages	Server Parameter	float	4.0
sort_buffer_pages	server parameter	int	16
temp_file_memory_size_in_pages	server parameter	int	4
thread_stack_size	server parameter	int	102400
garbage_collection	client parameter	bool	no
temp_file_max_size_in_pages	server parameter	int	-1
temp_volume_path	server parameter	string	NULL
unfill_factor	server parameter	float	0.1
volume_extension_path	server parameter	string	NULL
call_stack_dump_activation_list	client/server parameter	string	NULL
call_stack_dump_deactivation_list	client/server parameter	string	NULL
call_stack_dump_on_error	client/server parameter	bool	no
error_log	client/server parameter	string	cub_client.err, cub_server.err
error_log_level	client/server parameter	string	syntax
error_log_warning	client/server parameter	string	no
error_log_size	client/server parameter	int	8000000
auto_restart_server	server parameter	bool	yes
deadlock_detection_interval_in_secs	server parameter	int	1
file_lock	server parameter	bool	yes
isolation_level	client parameter	int	3
lock_escalation	server parameter	int	100000
lock_timeout_in_secs	client parameter	int	-1
lock_timeout_message_type	server parameter	int	0
background_archiving	server parameter	bool	yes
log_max_archives	server parameter	int	INT_MAX
page_flush_interval_in_msecs	server parameter	int	0
adaptive_flush_control	server parameter	bool	true
max_flush_pages_per_second	server parameter	int	10000
sync_on_nflush	server parameter	int	200
pthread_scope_process	server parameter	bool	yes
backup_volume_max_size_bytes	server parameter	int	-1

checkpoint_interval_in_mins	server parameter	int	720
checkpoint_every_npages	server parameter	int	10000
log_buffer_pages	server parameter	int	50
media_failure_support	client parameter	bool	yes
insert_execution_mode	client parameter	int	1
max_plan_cache_entries	client/server parameter	int	1000
max_query_cache_entries	server parameter	int	-1
query_cache_mode	server parameter	int	0
query_cache_size_in_pages	server parameter	int	-1
replication	server parameter	bool	no
index_scan_in_oid_order	client parameter	bool	no
single_byte_compare	server parameter	bool	no
compactdb_page_reclaim_only	server parameter	int	0
csql_history_num	client parameter	int	50
java_stored_procedure	server parameter	bool	no
async_commit	server parameter	bool	no
group_commit_interval_in_msecs	server parameter	int	0
index_unfill_factor	server parameter	float	0.20

Section by Parameter

Parameters specified in **cubrid.conf** have the following three sections:

- Used when the CUBRID service starts : [service] section
- Applied commonly to all databases : [common] section
- Applied individually to each database : [@<*database*>] section

Where <*database*> is the name of the database to which each parameter applies. If a parameter configured in [common] is the same as the one configured in [@<*database*>], the one configured in [@<*database*>] is applied.

Default Parameters

cubrid.conf, a default database configuration file created during the CUBRID installation, includes some default Database Server parameters that must be changed. You can change the value of a parameter that is not included as a default parameter by manually adding or editing one.

The following is the content of the **cubrid.conf** file.

```
# Copyright (C) 2008 Search Solution Corporation. All rights reserved by Search Solution.
#
# $Id$
#
# cubrid.conf#
# For complete information on parameters, see the CUBRID
# Database Administration Guide chapter on System Parameters
# Service section - a section for 'cubrid service' command
[service]
# The list of processes to be started automatically by 'cubrid service start' command
# Any combinations are available with server, broker and manager.
service=server,broker,manager
# The list of database servers in all by 'cubrid service start' command.
# This property is effective only when the above 'service' property contains 'server'
keyword.
#server=foo,bar
# Common section - properties for all databases
# This section will be applied before other database specific sections.
```

```
[common]
# Number of data buffer pages
# data_buffer_pages (25,000 pages) * DB page size (4KB) = 100M
data_buffer_pages=25000
# Number of sort buffer pages
# sort_buffer_pages (16 pages) * DB page size (4KB) * number of threads
sort_buffer_pages=16
# Number of log buffer pages.
# log_buffer_pages (50 pages) * DB page size (4KB) = 200KB
log_buffer_pages=50
# Maximum number of locks acquired on individual instances of a
# class before the locks on the instances are escalated to a class lock
lock_escalation=100000
# Minimal amount of time to wait for a lock (seconds).
# A negative value, indicates to wait indefinitely until the lock is
# granted or until the transaction is rolled back as a result of a deadlock.
# A value of 0, indicates not to wait for a lock.
lock_timeout_in_secs=-1
# Interval between attempts at deadlock detection (seconds).
# An approximate interval to attempt to run the deadlock detector.
deadlock_detection_interval_in_secs=1
# Checkpoint when the specified time has passed (minutes).
# Checkpoint will be done also when log has grown by specified pages.
checkpoint_interval_in_mins=720
# Transaction isolation level.
# Six levels of isolation are provided, represented by:
# "TRAN_SERIALIZABLE"
# "TRAN_REP_CLASS_REP_INSTANCE"
# "TRAN_REP_CLASS_COMMIT_INSTANCE"
# "TRAN_REP_CLASS_UNCOMMIT_INSTANCE"
# "TRAN_COMMIT_CLASS_COMMIT_INSTANCE"
# "TRAN_COMMIT_CLASS_UNCOMMIT_INSTANCE"
# For other aliases, or for more information on the levels, see the
# System Parameters chapter in the Database Administration Guide.
isolation_level="TRAN_REP_CLASS_UNCOMMIT_INSTANCE"
# TCP port id for the CUBRID programs (used by all clients).
cubrid_port_id=1523
# The maximum number of concurrent client connections the server will accept.
# This value also means the total # of concurrent transactions.
max_clients=50
# Restart the server process automatically
auto_restart_server=yes
# Become a master server for replication.
replication=no
# Enable Java Stored Procedure
java_stored_procedure=no
```

Connection-Related Parameters

The following are parameters related to the Database Server. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value	Min	Max
cubrid_port_id	int	1523	1	
db_hosts	string	NULL		
max_clients	int	50	10	1024

cubrid_port_id

cubrid_port_id is a parameter that configures the port to be used by the master process. The default value is **1,523**. If the port 1,523 is already being used on the server where CUBRID is installed or it is blocked by a firewall, an error message, which means the master server is not connected because the master process cannot be running properly, is outputted. If such port conflict occurs, the administrator must change the value of **cubrid_port_id** considering the server environment.

db_hosts

db_hosts is a parameter that specifies a list of Database Server hosts to which clients can connect, and the connection order. The server host list consists of more than one server host names, and host names are separated by spaces or colons (:). Duplicate or non-existent names are ignored.

The following is an example that shows the values of the **db_hosts** parameter. In this example, connections are attempted in the order of **host1** > **host2** > **host3**.

```
db_hosts="hosts1:hosts2:hosts3"
```

To connect to the server, the client first tries to connect to the specified server host referring to the database location file (**databases.txt**). If the connection fails, the client then tries to connect to the first one of the secondarily specified server hosts by referring to the value of the **db_hosts** parameter in the database configuration file (**cubrid.conf**).

max_clients

max_clients is a parameter that configures the maximum number of clients (usually Broker processes) which allow concurrent connections to the database server. The **max_clients** parameter refers to the number of concurrent transactions. The default value is **50**.

To grantee performance while increasing the number of concurrent users in CUBRID environment, you need to assign an approviate value to **max_clients** (**cubrid.conf**) and **MAX_NUM_APPL_SERVER** (**cubrid_broker.conf**) parameters. That is, you are required to modify the number of concurrent connections allowed by databases with the **max_clients** parameter. You should also modify the number of concurrent connections allowed by brokers with the **MAX_NUM_APPL_SERVER** parameter.

For example, in the **cubrid_broker.conf** file, the **MAX_NUM_APPL_SERVER** value of [%query_editor] is 50 and the **MAX_NUM_APPL_SERVER** value of [%BROKER1] is 50, you should specify the **max_clients** parameter value as 120 (100 multiplied by 2) in the **cubrid.conf** file so that it can have more free space.

Memory-Related Parameters

The following are parameters related to the memory used by the Database Server or client. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value	Min	Max
data_buffer_pages	int	25000	1	
index_scan_oid_buffer_pages	float	4.0	0.05	16.0
sort_buffer_pages	int	16	1	
temp_file_memory_size_in_pages	int	4	0	20
thread_stacksize	int	102400	65536	
garbage_collection	bool	no		

data_buffer_pages

data_buffer_pages is a parameter that configures the number of data pages to be cached in the memory by the Database Server. The greater the value of the **data_buffer_pages** parameter, the more data pages to be cached in the buffer, thus providing the advantage of decreased disk I/O cost. However, if this parameter is too large, the buffer pool can be swapped out by the operating system because the system memory is excessively occupied. It is recommended to configure the **data_buffer_pages** parameter in a way the required memory size is less than two-thirds of the system memory size. The default value is **25,000** pages.

- Required memory size = the number of buffer pages (**data_buffer_pages** * **page size**)
- The number of buffer pages = the value of the **data_buffer_pages** parameter
- Page size = the value of the page size specified by the **-s** option of the **cubrid createdb** utility during the database creation

index_scan_oid_buffer_pages

index_scan_oid_buffer_pages is a parameter that configures the number of buffer pages where the OID list is to be temporarily saved during the index scan. The default value is **4**. The minimum value is 0.05 and the maximum value is 16.0.

The size of the OID buffer tends to vary in proportion to the value of the **index_scan_oid_buffer_pages** parameter and the page size set when the database was created. In addition, the bigger the size of such OID buffer, the more the index scan cost. You can set the value of the **index_scan_oid_buffer_pages** by considering these factors.

sort_buffer_pages

sort_buffer_pages is a parameter that configures the number of buffer pages to be used when sorting. The default value is **16** and the minimum value is 1. The server assigns one sort buffer for each client request, and releases the assigned buffer memory when sorting is complete.

temp_file_memory_size_in_pages

temp_file_memory_size_in_pages is a parameter that configures the number of buffer pages to cache temporary result of a query. The default value is **4** and the maximum value is 20.

- Required memory size = the number of temporary memory buffer pages (**temp_file_memory_size_in_pages** * **page size**)
- The number of temporary memory buffer pages = the value of the **temp_file_memory_size_in_pages** parameter
- Page size = the value of the page size specified by the **-s** option of the **cubrid createdb** utility during the database creation

thread_stacksize

thread_stacksize is a parameter that configures the stack size of a thread. The default value is **100*1024**. The value of the **thread_stacksize** parameter must not exceed the stack size allowed by the operating system.

garbage_collection

garbage_collection is a parameter that specifies whether or not to collect garbage memory no longer used by the client. The default value is **no**.

Disk-Related Parameters

The following are disk-related parameters for defining database volumes and saving files. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value	Min	Max
temp_file_max_size_in_pages	int	-1		
temp_volume_path	string	NULL		
unfill_factor	float	0.1	0.0	0.3
volume_extension_path	string	NULL		
dont_reuse_heap_file	bool	no		

temp_file_max_size_in_pages

temp_file_max_size_in_pages is a parameter that configures the maximum number of pages to store temporary volumes in the disk, which are used for the execution of complex queries or sorting. The default value is **-1**. If this parameter is configured to the default value, unlimited number of temporary volumes are created and stored in the directory specified by the **temp_volume_path** parameter. If it is configured to 0, the administrator must create temporary volumes manually by using the **cubrid addvoldb** utility because temporary volumes are not created automatically.

temp_volume_path

temp_volume_path is a parameter that specifies the directory in which to create temporary volumes used for the execution of complex queries or sorting. The default value is the volume location configured during the database creation.

unfill_factor

unfill_factor is a parameter that defines the rate of disk space to be allocated in a heap page for data updates. The default value is **0.1**. That is, the rate of free space is configured to 10%. In principle, data in the table is inserted in physical order. However, if the size of the data increases due to updates and there is not enough space for storage in the given page, performance may degrade because updated data must be relocated to another page. To prevent such a problem, you can configure the rate of space for a heap page by using the **unfill_factor** parameter. The allowable maximum value is 0.3 (30%). In a database where data updates rarely occur, you can configure this parameter to 0.0 so that space will not be allocated in a heap page for data updates. If the value of the **unfill_factor** parameter is negative or greater than the maximum value, the default value (**0.1**) is used.

volume_extension_path

volume_extension_path is a parameter that specifies the directory where automatically extended volumes are to be created. The default value is the volume location configured during the database creation.

dont_reuse_heap_file

The parameter "**dont_reuse_heap_file**" specifies whether or not heap files, which are deleted when deleting the table (DROP TABLE), are to be reused when creating a new table (CREATE TABLE). If this parameter is set to 0, the deleted heap files can be reused; if it is set to 1, the deleted heap files are not used when creating a new table. The default value is 0.

Error Message-Related Parameters

The following are parameters related to processing error messages recorded by CUBRID. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value
call_stack_dump_activation_list	string	NULL
call_stack_dump_deactivation_list	string	NULL
call_stack_dump_on_error	bool	no
error_log	string	cub_client.err, cub_server.err
error_log_level	string	syntax
error_log_warning	string	no
error_log_size	string	8000000

call_stack_dump_activation_list

call_stack_dump_activation_list is a parameter that specifies a certain error number for which a call stack is to be dumped as an exception even when you configure that a call stack will not be dumped for any errors. Therefore, the **call_stack_dump_activation_list** parameter is effective only when **call_stack_dump_on_error=no**. The following is an example that configures the parameter so that call stacks will not be dumped for any errors, except for the ones whose numbers are -115 and -116.

```
call_stack_dump_on_error= no
call_stack_dump_activation_list=-115,-116
```

call_stack_dump_deactivation_list

call_stack_dump_deactivation_list is a parameter that specifies a certain error number for which a call stack is not to be dumped when you configure that a call stack will be dumped for any errors. Therefore, the **call_stack_dump_deactivation_list** parameter is effective only when **call_stack_dump_on_error=yes**. The following is an example that configures the parameter so that call stacks will be dumped for any errors, except for the ones whose numbers are -115 and -116.

```
call_stack_dump_on_error= yes
call_stack_dump_deactivation_list=-115,-116
```

call_stack_dump_on_error

call_stack_dump_on_error is a parameter that determines whether or not to dump a call stack when an error occurs in the Database Server. If this parameter is configured to no, a call stack for any errors is not dumped. If it is configured to yes, a call stack for all errors is dumped. The default value is **no**.

error_log

error_log is a server/client parameter that specifies the name of the error log file when an error occurs in the database server. The name of the error log file must be in the form of <*database_name*>_<*date*>_<*time*>**.err**. However, the naming rule of the error log file does not apply to errors for which the system cannot find the Database Server information. Therefore, error logs are recorded in the **cubrid.err** file. The error log file **cubrid.err** is stored in the **$CUBRID/log/server** directory.

error_log_level

error_log_level is a server parameter that specifies a error message to be stored based on severity. There are five different levels which ranges from **NOTIFICATION** (lowest level), **WARNING**, **SYNTAX**, **ERROR**, and **SYNTAX** (highest level). An error message with **SYNTAX**, **ERROR**, and FATAL levels are stored in the log file if severity of error is **SYNTAX**, default value.

error_log_warning

The server parameter **error_log_warning** specifies whether or not error messages with a severity level of **WARNING** are to be displayed. Its default value is no. Therefore, only error messages with levels other than **WARNING** will be saved even when it is set to **error_log_level = NOTIFICATION**. For this reason, you must set **error_log_warning = yes** to save WARNING messages to an error log file.

error_log_size

error_log_size is a parameter that specifies the maximum number of lines per an error log file. The default value is **8,000,000**. If it reaches up the specified number, the <*database_name*>_<*date*>_<*time*>.err.bak file is created.

Concurrency/Lock Parameters

The following are parameters related to concurrency control and locks of the Database Server. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value	Min	Max
deadlock_detection_interval_in_secs	int	1	1	
isolation_level	int	3	1	6
lock_escalation	int	100000	5	
lock_timeout_in_secs	int	-1	-1	
lock_timeout_message_type	int	0	0	2

deadlock_detection_interval_in_secs

deadlock_detection_interval_in_secs is a parameter that configures the interval (in seconds) in which deadlocks are detected for stopped transactions. If a deadlock occurs, CUBRID resolves the problem by rolling back one of the transactions. The default value is 1 second. Note that deadlocks cannot be detected if the detection interval is too long.

isolation_level

isolation_level is a parameter that configures the isolation level of a transaction. The higher the isolation level, the less concurrency and the less interruption by other concurrent transactions. The **isolation_level** parameter can be configured to an integer value from 1 to 6, which represent isolation levels, or character strings. The default value is **TRAN_REP_CLASS_UNCOMMIT_INSTANCE**. For details about each isolation level and parameter values, see Setting Isolation Level and the following table.

Isolation Level	isolation_level Parameter Value
SERIALIZABLE	"TRAN_SERIALIZABLE" or 6
REPEATABLE READ CLASS with REPEATABLE READ INSTANCES	"TRAN_REP_CLASS_REP_INSTANCE" or "TRAN_REP_READ" or 5
REPEATABLE READ CLASS with READ COMMITTED INSTANCES(or CURSOR STABILITY)	"TRAN_REP_CLASS_COMMIT_INSTANCE" or "TRAN_READ_COMMITTED" or "TRAN_CURSOR_STABILITY" or 4
REPEATABLE READ CLASS with READ UNCOMMITTED INSTANCES	"TRAN_REP_CLASS_UNCOMMIT_INSTANCE" or "TRAN_READ_UNCOMMITTED" or 3
READ COMMITTED CLASS with READ COMMITTED INSTANCES	"TRAN_COMMIT_CLASS_COMMIT_INSTANCE" or 2
READ COMMITTED CLASS with READ UNCOMMITTED INSTANCES	"TRAN_COMMIT_CLASS_UNCOMMIT_INSTANCE" or 1

- **TRAN_SERIALIZABLE** : This isolation level ensures the highest level of consistency. For more information, see SERIALIZABLE.
- **TRAN_REP_CLASS_REP_INSTANCE** : This isolation level can occur phantom read. For more information, see REPEATABLE READ CLASS with REPEATABLE READ INSTANCES.
- **TRAN_REP_CLASS_COMMIT_INSTANCE** : This isolation level can occur unrepeatable read. For more information, see REPEATABLE READ CLASS with READ COMMITTED INSTANCES.
- **TRAN_REP_CLASS_UNCOMMIT_INSTANCE** : This isolation level can occur dirty read. For more information, see REPEATABLE READ CLASS with READ UNCOMMITTED INSTANCES.
- **TRAN_COMMIT_CLASS_COMMIT_INSTANCE** : This isolation level can occur unrepeatable read. It allows modification of table schema by current transactions while data is being retrieved. For more information, see READ COMMITTED CLASS with READ COMMITTED INSTANCES.
- **TRAN_COMMIT_CLASS_UNCOMMIT_INSTANCE** : This isolation level can occur dirty read. It allows modification of table schema by current transactions while data is being retrieved. For more information, see READ COMMITTED CLASS with READ UNCOMMITTED INSTANCES.

lock_escalation

lock_escalation is a parameter that specifies the maximum number of locks permitted before row level locking is extended to table level locking. The default value is **100,000**. If the value of the **lock_escalation** parameter is small, the overhead by memory lock management is small as well; however, the concurrency decreases. On the other hand, if the configured value is large, the overhead is large as well; however, the concurrency increases.

lock_timeout_in_secs

lock_timeout_in_secs is a client parameter that configures the lock waiting time. If the lock is not permitted within the specified time period, the given transaction is canceled, and an error message is returned. If the parameter is configured to **-1**, which is the default value, the waiting time is infinite until the lock is permitted. If it is configured to 0, there is no waiting for locks.

lock_timeout_message_type

lock_timeout_message_type is a parameter that configures the level of information that is to be included in the message returned when a lock timeout occurs. If the parameter is configured to **0**, which is the default value, the information about lock ownership is not included in the message. If it is configured to 1, single lock ownership information is included. If it is configured to 2, all information about lock ownership is included.

- If **lock_timeout_message_type** = 0

```
ERROR: Your transaction (index 3, cub_user@cdbs006.cub|15668) timed out waiting
on    X_LOCK lock on instance 0|636|34 of class participant. You are waiting for
user(s)  to finish.
```

- If **lock_timeout_message_type** = 1

```
ERROR : Your transaction (index 3, cub_user@cdbs006.cub|15668) timed out waiting
on    X_LOCK lock on instance 0|636|34 of class participant. You are waiting for user(s)
cub_user@cdbs006.cub|15615 to finish.
```

- If **lock_timeout_message_type** = 2

```
ERROR: Your transaction (index 3, cub_user@cdbs006.cub|15668) timed out waiting
on    X_LOCK lock on instance 0|636|34 of class participant. You are waiting for user(s)
cub_user@cdbs006.cub|15615, cub_user@cdbs006.cub|15596 to finish.
```

Logging-Related Parameters

The following are parameters related to logs used for database backup and restore. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value	Min	Max
log_buffer_pages	int	50	3	
media_failure_support	bool	yes		
log_max_archives	int	INT_MAX	0	
background_archiving	bool	yes		
page_flush_interval_in_msecs	int	0	-1	
checkpoint_interval_in_mins	int	720	1	
checkpoint_every_npages	int	10000	10	
adaptive_flush_control	bool	true		
max_flush_pages_per_second	int	10000	1	INT_MAX
sync_on_nflush	int	200	1	INT_MAX

log_buffer_pages

log_buffer_pages is a parameter that configures the number of log buffer pages to be cached in the memory. The default value is **50**. If the value of the **log_buffer_pages** parameter is big, performance can be improved (due to the decrease in disk I/O) when transactions are long and numerous. It is recommended to configure an appropriate value considering the memory size and operations of the system where CUBRID is installed.

- Required memory size = the number of log buffer pages (**log_buffer_pages**) * database page size (**database_page_size**)
- The number of log buffer pages = the value of the **log_buffer_pages** parameter

- Database page size = the value of the page size specified by the **-s** option of the **cubrid createdb** utility during the database creation

media_failure_support

media_failure_support is a parameter that specifies whether or not to store archive logs in case of storage media failure. If the parameter is configured to **yes**, which is the default value, all active logs are copied to archive logs when the active logs are full and the transaction is active. If it is configured to no, archive logs created after the active logs are full are deleted automatically. Note that archive logs are deleted automatically if the value of the parameter is configured to no.

Note If you specify this parameter to no, the backgroud_archiving parameter is deactivated, accordingly.

log_max_archives

log_max_archives is a parameter that sets the maximum number of archive log files to record if **media_failure_support** is set to yes. The minimum value is set to zero, and the default is INT_MAX. For example, when **log_max_archives**=3 in cubrid.conf, the most recent three archive log files are recorded. If a fourth archiving log file is generated, the oldest archive log file is automatically deleted. The information about the deleted archive log is recorded into the *_login file.
However, if an active transaction still refers to an existing archive log, the archive log will not be deleted. That is, if a transaction starts at the point that the first archive log is generated, and it is still active until the fifth archive log is generated, the first archive log cannot be deleted.

background_archiving

background_archiving is a parameter that generate a temporary archive log periodically at a specific time if **media_failure_support** is set to yes. This is useful when balancing disk I/O load due to the archive log process. The default is **yes**.

checkpoint_interval_in_mins

checkpoint_interval_in_mins is a parameter that sets cycle (in minutes) for checkpoint to be executed. The default value is **720.**

Checkpoint flushes log files(dirty page) remained in data buffers to a disk. It can restore data back to the latest checkpoint if failure happens. If high volume of log files are stored in a disk due to checkpoint, it may cause disk I/O. Therefore, you should set the checkpoint cycle properly to prevent database operation failure.

The **checkpoint_interval_in_mins** and **checkpoint_every_npages** parameters are related to setting checkpoint cycle. The checkpoint is periodically executed whenever the time specified in **checkpoint_interval_in_mins** parameter has elapsed or the number of log pages specified in **checkpoint_every_npages** parameter has reached.

checkpoint_every_npages

checkpoint_every_npages is a parameter that sets checkpoint cycle by log page. The default value is **10,000**. You can distribute disk I/O overload at the checkpoint by specifying lower number in the **checkpoint_every_npages** parameter, especially in the environment where **INSERT**/**UPDATE** are heavily loaded at a specific time.

page_flush_interval_in_msecs

The parameter **page_flush_interval_in_msecs** specifies the interval in milliseconds (msec) at which dirty pages in a data buffer are flushed to a disk. Its default value is 0. If this parameter is set to -1 (the minimum value), dirty pages are flushed to the disk only at the checkpoint, or when pages are swapped.

This is a parameter that is related to I/O load and buffer concurrency. For this reason, you must set its value in consideration of the workload of the service environment.

adaptive_flush_control

The parameter **adaptive_flush_control** automatically adjusts the flush capacity at every 50 ms depending on the current status of the flushing operation. Its default value is **yes**. That is, this capacity is increased if a large number of **INSERT** or **UPDATE** operations are concentrated at a certain point of time and the number of flushed pages reaches the **max_flush_pages_per_second** parameter value; and is decreased otherwise. In the same way, you can distribute the I/O load by adjusting the flush capacity on a regular basis depending on the workload.

max_flush_pages_per_second

The parameter **max_flush_pages_per_second** specifies the maximum flush capacity when the flushing operation is performed from a buffer to a disk. Its default value is 10,000. That is, you can prevent concentration of I/O load at a certain point of time by setting this parameter to control the maximum flush capacity per second.

If a large number of **INSERT** or **UPDATE** operations are concentrated at a certain point of time, and the flush capacity reaches the maximum capacity set by this parameter, only log pages are flushed to the disk, and data pages are no longer flushed. Therefore, you must set an appropriate value for this parameter considering the workload of the service environment.

sync_on_nflush

The parameter **sync_on_nflush** sets the interval in pages between after data and log pages are flushed from buffer and before they are synchronized with FILE I/O of operating system. Its default value is 200. That is, the CUBRID Server performs synchronization with the FILE I/O of the operating system whenever 200 pages have been flushed. This is also a parameter related to I/O load.

Transaction Processing-Related Parameters

The following are parameters for improving transaction commit performance. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value	Min	Max
async_commit	bool	no		
group_commit_interval_in_msecs	int	0	0	

async_commit

async_commit is a parameter that activates the asynchronous commit functionality. If the parameter is configured to **no**, which is the default value, the asynchronous commit is not performed; if it is configured to yes, the asynchronous commit is executed. The asynchronous commit is a functionality that improves commit performance by completing the commit for the client before commit logs are flushed on the disk and having the log flush thread (LFT) perform log flushing in the background. Note that already committed transactions cannot be restored if a failure occurs on the Database Server before log flushing is performed.

group_commit_interval_in_msecs

group_commit_interval_in_msecs is a parameter that configures the interval (in milliseconds), at which the group commit is to be performed. If the parameter is configured to **0**, which is the default value, the group commit is not performed. The group commit is a functionality that improves commit performance by combining multiple commits that occurred in the specified time period into a group so that commit logs are flushed on the disk at the same time.

Statement/Type-Related Parameters

The following are parameters related to SQL statements and data types supported by CUBRID. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value
ansi_quotes	bool	yes
block_ddl_statement	bool	no
block_nowhere_statement	bool	no
compat_numeric_division_scale	bool	no
intl_mbs_support	bool	no
oracle_style_empty_string	bool	no
only_full_group_by	bool	no
pipes_as_concat	bool	yes

ansi_quotes

ansi_quotes is a parameter that enclose symbols and character string to handle identifiers. The default value is **yes**. If this parameter value is set to **yes**, double quotations are handled as identifier symbols and single quotations are handled as character string symbols. If it is set to **no**, double quotations are handled as character string symbols.

block_ddl_statement

block_ddl_statement is a parameter that restricts the execution of DDL (Data Definition Language) statements by the client. If the parameter is configured to no, the given client is allowed to execute DDL statements. If it is configured to yes, the client is not permitted to execute DDL statements. The default value is **no**.

block_nowhere_statement

block_nowhere_statement is a parameter that restricts the execution of **UPDATE**/**DELETE** statements without a condition clause (**WHERE**) by the client. If the parameter is configured to no, the given client is allowed to execute **UPDATE**/**DELETE** statements without a condition clause. If it is configured to yes, the client is not permitted to execute **UPDATE**/**DELETE** statements without a condition clause. The default value is **no**.

compact_numeric_division_scale

compat_numeric_division_scale is a parameter that configures the scale to be displayed in the result (quotient) of a division operation. If the parameter is configured to **no**, the scale of the quotient is 9 if it is configured to **yes**, the scale is determined by that of the operand. The default value is **no**.

intl_mbs_support

intl_mbs_support is a parameter that specifies whether or not to support multi-byte character set. If the parameter is configured to **no**, a multi-byte character set is not allowed if it is configured to yes, a multi-byte character set is allowed. To improve performance, it is recommended to configure the **intl_mbs_support** parameter to **no** and use alphabets for table and column names because operation cost for supporting multi-byte character set is high.

oracle_style_empty_string

oracle_style_empty_string is a parameter that improves compatibility with other DBMS (Database Management Systems) and specifies whether or not to process empty strings as **NULL** as in Oracle DBMS. If the **oracle_style_empty_string** parameter is configured to **no**, the character string is processed as a valid string if it is configured to **yes**, the empty string is processed as **NULL**.

only_full_group_by

only_full_group_by is a parameter that specifies whether extended syntax about using **GROUP BY** statement is used or not.

If this parameter value is set to **no**, an extended syntax is applied thus, a column that is not specified in the **GROUP BY** statement can be specified in the **SELECT** column list. If it is set to **yes**, a column that is only specified in the **GROUP BY** statement can be the **SELECT** column list.

The default value is no. Therefore, specify the **only_full_group_by** parameter value to **yes** to execute queries by SQL standards. Because the extended syntax is not applied in this case, an error below is displayed.

```
ERROR: Attributes exposed in aggregate queries must also appear in the group by clause.
```

pipes_as_concat

pipes_as_concat is a parameter about using a double pipe symbol. The default value is **yes**. If this parameter value is set to **yes**, a double pipe symbol is handled as a concatenation operator if no, it is handled as the **OR** operator.

Query Cache-Related Parameters

The following are parameters related to the query cache functionality that provides execution results cached for the same **SELECT** statement. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value	Min	Max
max_plan_cache_entries	int	1,000		
max_query_cache_entries	int	-1		
query_cache_mode	int	0	0	2
query_cache_size_in_pages	int	-1		

max_plan_cache_entries

max_plan_cache_entries is a parameter that configures the maximum number of query plans to be cached in the memory. If the **max_plan_cache_entries** parameter is configured to -1 or 0, generated query plans are not stored in the memory cache; if it is configured to an integer value equal to or greater than 1, a specified number of query plans are cached in the memory. Also, the value of this parameter must be configured to an integer value equal to or greater than 1 to use the query cache functionality that caches the results of the same query.

max_query_cache_entries

max_query_cache_entries is a parameter that configures the maximum number of query results to be cached. If the parameter is configured to -1 or 0, the query cache functionality is deactivated; if it is configured to an integer value equal to or greater than 1, the execution results of a specified number of queries are cached. With the query cache functionality, you can expect performance improvement in cases where query data does not change, and the same query is entered repeatedly. Note that the query cache functionality is activated only when the **max_plan_cache_entries** parameter, which activates the query plan cache functionality, is configured to an integer value equal to or greater than 1 because the query cache functionality is dependent of the query plan cache functionality.

query_cache_mode

query_cache_mode is a parameter that specifies one of two query cache modes. In the primary query cache mode, all queries are cached. In the second query cache mode, the query with the hint /*+QUERY_CACHE(1) */ is only cached. If this parameter is configured to **0**, which is the default value, the query cache functionality is deactivated. If it is configured to 1, the functionality is executed in the primary query cache mode. If it is configured to 2, it is executed in the secondary query cache mode. To activate the query cache functionality, configure **max_plan_cache_entries**, **max_query_cache_entries** and **query_cache_mode** parameters equal to or greater than 1 respectively. Note that the query cache functionality is deactivated if any of these parameters does not satisfy the condition.

```
// The following is an example of caching up to 1,000 for query plans, caching up to 100
for query results.
max_plan_cache_entries=1000
max_query_cache_entries=100
```

```
query_cache_mode=1
// The configured values for the two parameters are invalid because the plan cache
functionality is deactivated.
max_plan_cache_entries=-1
max_query_cache_entries=100
query_cache_mode=1
// The plan cache functionality is executed for up to 1,000 query plans, and the query
cache functionality is deactivated.
max_plan_cache_entries=1000
max_query_cache_entries=100
query_cache_mode=0
```

query_cache_size_in_pages

query_cache_size_in_pages is a parameter that specifies the number of pages of query results to be cached. A query is cached only when its results are within the specified page size. If the parameter is configured to **-1**, which is the default value, the query cache functionality is executed for all queries without any constraints for the size of the result page.

Utility-Related Parameters

The following are parameters related to utilities used in CUBRID. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value	Min	Max
compactdb_page_reclaim_only	int	0		
csql_history_num	int	50	1	200
communication_histogram	string	no		
backup_volume_max_size_bytes	int	-1	1024*32	

compactdb_page_reclaim_only

compactdb_page_reclaim_only is a parameter related to the **compactdb** utility, which compacts the storage of already deleted objects to reuse OIDs of the already assigned storage. Storage optimization with the **compactdb** utility can be divided into three steps. The optimization steps can be selected through the **compactdb_page_reclaim_only** parameter. If the parameter is configured to **0**, which is the default value, step 1, 2 and 3 are all performed, so the storage is optimized in data, table and file units. If it is configured to 1, step 1 is skipped to have the storage optimized in table and file units. If it is configured to 2, steps 1 and 2 are skipped to have the storage optimized only in file units.

- Step 1 : Optimizes the storage only in data units.
- Step 2 : Optimizes the storage in table units.
- Step 3 : Optimizes the storage in file (heap file) units.

csql_history_num

csql_history_num is a parameter related to the CSQL Interpreter, and configures the number of SQL statements to be stored in the history of the CSQL Interpreter. The default value is **50**.

communication_histogram

communication_histogram is a parameter related to the **cubrid statdump** utility. It is related to Session Commands **;.h** of the CSQL Interpreter and the default value is **no**. For more information, see Outputting Statistics Information of Server.

backup_volume_max_size_bytes

backup_volume_max_size_bytes is a parameter that configures the maximum size of the backup volume created by the **cubrid backupdb** utility in byte units. If the parameter is configured to **-1**, which is the default value, there is no limit to the size of the backup volume to be created. If it is not configured, the size of the backup volume is allowed up to the size limit of the storage media.

Other Parameters

The following are other parameters. The type and value range for each parameter are as follows:

Parameter Name	Type	Default Value	Min	Max
service	string			
server	string			
replication	bool	no		
index_scan_in_oid_order	bool	no		
single_byte_compare	bool	no		
insert_execution_mode	int	1	1	7
java_stored_procedure	bool	no		
pthread_scope_process	bool	yes		
auto_restart_server	bool	yes		
index_unfill_factor	float	0.20	0	0.35

service

service is a parameter that registers a process that starts automatically when the CUBRID service starts. There are three types of processes: **server**, **broker** and **manager**. All three processes are usually registered as in **service=server,broker,manager**.

- If the parameter is configured to **server**, the database process specified by the **@server** parameter gets started.
- If the parameter is configured to **broker**, the Broker process gets started.
- If the parameter is configured to **manager**, the manager process gets started.

server

server is a parameter that registers a Database Server process that starts automatically when the CUBRID service starts.

replication

replication is a parameter that activates the database replication feature. If the parameter is configured to **no**, which is the default value, the replication feature is deactivated; if it is configured to yes, the replication feature is activated. When the replication feature is activated, the given database acts as a replication master server that creates replication logs.

index_scan_in_oid_order

index_scan_in_oid_order is a parameter that configures the result data to be retrieved in OID order after the index scan. If the parameter is configured to **no**, which is the default value, results are retrieved in data order; if it is configured to yes, they are retrieved in OID order.

single_byte_compare

single_byte_compare is a parameter that determines whether or not to compare strings in single byte units. If the parameter is configured to **no**, which is the default value, strings are compared in two byte units; if it is configured to yes, they are compared in single byte units. That is, you can retrieve/compare strings on data stored as UTF-8.

insert_execution_mode

insert_execution_mode has execution modes ranging from 1 to 7. Queries are usually executed on the server according to the query plan created by the client, but this parameter is used to directly insert queries on the server side. A selected execution mode is executed directly on the server, and other execution modes are executed on the client. This parameter can be used to perform an INSERT operation to the server in an environment in which dirty reading of INSERTed data is required, or in which the memory capacity of the client is limited.

The following are three types of **INSERT** statements for execution modes. This parameter can be set through a combination of integer values corresponding to each execution mode.

- **INSERT_SELECT** : When using the **SELECT** statement in the **INSERT** statement.

```
INSERT INTO code2(s_name, f_name) SELECT s_name, f_name from code;
```

- **INSERT_VALUES** : The common **INSERT** statement.

```
INSERT INTO code2(s_name, f_name) VALUES ('S', 'Silver');
```

- **INSERT_DEFAULT** : When inserting the default value because a column with the default value is omitted in the **INSERT** statement.

```
CREATE TABLE code2(s_name char(1) DEFAULT '_', f_name varchar(40));
```

```
INSERT INTO code2(f_name) DEFAULT VALUES;
```

- **INSERT_REPLACE** : For example, when the **REPLACE** statement is executed, the corresponding integer value is 8.

```
CREATE TABLE code2(s_name char(1) NOT NULL UNIQUE, f_name varchar(40));
REPLACE INTO code2 VALUES ('S', 'Silver');
```

- **INSERT_ON_DUP_KEY_UPDATE** : In addition, when the **ON DUPLICATE KEY UPDATE** clause is specified in the **INSERT** statement, the corresponding integer value is 16.

```
CREATE TABLE code2(s_name char(1) NOT NULL UNIQUE, f_name varchar(40));
INSERT INTO code2 VALUES ('S', 'Silver') ON DUPLICATE KEY UPDATE f_name='Silver';
```

The sum of the execution mode values above is the execution mode to be configured.

- Example 1 : If you want to execute **INSERT_SELECT** and **INSERT_VALUES** on the server, the **insert_execution_mode** is 3. (1 + 2 = 3)
- Example 2 : If you want to execute **INSERT_SELECT**, **INSERT_DEFAULT**, **INSERT_REPLACE**, an **INSERT_ON_DUP_KEY_UPDATE** on the server, the **insert_execution_mode** is 29(1+4+8+16=29).

java_stored_procedure

java_stored_procedure is a parameter that determines whether or not to use Java stored procedures by running the Java Virtual Machine (JVM). If the parameter is configured to **no**, which is the default value, JVM is not executed; if it is configured to yes, JVM is executed so you can use Java stored procedures. Therefore, configure the parameter to yes if you plan to use Java stored procedures.

pthread_scope_process

pthread_scope_process is a parameter that configures the contention scope of threads. It only applies to AIX systems. If the parameter is configured to no, the contention scope becomes **PTHREAD_SCOPE_SYSTEM**; if it is configured to yes, it becomes **PTHREAD_SCOPE_PROCESS**. The default value is **yes**.

auto_restart_server

auto_restart_server is a parameter that specifies whether or not to restart the process when it stops due to a fatal error in the Database Server process. If **auto_restart_server** is configured to yes, the server process restarts automatically when it stopped due to abnormal causes other than the normal stop process (**STOP** command of the CUBRID Server).

index_unfill_factor

If there is no free space because index pages are full when the **INSERT** or **UPDATE** operation is executed after the first index is created, the split of index page nodes occurs. This substantially affects the performance by increasing the operation time. **index_unfill_factor** is a parameter that specifies the percent of free space defined for each index page node when an index is created. The **index_unfill_factor** value is applied only when an index is created for the first time. The percent of free space defined for the page is not maintained dynamically. Its value ranges between 0 and 0.35. The default value is **0.20**.

If an index is created without any free space for the index page node (**index_unfill_factor**=0), the split of index page nodes occurs every time an additional insertion is made. This may degrade the performance.

If the value of **index_unfill_factor** is large, a large amount of free space is available when an index is created. Therefore, better performance can be obtained because the split of index nodes does not occur for a relatively long period of time until the free space for the nodes is filled after the first index is created.

If this value is small, the amount of free space for the nodes is small when an index is created. Therefore, it is likely that the index nodes are splitted by **INSERT** or **UPDATE** because the free space for the index nodes is filled in a short period of time.

Changing Database Server Configuration

Editing the Configuration File

You can add/delete parameters or change parameter values by manually editing the database configuration file (**cubrid.conf**) in the **$CUBRID/conf** directory.

The following parameter syntax rules are applied when configuring parameters in the configuration file:

- Parameter names are not case-sensitive.
- The name and value of a parameter must be entered in the same line.
- An equal sign (=) can be used to configure the parameter value. Spaces are allowed before and after the equal sign.
- If the value of a parameter is a character string, enter the character string without quotes. However, use quotes if spaces are included in the character string.

Using SQL Statements

Description

You can configure a parameter value by using SQL statements in the CSQL Interpreter or CUBRID Manager's Query Editor. Note that only client parameters can be updated.

Syntax

```
SET SYSTEM PARAMETERS 'parameter_name=value [{; name=value}...]'
```

parameter_name is the name of a client parameter whose value is editable. In this syntax, value is the value of the given parameter. You can change multiple parameter values by separating them with semicolons (;).

Example

The following is an example of retrieving the result of an index scan in OID order and configuring the number of queries to be saved in the history of the CSQL Interpreter to 70.

```
SET SYSTEM PARAMETERS 'index_scan_in_oid_order=1; csql_history_num=70'
```

Using Session Commands of the CSQL Interpreter

Description

You can configure database parameter values by using session commands (;**SEt**) in the CSQL Interpreter. Note that only client parameters can be updated.

Example

The following is an example of configureing the **block_ddl_statement** parameter to 1 so that execution of DDL statements is not allowed.

```
csql> ;se block_ddl_statement=1
=== Set Param Input ===
block_ddl_statement=1
```

Broker Configuration

cubrid_broker.conf Configuration File and Default Parameters

Broker System Parameters

The following are Broker parameters that can be used in the **cubrid_broker.conf** configuration file. For description of each parameter, see **Parameter Description** in Parameter by Broker.

Parameter Name	Type	Default Value
MASTER_SHM_ID	int	30001
ADMIN_LOG_FILE	string	log/broker/cubrid_broker.log
SERVICE	string	ON
BROKER_PORT	int	30000 (MAX : 65535)
MIN_NUM_APPL_SERVER	int	5
MAX_NUM_APPL_SERVER	int	40
APPL_SERVER_SHM_ID	int	30000
APPL_SERVER_MAX_SIZE	int	For Windows : 40 (32 bit), 80 (64 bit) For Linux : 0
LOG_DIR	string	log/broker/sql_log
ERROR_LOG_DIR	string	log/broker/error_log
SQL_LOG	string	ON
TIME_TO_KILL	int	120
SESSION_TIMEOUT	int	300
KEEP_CONNECTION	string	AUTO
ACCESS_LIST	string	-
ACCESS_LOG	string	ON
APPL_SERVER_PORT	int	BROKER_PORT+1
APPL_SERVER	string	CAS
LOG_BACKUP	string	OFF
SQL_LOG_MAX_SIZE	int	100000
MAX_STRING_LENGTH	int	-1
SOURCE_ENV	string	cubrid.env
STATEMENT_POOLING	string	ON
CCI_PCONNECT	string	OFF
SELECT_AUTO_COMMIT	string	OFF
LONG_QUERY_TIME	int	60
LONG_TRANSACTION_TIME	int	60
ACCESS_MODE	string	RW

Default Parameters

cubrid_broker.conf, a default broker configuration file created during the CUBRID installation, includes some default Broker parameters that must be changed. You can change the value of a parameter that is not included as a default parameter by manually adding or editing one.

The following is the content of the **cubrid_broker.conf** file that is created by default during the installation.

```
[broker]
MASTER_SHM_ID           =30001
ADMIN_LOG_FILE          =log/broker/cubrid_broker.log

[%query_editor]
SERVICE                 =ON
BROKER_PORT             =30000
MIN_NUM_APPL_SERVER     =5
MAX_NUM_APPL_SERVER     =40
APPL_SERVER_SHM_ID      =30000
LOG_DIR                 =log/broker/sql_log
ERROR_LOG_DIR           =log/broker/error_log
SQL_LOG                 =ON
TIME_TO_KILL            =120
SESSION_TIMEOUT         =300
KEEP_CONNECTION         =AUTO

[%BROKER1]
SERVICE                 =ON
BROKER_PORT             =33000
MIN_NUM_APPL_SERVER     =5
MAX_NUM_APPL_SERVER     =40
APPL_SERVER_SHM_ID      =33000
LOG_DIR                 =log/broker/sql_log
ERROR_LOG_DIR           =log/broker/error_log
SQL_LOG                 =ON
TIME_TO_KILL            =120
SESSION_TIMEOUT         =300
KEEP_CONNECTION         =AUTO
```

Environment Variables related to the Broker Configuration File

You can specify the **cubrid_broker.conf** file location by using the CUBRID_BROKER_CONF_FILE variable to executing various Brokers with different configuration.

Common Parameters

The following are parameters commonly applied to all Brokers, and they are listed under [broker] section in the **cubrid_broker.conf** file.

MASTER_SHM_ID

MASTER_SHM_ID is a parameter that specifies the identifier of shared memory which is used to manage the CUBRID Broker. Its value must be unique in the system. The default value is **30001**.

ADMIN_LOG_FILE

ADMIN_LOG_FILE is a parameter that specifies the file where the time information related with the CUBRID Broker running is stored. The default value is **log/broker/cubrid_broker.log** file.

Parameter by Broker

The following describes parameters to configure the environment variables of Brokers; each parameter is located under [%*broker_name*].

SERVICE

SERVICE is a parameter that determines whether to run the given Broker. It can be configured to either **ON** or **OFF**. The default value is **ON**. The Broker can run only when this parameter is configured to **ON**.

BROKER_PORT

BROKER_PORT is a parameter that configures the port number of the given Broker. Its value must be unique in the system and equal to or smaller than 65,535. By default, the broker port for **query_editor** is configured to **30000**, and the port for broker1 is configured to **33000**.

MIN_NUM_APPL_SERVER

MIN_NUM_APPL_SERVER is a parameter that configures the minimum number of application servers (CAS) even if any request to connect the broker has not been made. The default value is **5**.

MAX_NUM_APPL_SERVER

MAX_NUM_APPL_SERVER is a parameter that configures the maximum number of application servers (CAS). The default value is **40**. In an evnironment where connection pool is maintained by using a middleware such as WAS, you must specify the value of **MAX_NUM_APPL_SERVER** parameter as same as that of connection pool.

APPL_SERVER_SHM_ID

APPL_SERVER_SHM_ID is a parameter that configures the shared memory ID to be used by application servers (CAS). Its value must be unique in the system. The default value is the same as the port of the given Broker.

APPL_SERVER_MAX_SIZE

APPL_SERVER_MAX_SIZE is a parameter that specifies the maximum size of the process memory usage provided by the application server (CAS). The unit is MB. This value should be configured in the consideration of server operation environment because it affects the policy, CAS restart, in force. Especially, if you configure this value too low, applications can frequently be restarted. Note that the default value for Windows and Linux is different.
For Windows, the 32-bit CUBRID has 40 (MB) for the **APPL_SERVER_MAX_SIZE** value by default; 64-bit CUBRID has 80 (MB). If the current process memory usage exceeds the value of **APPL_SERVER_MAX_SIZE**, the Broker restarts the application server. For Linux, 0 is the default value for **APPL_SERVER_MAX_SIZE**; and it restarts the application server in the following conditions:

- Zero or negative : In case the current process is twice as large as the initial memory
- Positive : In case a value exceeds the number specified in **APPL_SERVER_MAX_SIZE**

LOG_DIR

LOG_DIR is a parameter that specifies the directory where SQL logs are stored. The default value is **log/broker/sql_log**. The file name of the SQL logs is *broker_name_id*.*sql*.**log**.

ERROR_LOG_DIR

ERROR_LOG_DIR is a parameter that specifies the directory where error logs for the Broker are stored. The default value is **log/broker/error_log**. The name of the error log file for the Broker is *broker_ name_id*.**err**.

SQL_LOG

SQL_LOG is a parameter that determines whether to leave logs for SQL statements processed by the application server (CAS) when an application server handles requests from a client. The default value is **ON**. When this parameter is configured to **ON**, all logs are stored. Log file name becomes *broker_name_id*.*sq*l.**log**. The file is created in the **log/broker/sql_log** directory under the installation directory. The parameter values are as follows:

- **OFF** : Does not leave any logs

- **ERROR** : Leaves logs for queries which occur an error. only queries where an error occurs
- **NOTICE** : Leaves logs for the long-duration execution queries which exceeds the configured time/transaction, or leaves logs for queries which occur an error
- **TIMEOUT** : Leaves logs for the long-duration execution queries which exceeds the configured time/transaction
- **ON/ALL** : Leaves all logs

TIME_TO_KILL

TIME_TO_KILL is a parameter that configures the time to remove application servers (CAS) in idle state among application servers added dynamically. The default value is **120** (sec). An idle state is one in which the server is not involved in any jobs. If this state continues exceeding the value specified in **TIME_TO_KILL**, the application server (CAS) is added or removed.

The value configured in this parameter affects only application server added dynamically, so it applies only when the **AUTO_ADD_APPL_SERVER** parameter is configured to **ON**. Note that times to add or remove the application servers (CAS) will be increased more if the **TIME_TO_KILL** value is so small.

SESSION_TIMEOUT

SESSION_TIMEOUT is a parameter that configures a timeout value for the session of the given Broker. The default value is **300** (sec). The given session is terminated if there is no response to the job request for the specified time period.

KEEP_CONNECTION

KEEP_CONNECTION is a parameter that specifies how application servers (CAS) and application clients are connected. It can be configured to **ON**, **OFF** or **AUTO**. If this parameter is configured to **OFF**, clients are connected to an application server in a transaction unit. If it is configured to **ON**, they are connected in a connection unit. If it is configured to **AUTO**, and then the number of application servers is more than that of clients, it will act as if ON; in the reverse case that clients are more than CASs, it will act as if OFF. The default value is **AUTO**.

ACCESS LIST

ACCESS_LIST is a parameter that specifies the name of the file where IP addresses of application client which allows access of the CUBRID Broker is to be saved. To allow access by IP addresses 210.192.33.* and 210.194.34.*, save them to a file (ip_lists.txt) and then configure the file name with the value of this parameter.

ACCESS_LOG

ACCESS_LOG is a parameter that specifies whether or not to store access log. The default value is **ON**. The name of the access log file for the Broker is *broker_name_id*.**access**, and the file is stored in the **log/broker** directory.

LOG_BACKUP

LOG_BACKUP is a parameter that specifies whether or not to back up access and error log files of the Broker. The default value is **OFF**. If this parameter is configured to **ON**, access and error logs are backed up when the CUBRID Broker terminates. The backup file name for access logs becomes *broker_name_id* **access**, and the one for error logs becomes *broker_ name_id*. **error**.

SQL_LOG_MAX_SIZE

SQL_LOG_MAX_SIZE is a parameter that specifies the maximum size of the SQL log file. The default value is **100,000** (KB). If the size of the SQL log file, which is created when the **SQL_LOG** parameter is configured to **ON**, reaches the value configured by the parameter, *broker_name_id*. **sql.log.bak** is created.

APPL_SERVER_PORT

APPL_SERVER_PORT, which can be added only in the Windows operating system, is a parameter that specifies the connection port for the application server (CAS) which communicates with the application client. The default is configured to add 1 to the specified **BROKER_PORT** parameter. As the maximum number of application servers is limited by the **MAX_NUM_APPL_SERVER** parameter of the **cubrid_broker_conf** file, the maximum number of connection ports for the application server (CAS) is also limited by the value of the **MAX_NUM_APPL_SERVER** parameter. If there is a firewall in the Windows operating system between application client and the CUBRID Broker, the connection port specified by **BROKER_PORT** and **APPL_SERVER_PORT** must be open.

APPL_SERVER

APPL_SERVER is a parameter that specifies the type of application servers created and managed by the CUBRID Broker. The default value is **CAS**.

MAX_STRING_LENGTH

MAX_STRING_LENGTH is a parameter that configures the maximum string length for bit, varbit, char, varchar, nchar, nchar varying data types. If this parameter is configured to **-1**, which is the default value, the length defined in the database is used. If the parameter is configured to **100**, the value 100 is applied even when a certain attribute is defined as varchar(1000).

SOURCE_ENV

SOURCE_ENV is a parameter that specifies the file to independently configure operating system environment variables for each broker. The extension of the file must be **env**. All parameters specified in **cubrid.conf** can also be configured by environment variables. For example, the **lock_timeout_in_secs** parameter in **cubrid.conf** can also be configured by the **CUBRID_LOCK_TIMEOUT_IN_SECS** environment variable. As another example, to block execution of DDL statements on broker1, you can configure **CUBRID_BLOCK_DDL_STATEMENT 1** in the file specified by **SOURCE_ENV**.

An environment variable, if exists, has priority over **cubrid.conf**.

The default value is **cubrid.env**.

STATEMENT_POOLING

STATEMENT_POOLING is a parameter that specifies whether or not to use statement pooling. The default value is **ON**.

When a transaction is committed or rolled back, CUBRID closes all the prepared statement handles that exist in the client session. However, if the parameter is set to **STATEMENT_POOLING=ON**, the prepared statement handles remain in the pool, so that the handles can be reused. Therefore, you must maintain the default setting (*STATEMENT_POOLING=ON*) in general applications that reuse prepared statements or in environments in which a library such as DBCP, in which the statement pooling is implemented, is applied.

When the parameter is set to **STATEMENT_POOLING=OFF** and the prepared statement is executed after the transaction is committed or terminated, the following message is displayed.

```
Caused by: cubrid.jdbc.driver.CUBRIDException: Attempt to access a closed Statement.
```

CCI_PCONNECT

CCI_PCONNECT is a parameter that specifies whether or not to use the CCI connection pooling. The default value is **OFF**.

SELECT_AUTO_COMMIT

SELECT_AUTO_COMMIT is a parameter that sets auto-commit mode for **SELECT** statements in **CCI** or **PHP**. Its default value is **OFF**. However, note that auto-commit is performed only at the point at which the result set for all n query statements is fetched from the server when there are n prepared statements. An example is as follows. For more information, see cci_end_tran.

```
SELECT_1 prepare
SELECT_1 execute // AUTO COMMIT O
```

```
SELECT_1 prepare
SELECT_2 prepare
SELECT_1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed
SELECT_2 execute // AUTO COMMIT O

SELECT_1 prepare
SELECT_1 execute // AUTO COMMIT O
INSERT_1 prepare
INSERT_1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed

INSERT_1 prepare
INSERT_1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed
SELECT_1 prepare
SELECT_1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed

SELECT_1 prepare
INSERT_1 prepare
SELECT_1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed
INSERT_1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed

INSERT_1 prepare
SELECT_1 prepare
INSERT_1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed
SELECT_1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed
```

LONG_QUERY_TIME

LONG_QUERY_TIME is a parameter that specifies execution time which is evaluated as long-duration queries. The default value is **60000** in ms. Note that a parameter value is configured to 0, it is not evaluated as a long-duration query.

LONG_TRANSACTION_TIME

LONG_TRANSACTION_TIME is a parameter that specifies execution time which is evaluated as long-duration transactions. The default value is **60000** in ms. Note that a parameter is configured to 0, it is not evaluated as a long-duration transaction.

CUBRID Manager Server Configuration

cm.conf Configuration File and Default Parameters

CUBRID Manager Server System Parameters

The following are parameters required to run the CUBRID Manager server. These parameters can be edited in the configuration file (**cm.conf**).

Parameter Name	Type	Default Value
cm_port	int	8001
monitor_interval	int	5
allow_user_multi_connection	string	YES
auto_start_broker	string	YES
server_long_query_time	int	10

Parameter Syntax Rules

The following are parameter syntax rules applied to the CUBRID Manager server configuration file.

- Parameter names are not case-sensitive.
- The name and value of a parameter must be entered in the same line.
- A space character and an equal sign (=) are allowed to configure a parameter value.
- If the value of a parameter is a character string, enter the character string without quotes. However, use quotes if spaces are included in the character string.

Default Parameters

cm.conf, a default CUBRID Manager configuration file created during the CUBRID installation, includes some default manager server parameters that must be changed. You can change the value of a parameter that is not included as a default parameter by manually adding or editing one.

The following is the content of the **cm.conf** file that is created by default during the installation.

```
# cm.conf
#     -- CUBRID database management tool server configuration file
#
#
# When server starts, it looks for the environment variable
# 'CUBRID_MANAGER' and use it to locate this file. It is assumed that
# 'CUBRID_MANAGER' is the root directory of all CUBRID Manager related files.
#

#
# Port number designation
#
cm_port 8001

#
# Monitoring interval setting
#
monitor_interval 5

#
# Allowing Multiple connection with one CUBRID Manager user.
#
allow_user_multi_connection YES

###############################
# diagnostics parameter
###############################
```

```
#
# turn on/off diag
#
#execute_diag ON

#
# server long query time (sec)
#
server_long_query_time 10
```

Parameters

cm_port

cm_port is a parameter that sets the port to be used between the CUBRID Manager server and client. Two ports are actually used: the port set by the **cm_port** parameter and other port added by 1. For example, if the default value is set to **8001**, both 8001 and 8002 ports are actually used. **cm_port** is used by **cub_auto**, and **cm_port**+1 is used by **cub_js**.

monitor_interval

monitor_interval is a parameter that sets the monitoring interval (in seconds) of the **cub_auto** process of the CUBRID Manager sever. The default value is **5** (sec), and the minimum value is **1** (sec). The shorter the **monitor_interval**, the greater the system load.

allow_user_multi_connection

allow_user_multi_connection is a parameter that allows connection by using CUBRID Manager clients. If the parameter is set to **YES**, which is the default value, all users in the server can use the same user name to connect to the system from more than one CUBRID Manager clients.

auto_start_broker

auto_start_broker is a parameter that sets whether to startthe CUBRID Broker automatically accompanying with the CUBRID Manager server. If the parameter is set to **YES**, which is the default value, the CUBRID Broker starts together with the CUBRID Manager server.

server_long_query_time

server_long_query_time is a parameter that sets a reference time determined by a slow query, a diagnosis operation performed by the database. The default value is **10** (sec). In this case, if the execution time of a query exceeds 10 seconds, it is determined as a slow query. The **server_long_query_time** parameter is valid only when the **execute_diag** parameter is set to **ON** because it is applied when the database diagnosis functionality by the CUBRID Manager is activated.

API Reference

This chapter covers the following APIs:

- JDBC API
- ODBC API
- OLE DB API
- PHP API
- CCI API

JDBC API

JDBC Programming

CUBRID JDBC Driver

The CUBRID JDBC driver (**cubrid_jdbc.jar**) enables the system to make a connection to the CUBRID database in an application written in Java. The driver is located in the "location of CUBRID installed/jdbc" directory.

The CUBRID JDBC driver has been developed based on the JDBC 2.0 specification and provides compilation output generated in JDK version 1.6.

Checking the CUBRID JDBC Driver Version

You can check the JDBC driver version as follows:

```
% jar -tf cubrid_jdbc.jar
META-INF/ META-INF/MANIFEST.MF
cubrid/ cubrid/jdbc/
cubrid/jdbc/driver/
cubrid/jdbc/jci/
cubrid/sql/
CUBRID-JDBC-8.1.4.1032
cubrid/jdbc/driver/CUBRIDBlob.class
...
```

Registering the CUBRID JDBC Driver

Use the **Class.forName** (*driver-class-name*) command to register the JDBC driver. The following is an example of loading the cubrid.jdbc.driver.CUBRIDDriver class to register the CUBRID JDBC driver.

```
import java.sql.*;
import cubrid.jdbc.driver.*;

public class LoadDriver {
   public static void main(String[] Args) {
       try {
           Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
       } catch (Exception e) {
           System.err.println("Unable to load driver.");
           e.printStackTrace();
       }
       ...
```

CUBRID JDBC Interface

The following table shows the JDBC standard and extended interfaces supported by CUBRID JDBC. Note that some methods are not supported even though they are specified in the JDBC 2.0 specification.

Supported Inferface by CUBRID

JDBC Standard Interface	JDBC Extended Interface	Supported
java.sql.Blob java.sql.CallableStatement java.sql.Clob java.sql.Connection java.sql.DatabaseMetaData java.sql.Driver java.sql.PreparedStatement java.sql.ResultSet java.sql.ResultSetMetaData	java.sql.CUBRIDConnection java.sql.CUBRIDPreparedStatement java.sql.CUBRIDResultSet java.sql.CUBRIDResultSetMetaData CUBRIDOID	Supported.
java.sql.Statement	java.sql.CUBRIDStatement	The getGeneratedKeys()

		method of JDBC 3.0 is supported.
java.sql.DriverManager		Supported
Java.sql.SQLException	Java.sql.CUBRIDException	Supported
java.sql.Array java.sql.ParameterMetaData java.sql.Refava.sql.Savepoint java.sql.SQLData java.sql.SQLInput java.sql.Struct		Not Supported

Connection Configuration

The **DriverManager** is a basic interface for JDBC driver management and performs functions such as selecting a database driver and creating a new database connection. If the CUBRID JDBC driver is registered, database connection is made by calling the **DriverManager.getConnection** (*db-url*, *user-id*, *password*) function. The **getConnection** function returns the **Connection** object, which is used for query and command executions and transaction commit or rollback. The parameter *db-url*, which is for connection configuration, is as follows:

```
jdbc:cubrid:<host>:<port>:<db-name>:[user-id]:[password]:[?<property> [& <property>]]

<host> ::=
hostname | ip_address

<property> ::=
althosts= <alternative_hosts> | rctime= <second> | charset= <character_set>

<alternative_hosts> :
<standby_broker1_host>:<port> [,<standby_broker2_host>:<port>]
```

- *<host>* : IP address or host name where the CUBRID Broker is running
- *<port>* : Broker port number (default : 33000)
- *<db-name>* : The name of the database to connect
- *[user-id]* : The user that will be connected to the database. There are two users in the database by default: DBA and PUBLIC. If you enter an empty string (" "), you will connect to the database as a PUBLIC user.
- *[password]* : If there is no password set for the user, enter an empty string (" ").
- *althosts* : One or more host IP of standby broker and connection port to be failed over in HA environment
- *rctime* : Interval time (in seconds) to fail over an active server during system failure. For more information, see the example in the HA-Related JDBC Configuration.
- *charset* : Character set (charset) of database to be connected

Example 1

```
--connection URL string when user name and password omitted

URL=jdbc:CUBRID:127.0.0.1:31000:db1:::

--connection URL string when charset property specified

URL=jdbc:CUBRID:127.0.0.1:31000:db1:::?charset=utf-8

--connection URL string when a property(althosts) specified for HA
URL=jdbc:CUBRID:127.0.0.1:31000:db1:::?althosts=127.0.0.2:31000,127.0.0.3:31000

--connection URL string when properties(althosts,rctime) specified for HA
URL=jdbc:CUBRID:127.0.0.1:31000:db1:::?althosts=127.0.0.2:31000,127.0.0.3:31000&rctime=600

--connection URL string when properties(althosts,rctime, charset) specified for HA
URL=jdbc:CUBRID:127.0.0.1:31000:db1:::?althosts=127.0.0.2:31000,127.0.0.3:31000&rctime=600
&charset=utf-8
```

Example 2

```
String url = "jdbc:cubrid:210.216.33.250:43300:demodb:::";
String userid = "";
String password = "";

try {
   Connection conn =
           DriverManager.getConnection(url,userid,password);
   // Do something with the Connection

   ...

   } catch (SQLException e) {
       System.out.println("SQLException:" + e.getMessage());
       System.out.println("SQLState: " + e.getSQLState());
   }
   ...
```

Note The rollback function, which requests the transaction rollback, exits when the server completes the work.

Verifying Foreign Key Information

Description

You can verify foreign key information by using **getImportedKeys**, **getExportedKeys**, and **getCrossReference** methods provided by **DatabaseMetaData** interface. Usage and examples of each method are as follows:

Syntax

```
getImportedKeys(String catalog, String schema, String table)
getExportedKeys(String catalog, String schema, String table)
getCrossReference(String parentCatalog, String parentSchema, String parentTable, String
foreignCatalog, String foreignSchema, String foreignTable)
```

- **getImportedKeys method** : A method that retrieves the information of primary key columns which are referred by foreign key columns in a given table. The results are sorted by **PKTABLE_NAME** and **KEY_SEQ**.
- **getExportedKeys method** : A method that retrieves the information of all foreign key columns which refer to primary key columns in a given table. The results are sorted by **FKTABLE_NAME** and **KEY_SEQ**.
- **getCrossReference method** : A method that retroeves the information of primary key columns which are referred by foreign key columns in a given table. The results are sorted by **PKTABLE_NAME** and **KEY_SEQ**.

Return Value

When the methods above are called, the following ResultSet, consisting of 14 columns, is returned.

Name	**Type**	**Note**
PKTABLE_CAT	String	Always null
PKTABLE_SCHEM	String	Always null
PKTABLE_NAME	String	Table name of primary key
PKCOLUMN_NAME	String	Table name of primary key
FKTABLE_CAT	String	Always null
FKTABLE_SCHEM	String	Always null
FKTABLE_NAME	String	Table name of foreign key
FKCOLUMN_NAME	String	Column name of foreign key
KEY_SEQ	short	Sequence of foreign or primary keys (starting from 1)
UPDATE_RULE	short	A corresponding value to referring action defined as to foreign keys when primary keys are updated Cascade=0, Restrict=2, No action=3, Set null=4

DELETE_RULE	short	A corresponding value to referring action defined as to foreign keys when primary keys are deleted Cascade=0, Restrict=2, No action=3, Set null=4
FK_NAME	String	Foreign key name
PK_NAME	String	Primary key name
DEFERRABILITY	short	Always 6(DatabaseMetaData.importedKeyInitiallyImmediate)

Example

```
ResultSet rs = null;

                DatabaseMetaData dbmd = conn.getMetaData();

                System.out.println("\n===== Test getImportedKeys");
                System.out.println("=====");
                rs = dbmd.getImportedKeys(null, null, "pk_table");
                Test.printFkInfo(rs);
                rs.close();

                System.out.println("\n===== Test getExportedKeys");
                System.out.println("=====");
               rs = dbmd.getExportedKeys(null, null, "fk_table");
                Test.printFkInfo(rs);
                rs.close();

                System.out.println("\n===== Test getCrossReference");
                System.out.println("=====");
                rs = dbmd.getCrossReference(null, null, "pk_table", null, null,
"fk_table");

                Test.printFkInfo(rs);
                rs.close();
```

Using OIDs and Collections

In addition to the methods defined in the JDBC specification, the CUBRID JDBC driver provides methods that handle OIDs and collections (set, multiset and sequence).

To use these methods, you must import **cubrid.sql.*;** in addition to the CUBRID JDBC driver classes which are imported by default. In addition, to get the results, you must convert **ResultSet** to **CUBRIDResultSet** first. (**ResultSet** is provided by the standard JDBC API, by default.)

```
import cubrid.jdbc.driver.* ;
import cubrid.sql.* ;
...
CUBRIDResultSet urs = (CUBRIDResultSet) stmt.executeQuery(
"SELECT city FROM location");
```

Caution **AUTO COMMIT** does not work even though it is configured to **TRUE** if CUBRID extended APIs are used. Therefore, you must manually commit open connections. The CUBRID extended APIs are methods that handle OIDs and collections.

Using OIDs

You must follow the following rules to use OIDs.

- To use **CUBRIDOID**, you should import **cubrid.sql.*;**. (a)
- You can retrieve an OID by specifying a class name in the **SELECT** statement. The name can be used together with other attributes. (b)
- The **ResultSet** of a query must be **CUBRIDResultSet**. (c)
- The method that retrieves the OID from the **CUBRIDResultSet** is **getOID**(). (d)
- To retrieve a value from an OID, use the **getValues**() method. Its result is **ResultSet**. (e)
- To substitute a value for an OID, use the **setValues**() method. (f)
- When you use the extended APIs, you must always perform **commit**() to make connection. (g)

```
import java.sql.*;
import cubrid.sql.*; //a
import cubrid.jdbc.driver.*;

/*
CREATE TABLE oid_test(
   id INTEGER,
   name VARCHAR(10),
   age INTEGER
);

INSERT INTO oid_test VALUES(1, 'Laura', 32);
INSERT INTO oid_test VALUES(2, 'Daniel', 39);
INSERT INTO oid_test VALUES(3, 'Stephen', 38);
*/

class OID_Sample
{
   public static void main (String args [])
   {
      // Making a connection
      String url= "jdbc:cubrid:localhost:33000:demodb:::";
      String user = "dba";
      String passwd = "";

      // SQL statement to get OID values
      String sql = "SELECT oid_test from oid_test"; //b
      // columns of the table
      String[] attr = { "id", "name", "age" } ;


      // Declaring variables for Connection and Statement
      Connection con = null;
      Statement stmt = null;
      CUBRIDResultSet rs = null;
      ResultSetMetaData rsmd = null;

      try {
         Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
      } catch (ClassNotFoundException e) {
         throw new IllegalStateException("Unable to load Cubrid driver", e);
      }

      try {
         con = DriverManager.getConnection(url, user, passwd);
         stmt = con.createStatement();
         rs = (CUBRIDResultSet)stmt.executeQuery(sql); //c
         rsmd = rs.getMetaData();

         // Printing columns
         int numOfColumn = rsmd.getColumnCount();
         for (int i = 1; i <= numOfColumn; i++ ) {
            String ColumnName = rsmd.getColumnName(i);
            String JdbcType = rsmd.getColumnTypeName(i);
            System.out.print(ColumnName );
            System.out.print("("+ JdbcType + ")");
            System.out.print(" | ");
         }
         System.out.print("\n");
         // Printing rows
         CUBRIDResultSet rsoid = null;
         int k = 1;

         while (rs.next()) {
            CUBRIDOID oid = rs.getOID(1); //d
            System.out.print("OID");
            System.out.print(" | ");
            rsoid = (CUBRIDResultSet)oid.getValues(attr); //e

            while (rsoid.next()) {
               for( int j=1; j <= attr.length; j++ ) {
                  System.out.print(rsoid.getObject(j));
                  System.out.print(" | ");
               }
            }
            System.out.print("\n");

            // New values of the first row
            Object[] value = { 4, "Yu-ri", 19 };
            if (k == 1) oid.setValues(attr, value); //f

            k = 0;
         }
         con.commit(); //g

      } catch(CUBRIDException e) {
         e.printStackTrace();

      } catch(SQLException ex) {
         ex.printStackTrace();

      } finally {
         if(rs != null) try { rs.close(); } catch(SQLException e) {}
         if(stmt != null) try { stmt.close(); } catch(SQLException e) {}
         if(con != null) try { con.close(); } catch(SQLException e) {}
      }
   }
}
```

Using Collections

The line marked by 'a' in the example 1 below is where data of a collection type is fetched from the **CUBRIDResultSet**. The results are returned as array format.

Example 1

```
import java.sql.*;
import java.lang.*;
import cubrid.sql.*;
import cubrid.jdbc.driver.*;

// create class collection_test(
// settest set(integer),
// multisettest multiset(integer),
// listtest list(Integer)
// );
//

// insert into collection_test values({1,2,3},{1,2,3},{1,2,3});
// insert into collection_test values({2,3,4},{2,3,4},{2,3,4});
// insert into collection_test values({3,4,5},{3,4,5},{3,4,5});

class Collection_Sample
{
   public static void main (String args [])
   {
       String url= "jdbc:cubrid:210.216.33.250:43300:demodb:::";
       String user = "";
       String passwd = "";
       String sql = "select settest,multisettest,listtest from collection_test";
       try {
           Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
       } catch(Exception e){
           e.printStackTrace();
       }
       try {
           Connection con = DriverManager.getConnection(url,user,passwd);
           Statement stmt = con.createStatement();
           CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);
           CUBRIDResultSetMetaData rsmd = (CUBRIDResultSetMetaData) rs.getMeta Data();
           int numbOfColumn = rsmd.getColumnCount();
           while (rs.next ()) {
               for (int j=1; j<=numbOfColumn; j++ ) {
                   Object[] reset = (Object[]) rs.getCollection(j); //a
                   for (int m=0 ; m < reset.length ; m++)
                       System.out.print(reset[m] +",");
                   System.out.print(" | ");
               }
               System.out.print("\n");
           }
           rs.close();
           stmt.close();
           con.close();
       } catch(SQLException e) {
           e.printStackTrace();
       }
   }
}
```

Example 2

```
import java.sql.*;
import java.io.*;
import java.lang.*;
import cubrid.sql.*;
import cubrid.jdbc.driver.*;

// create class collection_test(
// settest set(integer),
// multisettest multiset(integer),
// listtest list(Integer)
// );
//
// insert into collection_test values({1,2,3},{1,2,3},{1,2,3});
// insert into collection_test values({2,3,4},{2,3,4},{2,3,4});
// insert into collection_test values({3,4,5},{3,4,5},{3,4,5});

class SetOP_Sample
{
   public static void main (String args [])
   {
       String url = "jdbc:cubrid:127.0.0.1:33000:demodb:::" ;
       String user = "";
       String passwd = "";
       String sql = "select collection_test from collection_test";
       try {
           Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
       } catch(Exception e){
           e.printStackTrace();
       }
       try {
           CUBRIDConnection con =(CUBRIDConnection)
           DriverManager.getConnection(url,user,passwd);
           Statement stmt = con.createStatement();
           CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);
           while (rs.next ()) {
               CUBRIDOID oid = rs.getOID(1);
               oid.addToSet("settest",new Integer(10));
               oid.addToSet("multisettest",new Integer(20));
               oid.addToSequence("listtest",1,new Integer(30));
               oid.addToSequence("listtest",100,new Integer(100));
               oid.putIntoSequence("listtest",99,new Integer(99));
               oid.removeFromSet("settest",new Integer(1));
               oid.removeFromSet("multisettest",new Integer(2));
               oid.removeFromSequence("listtest",99);
               oid.removeFromSequence("listtest",1);
           }
           con.commit();
           rs.close();
           stmt.close();
           con.close();
       } catch(SQLException e) {
           e.printStackTrace();
       }
   }
}
```

Getting Auto-Increment Column Values

Auto-increment Feature

The auto-increment feature (**AUTO_INCREMENT**) is a column-related feature that increments the numeric value of each row. For more information, see Column Definition in Creating Tables. This feature can be defined only for numeric domains (**SMALLINT**, **INTEGER**, **DECIMAL**(*p*, 0), **NUMERIC**(*p*, 0)).

The auto-increment feature is recognized as an automatically created key in a JDBC program. To retrieve the key, you need to specify the time to insert a row from which the automatically created key value is to be retrieved. To perform it, you must set the flag by calling **Connection.prepareStatement** and **Statement.execute**. In this case, the command to be executed should be the **INSERT** statement or **INSERT** within **SELECT** statement. For other commands, the JDBC driver ignores the flag-setting parameter.

Steps

- Use one of the followings to indicate whether or not to return a key created automatically. The following method forms are used for tables of the database server that supports the auto-increment columns. Each method form can be applied only to a single-row **INSERT** statement.
  - Create a **PreparedStatement** object by referring to the followings:
    Connection.prepareStatement(*sql statement*, **Statement.RETURN_GENERATED_KEYS**);
  - To insert a row using the **Statement.execute** method, use one of the forms of the **Statement.execute** method by referring to the followings:
    Statement.execute(*sql statement*, **Statement.RETURN_GENERATED_KEYS**);
- Retrieve a **ResultSet** object that contains a automatically created key value by calling the **PreparedStatement.getGeneratedKeys** or **Statement.getGeneratedKeys** method. Note that the data type of the automatically created key in **ResultSet** is **DECIMAL** regardless of the data type of the given domain.

Example

The following is an example of creating a table with the auto-increment feature, entering data into the table so that automatically created key values are entered into auto-increment columns, and checking whether the key values are successfully retrieved by using the **Statement.getGeneratedKeys**() method. Each step is explained in the comments for commands that correspond to the steps above.

```
import java.sql.*;
import java.math.*;
import cubrid.jdbc.driver.*;

Connection con;
Statement stmt;
ResultSet rs;
java.math.BigDecimal iDColVar;
...
stmt = con.createStatement();     // Create a Statement object

stmt.executeUpdate(
"CREATE TABLE EMP_PHONE (EMPNO CHAR(6), PHONENO CHAR(4), "
+   "IDENTCOL INTEGER AUTO_INCREMENT)"); // Create table with identity column

stmt.execute(
"INSERT INTO EMP_PHONE (EMPNO, PHONENO) "
+   "VALUES ('000010', '5555')",          // Insert a row  <Step 1>
Statement.RETURN_GENERATED_KEYS);            // Indicate you want automatically


rs = stmt.getGeneratedKeys();    // generated keys
                                       // Retrieve the automatically  <Step 2>
                                       // generated key value in a ResultSet.
                                       // Only one row is returned.
                                       // Create ResultSet for query
while (rs.next()) {
  java.math.BigDecimal idColVar = rs.getBigDecimal(1);
                                       // Get automatically generated key
                                       // value
  System.out.println("automatically generated key value = " + idColVar);
}
rs.close();                           // Close ResultSet
stmt.close();                         // Close Statement
```

Using BLOB/CLOB

The interfaces that porcess **LOB** data in JDBC is implemented based on JDBC 4.0 specification. The constraints of interfaces are as follows:

- It supports sequential writes only when creating the objects of **BLOB** or **CLOB**. Writing to arbitary locations are not supported.
- You cannot change the data of **BLOB** or **CLOB** by calling methods of **BLOB** or **CLOB** object which are received from **ResultSet**.
- It does not support **Blob.truncate**, **Clob.truncate**, **Blob.position**, and **Clob.position**.
- You cannot bind **LOB** data by calling **PreapredStatement.setAsciiStream**, **PreparedStatement.setBinaryStream**, and **PreparedStatement.setCharacterStream** methods of **BLOB/CLOB** type columns.
- To use **BLOB/CLOB** types in an environment where JDBC 4.0 specification is not supported such as JDB version 1.5 or earlier, you must convert a conn object to **CUBRIDConnection**, explicitly. See the example below.

```
// JDK 1.6 or later
import java.sql.*;
Connection conn = DriverManager.getConnection(url, id, passwd);
Blob blob = conn.createBlob();
…
// JDK 1.5 or earlier
import java.sql.*;
import cubrid.jdbc.driver.*;

Connection conn = DriverManager.getConnection(url, id, passwd);
Blob blob = ((CUBRIDConnection)conn).createBlob();
…
```

Saving LOB Data

The way to bind **LOB** type data is as follows:

- Create **java.sql.Blob** or **java.sql.Clob** object and save the file contents in the object. Use, then, **setBlob**() or **setClob**() of **PreparedStatement** (example 1).
- Perform query and get **java.sql.Blob** or **java.sql.Clob** object from the **ResultSet object**. Bind, then, the object in **PreparedStatement** (example 2).

Example 1

```
Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
Connection conn = DriverManager.getConnection ("jdbc:cubrid:localhost:33000:image_db:::",
"", "");
PreparedStatement pstmt1 = conn.prepareStatement("INSERT INTO doc(image_id, doc_id, image)
VALUES (?,?,?)");
pstmt1.setString(1, "image-21");
pstmt1.setString(2, "doc-21");

//Creating an empty file in the file system
Blob bImage = conn.createBlob();
byte[] bArray = new byte[256];
…

//Inserting data into the external file. Position is start with 1.
bImage.setBytes(1, bArray);
//Appending data into the external file
bImage.setBytes(257, bArray);
…
pstmt1.setBlob(3, bImage);
pstmt1.executeUpdate();
…
```

Example 2

```
Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
Connection conn = DriverManager.getConnection ("jdbc:cubrid:localhost:33000:image_db:::",
"", "");
conn.setAutoCommit(false);
PreparedStatement pstmt1 = conn.prepareStatement("SELECT image FROM doc WHERE image_id = ?
");
pstmt1.setString(1, "image-21");
ResultSet rs = pstmt1.executeQuery();

while (rs.next())
{
Blob bImage = rs.getBlob(1);
PreparedStatement pstmt2 = conn.prepareStatement("INSERT INTO doc(image_id, doc_id, image)
VALUES (?,?,?)");
pstmt2.setString(1, "image-22")
pstmt2.setString(2, "doc-22")
pstmt2.setBlob(3, bImage);
pstmt2.executeUpdate();
pstmt2.close();
}
pstmt1.close();
conn.commit();
conn.setAutoCommit(true);
conn.close();
…
```

Getting LOB Data

The way to get **LOB** type data is as follows:

- Get data directly from **ResultSet** by using **getBytes**() or **getString**() method (example 1).
- Get the **java.sql.Clob** object from **ResultSet** by calling **getBlob**() or **getClob**() method and then get data by using **getBytes**() or **getSubString**() method for this object (example 2).

Example 1

```
Connection conn = DriverManager.getConnection ("jdbc:cubrid:localhost:33000:image_db:::",
"", "");

// Getting data directly from ResetSet
PrepareStatement pstmt1 = conn.prepareStatement("SELECT content FROM doc_t WHERE doc_id = ?
");
pstmt2.setString(1, "doc-10");
ResultSet rs = pstmt1.executeQuery();
while (rs.next())
  {
    String sContent = rs.getString(1);
    System.out.println("doc.content= "+sContent.);
  }
```

Example 2

```
Connection conn = DriverManager.getConnection ("jdbc:cubrid:localhost:33000:image_db:::",
"", "");

//Getting Blob data from ResultSet and getting data from the Blob object
PrepareStatement pstmt2 = conn.prepareStatement(“SELECT image FROM image_t WHERE image_id
= ?”);
pstmt2.setString(1,”image-20”);
ResultSet rs = pstmt1.executeQuery();
while (rs.next())
  {
    Blob bImage = rs.getBlob(1);
    Bytes[] bArray = bImange.getBytes(1, (int)bImage.length());
  }
```

CUBRIDOID

Overview

A **CUBRIDOID** class contains the following methods to process OIDs.

Return Type	Method Name
void	addToSequence(String attrName, int index, Object value)
void	addToSet(String attrName, Object value)
static CUBRIDOID	getNewInstance(CUBRIDConnection con, String oidStr)
String	getOidString()
String	getTableName()
ResultSet	getValues(String[] attrNames)
Boolean	isInstance()
void	putIntoSequence(String attrName, int index, Object value)
void	remove()
void	removeFromSequence(String attrName, int index)
void	removeFromSet(String attrName, Object value)
void	setReadLock()
void	setValues(String[] attrNames, Object[] values)
void	setWriteLock()

addToSequence

Description

This function is used to insert the value specified in *value* into the attribute named *attrName* and associated with **SEQUENCE** constraints on the **CUBRIDOID** instance, specifically in front of the *index*-th element in the **SEQUENCE** attribute.

Syntax

```
void addToSequence(String attrName, int index, Object value)
```

Example

```
//create class foo(c list of int  )
//insert into foo values({3})

String sql = "select foo from foo" ;

Connection con = DriverManager.getConnection(url,user,passwd);

Statement stmt = con.createStatement();
CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID oid = rs.getOID(1);               // get OID
   oid.addToSequence("c",1, new Integer(22)); // c: {3}-> {22,3}
}
```

addToSet

Description

This function is used to insert the value specified in *value* into the attribute named *attrName* and associated with **SET** or **MULTISET** constraints on the **CUBRIDOID** instance.

Syntax

```
void addToSet(String attrName, Object value)
```

Example

```
//create class foo(a set of int, b multiset of int )
//insert into foo values({1},{2})
String sql = "select foo from foo" ;

Connection con = DriverManager.getConnection(url,user,passwd);

Statement stmt = con.createStatement();
CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);
while (rs.next ()) {
   CUBRIDOID oid = rs.getOID(1);        // get OID
   oid.addToSet("a",new Integer(11));  // a : {1} -> {1,11}
   oid.addToSet("b",new Integer(13));  // b : {2} -> {2, 13}
}
```

getNewInstance

Description

This function is used to convert an OID string to a **CUBRIDOID** object, and then returns the **CUBRIDOID** object.

Syntax

```
static CUBRIDOID getNewInstance(CUBRIDConnection con, String oidStr)
```

Return Value

- **CUBRIDOID** object

Example

```
String sql = "select foo from foo" ;

CUBRIDConnection con = (CUBRIDConnection)
                    DriverManager.getConnection(url,user,passwd);

Statement stmt = con.createStatement();

CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID realoid = rs.getOID(1);  // get OID (CUBRIDOID)
   // CUBRIDOID -> OID string
   String stringoid =  realoid.getOidString();
   // OID string -> CUBRIDOID
   realoid = CUBRIDOID.getNewInstance(con, stringoid);
}
```

getOidString

Description

This function is used to convert a **CUBRIDOID** object to an OID string, and then returns the string.

Syntax

```
String getOidString()
```

Return Value

- Character string

Example

```
String sql = "select foo from foo" ;

CUBRIDConnection con = (CUBRIDConnection)
                    DriverManager.getConnection(url,user,passwd);

Statement stmt = con.createStatement();
CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID realoid = rs.getOID(1);    // get OID
   // CUBRIDOID -> OID string
   String stringoid = realoid.getOidString();
   // OID string -> CUBRIDOID
   realoid = CUBRIDOID.getNewInstance(con,stringoid);
}
```

getTableName

Description

This function is used to returns the table name of the instance corresponding to the **CUBRIDOID** object.

Syntax

```
String getTableName()
```

Return Value

- A table name of an instance that corresponds to **CUBRIDOID**

Example

```
String sql = "select foo from foo" ;

CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID oid = rs.getOID(1);

   String tablename = oid.getTableName();
   System.out.println(tablename );
}
```

getValues

Description

This function is used to return the **ResultSet** which contains values of the requested attribute.

Syntax

```
ResultSet getValues(String[] attrNames)
```

Return Value

- **ResultSet**

Example

```
// create class foo ( a string, b int )
// insert into foo values('CUBRID', 2001)

String sql = "select foo from foo";
String[] attr = { "a", "b" };  // class's column name list
CUBRIDResultSet rs= (CUBRIDResultSet) stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID oid = rs.getOID(1);
   ResultSet rsoid = oid.getValues(attr);
}
```

isInstance

Description

This function is used to return true if the instance corresponding to the **CUBRIDOID** exists. If otherwise, it returns false.

Syntax

```
Boolean isInstance()
```

Return Value

- TRUE : An instance that corresponds to **CUBRIDOID** exists.
- FALSE : An instance that corresponds to **CUBRIDOID** does not exist.

Example

```
String sql = "select foo  from foo" ;

CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID oid = rs.getOID(1);
   System.out.print("isInstance : " + oid.isInstance()); // true
   oid.remove();  // remove the object in the oid
   System.out.print("After remove, isInstance : "
                    + oid.isInstance()); // false
}
```

putIntoSequence

Description

This function is used to modify the *index*-th value in the attribute associated with the **SEQUENCE** constraint on the **CUBRIDOID** instance as the value specified in *value*.

Syntax

```
void putIntoSequence(String attrName, int index, Object value)
```

Example

```
//create class foo(c list of int  )
//insert into foo values({1})

String sql = "select foo from foo" ;

Connection con = DriverManager.getConnection(url,user,passwd);

Statement stmt = con.createStatement();
CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID oid = rs.getOID(1);                // get OID
   oid.putIntoSequence("c",1, new Integer(10)); // c:{1}->{10}
}
```

remove

Description

This function is used to remove the instance corresponding to the **CUBRIDOID**.

Syntax

```
void remove()
```

Example

```
String sql = "select foo from foo" ;

CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID oid = rs.getOID(1);
   System.out.print("isInstance : " + oid.isInstance()); // true
   oid.remove();  // remove the object in the oid
   System.out.print("  After remove .isInstance : " +
                    oid.isInstance()); // false
}
```

removeFromSequence

Description

This function is used to remove the *index*-th value from the attribute associated with the **SEQUENCE** constraint on the **CUBRIDOID** instance.

Syntax

```
void removeFromSequence(String attrName, int index)
```

Example

```
//create class foo(c list of int  )
//insert into foo values(1,3)

String sql = "select foo from foo" ;

Connection con = DriverManager.getConnection(url,user,passwd);

Statement stmt = con.createStatement();
CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID oid = rs.getOID(1);      // get OID
   oid.removeFromSequence("c",1);     // c: {1,3} -> {3}
}
```

removeFromSet

Description

This function is used to remove the corresponding value specified in *value* from the attribute associated with the **SET** constraint on the **CUBRIDOID** instance. If the corresponding value is more than one, the very value found for the first time becomes removed.

Syntax

```
void removeFromSet(String attrName, Object value)
```

Example

```
//create class foo(a set of int, b multiset of int )
//insert into foo values({1,11},{2,13})

String sql = "select foo  from foo"
Connection con = DriverManager.getConnection(url,user,passwd)

Statement stmt = con.createStatement()
CUBRIDResultSet rs= (CUBRIDResultSet) stmt.executeQuery(sql)

while (rs.next ()) {
    CUBRIDOID oid = rs.getOID(1)               // get OID
    oid.removeFromSet("a",new Integer(11)) // a: {1,11} -> {1}
    oid.removeFromSet("a",new Integer(13)) // b: {2,13} -> {2}
}
```

setReadLock

Description

This function is used to set a read-lock on the instance corresponding to the **CUBRIDOID**.

Syntax

```
void setReadLock()
```

Example

```
String sql = "select foo from foo" ;

CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID oid = rs.getOID(1);
   oid.setReadLock();
}
```

setValues

Description

This function is used to replace the value specified in the *attrNames* with the value specified in the *values*.

Syntax

```
void setValues(String[] attrNames, Object[] values)
```

Example

```
// create class foo ( a string, b int )
String sql = "select foo from foo";
String[] attr = { "a", "b" };  // a list of attribute names
String[] values = {"CUBRID", new Integer(2001)};

CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID oid = rs.getOID(1);
   oid.setValues(attr, values );
}
```

setWriteLock

Description

This function is used to set a write-lock on the instance corresponding to the **CUBRIDOID**.

Syntax

```
void setWriteLock()
```

Example

```
String sql = "select foo from foo" ;

CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);

while (rs.next ()) {
   CUBRIDOID oid = rs.getOID(1);
   oid.setWriteLock();
}
```

CUBRIDPreparedStatement

Overview

The **CUBRIDPreparedStatement** class extends the standard **PreparedStatement** and contains the following additional methods.

Return Type	Method Name
CUBRIDOID	executeInsert()
void	setCollection(int index, Object[] array)
void	setOID(int index, CUBRIDOID oid)

executeInsert

Description

This function is used to execute an **INSERT** statement within the **CUBRIDPreparedStatement** object and returns the **CUBRIDOID** corresponding to the inserted object.

Syntax

```
CUBRIDOID executeInsert()
```

Return Value

- A **CUBRIDOID** that corresponds to the inserted object

Example

```
String sql  = "insert into testtable(a) values(?)";

CUBRIDPreparedStatement pstmt = (CUBRIDPreparedStatement)
                                con.prepareStatement(sql);
pstmt.setString(1, "CUBRID");
CUBRIDOID oid = pstmt.executeInsert();
```

setCollection

Description

This function is used to specify the *index*-th parameter in the prepared statement as a collection corresponding to *array*. CUBRID has three types of collections: Set, Multiset and Sequence.

Syntax

```
void setCollection(int index, Object[] array)
```

Example

```
String[] strs = { "abc", "def"};

psmt.setCollection(1, strs);
```

setOID

Description

This function is used to specify the *index*-th parameter in the prepared statement as the **CUBRIDOID** specified in *oid*.

Syntax

```
void setOID(int index, CUBRIDOID oid)
```

CUBRIDResultSet

Overview

The **CUBRIDResultSet** class is extended from the standard **ResultSet** class and has the following additional methods.

Return Type	Method Name
Object	getCollection(int attrIndex)
Object	getCollection(String attrName)
CUBRIDOID	getOid()
CUBRIDOID	getOid(int attrIndex)
CUBRIDOID	getOid(String attrName)

getCollection

Description

This function is used to return the index specified in *attrIndex* or the attribute value specified in *attrName*. The returned object can be converted to an array such as String[].

Syntax

```
Object getCollection(int attrIndex)
Object getCollection(String attrName)
```

Return Value

- An index specified by *attrIndex* or a value of the column that corresponds to the column name specified by *attrName*

getOID

Description

This function is used to return the index specified in *attrIndex* or the attribute value specified in *attrName* to **CUBRIDOID**, thus it returns the **CUBRIDOID**.

If *attrIndex* or *attrName* is not specified, **CUBRIDOID** of the current row of **ResultSet** is returned. This is valid only when **ResultSet** is **TYPE_SCROLL_SENSITIVE** or **CONCUR_UPDATABLE**.

Syntax

```
CUBRIDOID getOID(int attrIndex)
CUBRIDOID getOID(String attrName)
CUBRIDOID getOID()
```

Return Value

- **CUBRIDOID**

CUBRIDResultSetMetaData

Overview

The **CUBRIDResultSetMetaData** class is extended from the standard **ResultSetMetaData** and has the following additional methods.

Return Type	Method Name
int	getElementType(int columnIndex)
String	getElementTypeName(int columnIndex)

getElementType

Description

This function is used to return a type of the COLLECTION element as *int* defined in the **java.sql.Types**. If a domain of the *columnIndex*-th attribute is not COLLECTION such as **SET**, **MULTISET**, or **SEQUENCE**, **SQLException** occurs in the end.

Syntax

```
int getElementType(int columnIndex)
```

Return Value

- Collection element type (int)

getElementTypeName

Description

This function is used to return the name of the type in the COLLECTION elements. If a domain of the *columnIndex*-th attribute is not COLLECTION such as **SET**, **MULTISET**, or **SEQUENCE**, **SQLException** occurs in the end.

Syntax

```
String getElementTypeName(int columnIndex)
```

Return Value

- Collection element's type name

Example

```
// The following schema is used in this example.
//
// create class foo(
//    a  set(int),
//    b  multiset(int),
//    c  sequence(int)
// );

String sql = "select * from foo" ;
Connection con = DriverManager.getConnection(url,user,passwd);

Statement stmt = con.createStatement();
```

```
CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);
CUBRIDResultSetMetaData rsmd = (CUBRIDResultSetMetaData)
                               rs.getMetaData();

int numberofColumn = rsmd.getColumnCount();
for (int i=1; i <= numberofColumn; i++ ) {
   System.out.println(rsmd.getElementType(i) );
   System.out.println(rsmd.getElementTypeName(i) );
}
```

CUBRIDStatement

Overview

The **CUBRIDStatement** class is extended from the standard **Statement** class and has the following additional methods.

Return Type	Method Name
CUBRIDOID	executeInsert(String insertStmt)

executeInsert

Description

This function is used to return the **CUBRIDOID** corresponding to a new tuple (row) inserted by the SQL statement, *insertStmt*.

Syntax

```
CUBRIDOID executeInsert(String insertStmt)
```

Return Value

- **CUBRIDOID** of the added row

Example

```
String sql = "insert into testable(a) values (1)"

CUBRIDStatement stmt = (CUBRIDStatement) con.createStatement();
CUBRIDOID oid = stmt.executeInsert(sql);
```

ODBC API

ODBC Programming

CUBRID ODBC Driver

Description

The CUBRID ODBC driver supports ODBC version 3.52, ODBC core, and some of Level 1 and Level 2 APIs. Since it has been developed based on ODBC Spec 3.x, backward compatibility is not completely ensured for programs written using ODBC Spec 2.x. Only 32 bit are supported.

For more information on configuring CUBRID ODBC driver, see Configuring the Environment of ODBC and ASP.

Data Type Mapping of CUBRID and ODBC

The following table shows the data mapping relationship between data types of ODBC and those supported by CUBRID.

CUBRID Data Type	ODBC Data Type
Char	SQL_CHAR
Varchar	SQL_VARCHAR
String	SQL_LONGVARCHAR
Nchar	SQL_CHAR
Varnchar	SQL_VARCHAR
Bit	SQL_BINARY
varying bit	SQL_VARBINARY
Numeric	SQL_NUMERIC
Int	SQL_INTEGER
Short	SQL_SMALLINT
Float	SQL_FLOAT
Double	SQL_DOUBLE
Bigint	SQL_BIGINT
Date	SQL_TYPE_DATE
Time	SQL_TYPE_TIME
Timestamp	SQL_TYPE_TIMESTAMP
Datetime	SQL_TYPE_TIMESTAMP
Monetary	SQL_DOUBLE
Oid	SQL_CHAR(32)
set, multiset, sequence	SQL_VARCHAR(MAX_STRING_LENGTH)

Configuring Connection Strings

When you are programming CUBRID ODBC, you can write connection strings as follows:

Item	Example	Description
Driver	CUBRID Driver	Driver name

UID	user1	User ID
PWD	xxx	Password
FETCH_SIZE	100	Fetch size
PORT	30000	Broker port number
SERVER	192.168.1.11	IP address or host name of a CUBRID Broker server
DB_NAME	demodb	Database name
DESCRIPTION	cubrid_test	Description
CHARSET	utf-8	Character set

The following example shows how to use connecting strings above.

```
"DRIVER=CUBRID
Driver;UID=user1;PWD=xxx;FETCH_SIZE=100;PORT=30000;SERVER=192.168.1.11;DB_NAME=demodb;DESC
RIPTION=cubrid_test;CHARSET=utf-8"
```

Supported Functions and Backward Compatibility

Information on supported functions by CUBRID ODBC, versions, compatibility with ODBC Spec is as follows:

API	Version Introduced	Standards Compliance	Support
SQLAllocHandle	3.0	ISO 92	YES
SQLBindCol	1.0	ISO 92	YES
SQLBindParameter	2.0	ODBC	YES
SQLBrowseConnect	1.0	ODBC	NO
SQLBulkOperations	3.0	ODBC	YES
SQLCancel	1.0	ISO 92	YES
SQLCloseCursor	3.0	ISO 92	YES
SQLColAttribute	3.0	ISO 92	YES
SQLColumnPrivileges	1.0	ODBC	NO
SQLColumns	1.0	X/Open	YES
SQLConnect	1.0	ISO 92	YES
SQLCopyDesc	3.0	ISO 92	YES
SQLDescribeCol	1.0	ISO 92	YES
SQLDescribeParam	1.0	ISO 92	NO
SQLDisconnect	1.0	ISO 92	YES
SQLDriverConnect	1.0	ISO 92	YES
SQLEndTran	3.0	ISO 92	YES
SQLExecDirect	1.0	ISO 92	YES
SQLExecute	1.0	ISO 92	YES
SQLFetch	1.0	ISO 92	YES
SQLFetchScroll	3.0	ISO 92	YES
SQLForeignKeys	1.0	ODBC	YES (2008 R3.1 or later)
SQLFreeHandle	3.0	ISO 92	YES
SQLFreeStmt	1.0	ISO 92	YES

SQLGetConnectAttr	3.0	ISO 92	YES
SQLGetCursorName	1.0	ISO 92	YES
SQLGetData	1.0	ISO 92	YES
SQLGetDescField	3.0	ISO 92	YES
SQLGetDescRec	3.0	ISO 92	YES
SQLGetDiagField	3.0	ISO 92	YES
SQLGetDiagRec	3.0	ISO 92	YES
SQLGetEnvAttr	3.0	ISO 92	YES
SQLGetFunctions	1.0	ISO 92	YES
SQLGetInfo	1.0	ISO 92	YES
SQLGetStmtAttr	3.0	ISO 92	YES
SQLGetTypeInfo	1.0	ISO 92	YES
SQLMoreResults	1.0	ODBC	YES
SQLNativeSql	1.0	ODBC	YES
SQLNumParams	1.0	ISO 92	YES
SQLNumResultCols	1.0	ISO 92	YES
SQLParamData	1.0	ISO 92	YES
SQLPrepare	1.0	ISO 92	YES
SQLPrimaryKeys	1.0	ODBC	YES (2008 R3.1 or later)
SQLProcedureColumns	1.0	ODBC	YES (2008 R3.1 or later)
SQLProcedures	1.0	ODBC	YES (2008 R3.1 or later)
SQLPutData	1.0	ISO 92	YES
SQLRowCount	1.0	ISO 92	YES
SQLSetConnectAttr	3.0	ISO 92	YES
SQLSetCursorName	1.0	ISO 92	YES
SQLSetDescField	3.0	ISO 92	YES
SQLSetDescRec	3.0	ISO 92	YES
SQLSetEnvAttr	3.0	ISO 92	NO
SQLSetPos	1.0	ODBC	YES
SQLSetStmtAttr	3.0	ISO 92	YES
SQLSpecialColumns	1.0	X/Open	YES
SQLStatistics	1.0	ISO 92	YES
SQLTablePrivileges	1.0	ODBC	YES (2008 R3.1 or later)
SQLTables	1.0	X/Open	YES

Some functions for which backward compatibility is not supported must be converted into appropriate ones by using the mapping table below.

ODBC 2.x Function	ODBC 3.x Function

SQLAllocConnect	SQLAllocHandle
SQLAllocEnv	SQLAllocHandle
SQLAllocStmt	SQLAllocHandle
SQLBindParam	SQLBindParameter
SQLColAttributes	SQLColAttribute
SQLError	SQLGetDiagRec
SQLFreeConnect	SQLFreeHandle
SQLFreeEnv	SQLFreeHandle
SQLFreeStmt with SQL_DROP	SQLFreeHandle
SQLGetConnectOption	SQLGetConnectAttr
SQLGetStmtOption	SQLGetStmtAttr
SQLParamOptions	SQLSetStmtAttr
SQLSetConnectOption	SQLSetConnectAttr
SQLSetParam	SQLBindParameter
SQLSetScrollOption	SQLSetStmtAttr
SQLSetStmtOption	SQLSetStmtAttr
SQLTransact	SQLEndTran

Using OIDs and Collections

ODBC is designed for relational DBMSs. Therefore, CUBRID ODBC does not support some object-oriented features such as CUBRID OIDs and collections. It is because CUBRID is an object-relational DBMS that integrates relational and object-oriented data models.

Using OIDs

Because the CUBRID ODBC driver considers an OID as a string (char(32)), the **INSERT**, **UPDATE** and **DELETE** statements containing OIDs can be used as follows. The OID string should be used with single quotes ("). The domain of the member attribute in the following example is the same as the OID.

```
insert into foo(member) values('@12|34|56')
delete from foo where member = '@12|34|56'
update foo set age = age + 1 where member = '@12|34|56'
```

Using Collections

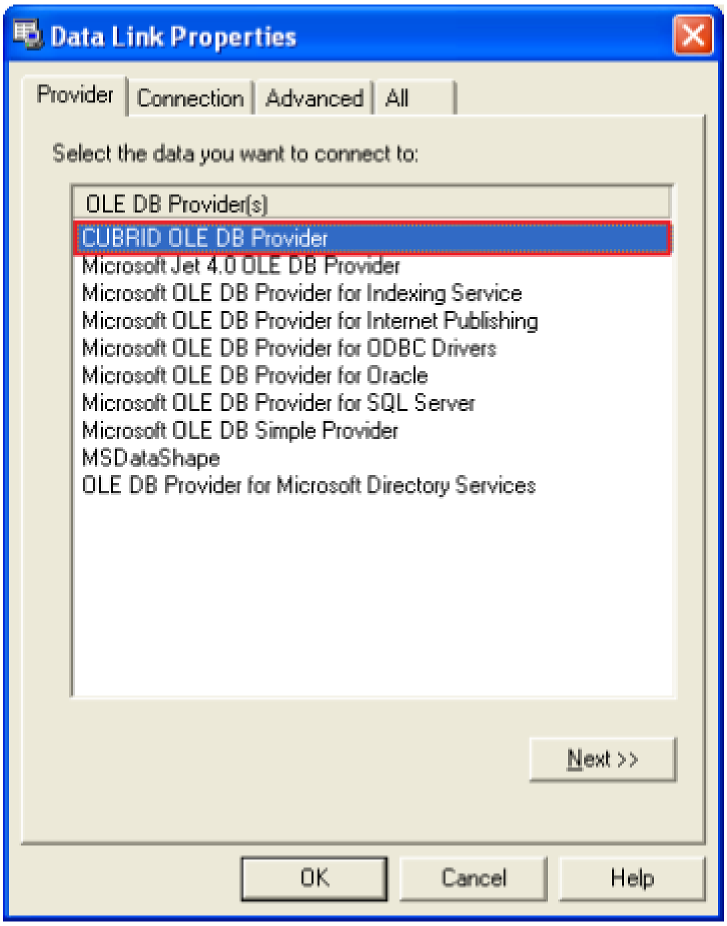
Collection types : **SET**, **MULTISET** and **SEQUENCE** are supported. The CUBRID ODBC driver considers a collection as a string (longvarchar). You can obtain a collection by separating each element in the **SELECT** statement using commas in braces as with "{value_1, value_2, ...value_n}."OLE DB API

OLE DB Programming

Using Data Link Property Dialog Box

In the [Data Link Properties] dialog box, you can check and configure various OLE DB providers provided by the current Windows operating system.

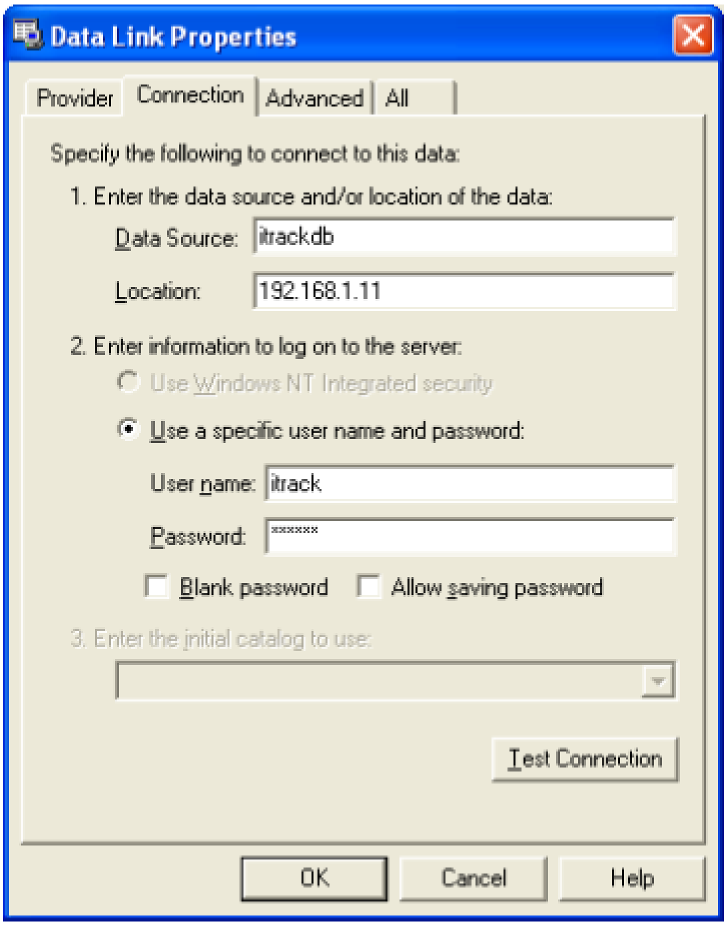
If you have properly installed the CUBRID OLE DB Provider for Windows, 'CUBRID OLE DB Provider' is displayed in the provider list of the [Data Link Properties] dialog box, as shown below.



Select the data you want to connect to:

OLE DB Provider(s)
CUBRID OLE DB Provider
Microsoft Jet 4.0 OLE DB Provider
Microsoft OLE DB Provider for Indexing Service
Microsoft OLE DB Provider for Internet Publishing
Microsoft OLE DB Provider for ODBC Drivers
Microsoft OLE DB Provider for Oracle
Microsoft OLE DB Provider for SQL Server
Microsoft OLE DB Simple Provider
MSDataShape
OLE DB Provider for Microsoft Directory Services

If you click the [Next] button after selecting 'CUBRID OLE DB Provider', the [Connection] tab appears as shown below. Set the desired link properties in the [Connection] tab.



Specify the following to connect to this data:

1. Enter the data source and/or location of the data:

Data Source: itrackdb

Location: 192.168.1.11

2. Enter information to log on to the server:

○ Use Windows NT Integrated security

● Use a specific user name and password:

User name: itrack

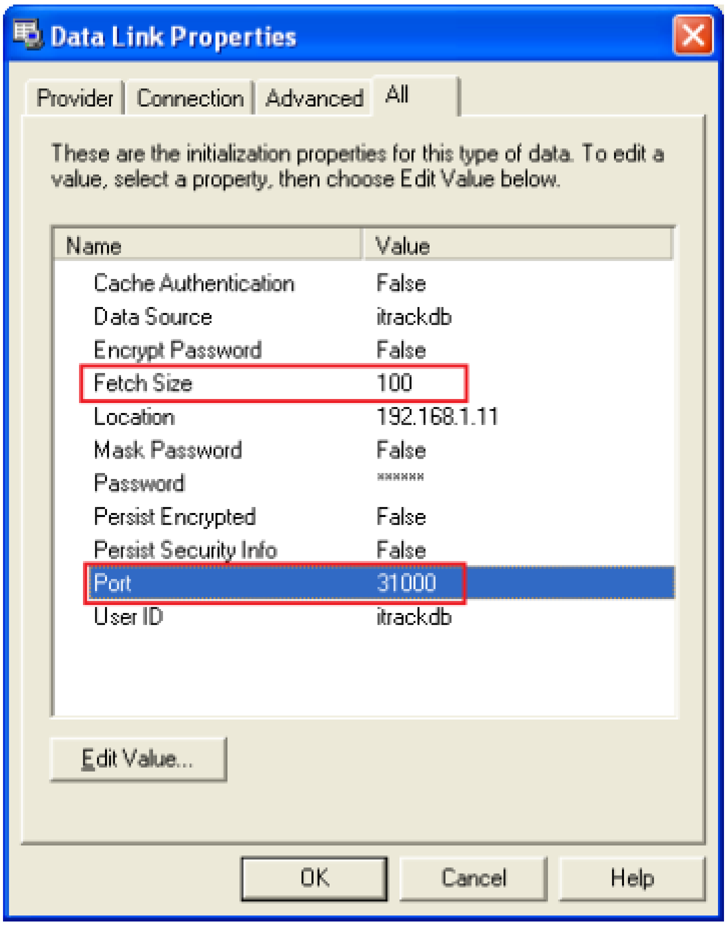
Password: ******

□ Blank password □ Allow saving password

3. Enter the initial catalog to use:

- **Data source** : Enter the name of the CUBRID database.
- **Location** : Enter the IP address or host name of the server where the CUBRID Broker is running.
- **User name** : Enter the name of the user who will log on to the database server.
- **Password** : Enter the password to be used for the database server logon.

Select all connection properties and then click the [All] tab.

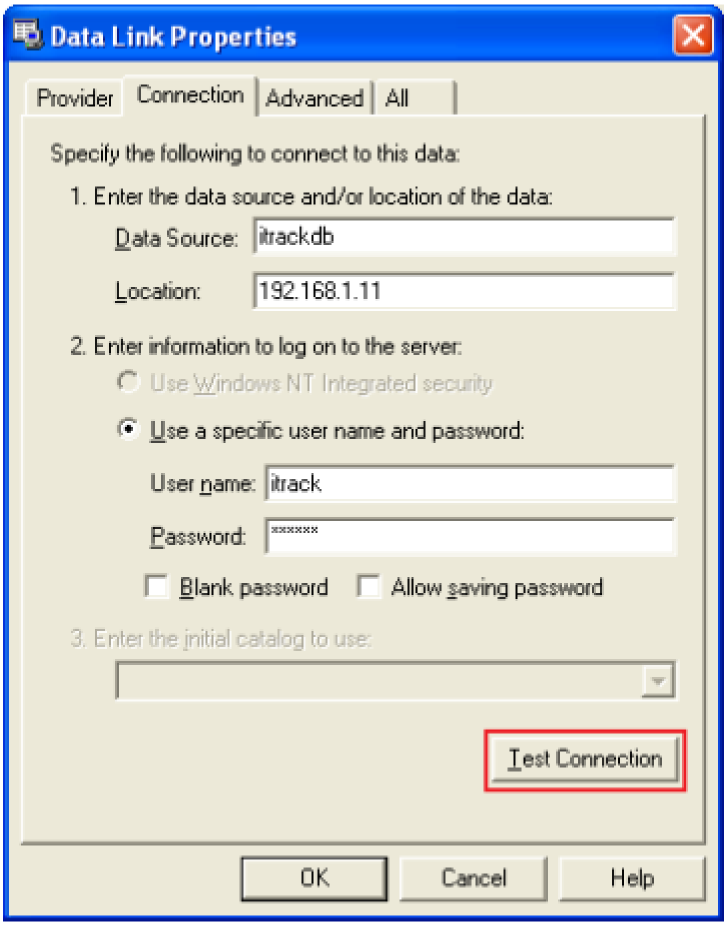


These are the initialization properties for this type of data. To edit a value, select a property, then choose Edit Value below.

Name	Value
Cache Authentication	False
Data Source	itrackdb
Encrypt Password	False
Fetch Size	100
Location	192.168.1.11
Mask Password	False
Password	******
Persist Encrypted	False
Persist Security Info	False
Port	31000
User ID	itrackdb

To check every value currently configured, click the [All] tab; to edit the value, double-click the item you want. When the [Edit Property Value] dialog box appears, enter the desired value and then click [OK]. The figure above shows an example that configures the [Port] to "31000," and [Fetch Size] to "100."

You can check whether the connection is working properly by clicking the [Test Connection] button in the [Connection] tab after completing all configuration.



Specify the following to connect to this data:

1. Enter the data source and/or location of the data:

Data Source: itrackdb

Location: 192.168.1.11

2. Enter information to log on to the server:

○ Use Windows NT Integrated security

● Use a specific user name and password:

User name: itrack

Password: ******

□ Blank password □ Allow saving password

3. Enter the initial catalog to use:

Configuring Connection String

When you program the CUBRID OLE DB Provider using ADO (ActiveX Data Object) or ADO.net, write the connection string as follows:

Item	Example	Description
Provider	CUBRIDProvider	Provider name
Data source	demodb	Database name
Location	192.168.1.11	The IP address of the CUBRID Broker Server
User ID	PUBLIC	User ID
Password	xxx	Password
Port	30000	Broker port number
Fetch Size	100	Fetch size

A connection string using the above example is as follows:

```
"Provider = CUBRIDProvider;Data Source = demodb;Location = 192.168.1.11;User ID =
PUBLIC;Password =xxx;Port = 30000;Fetch Size = 100"
```

Multi-Thread Programming in .NET Environment

To develop programs by using the CUBRID OLE DB Provider in the Microsoft .NET, you should consider the followings:

If you develop multi-thread programs by using ADO.NET in the management environment, you need to change the value of the ApartmentState attribute of the Thread object to a ApartmentState.STA value because the CUBRID OLE DB Provider supports only Single Threaded Apartment (STA) attributes.

Without any change of given values, the default value of the attribute in the Thread object returns Unknown value, thereby causing abnormal process or errors during multi-threads programming.

Caution All OLE DB objects are COM objects. Currently, the CUBRID OLE DB Provider supports only the apartment threading model among COM threading models. It does not support the free threading model. This applies to not only the .NET but all multi-threaded environment.

PHP API

PHP Programming

General Features

Connection/Transaction

- Connecting to a database : The first step of a database application is to use the cubrid_connect() function which provides a database connection. Once the cubrid_connect() function is executed successfully, you can use any functions available in the database. It is very important to call the cubrid_disconnect() function before the application is terminated completely. The cubrid_disconnect() function terminates the current transaction as well as the connection handle and all request handles created by the cubrid_connect() function.

Note Executing the cubrid_connect() function does not create the connection with the database server automatically. The actual connection is made when the necessary function for the database server connection is called.

- The cubrid_commit() or cubrid_rollback() function is used to commit or roll back a transaction. The cubrid_disconnect() function terminates the transaction and rolls back uncommitted ones.

Processing Queries

The following are basic steps of query execution.

- Creating a connection handle
- Creating a request handle for an SQL query request
- Fetching the result
- Terminating the request handle

```
$con = cubrid_connect("192.168.1.12", 12345, "demodb")
if($con) {
      $req = cubrid_execute($con, "select * from dept")
      if($req) {
              while ($row = cubrid_fetch($req)) {
              echo $row["name"]
              echo $row["position"]
              }
              cubrid_close_request($req)
      }
      cubrid_disconnect($con)
}
```

Column types and names of the query result

The cubrid_column_types() function is used to get an array containing column types, and the cubrid_column_names() function is used to get an array containing column names.

```
$req = cubrid_execute($con, "select * from person")
if($req) {
      $coltypes = cubrid_column_types($req)
      $colnames = cubrid_column_names($req)

      while (list($key, $coltype) = each ($coltypes))
      echo $coltype

      while (list($key, $colname) = each ($colnames))
      echo $colname

      cubrid_close_request(#req)
}
```

Adjusting the cursor

You can configure the position of the query result. The cubrid_move_cursor() function is used to move the cursor to a certain position from one of three points: the beginning of the query result, the current cursor position and the end of the query result.

```
$req = cubrid_execute($con, "select * from person")
if($req) {
      cubrid_move_cursor ($req, 10, CUBRID_CURSOR_CURRENT)
      while ($row = cubrid_fetch($req, CUBRID_ASSOC)) {
              echo $row["id"]
              echo $row["name"]
      }
}
```

Result array types

One of the following three types of arrays is used in the result of the cubrid_fetch() function. The type of the array can be determined when the cubrid_fetch() function is called. The associative array uses character string indexes. The numeric array uses numeric order indexes. The last array type includes both associative and numeric arrays.

- Numeric array

```
while (list($id, $name) = cubrid_fetch($req, CUBRID_NUM)) {
      echo $id
      echo $name
}
```

- Associative array

```
while ($row = cubrid_fetch($req, CUBRID_ASSOC)) {
      echo $row["id"]
      echo $row["name"]
}
```

Catalog Operation

Information about the database schema such as classes, virtual classes, attributes, functions, triggers and constraints can be obtained by calling the cubrid_schema() function. The return value of the cubrid_schema() function is a two-dimensional array.

```
$attrs=cubrid_schema($con,CUBRID_SCH_ATTRIBUTE,"person")
if ($attrs != -1) {
      while (list ($key, $attr) = each($attrs)) {
              echo $row["NAME"]
              echo $row["DOMAIN"]
      }
}
```

Processing Errors

When an error occurs, most PHP interface functions display the error message and return false or -1. Each error message, error code or error facility code can be checked by using the cubrid_error_msg(), cubrid_error_code(), and cubrid_error_code_facility() functions.

The return value of the cubrid_error_code_facility() function is one of **CUBRID_FACILITY_DBMS** (DBMS error), **CUBRID_FACILITY_CAS** (CAS server error), **CUBRID_FACILITY_CCI** (CCI error) and **CUBRID_FACILITY_CLIENT** (PHP module error).

CUBRID Features

Using OIDs

With a query that can update the **CUBRID_INCLUDE_OID** option in the cubrid_execute() function, you can get the OID value of the current row updated by the executing cubrid_current_oid().

```
CUBRID_INCLUDE_OID);
if ($req) {
while ($row = cubrid_fetch ($req)) {
echo cubrid_current_oid ($req);
echo $row["id"];
echo $row["name"];
}
cubrid_close_request ($req);
}
```

You can get all attributes, the specified attribute or an attribute of an instance by using the OID.

If you don't specify any attribute in the cubrid_get() function, the values of all attributes are returned (a). If you specify an attribute as an array data type, an associative array containing the values of the specified attribute is returned (b). If you specify an attribute as a character string array, the value of the attribute is returned (c).

```
$attrarray = cubrid_get ($con, $oid); // (a)
$attrarray = cubrid_get ($con, $oid, array("id", "name")); // (b)
$attrarray = cubrid_get ($con, $oid, "id"); // (c)
```

You can also update an attribute value of an instance by using the OID. To update a single attribute value, specify the attribute name as a character string type and its value (a). To set multiple attribute values, specify an associative array containing the attribute names and values (b).

```
$cubrid_put ($con, $oid, "id", 1); // (a)
$cubrid_put ($con, $oid, array("id"=>1, "name"=>"Tomas")); // (b)
```

Using Collections

- Collection data types can be used by using either PHP array data types or PHP functions that support array data types. The following is an example of fetching the query result with the cubrid_fetch() function.

```
$row = cubrid_fetch ($req);
$col = $row["customer"];
while (list ($key, $cust) = each ($col)) {
echo $cust;
}
```

- You can also get values of collection attributes. The following is an example of getting collection attribute values with the cubrid_col_get() function.

```
$tels = cubrid_col_get ($con, $oid, "tels");
while (list ($key, $tel) = each ($tels)) {
echo $tel."\n";
}
```

- You can directly update collection type values with cubrid_set_add() and cubrid_set_drop() functions.

```
$tels = cubrid_col_get ($con, $oid, "tels");
while (list ($key, $tel) = each ($tels)) {
$res = cubrid_set_drop ($con, $oid, "tel", $tel);
}
cubrid_commit ($con);
```

cubrid_affected_rows

Description

This function is used to get the number of rows that have been affected by the SQL statements (**INSERT**, **DELETE**, **UPDATE**).

Syntax

```
int cubrid_affected_rows (int req_handle)
```

- *req_handle* : Request handle

Return Value

- Success : Returns the number of rows affected by the SQL statement.
- Failure : -1

Example

```
$req = cubrid_execute ($con, "delete from person where name like 'j%' ");
if ($req) {
   $row_count = cubrid_affected_rows ($req);
   echo $row_count;
   cubrid_close_request ($req);
}
```

See Also

- cubrid_execute

cubrid_bind

Description

This functiont is used to substitute a value for a variable of the cubrid_prepare with parameters. The following table shows the types of substitute values.

Support	Bind type	Corresponding SQL type
Supported	STRING	CHAR, VARCHAR
	NCHAR	NCHAR, NVARCHAR
	BIT	BIT, VARBIT
	NUMERIC or NUMBER	SHORT, INT, NUMERIC
	FLOAT	FLOAT
	DOUBLE	DOUBLE
	TIME	TIME
	DATE	DATE
	TIMESTAMP	TIMESTAMP
	OBJECT	OBJECT
	NULL	NULL
Not supported	SET	SET
	MULTISET	MULTISET
	SEQUENCE	SEQUENCE

Syntax

```
int cubrid_bind (int req_handle,int bind_index,string bind_value [,string bind_value_type])
```

- Request handle obtained as the result of *req_handle* : cubrid_prepare
- *bind_index* : Binding location
- *bind_ value* : Actual value to be bound
- *bind_value_type* : Type of the value to be bound. It can be omitted by default. If it is omitted, the type is automatically cast to an appropriate one. However, **NCHAR** and **BIT** types must be passed as arguments.

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
$con = cubrid_connect ("dbsvr.cubrid.com", 12345, "demodb");
if ($con) {
$sql = "insert into tbl values ( ?,?,?)";
    $req = cubrid_prepare( $con, $sql, CUBRID_INCLUDE_OID );
    $i = 0;
    while ( $i < 2 ) {
    $res = cubrid_bind( $req, 1, "1", "NUMBER");
```

```
   $res = cubrid_bind( $req, 2, "2");
   $res = cubrid_bind( $req, 3, "04:22:34 PM 08/07/2007");
   $res = cubrid_execute( $req );
   $i = $i + 1;
}}
```

See Also

- cubrid_execute
- cubrid_prepare

cubrid_close_request

Description

This function is used to close the request handle given to the *req_handle* parameter and releases the memory area related to the handle.

Syntax

```
int cubrid_close_request (int req_handle)
```

- *req_handle* : Request handle

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
$con = cubrid_connect ("dbsvr.cubrid.com", 12345, "demodb");
if ($con) {
   echo "connected successfully";
   $req = cubrid_execute ( $con, "select * from members",
                           CUBRID_INCLUDE_OID | CUBRID_ASYNC);
   if ($req) {
      while ( list ($id, $name) = cubrid_fetch ($req) ){
         echo $id;
         echo $name;
      }
      cubrid_close_request($re1);
   }
   cubrid_discommect($con);
}
```

See Also

- cubrid_execute

cubrid_col_get

Description

This function is used to get the elements of the given collection type (set, multiset, sequence) attribute in the form of an array.

Syntax

```
int cubrid_col_get(int conn_handle, string oid, string attr_name)
```

- *conn_handle* : Connection handle
- *oid* : OID of the desired instance
- *attr_name* : Name of the attribute to be read from the instance

Return Value

- Success : An array that contains the desired elements (0 : default numeric array)
- Failure : FALSE. If an error occurs, a warning message is displayed to distinguish it from a collection without attributes or **NULL**. You can check the error with cubrid_error_code.

Example

```
$elem_array = cubrid_col_get ($con, $oid, "tel");
while (list ($key, $val) = each ($elem_array)) {
   echo "tel: $val\n";
}
```

cubrid_col_size

Description

This function is used to get the number of elements of a collection type (set, multiset, sequence) attribute.

Syntax

```
int cubrid_col_size(int conn_handle, string oid, string attr_name)
```

- *conn_handle* : Connection handle
- *oid* : OID of the desired instance
- *attr_name* : Name of the desired attribute of the instance

Return Value

- Success : The number of elements
- Failure : -1

Example

```
$elem_count = cubrid_col_size ($con, $oid, "tel");
echo "$oid (tel) has $elem_count elements\n";
```

cubrid_column_names

Description

This function is used to get column names in the query results by using *req_handle*.

Syntax

```
mixed cubrid_column_names (int req_handle)
```

- *req_handle* : Request handle

Return Value

- Success : An array that contains the column names
- Failure : FALSE

Example

```
$req = cubrid_execute ($con, "select * from person");
if ($req) {
   $coltypes = cubrid_column_types ($req);
   $colnames = cubrid_column_names ($req);
   while (list ($key, $coltype) = each ($coltypes))
      echo $coltype;
   while (list ($key, $colname) = each ($colnames))
      echo $colname;
   cubrid_close_request ($req);
}
```

See Also

- cubrid_execute
- cubrid_prepare
- cubrid_column_types

cubrid_column_types

Description

This function is used to get column types in the query results by using *req_handle*.

Syntax

```
mixed cubrid_column_types (int req_handle)
```

- *req_handle* : Request handle

Return Value

- Success : An array that contains the column types
- Failure : FALSE

Example

```
$req = cubrid_execute ($con, "select * from person");
if ($req) {
   $coltypes = cubrid_column_types ($req);
   $colnames = cubrid_column_names ($req);
   while (list ($key, $coltype) = each ($coltypes))
      echo $coltype;
   while (list ($key, $colname) = each ($colnames))
      echo $colname;
   cubrid_close_request ($req);
}
```

See Also

- cubrid_execute
- cubrid_prepare
- cubrid_column_names

cubrid_commit

Description

This function is used to commit transactions being performed currently in the connection referred to by *conn_handle*. The connection with the server is terminated after the **cubrid_commit** function is called, but the connection handle remains valid.

Syntax

```
int cubrid_commit (int conn_handle)
```

- *conn_handle* : Connection handle

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
$req = cubrid_execute ($oid, "insert into person values
    (2,'John')");
if ($req) {
   cubrid_close_request ($req);
   if ($failed) {
      cubrid_rollback ($con);
  } else {
      cubrid_commit ($con);
   }
}
```

See Also

- cubrid_rollback

cubrid_connect

Description

This function is used to configure the connection environment with the server by using the given information such as the server address, port number, database name, user name and password. If the user name and password are not set, **PUBLIC** is used as default.

Syntax

```
int cubrid_connect (string host, int port, string dbname[, string userid[, string passwd]])
```

- *host* : IP address and host name of the Broker Server
- *port* : Port number of the Broker Server
- *dbname* : Database name
- *userid* : Database user name
- *passwd* : Database user password

Return Value

- Success : Connection handle
- Failure : FALSE

Example

```
$con = cubrid_connect ("210.211.133.100", 12345, "demodb");

if ($con) {
   echo "connected successfully";
   $req =cubrid_execute($con, "insert into person values
   (1,'James')");

   if ($req) {
      cubrid_close_request ($req);
      cubrid_commit ($con);
   } else {
      cubrid_rollback ($con);
   }
   cubrid_disconnect ($con);
}
```

See Also

- cubrid_disconnect

cubrid_connect_with_url

Description

cubrid_connect_with_url tries to connect a database by using connection information passed with an url string argument. If the HA feature is enabled in PHP, you must specify connection information of the active server and connection information of the standby server, which is used for failover when failure occurs, in the *url* string argument of this function. If it has succeeded, the ID of connection handle is returned; if it fails, an error code is returned.

Syntax

```
int cubrid_connect_with_url (char *url [, char *db_user, char *db_password ])

<url> ::=
cci:cubrid:<host>:<db_name>:<db_user>:<db_password>:[?<properties>]

<properties> ::= <property> [&<property>]
<property> ::= althosts=<alternative_hosts> [&rctime=<time>]
<alternative_hosts> ::= <standby_broker1_host>:<port> [,<standby_broker2_host>:<port>]

<host> := HOSTNAME | IP_ADDR
<time> := SECOND
```

- *url* : (IN) A character string that contains server connection information
- *host* : A host name or IP address of the master database
- *db_name* : A name of the database
- *db_user* : A name of the database user
- *db_password* : A database user password
- **althosts** =*standby_broker1_host, standby_broker2_host,* . . . : Specifies the broker information of the standby server, which is used for failover when it is impossible to connect to the active server. You can specify multiple brokers for failover, and the connection to the brokers is attempted in the order listed in **alhosts**.
- **rctime** : An interval between the attempts to connect to the active broker in which failure occurred. After a failure occurs, the system connects to the broker specified by **althosts** (failover), terminates the transaction, and then attempts to connect to the active broker of the master database at every **rctime**. The default value is 600 seconds.
- *db_user* : (IN) A name of the database user
- *db_passwd* : (IN) A database user password

Return Value

- Success : Connection handle ID (int)
- Failure : Error code

cubrid_current_oid

Description

This function is used to get the OID of the current cursor position from the query results. To use **cubrid_current_oid**, an updatable query must be executed with the **CUBRID_INCLUDE_OID** option.

Syntax

```
mixed cubrid_current_oid (int req_handle)
```

- *req_handle* : Request handle

Return Value

- Success : OID of the current cursor position
- Failure : FALSE

Example

```
$req =cubrid_execute($con,"select * from person where id =1",
                           CUBRID_INCLUDE_OID);
if ($req) {
   cubrid_fetch ($req);
   $oid = cubrid_current_oid ($req);
   cubrid_close_request ($req);
   echo "OID is $oid";
}
```

See Also

- cubrid_execute

cubrid_data_seek

Description

cubrid_data_seek moves the internal row pointer of the CUBRID result associated with the specified result identifier to point to the specified *row* number. The next call to a CUBRID fetch function, such as cubrid_fetch_assoc(), would return that row.

Syntax

```
bool cubrid_data_seek (int $result, int $row_number)
```

- *result* : The result handle that is being evaluated. This result comes from a call to cubrid_execute().
- *row_number* : The desired row number of the new result pointer

Return Value

- Success : true
- Failure : false

Example

```
<?php
$link = cubrid_connect('localhost', 8080, ‘cubrid_database’, 'cubrid_user',
'cubrid_password');
if (!$link) {
    die('Could not connect: ' , cubrid_error_msg ());
}
$query = 'SELECT last_name, first_name FROM friends';
$result = cubrid_execute($link, $query);
if (!$result) {
    die('Query failed: ' . cubrid_error_msg ());
}
/* fetch rows in reverse order */
for ($i = cubrid_num_rows ($result) - 1; $i >= 0; $i--) {
    if (!cubrid_data_seek($result, $i)) {
        echo "Cannot seek to row $i: " . cubrid_error_msg () . "\n";
        continue;
    }

    if (!($row = cubrid_fetch_assoc($result))) {
        continue;
    }

    echo $row['last_name'] . ' ' . $row['first_name'] . "<br />\n";
}
?>
```

cubrid_disconnect

Description

This function is used to stop transactions currently being executed, terminates the connection with the server and closes the connection handle. All request handles that are still open will be closed.

Syntax

```
int cubrid_disconnect (int conn_handle)
```

- *conn_handle* : Connection handle

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
$con = cubrid_connect ("210.211.133.100", 12345, "demodb");
if ($con) {
   echo "connected successfully";
   $req = cubrid_execute ( $con, "insert into person values(1,'James')");
   if ($req) {
      cubrid_close_request ($req);
      cubrid_commit ($con);
   } else {
      cubrid_rollback ($con);
   }
   cubrid_disconnect ($con);
}
```

See Also

- cubrid_connect

cubrid_drop

Description

This function is used to drop the desired instance from the database by using the OID.

Syntax

```
int cubrid_drop (int conn_handle, string oid)
```

- *conn_handle* : Connection handle
- oid : OID of the instance to be deleted

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
$deloid = cubrid_get ($con, $oid, "order");
$res = cubrid_drop ($con, $deloid);
if ($res) {
   cubrid_commit ($con);
} else {
   cubrid_rollback ($con);
}
```

See Also

- cubrid_is_instance

cubrid_error_code

Description

This function is used to get the code of the error that occurred during the API execution. Usually, the error message can be fetched when the API returns **FALSE**.

Syntax

```
int cubrid_error_code ()
```

Return Value

- Error code

Example

```
$req = cubrid_execute ($con, "select id, name from person");
if ($req) {
   while (list ($id, $name) = cubrid_fetch($req))
       echo $id, $name;
} else {
    echo "Error Code: ", cubrid_error_code ();
    echo "Error Facility: ", cubrid_error_code_facility ();
    echo "Error Message: ", cubrid_error_msg ();
}
```

See Also

- cubrid_error_code_facility
- cubrid_error_msg

cubrid_error_code_facility

Description

This function is used to get a facility code (level at which the error occurred) from the code of the error that occurred during the API execution. Usually, the error code can be fetched when the API returns **FALSE**.

Syntax

```
int cubrid_error_code_facility ()
```

Return Value

- Facility code of the occurred error code :
  CUBRID_FACILITY_DBMS, CUBRID_FACILITY_CAS,
  CUBRID_FACILITY_CCI, CUBRID_FACILITY_CLIENT

Example

```
$req = cubrid_execute ($con, "select id, name from person");
if ($req) {
   while (list ($id, $name) = cubrid_fetch($req))
      echo $id, $name;
} else {
   echo "Error Code: ", cubrid_error_code ();
   echo "Error Facility: ", cubrid_error_code_facility ();
   echo "Error Message: ", cubrid_error_msg ();
}
```

See Also

- cubrid_error_code
- cubrid_error_msg

cubrid_error_msg

Description

This function is used to get the error message that occurred during the API execution. Usually, the error message can be fetched when the API returns **FALSE**.

Syntax

```
string cubrid_error_msg ()
```

Return Value

- Occurred error message

Example

```
$req = cubrid_execute ($con, "select id, name from person");
if ($req) {
  while (list ($id, $name) = cubrid_fetch($req))
    echo $id, $name;
} else {
  echo "Error Code: ", cubrid_error_code ();
  echo "Error Facility: ", cubrid_error_code_facility ();
  echo "Error Message: ", cubrid_error_msg ();
}
```

See Also

- cubrid_error_code
- cubrid_error_code_facility

cubrid_execute

Description

This function is used to execute a given SQL statement. It executes a query by using *conn_handle* and SQL and then returns the request handle created. This is an appropriate way to simply execute a query when parameter binding is not necessary.

cubrid_execute is also used when executing **Prepared Statement** with cubrid_prepare and cubrid_bind. In this case, required parameters are *request_handle* and *option*.

The *option* parameter is used to determine whether to get OID after query execution and whether to execute the query in asynchronous mode. **CUBRID_INCLUDE_OID** and **CUBRID_ASYNC** can be specified by using a bitwise OR operator (|). If not specified, neither of them are selected.

If *request_handle* is the first argument for the execution of cubrid_prepare, only CUBRID_ASYNC can be used as an option.

Syntax

```
int cubrid_execute (int conn_handle, string SQL [, int option])
```

- *conn_handle* : Connection handle
- *SQL* : SQL statement to be executed
- *option* : Query execution option - CUBRID_INCLUDE_OID, CUBRID_ASYNC

```
int cubrid_execute (int request_handle[, int option])
```

- *request _handle* : **cubrid_prepare** handle
- *option* : Query execution option - CUBBRID_ASYNC

Return Value

- Success : Request handle
- Failure : FALSE

Example

```
$con = cubrid_connect ("dbsvr.cubrid.com", 33000, "demodb");

if ($con) {    echo "connected successfully";

   $req = cubrid_execute ( $con, "select * from members",
                            CUBRID_INCLUDE_OID |
                            CUBRID_ASYNC);    if ($req) {
while ( list ($id, $name) = cubrid_fetch ($req) ){   echo $id;
       echo $name;
}
  cubrid_close_request ($req); }   cubrid_disconnect ($con);
} $con = cubrid_connect ("dbsvr.cubrid.com", 33000, "demodb");

if ($con) {  echo "connected successfully";

$sql = "insert into tbl values ( ?,?,?)";  $req = cubrid_prepare
       ( $con, $sql, CUBRID_INCLUDE_OID );

 $i = 0;
 while ( $i < 2 ) {
 $res = cubrid_bind( $req, 1, "1", "NUMBER");
 $res = cubrid_bind( $req, 2, "2");
 $res = cubrid_bind( $req, 3, "04:22:34 PM 08/07/2007");
 $res = cubrid_execute( $req );
 $i = $i + 1;
}}
```

See Also

- cubrid_close_request
- cubrid_commit
- cubrid_rollback
- cubrid_prepare
- cubrid_bind

cubrid_fetch

Description

This function is used to fetch one row from the query result. After the fetch, the cursor automatically moves to the next row.

Syntax

```
mixed cubrid_fetch (int req_handle[, int type])
```

- *req_handle* : Request handle
- *type* : Type of the result array to be fetched
  CUBRID_NUM, CUBRID_ASSOC,
  CUBRID_BOTH, CUBRID_OBJECT

Return Value

- Success : Result array or object.

It is determined by the *type* parameter. If the *type* parameter is omitted, **CUBRID_BOTH** is used. If you want to get the query result as an object data type, column names must comply with identifier name rules allowed in PHP. For example, a column name "count(*)" cannot be fetched and used as an object type.

The following are different result types depending on *type*.

- CUBRID_NUM : Numeric array (0-default)
- CUBRID_ASSOC : Associative array
- CUBRID_BOTH : Numeric and associative arrays (default value)
- CUBRID_OBJECT : An object that has the attribute whose name is the same as the column name of the query result

- Failure : FALSE

Example

```
$req = cubrid_execute ( $con, "select * from members",
                               CUBRID_INCLUDE_OID | CUBRID_ASYNC);
if ($req) {
   while ( list ($id, $name) = cubrid_fetch ($req) ){
      echo $id;
      echo $name;
   }
   cubrid_close_request ($req);
}
$req = cubrid_execute ($con, "select * from teams");
if ($req) {
   while ($row = cubrid_fetch ($req, CUBRID_OBJECT)) {
      echo $row->id;
      echo $row->name;
   }
}
```

See Also

- cubrid_execute

cubrid_fetch_assoc

Description

cubrid_fetch_assoc returns an associative array that corresponds to the fetched row and moves the internal data pointer ahead or **FALSE** if there are no more rows.

Syntax

```
array cubrid_fetch_assoc ( int $result )
```

- *result* : The result handle that is being evaluated. This result comes from a call to cubrid_execute.

Example

```
<?php
$query = 'SELECT id,email FROM people WHERE id = '42'';
$result = cubrid_execute($link, $query);
if (!$result) {
    die('Query failed: ' . cubrid_error_msg ());
}
$row= cubrid_fetch_assoc($result);
$lengths = cubrid_fetch_lengths($result);

print_r($row);
print_r($lengths);
?>

Output:
Array
(
    [id] => 42
    [email] => user@example.com
)
Array
(
    [0] => 2
    [1] => 16
)
```

cubrid_fetch_field

Description

cubrid_fetch_field returns an object containing field information. This function can be used to obtain information about fields in the provided query result. The properties of the object are:

- name : Column name
- table : Name of the table where the column belongs
- def : Default value of the column
- max_length : Maximum length of the column
- not_null : 1 if the column cannot be **NULL**
- unique_key : 1 if the column is a unique key
- multiple_key : 1 if the column is a non-unique key
- numeric : 1 if the column is numeric
- type : The type of the column

Syntax

```
object cubrid_fetch_field ( int $result [, int $field_offset= 0 ] )
```

- *result* : The result handle that is being evaluated. This result comes from a call to cubrid_execute.
- *field_offset* : The numerical field offset. If the field offset is not specified, the next field that was not yet retrieved by this function is retrieved. The field_offset starts at 0.

Example

```
< ?php
$link = cubrid_connect('localhost', 8080, ‘cubrid_database’, 'cubrid_user',
'cubrid_password')
if (!$link) {
        die('Could not connect: ' , cubrid_error_msg ())
}
$query = 'SELECT last_name, first_name FROM friends'
$result = cubrid_execute($link, $query)
if (!$result) {
        die('Query failed: ' . cubrid_error_msg ())
}
/* fetch rows in reverse order */
$i = 0
while ($i < cubrid_num_fields($result)) {
        echo "Information for column $i:< br /> \n"
        $meta = cubrid_fetch_field($result, $i)
        if (!$meta) {
                echo "No information available< br /> \n"
        }
        echo "< pre>
                max_length:          $meta-> max_length
                multiple_key:      $meta-> multiple_key
                name:                      $meta-> name
                not_null:              $meta-> not_null
                numeric:                $meta-> numeric
                table:            $meta-> table
                type:                      $meta-> type
                default:                $meta-> def
                unique_key:          $meta-> unique_key
< /pre> "
```

```
        $i++
}
?>
```

cubrid_fetch_lengths

Description

cubrid_fetch_lengths returns an array that corresponds to the lengths of each field in the last row fetched by CUBRID or **FALSE** on failure.

Syntax

```
array cubrid_fetch_lengths ( int $result )
```

- *result* : The result handle that is being evaluated. This result comes from a call to cubrid_execute.

Example

```
<?php
$query = 'SELECT id,email FROM people WHERE id = '42'';
$result = cubrid_execute($link, $query);
if (!$result) {
    die('Query failed: ' . cubrid_error_msg ());
}
$row= cubrid_fetch_assoc($result);
$lengths = cubrid_fetch_lengths($result);

print_r($row);
print_r($lengths);
?>

Output:
Array
(
    [id] => 42
    [email] => user@example.com
)
Array
(
    [0] => 2
    [1] => 16
)
```

cubrid_fetch_object

Description

cubrid_fetch_object returns an object with properties that correspond to the fetched row and moves the internal data pointer ahead or **FALSE** if there are no more rows. cubrid_fetch_row() fetches one row of data from the result associated with the specified result identifier. The row is returned as an array. Each result column is stored in an array offset, starting at offset 0.

Syntax

```
object cubrid_fetch_object ( int $result )
```

- *result* : The result handle that is being evaluated. This result comes from a call to cubrid_execute.

Example

```
<?php
$link = cubrid_connect('localhost', 8080, ‘cubrid_database’, 'cubrid_user',
'cubrid_password');
if (!$link) {
    die('Could not connect: ' , cubrid_error_msg ());
}
```

```
$result = cubrid_execute($link , "select * from mytable");
while ($row = cubrid_fetch_object($result)) {
    echo $row->user_id;
    echo $row->fullname;
}
?>
```

cubrid_fetch_row

Description

cubrid_fetch_row returns a numerical array that corresponds to the fetched row and moves the internal data pointer ahead or **FALSE** if there are no more rows. **cubrid_fetch_row()** fetches one row of data from the result associated with the specified result identifier. The row is returned as an array. Each result column is stored in an array.

Syntax

```
array cubrid_fetch_row ( int $result )
```

- *result* : The result handle that is being evaluated. This result comes from a call to cubrid_execute.

Example

```
<?php
$link = cubrid_connect('localhost', 8080, cubrid_database, 'cubrid_user',
'cubrid_password');
if (!$link) {
    die('Could not connect: ' , cubrid_error_msg ());
}
$result = cubrid_execute($link , "SELECT id,email FROM people WHERE id = '42'");
if (!$result) {
    echo 'Could not run query: ' . cubrid_error_msg ();
    exit;
}
$row = cubrid_fetch_row($result);

echo $row[0]; // 42
echo $row[1]; // the email value
?>
```

cubrid_field_flags

Description

cubrid_field_flags () returns the field flags of the specified field. The flags are reported as a single word per flag separated by a single space, so that you can split the returned value using **explode()**.

Syntax

```
string cubrid_field_flags ( int $result , int $field_offset )
```

- *result* : The result handle that is being evaluated. This result comes from a call to cubrid_execute.
- *field_offset* : The field_offset starts at 0.

Example

```
<?php
$result = cubrid_execute($link, "SELECT id,email FROM people WHERE id = '42'");
if (!$result) {
    echo 'Could not run query: ' .cubrid_error_msg();
    exit;
}
$flags = cubrid_field_flags($result, 0);

echo $flags;
print_r(explode(' ', $flags));
?>
```

```
Array
(
    [0] => not_null
    [1] => primary_key
    [2] => auto_increment
)
```

cubrid_field_len

Description

cubrid_field_len returns the length of the specified field on success, or **FALSE** on failure.

Syntax

```
string cubrid_field_len ( int $result , int $field_offset )
```

- *result* : The result handle that is being evaluated. This result comes from a call to cubrid_execute.
- *field_offset* : The numerical field offset. The field_offset starts at 0. If field_offset does not exist, an error occurs.

Example

```
<?php
$result = cubrid_execute($link, "SELECT id,email FROM people WHERE id = '42'");
if (!$result) {
echo 'Could not run query: ' .cubrid_error_msg();
exit;
}
// Will get the length of the id field as specified in the database
// schema.
$length = cubrid_field_len($result, 0);
echo $length;
?>
```

cubrid_field_name

Description

cubrid_field_name returns the name of the specified field index on success, or **FALSE** on failure.

Syntax

```
string cubrid_field_name ( int $result , int $field_offset )
```

- *result* : The result handle that is being evaluated. This result comes from a call to cubrid_execute.
- *field_offset* : The numerical field offset. The field_offset starts at 0. If field_offset does not exist, an error occurs.

Example

```
<?php
$result = cubrid_execute($link, "select * from users");
if (!$result) {
echo 'Could not run query: ' .cubrid_error_msg();
exit;
}

echo mysql_field_name($result, 0) . "\n";
echo mysql_field_name($result, 2);
?>

Output:
user_id
password
```

cubrid_field_seek

Description

cubrid_field_seek sets a field offset value to be used in cubrid_fetch_field function. If the **cubrid_fetch_field** function that does not include a field offset is called, the field offset specified in this function is returned.

Syntax

```
string cubrid_field_seek ( int $result , int $field_offset )
```

- *result* : The result handle that is being evaluated. This result comes from a call to cubrid_execute api_php_execute.htm.
- *field_offset* : The numerical field offset. The field_offset starts at 0. If field_offset does not exist, an error occurs.

Return Value

- Success : true
- Failure : false

Example

```
<?php
$result = cubrid_execute($link, "select * from users");
if (!$result) {
echo 'Could not run query: ' .cubrid_error_msg();
exit;
}

$row = cubrid_fetch_array($result);
echo $row['ID'] . ' ' . $row['Name'];
cubrid_field_seek($result,2);
echo $row['ID'] . ' ' . $row['Name'];
?>
```

cubrid_field_table

Description

cubrid_field_table returns the name of the table that the specified field is in.

Syntax

```
string cubrid_field_table ( int $result , int $field_offset )
```

- *result* : The result handle that is being evaluated. This result comes from a call to cubrid_execute.
- *field_offset* : The numerical field offset. The field_offset starts at 0. If field_offset does not exist, an error occurs.

Example

```
<?php
$query = "SELECT account.*, country.* FROM account, country WHERE country.name =
'Portugal' AND account.country_id = country.id";

$result = cubrid_execute($link, $query);
if (!$result) {
echo 'Could not run query: ' .cubrid_error_msg();
exit;
}

// Lists the table name and then the field name
for ($i = 0; $i < cubrid_num_fields($result); ++$i) {
$table = cubrid_field_table($result, $i);
$field = cubrid_field_name($result, $i);

echo  "$table: $field\n";
}
```

```
?>
```

cubrid_field_type

Description

cubrid_field_type is similar to the cubrid_field_name() function. The arguments are identical, but the field type is returned instead. The returned field type will be one of "int", "real", "string", etc.

Syntax

```
string cubrid_field_type ( int $result , int $field_offset )
```

- *result* : The result handle that is being evaluated. This result comes from a call to cubrid_execute.
- *field_offset* : The numerical field offset. The field_offset starts at 0. If field_offset does not exist, an error occurs.

Example

```
<?php
$query = "SELECT * FROM func";

$result = cubrid_execute($link, $query);
if (!$result) {
echo 'Could not run query: ' .cubrid_error_msg();
exit;
}

$fields = cubrid_num_fields($result);
$rows   = cubrid_num_rows($result);
$table  = cubrid_field_table($result, 0);
echo "Your '" . $table . "' table has " . $fields . " fields and " . $rows . " 
record(s)\n";
echo "The table has the following fields:\n";
for ($i=0; $i < $fields; $i++) {
$type  = cubrid_field_type($result, $i);
$name  = cubrid_field_name($result, $i);
$len   = cubrid_field_len($result, $i);
$flags = cubrid_field_flags($result, $i);
echo $type . " " . $name . " " . $len . " " . $flags . "\n";
}
?>

Output:
Your 'func' table has 4 fields and 1 record(s)
The table has the following fields:
string name 64 not_null primary_key binary
int ret 1 not_null
string dl 128 not_null
string type 9 not_null enum
```

cubrid_free_result

Description

cubrid_free_result Frees result memory. Returns TRUE on success or **FALSE** on failure.

Syntax

```
bool cubrid_free_result ( int $result )
```

- *result* : The result handle that is being evaluated. This result comes from a call to cubrid_execute.

Example

```
<?php
$result = cubrid_execute("SELECT id,email FROM people WHERE id = '42'");
/* Use the result, assuming we're done with it afterwards */
```

```
$row = cubrid_fetch_assoc($result);

/* Now we free up the result and continue on with our script */
cubrid_free_result($result);

echo $row['id'];
echo $row['email'];
?>
```

cubrid_get

Description

This function is used to get a desired attribute of an instance by using OID. You can get a single attribute by using a character string type for the *attr* argument, or multiple attributes by using an array type.

Syntax

```
mixed cubrid_get (int conn_handle, string oid[, mixed attr])
```

- *conn_handle* : Connection handle
- *oid* : OID of the instance whose value you want to get
- *attr* : Name of the attribute whose value you want to get

Return Value

A character string is returned if a character string type is set for the *attr* argument; an associative array is returned if an array type (0 - default numeric array) is set. If the *attr* argument is omitted, all attributes of the instance are returned as an associative array.

- Success : Content of the attribute(s) requested
- Failure : FALSE. If an error occurs, a warning message is displayed to distinguish it from an empty character string or **NULL**. You can check the error with cubrid_error_code.

Example

```
$attrarray = cubrid_get ($con, $oid);
echo $attrarray["id"];
echo $attrarray["name"];
```

See Also

- cubrid_put

cubrid_get_charset

Description

cubrid_get_charset gets CUBRID current connection charset.

Syntax

```
string cubrid_get_charset ( void )
```

Example

```
<?php
printf("current charset: %s\n", cubrid_get_charset($req));
?>
```

cubrid_get_class_name

Description

This function is used to get a class name from an OID.

Syntax

```
mixed  cubrid_is_instance (int conn_handle, string oid)
```

- *conn_handle* : Connection handle
- *oid* : OID of an instance, for which you want to check whether it exists

Return Value

- Success : Class name
- Failure : FALSE

Example

```
$target_oid = cubrid_get ($con, $oid, "customer");
$class_name = cubrid_get_class_name ($con, $target_oid);
if ($class_name) {
  echo "class name of $oid is $class_name\n";
} else {
  echo "error\n";
}
```

See Also

- cubrid_is_instance
- cubrid_drop

cubrid_get_client_info

Description

cubrid_get_client_info() returns a string that represents the cci library version.

```
string cubrid_get_client_info ( void )
```

Example

```
<?php
printf("CUBRID client info: %s\n", cubrid_get_client_info());
?>
```

cubrid_get_db_parameter

Description

cubrid_get_server_info() returns a string that represents the CUBRID server version.

Syntax

```
array cubrid_get_db_parameter ( $req )
```

Example

```
<?php
printf("CUBRID DB parameters: ");
printf_r(cubrid_get_db_parameter($req));
?>
```

cubrid_get_server_info

Description

cubrid_get_server_info() returns a string that represents the CUBRID server version.

Syntax

```
string cubrid_get_server_info ( void )
```

Examples

```
<?php
printf("CUBRID server info: %s\n", cubrid_get_server_info());
?>
```

cubrid_insert_id

Description

cubrid_insert_id retrieves the IDs specified or generated for the **AUTO_INCREMENT** columns updated by the previous **INSERT** query. It returns an array with all the **AUTO_INCREMENT** columns and their values. It returns 0 if the previous query does not generate new rows, or **FALSE** on failure.

Syntax

```
array cubrid_insert_id ( string $class_name [, int $connection_handle] )
```

- *class_name* : The name of the class (table) that was used in the last **INSERT** statement for which the auto increment values are retrieved.
- *connection_handle* : The connection handle previously obtained from a call to cubrid_connect().

Example

```
<?php
$result = cubrid_execute($link, "INSERT INTO mytable (product) values ('kossu')");
if (!$result) {
echo 'Could not run query: ' .cubrid_error_msg();
exit;
}
print_r(cubrid_insert_id("mytable"));
?>
```

cubrid_is_instance

Description

This function is used to check whether an instance referred to by an OID exists in the database.

Syntax

```
int cubrid_is_instance (int conn_handle, string oid)
```

- *conn_handle* : Connection handle
- *oid* : OID of an instance, for which you want to check whether it exists

Return Value

- 1 : An instance exists.
- 0 : An instance does not exist.
- -1 : An error occurs.

Example

```
$target_oid = cubrid_get ($con, $oid, "customer");
$res = cubrid_is_instance ($con, $target_oid);
if ($res == 1) {
   echo "$oid is presents.\n";
} else if ($res == 0){
   echo "$oid is not presents.\n";
} else {
   echo "error\n";
}
```

See Also

- cubrid_drop
- cubrid_get_class_name

cubrid_list_dbs

Description

cubrid_list_dbs gets CUBRID server info. cubrid_get_server_info() returns a string that represents the CUBRID server version.

Syntax

```
array cubrid_list_dbs ( )
```

Example

```
<?php
printf(“CUBRID Databases: “);
printf_r(cubrid_list_dbs());
?>
```

cubrid_lock_read

Description

This function is used to configure a read lock on the given instance by using an OID.

Syntax

```
int cubrid_lock_read (int conn_handle, string oid)
```

- *conn_handle* : Connection handle
- *oid* : OID of an instance on which you want to configure a lock

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
$lock_oid = cubrid_get ($con, $oid, "next_id");
$res = cubrid_lock_read ($con, $lock_oid);
```

See Also

- cubrid_lock_write

cubrid_lock_write

Description

This function is used to configure a write lock on the given instance using an OID.

Syntax

```
int cubrid_lock_write (int conn_handle, string oid)
```

- *conn_handle* : Connection handle
- *oid* : OID of an instance on which you want to configure a lock

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
$lock_oid = cubrid_get ($con, $oid, "next_id");
$res = cubrid_lock_write ($con, $lock_oid);
```

See Also

- cubrid_lock_read

cubrid_move_cursor

Description

This function is used to move the current cursor position of req_handle to the distance configured by the offset argument in the direction in the origin argument. For origin, the first position in the result (**CUBRID_CURSOR_FIRST**), the current position in the result (**CUBRID_CURSOR_CURRENT**) and the last position in the result (**CUBRID_CURSOR_LAST**) can be used. If origin is not specified, **CUBRID_CURSOR_CURRENT** is used by default.

If the amount of cursor movement exceeds the range of the result, the cursor moves to a position next to the end of the result range. For example, if the cursor moves to the position 20 when the size of the result is 10, it moves to the 11th position and returns **CUBRID_NO_MORE_DATA**.

Syntax

```
int cubrid_move_cursor (int req_handle, int offset[, int origin])
```

- *req_handle* : Request handle
- *offset* : The number of positions to which the cursor is to be moved
- *origin* : Origin of the cursor movement
  CUBRID_CURSOR_FIRST,
  CUBRID_CURSOR_CURRENT,
  CUBRID_CURSOR_LAST

Return Value

- CUBRID_CURSOR_SUCCESS
- Failure : CUBRID_ NO_MORE_DATA

Example

```
cubrid_move_cursor ($req_handle, 1, CUBRID_CURSOR_FIRST);
// Move the cursor to the first position.
$row = cubrid_fetch ($req_handle);
echo $row["id"], $row["name"];
```

```
cubrid_move_cursor ($req_handle, 1, CUBRID_CURSOR_LAST);
// Move the cursor to the last position.
$row = cubrid_fetch ($req_handle);
echo $row["id"], $row["name"];
```

See Also

- cubrid_execute

cubrid_num_cols

Description

This function is used to return the number of columns in the query result. This method is available only with the **SELECT** statement.

Syntax

```
int cubrid_num_cols (int req_handle)
```

- *req_handle* : Request handle

Return Value

- Success : The number of columns
- Error occurs : -1

Example

```
$req = cubrid_execute ($con, "select * from member");
if ($req) {
  $rows_count = cubrid_num_rows ($req);
  $cols_count = cubrid_num_cols ($req);
  echo "result set rows count : $rows\n";
  echo "result set columns count : $cols\n";
  cubrid_close_request ($req);
}
```

See Also

- cubrid_execute
- cubrid_num_rows

cubrid_num_fields

Description

cubrid_num_fields returns the number of fields in the result set resource on success, or **FALSE** on failure.

Syntax

```
int cubrid_num_fields ( int $result )
```

- *result* : Specifies the result handle. This result comes from a call to cubrid_execute.

Example

```
<?php
$result = cubrid_execute($link, "SELECT id,email FROM people WHERE id = '42'");
if (!$result) {
echo 'Could not run query: ' . cubrid_error_msg();
exit;
}

/* returns 2 because id,email === two fields */
```

```
echo cubrid_num_fields($result);
?>
```

cubrid_num_rows

Description

This function is used to return the number of rows in the query result. This is available only with the **SELECT** statement. Use cubrid_affected_rows if you want to know the results of **INSERT**, **UPDATE** and **DELETE** queries. **cubrid_num_rows** can be used only with synchronous queries. It returns 0 if the query is asynchronous.

Syntax

```
int cubrid_num_rows (int req_handle)
```

- *req_handle* : Request handle

Return Value

- Success : The number of rows
- 0 : Asynchronous query
- -1 : Error occurs

Example

```
$req = cubrid_execute ($con, "select * from member");
if ($req) {
   $rows_count = cubrid_num_rows ($req);
   $cols_count = cubrid_num_cols ($req);
   echo "result set rows count : $rows\n";
   echo "result set columns count : $cols\n";
   cubrid_close_request ($req);
}
```

See Also

- cubrid_execute
- cubrid_num_cols
- cubrid_affected_rows

cubrid_prepare

Description

This function is an API that represents a precompiled SQL statement on the given connection handle. The SQL statement is precompiled and then included in **cubrid_prepare**. This method can be used to efficiently execute the statement multiple times or to effectively process Long Data. You can use only a single statement and a parameter can insert a question mark (?) into appropriate position in the SQL statement. You can also add a parameter to the position in the **VALUES** clause of the **INSERT** statement or in the **WHERE** clause of the SQL statement, for which the value is to be substituted. Substituting a value for a question mark (?) can be performed only by cubrid_bind.

Syntax

```
int cubrid_prepare (int conn_handle,string prepare_stmt [, int option])
```

- *conn_handle* : Connection handle
- *prepare_stm* : A prepare query
- *option* : OID return option - CUBRID_INCLUDE_OID

Return Value

- Success : Request handle

- Failure : FALSE

Example

```
if ($con) {
$sql = "insert into tbl values ( ?,?,?)";
 $req = cubrid_prepare( $con, $sql, CUBRID_INCLUDE_OID );
 $i = 0;
 while ( $i < 2 ) {
 $res = cubrid_bind( $req, 1, "1", "NUMBER");
 $res = cubrid_bind( $req, 2, "2");
 $res = cubrid_bind( $req, 3, "04:22:34 PM 08/07/2007");
 $res = cubrid_execute( $req );
$i = $i + 1;
}}
```

See Also

- cubrid_execute
- cubrid_bind

cubrid_put

Description

This function is used to change attribute values of an instance by using the given OID. You can update single attribute by using string data type to set attr. In such case, you can use integer, floating-point, or character string data type for the value argument. To change multiple attributes simultaneously, pass value argument in the form of associative array data type without specifying the attr argument. However, this method cannot be used for attributes of a collection type. For attributes of a collection type, you should use collection-related APIs (cubrid_set_add, cubrid_set_drop, etc.).

Syntax

```
int cubrid_put (int conn_handle, string oid[, string attr], mixed value)
```

- *conn_handle* : Connection handle
- *oid* : OID of the instance whose value you want to change
- *attr* : Name of the attribute whose value you want to change
- *value* : Value of the attribute you want to change

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
$attrarray = cubrid_get ($con, $oid);
$attrarray["name"] = "New Name";
cubrid_put ($con, $oid, $attrarray);
cubrid_put ($con, $oid, "name", "New Name2");
cubrid_put ($con, $oid, "hobbies", array("aa", "bb"));
```

See Also

- cubrid_get
- cubrid_set_add
- cubrid_set_drop
- cubrid_seq_insert
- cubrid_seq_drop
- cubrid_seq_put

cubrid_real_escape_string

Description

The cubrid_real_escape_string function performs safe query by adding an additional single quotation ('') if a string contains a single quotation(') before sending the query to a server.

To make a underline and a percentage symbols to be recognized as a string (not a wild card) in a query statement that contains the **LIKE** conditional expression, you must use the **ESCAPE** statement together.

- \x00
- \n
- \r
- \
- '
- "

This function must always (with few exceptions) be used to make data safe before sending a query to CUBRID.

Syntax

```
string cubrid_real_escape_string (string $unescaped_string [, resource $link_identifier ] )
```

- *unescaped_string* : The string that is to be escaped.
- *link_identifier* : The CUBRID connection. If the link identifier is not specified, the last link opened by cubrid_connect is assumed.

Return Value

- Success : Escaped characters
- Failure : FALSE

Example

```
< ?php
$query = sprintf("SELECT * FROM users WHERE user='%s' AND password='%s'",
cubrid_real_escape_string($user),
cubrid_real_escape_string($password))
?>
```

cubrid_result

Description

cubrid_result retrieves the contents of one cell from a CUBRID result set on success, or **FALSE** on failure.

When working on large result sets, you should consider using one of the functions that fetch an entire row. As these functions return the contents of multiple cells in one function call, they're MUCH quicker than **cubrid_result()**. Also, note that specifying a numeric offset for the field argument is much quicker than specifying a fieldname or tablename.fieldname argument.

Syntax

```
string cubrid_result ( int $result , int $row [, mixed $field= 0 ] )
```

- *result* : The result handle that is being evaluated. This result comes from a call to cubrid_execute.
- *row* : The row number from the result that's being retrieved. Row numbers start at 0.
- *field* : The name or offset of the field being retrieved. It can be the field's offset, the field's name, or the field's table dot field name (tablename.fieldname). If the column name has been aliased ('select foo as bar from...'), use the alias instead of the column name. If undefined, the first field is retrieved.

Example

```
<?php
$result = cubrid_execute($link, “SELECT name FROM work.employee”);
if (!$result) {
echo 'Could not run query: ' . cubrid_error_msg();
exit;
}
if (!$result) {
die('Could not query:' . mysql_error());
}
echo mysql_result($result, 2); // outputs third employee's name
?>
```

cubrid_rollback

Description

This function is used to roll back the transaction being executed in the connection referred by the *conn_handle*. The connection with the server is terminated after the **cubrid_rollback** method is called, but the connection handle remains valid.

Syntax

```
int cubrid_rollback (int conn_handle)
```

- *conn_handle* : Connection handle

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
$req = cubrid_execute ($oid, "insert into person values (2,'John')");
if ($req) {
   cubrid_close_request ($req);
   if ($failed) {
      cubrid_rollback ($con);
   } else {
      cubrid_commit ($con);
   }
}
```

See Also

- cubrid_commit
- cubrid_disconnect

cubrid_schema

Description

This function is used to get specific schema information of a database. You should specify *class_name* to get information related to a specific class, and *attr_name* to get information related to a specific attribute (currently, only used with **CUBRID_SCH_ATTR_PRIVILEGE**).

Syntax

```
mixed cubrid_schema (int conn_handle, int schema_type[, string class_name[, string
attr_name]])
```

- *conn_handle* : Connection handle
- *schema_type* : Type of schema you want to get

- *class_name* : Class from which schema is to be obtained
- *attr_name* : Attribute from which schema is to be obtained

Return Value

- Success : Array in which schema information is contained
- Failure : -1

The result of the **cubrid_schema** function is returned as a two-dimensional array (column (associative array) * row (numeric array)). The following table shows types of schema and the column structure of the result array to be returned based on the schema type.

Schema	Column Number	Column Name	Value
CUBRID_SCH_CLASS	1	NAME	0 : System class 1 : vclass 2 : class
	2	TYPE	
CUBRID_SCH_VCLASS	1	NAME	1 : vclass
	2	TYPE	
CUBRID_SCH_QUERY_SPEC	1	QUERY_SPEC	
CUBRID_SCH_ATTRIBUTE	1	ATTR_NAME	
	2	DOMAIN	
	3	SCALE	
	4	PRECISION	
	5	INDEXED	1 : indexed
	6	NON NULL	1 : non null
	7	SHARED	1 : shared
	8	UNIQUE	1 : unique
	9	DEFAULT	
	10	ATTR_ORDER	base : 1
	11	CLASS_NAME	
	12	SOURCE_CLASS	
CUBRID_SCH_CLASS_ATTRIBUTE	1	ATTR_NAME	
	2	DOMAIN	
	3	SCALE	
	4	PRECISION	
	5	INDEXED	1 : indexed
	6	NON NULL	1 : non null
	7	SHARED	1 : shared
	8	UNIQUE	1 : unique
	9	DEFAULT	
	10	ATTR_ORDER	base : 1
	11	CLASS_NAME	
	12	SOURCE_CLASS	
CUBRID_SCH_METHOD	1	NAME	

	2	RET_DOMAIN	
	3	ARG_DOMAIN	
CUBRID_SCH_METHOD_FILE	1	METHOD_FILE	
CUBRID_SCH_super class	1	CLASS_NAME	
	2	TYPE	
CUBRID_SCH_SUBCLASS	1	CLASS_NAME	
	2	TYPE	
CUBRID_SCH_CONSTRAINT	1	TYPE	0 : unique 1 : index
	2	NAME	
	3	ATTR_NAME	
CUBRID_SCH_TRIGGER	1	NAME	
	2	STATUS	
	3	EVENT	
	4	TARGET_CLASS	
	5	TARGET_ATTR	
	6	ACTION_TIME	
	7	ACTION	
	8	PRIORITY	
	9	CONDITION_TIME	
	10	CONDITION	
CUBRID_SCH_CLASS_PRIVILEGE	1	CLASS_NAME	
	2	PREVILEGE	
	3	GRANTABLE	
CUBRID_SCH_ATTR_PRIVILEGE	1	ATT_NAME	
	2	PREVILEGE	
	3	GRANTABLE	

Example

```
$attrs = cubrid_schema ($con, CUBRID_SCH_ATTRIBUTE, "person")
while (list($key, $value) = each($attrs)) {
      echo $value["NAME"]
      echo $value["DOMAIN"]
}
```

cubrid_seq_drop

Description

This function is used to drop elements from the given **SEQUENCE** type attribute in the database.

Syntax

```
int cubrid_seq_drop(int conn_handle, string oid, string attr_name, int index)
```

- *conn_handle* : Connection handle
- *oid* : OID of the desired instance

- *attr_name* : Name of the desired attribute of the instance
- *index* : Index of the element to be dropped. The default value is 1.

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
$elems = cubrid_col_get ($con, $oid, "style");
$i = 1;
while (list ($key, $val) = each($elems)) {
if ($val == "1") {
    echo $val;
    cubrid_seq_drop ($con, $oid, "style", $i);
  }
  $i++;
}
```

See Also

- cubrid_seq_insert
- cubrid_seq_put

cubrid_seq_insert

Description

This function is used to insert an element into a specific position of a SEQUENCE type attribute.

Syntax

```
int cubrid_seq_insert (int conn_handle, string oid, string attr_name, int index, string
seq_element)
```

- *conn_handle* : Connection handle
- *oid* : OID of the desired instance
- *attr_name* : Name of the desired attribute of the instance
- *index* : Position into which the new element is to be inserted (default value: 1)
- *seq_string* : Content of the element to be inserted

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
cubrid_seq_insert ($con, $oid, "tel", 1, "02-3430-1200");
cubrid_seq_insert ($con, $oid, "tel", 1, "02-3430-1300");
```

See Also

- cubrid_seq_drop
- cubrid_seq_put

cubrid_seq_put

Description

This function is used to change the content of an element of the given SEQUENCE type attribute.

Syntax

```
int cubrid_seq_put (int conn_handle, string oid, string attr_name, int index, string
seq_element)
```

- *conn_handle* : Connection handle
- *oid* : OID of the desired instance
- *attr_name* : Name of the desired attribute of the instance
- *index* : Index of the element to be changed (default value: 1)
- *seq_element* : Content of the element to be changed

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
cubrid_seq_put ($con, $oid, "tel", 1, "02-3430-1200");
cubrid_seq_put ($con, $oid, "tel", 2, "02-3430-1300");
```

See Also

- cubrid_seq_insert
- cubrid_seq_drop

cubrid_set_add

Description

This function is used to insert an element to the given SET type (set, multiset) attribute.

Syntax

```
int cubrid_set_add (int conn_handle, string oid, string attr_name, string set_element)
```

- *conn_handle* : Connection handle
- *oid* : OID of the desired instance
- *attr_name* : Name of the desired attribute of the instance
- *set_string* : Content of the element to be inserted

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
cubrid_set_add ($con, $oid, "friend", "James");
cubrid_set_add ($con, $oid, "friend", "Michael");
```

See Also

- cubrid_set_drop

cubrid_set_drop

Description

This function is used to drop an element from the given SET type (set, multiset) attribute.

Syntax

```
int cubrid_set_drop (int conn_handle, string oid, string attr_name, string set_element)
```

- *conn_handle* : Connection handle
- *oid* : OID of the desired instance
- *attr_name* : Name of the desired attribute of the instance
- *set_element* : Content of the element to be dropped.

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
cubrid_set_drop ($con, $oid, "friend", "James");
cubrid_set_drop ($con, $oid, "friend", "Michael");
```

See Also

- cubrid_set_add

cubrid_unbuffered_query

Description

cubrid_unbuffered_query sends a SQL query to CUBRID, without fetching and buffering the result rows automatically, as cubrid_execute does. On the one hand, this saves a considerable amount of memory with SQL queries that produce large result sets. On the other hand, you can start working on the result set immediately after the first row has been retrieved: you don't have to wait until the complete SQL query has been performed. When using multiple DB-connects, you have to specify the optional parameter *link_identifier*.

Syntax

```
resource cubrid_unbuffered_query ( string $query [, int $link_identifier ] )
```

- *query* : A SQL query
- *link_identifier* : The CUBRID connection. If the *link identifier* is not specified, the last link opened by cubrid_connect is assumed.

Return Value

- Returns **TRUE** on success, or **FALSE** on error.
- For other type of SQL statements such as **UPDATE**, **DELETE**, **DROP**, etc, **cubrid_unbuffered_query()** returns **TRUE** on success or **FALSE** on error.

Example

```
< ?php
$result = cubrid_unbuffered_query("INSERT INTO mytable (product) values ('kossu')", $link)
if (!$result) {
echo 'Could not run query: ' .cubrid_error_msg()
exit
}
printf("Last inserted record has id %d\n", cubrid_insert_id())
?>
```

cubrid_version

Description

This function is used to check the version information of the CUBRID PHP module.

Syntax

```
string cubrid_version ()
```

Return Value

- n version information (e.g. "1.2.0")

Example

```
echo cubrid_version();
```

See Also

- cubrid_error_code
- cubrid_error_code_facility

CCI API

CCI Overview

Overview

The CCI (C Client Interface) is an interface that exists between the CUBRID broker and the application client, through which a C-based application client can access the CUBRID database server using a broker. This interface is also used as an infrastructure for making tools that utilize CAS (e.g. PHP and ODBC). The CUBRID broker delivers the query received from an application client to the broker, and transfers the execution result to the client.

A header file and library files are required to use CCI.

- Header file : $CUBRID/include/**cas_cci.h**
- Library file :
- $CUBRID/lib/**libcascci.so** (Windows : **cascci.dll**)
- $CUBRID/lib/**libcascci.a** (Windows : **cascci.lib**)

Writing CCI Application Program

The basic steps used for writing programs are as follows, and a step for binding the data to a variable is added to use the prepared statement. The steps are implemented in example codes 1 and 2. Auto-commit mode is supported only for SELECT statements when the SELECT_AUTO_COMMIT parameter has been set to ON. Otherwise, you must explicitly commit or roll back the transaction by using the cci_end_tran() function. For details about how to configure and use auto-commit mode in CCI, see [Performance Tuning > Broker Configuration > Parameter By Broker > SELECT_AUTO_COMMIT] and cci_end_tran().

- Opening a database connection handle (related function : cci_connect(), cci_connect_with_url())
- Preparing an SQL statement (related function : cci_prepare())
- Binding data to a prepared SQL statement (related function : cci_bind_param())
- Executing a prepared SQL statement (related function : cci_execute())
- Processing the execution result (related function : cci_cursor(), cci_fetch(), cci_get_data(), cci_get_result_info())
- Closing the request handle (related function : cci_close_req_handle())
- Closing a database connection handle (related function : cci_disconnect())

Example 1

```
//Example to execute a simple query
#include <stdio.h>
#include "cas_cci.h"
#define BUFSIZE  (1024)

int
main (void)
{
  int con = 0, req = 0, col_count = 0, i, ind;
  int error;
  char *data;
  T_CCI_ERROR cci_error;
  T_CCI_COL_INFO *col_info;
  T_CCI_SQLX_CMD cmd_type;
  char *query = "select * from code";

//getting a connection handle for a connection with a server
  con = cci_connect ("localhost", 44000, "demodb", "dba", "");
  if (con < 0)
    {
      printf ("cannot connect to database\n");
      return 1;
    }
```

```
//preparing the SQL statement
  req = cci_prepare (con, query, 0, &cci_error);
  if (req < 0)
    {
      printf ("prepare error: %d, %s\n", cci_error.err_code,
              cci_error.err_msg);
      goto handle_error;
    }

//getting column information when the prepared statement is the SELECT query
  col_info = cci_get_result_info (req, &cmd_type, &col_count);
  if (col_info == NULL)
    {
      printf ("get_result_info error: %d, %s\n", cci_error.err_code,
              cci_error.err_msg);
      goto handle_error;
    }

//Executing the prepared SQL statement
  error = cci_execute (req, 0, 0, &cci_error);
  if (error < 0)
    {
      printf ("execute error: %d, %s\n", cci_error.err_code,
              cci_error.err_msg);
      goto handle_error;
    }
  while (1)
    {

//Moving the cursor to access a specific tuple of results
      error = cci_cursor (req, 1, CCI_CURSOR_CURRENT, &cci_error);
      if (error == CCI_ER_NO_MORE_DATA)
        {
          break;
        }
      if (error < 0)
        {
          printf ("cursor error: %d, %s\n", cci_error.err_code,
                  cci_error.err_msg);
          goto handle_error;
        }

//Fetching the query result into a client buffer
      error = cci_fetch (req, &cci_error);
      if (error < 0)
        {
          printf ("fetch error: %d, %s\n", cci_error.err_code,
                  cci_error.err_msg);
          goto handle_error;
        }
      for (i = 1; i <= col_count; i++)
        {

//Getting data from the fetched result
          error = cci_get_data (req, i, CCI_A_TYPE_STR, &data, &ind);
          if (error < 0)
            {
              printf ("get_data error: %d, %d\n", error, i);
              goto handle_error;
            }
          printf ("%s\t|", data);
        }
      printf ("\n");
    }

//Closing the request handle
  error = cci_close_req_handle (req);
  if (error < 0)
    {
      printf ("close_req_handle error: %d, %s\n", cci_error.err_code,
              cci_error.err_msg);
      goto handle_error;
    }
```

```
//Disconnecting with the server
  error = cci_disconnect (con, &cci_error);
  if (error < 0)
    {
      printf ("error: %d, %s\n", cci_error.err_code, cci_error.err_msg);
      goto handle_error;
    }

  return 0;

handle_error:
  if (req > 0)
    cci_close_req_handle (req);
  if (con > 0)
    cci_disconnect (con, &cci_error);

  return 1;
}
```

Example 2

```
//Example to execute a query with a bind variable

char *query = "select * from nation where name = ?";
  char namebuf[128];

//getting a connection handle for a connection with a server
  con = cci_connect ("localhost", 44000, "demodb", "dba", "");
  if (con < 0)
    {
      printf ("cannot connect to database ");
      return 1;
    }

//preparing the SQL statement
  req = cci_prepare (con, query, 0, &cci_error);
  if (req < 0)
    {
      printf ("prepare error: %d, %s ", cci_error.err_code,
              cci_error.err_msg);
      goto handle_error;
    }

//Binding date into a value
  strcpy (namebuf, "Korea");
  error =
    cci_bind_param (req, 1, CCI_A_TYPE_STR, &namebuf, CCI_U_TYPE_STRING,
                    CCI_BIND_PTR);
  if (error < 0)
    {
      printf ("bind_param error: %d ", error);
      goto handle_error;
    }
```

Using BLOB/CLOB with CCI

Storing LOB Data

You can create LOB data file and bind the data by using the following functions in CCI applications.

- Creating LOB data file (related function : cci_blob_new( ), cci_blob_write( ))
- Binding LOB data (related function : cci_bind_param( ))
- Freeing memory of LOB structure (related function : cci_blob_free( ))

Example 1

```
int con = 0; /* connection handle */
int req = 0; /* request handle */
int res;
int n_executed;
int i;
T_CCI_ERROR error;
T_CCI_BLOB blob = NULL;
char data[1024] = “bulabula”;

con = cci_connect ("localhost", 33000, “tdb", "PUBLIC", "");
if (con < 0) {
goto handle_error;
}
req = cci_prepare (con, “insert into doc (doc_id, content) values (?,?)”, 0, &error);
if (req< 0)
{
goto handle_error;
}

res = cci_bind_param (req, 1 /* binding index*/, CCI_A_TYPE_STR, “doc-10”, &ind,
CCI_U_TYPE_STRING);

/* Creating an empty LOB data file
res = cci_blob_new (con, &blob, &error);
res = cci_blob_write (con, blob, 0 /* start position */, 1024 /* length */, data, &error);

/* Binding BLOB data */
res = cci_bind_param (req, 2 /* binding index*/, CCI_A_TYPE_BLOB, (void *)blob,
CCI_U_TYPE_BLOB, CCI_BIND_PTR);

n_executed = cci_execute (req, 0, 0, &error);
if (n_executed < 0)
{
goto handle_error
}

/* Memory free */
cci_blob_free(blob);
return 0;

handle_error:
if (blob != NULL)
{
cci_blob_free(blob);
}
if (req > 0)
{
cci_close_req_handle (req);
}
if (con > 0)
{
cci_disconnect(con, &error);
}
return -1;
```

Getting LOB Data

You can get LOB data by using the following functions in CCI applications. Note that if you enter data in LOB type colulmn, the actual LOB data is stored externally and Locator value referring to the file is stored in LOB type column itself. Therefore, you must call the cci_blob_read() function (not the cci_get_data() function) to get LOB data stored in the file.

- Getting LOB type column value (Locator) (related function : cci_get_data( ))
- Getting LOB data (related function : cci_blob_read( ))
- Freeing memory of LOB structure (related function : cci_blob_free( ))

Example

```
int con = 0; /* connection handle */
int req = 0; /* request handle */
int ind; /* NULL indicator, 0 if not NULL, -1 if NULL*/
int res;
int i;
```

```
T_CCI_ERROR error;
T_CCI_BLOB blob;
char buffer[1024];

con = cci_connect ("localhost", 33000, "image_db", "PUBLIC", "");
if (con < 0)
 {
  goto handle_error;
 }
req = cci_prepare (con, "select content from doc_t", 0 /*flag*/, &error);
if (req< 0)
 {
  goto handle_error;
 }
res = cci_execute (req, 0/*flag*/, 0/*max_col_size*/, &error);
res = cci_fetch_size (req, 100 /* fetch size */);

while (1) {
  res = cci_cursor (req, 1/* offset */, CCI_CURSOR_CURRENT/* cursor position */, &error);
  if (res == CCI_ER_NO_MORE_DATA)
     {
      break;
     }
  res = cci_fetch (req, &error);

/* Fetching CLOB Locator */
  res = cci_get_data (req, 1 /* colume index */, CCI_A_TYPE_BLOB,
        (void *)&blob /* BLOB handle */, &ind /* NULL indicator */);

/* Fetching CLOB data */
res = cci_blob_read (con, blob, 0 /* start position */, 1024 /* length */, buffer, &error);
printf ("content = %s\n", buffer);
}
/* Memory free */
cci_blob_free(blob);
res=cci_close_req_handle(req);
res = cci_disconnect (con, &error);
return 0;

handle_error:
if (req > 0)
    {
     cci_close_req_handle (req);
    }
if (con > 0)
   {
     cci_disconnect(con, &error);
   }
return -1;
```

CCI Error Code and Message

The following table shows the error codes and their messages of CCI.

Error Code	Error Message	Note
CCI_ER_ALLOC_CON_HANDLE	"Cannot allocate connection handle"	
CCI_ER_ATYPE	"Invalid T_CCI_A_TYPE value"	
CCI_ER_BIND_ARRAY_SIZE	"Array binding size is not specified"	
CCI_ER_BIND_INDEX	"Parameter index is out of range"	Index that binds data is not valid.
CCI_ER_COLUMN_INDEX	"Column index is out of range"	
CCI_ER_COMMUNICATION	"Cannot communicate with	

	server"	
CCI_ER_CON_HANDLE	"Invalid connection handle"	
CCI_ER_CONNECT	"Cannot connect to CUBRID CAS"	Fails to connect the CAS when trying connection to the server.
CCI_ER_DELETED_TUPLE	"Current row was deleted"	
CCI_ER_FILE	"Cannot open file"	Fails to open/read/write a file.
CCI_ER_HOSTNAME	"Unknown host name"	
CCI_ER_INVALID_CURSOR_POS	"Invalid cursor position"	
CCI_ER_INVALID_URL	"Invalid url string"	
CCI_ER_ISOLATION_LEVEL	"Unknown transaction isolation level"	
CCI_ER_NO_MORE_DATA	"Invalid cursor position"	
CCI_ER_NO_MORE_MEMORY	"Memory allocation error"	Insufficient memory
CCI_ER_OBJECT	"Invalid oid string"	
CCI_ER_OID_CMD	"Invalid T_CCI_OID_CMD value"	
CCI_ER_TRAN_TYPE	"Unknown transaction type"	
CCI_ER_PARAM_NAME	"Invalid T_CCI_DB_PARAM value"	
CCI_ER_REQ_HANDLE	"Cannot allocate request handle"	
CCI_ER_SAVEPOINT_CMD	"Invalid T_CCI_SAVEPOINT_CMD value"	Invalid T_CCI_SAVEPOINT_CMD value is used as an argument of cci_savepoint() function.
CCI_ER_SET_INDEX	"Invalid set index"	Invalid index is specified when an set element in the T_SET is retrieved.
CCI_ER_STRING_PARAM	"Invalid string argument"	string parameter is NULL or an empty string.
CCI_ER_THREAD_RUNNING	"Thread is running"	The thread is still executed when cci_execute() is executed with CCI_EXEC_THREAD flaged and check the result of thread execution through cci_get_thread_result().
CCI_ER_TRAN_TYPE	"Unknown transaction type"	Connection to the server has succeeded, connection to a database fails.
CCI_ER_TYPE_CONVERSION	"Type conversion error"	Cannot convert the given value into an actual data type.
CCI_ER_DBMS CAS_ER_DBMS	"CUBRID DBMS Error"	Fails to database connection.
CAS_ER_COLLECTION_DOMAIN	"Heterogeneous set is not supported"	Not supported set type.
CAS_ER_COMMUNICATION	"Cannot receive data from	

	client"	
CAS_ER_DB_VALUE	"Cannot make DB_VALUE"	
CAS_ER_DBSERVER_DISCONNECTED	"Cannot communicate with DB Server"	
CAS_ER_FREE_SERVER	"Cannot process the request. Try again later"	Cannot assign CAS.
CAS_ER_INVALID_CALL_STMT	"Illegal CALL statement"	
CAS_ER_NO_MORE_DATA	"Invalid cursor position"	
CAS_ER_NO_MORE_MEMORY	"Memory allocation error"	
CAS_ER_NO_MORE_RESULT_SET	"No More Result"	
CAS_ER_NOT_AUTHORIZED_CLIENT	"Authorization error"	Access is denied.
CAS_ER_NOT_COLLECTION	"The attribute domain must be the set type"	No set type.
CAS_ER_NUM_BIND	"Invalid parameter binding value argument"	The number of data to be bound is not matched with the number of delivered data.
CAS_ER_OBJECT	"Invalid oid"	
CAS_ER_OPEN_FILE	"Cannot open file"	
CAS_ER_PARAM_NAME	"Invalid T_CCI_DB_PARAM value"	Invalid get_db_parameter and , set_db_parameter parameter name.
CAS_ER_QUERY_CANCEL	"Cannot cancel the query"	
CAS_ER_UNKNOWN_U_TYPE	"Invalid T_CCI_U_TYPE value"	
CAS_ER_TYPE_CONVERSION	"Type conversion error"	
CAS_ER_SCHEMA_TYPE	"Invalid T_CCI_SCH_TYPE value"	
CAS_ER_STMT_POOLING	"Invalid plan"	
CAS_ER_TRAN_TYPE	"Invalid transaction type argument"	
CAS_ER_TYPE_CONVERSION	"Type conversion error"	
CAS_ER_UNKNOWN_U_TYPE	"Invalid T_CCI_U_TYPE value"	
CAS_ER_VERSION	"Version mismatch"	Invalid Server and Client version.

C Type Definition

Name	Type	Member	Description
T_CCI_ERROR	struct	char err_msg[1024]	Representation of database error info
		int err_code	
T_CCI_BIT	struct	int size	Representation of bit type
		char *buf	
T_CCI_DATE	struct	short yr	Representation of timestamp, date,
		short mon	

		short day	time type
		short hh	
		short mm	
		short ss	
		short ms	
T_CCI_SET	void*		Representation of set type
T_CCI_COL_INFO	struct	**T_CCI_U_TYPE** type	Representation of column information for the **SELECT** statement
		char is_non_null	
		short scale	
		int precision	
		char *col_name	
		char *real_attr	
		char *class_name	
T_CCI_QUERY_RESULT	struct	int result_count	Results of batch execution
		int stmt_type	
		char *err_msg	
		char oid[32]	
T_CCI_PARAM_INFO	struct	**T_CCI_PARAM_MODE** mode	Representation of input parameter info
		T_CCI_U_TYPE type	
		short scale	
		int precision	
T_CCI_U_TYPE	enum	**CCI_U_TYPE_UNKNOWN**	Database type info
		CCI_U_TYPE_NULL	
		CCI_U_TYPE_CHAR	
		CCI_U_TYPE_STRING	
		CCI_U_TYPE_NCHAR	
		CCI_U_TYPE_VARNCHAR	
		CCI_U_TYPE_BIT	
		CCI_U_TYPE_VARBIT	
		CCI_U_TYPE_NUMERIC	
		CCI_U_TYPE_INT	
		CCI_U_TYPE_SHORT	
		CCI_U_TYPE_MONETARY	
		CCI_U_TYPE_FLOAT	
		CCI_U_TYPE_DOUBLE	
		CCI_U_TYPE_DATE	
		CCI_U_TYPE_TIME	
		CCI_U_TYPE_TIMESTAMP	
		CCI_U_TYPE_SET	

		CCI_U_TYPE_MULTISET	
		CCI_U_TYPE_SEQUENCE	
		CCI_U_TYPE_OBJECT	
		CCI_U_TYPE_BIGINT	
		CCI_U_TYPE_DATETIME	
T_CCI_A_TYPE	enum	**CCI_A_TYPE_STR**	Representation of type info used in API
		CCI_A_TYPE_INT	
		CCI_A_TYPE_FLOAT	
		CCI_A_TYPE_DOUBLE	
		CCI_A_TYPE_BIT	
		CCI_A_TYPE_DATE	
		CCI_A_TYPE_SET	
		CCI_A_TYPE_BIGINT	
		CCI_TYPE_BLOB	
		CCI_TYPE_CLOB	
T_CCI_DB_PARAM	enum	**CCI_PARAM_ISOLATION_LEVEL**	Database parameter name
		CCI_PARAM_LOCK_TIMEOUT	
		CCI_PARAM_MAX_STRING_LENGTH	
		CCI_PARAM_AUTO_COMMIT	
T_CCI_SCH_TYPE	enum	**CCI_SCH_CLASS**	
		CCI_SCH_VCLASS	
		CCI_SCH_QUERY_SPEC	
		CCI_SCH_ATTRIBUTE	
		CCI_SCH_CLASS_ATTRIBUTE	
		CCI_SCH_METHOD	
		CCI_SCH_CLASS_METHOD	
		CCI_SCH_METHOD_FILE	
		CCI_SCH_SUPERCLASS	
		CCI_SCH_SUBCLASS	
		CCI_SCH_CONSTRAIT	
		CCI_SCH_TRIGGER	
		CCI_SCH_CLASS_PRIVILEGE	
		CCI_SCH_ATTR_PRIVILEGE	
		CCI_SCH_DIRECT_SUPER_CLASS	
		CCI_SCH_PRIMARY_KEY	
		CCI_SCH_IMPORTED_KEYS	
		CCI_SCH_EXPORTED_KEYS	
		CCI_SCH_CROSS_REFERENCE	
T_CCI_CUBRID_STMT	enum	**CUBRID_STMT_ALTER_CLASS**	
		CUBRID_STMT_ALTER_SERIAL	

	CUBRID_STMT_COMMIT_WORK
	CUBRID_STMT_REGISTER_DATABASE
	CUBRID_STMT_CREATE_CLASS
	CUBRID_STMT_CREATE_INDEX
	CUBRID_STMT_CREATE_TRIGGER
	CUBRID_STMT_CREATE_SERIAL
	CUBRID_STMT_DROP_DATABASE
	CUBRID_STMT_DROP_CLASS
	CUBRID_STMT_DROP_INDEX
	CUBRID_STMT_DROP_LABEL
	CUBRID_STMT_DROP_TRIGGER
	CUBRID_STMT_DROP_SERIAL
	CUBRID_STMT_EVALUATE
	CUBRID_STMT_RENAME_CLASS
	CUBRID_STMT_ROLLBACK_WORK
	CUBRID_STMT_GRANT
	CUBRID_STMT_REVOKE
	CUBRID_STMT_STATISTICS
	CUBRID_STMT_INSERT
	CUBRID_STMT_SELECT
	CUBRID_STMT_UPDATE
	CUBRID_STMT_DELETE
	CUBRID_STMT_CALL
	CUBRID_STMT_GET_ISO_LVL
	CUBRID_STMT_GET_TIMEOUT
	CUBRID_STMT_GET_OPT_LVL
	CUBRID_STMT_SET_OPT_LVL
	CUBRID_STMT_SCOPE
	CUBRID_STMT_GET_TRIGGER
	CUBRID_STMT_SET_TRIGGER
	CUBRID_STMT_SAVEPOINT
	CUBRID_STMT_PREPARE
	CUBRID_STMT_ATTACH
	CUBRID_STMT_USE
	CUBRID_STMT_REMOVE_TRIGGER
	CUBRID_STMT_RENAME_TRIGGER
	CUBRID_STMT_ON_LDB
	CUBRID_STMT_GET_LDB
	CUBRID_STMT_SET_LDB
	CUBRID_STMT_GET_STATS

		CUBRID_STMT_CREATE_USER
		CUBRID_STMT_DROP_USER
		CUBRID_STMT_ALTER_USER
T_CCI_CURSOR_POS	enum	**CCI_CURSOR_FIRST**
		CCI_CURSOR_CURRENT
		CCI_CURSOR_LAST
T_CCI_TRAN_ISOLATION	enum	**TRAN_COMMIT_CLASS_UNCOMMIT_INSTANCE**
		TRAN_COMMIT_CLASS_COMMIT_INSTANCE
		TRAN_REP_CLASS_UNCOMMIT_INSTANCE
		TRAN_REP_CLASS_COMMIT_INSTANCE
		TRAN_REP_CLASS_REP_INSTANCE
		TRAN_SERIALIZABLE
T_CCI_PARAM_MODE	enum	**CCI_PARAM_MODE_UNKNOWN**
		CCI_PARAM_MODE_IN
		CCI_PARAM_MODE_OUT
		CCI_PARAM_MODE_INOUT

cci_bind_param

Description

This function is used to bind data in the *bind* variable of prepared statement. Converts *value* of the given *a_type* to an actual binding type and saves it. Subsequently, whenever cci_execute() is called, the saved data is sent to the server. If **cci_bind_param**() is called multiple times for the same *index*, the last set value is configured.

If **NULL** is bound to the database, there can be two scenarios.

- *value* is a **NULL** pointer.
- *u_type* is **CCI_U_TYPE_NULL**.

If **CCI_BIND_PTR** is configured for *flag*, the pointer of *value* variable is copied (shallow copy), but no value is copied. If it is not configured for *flag*, the value of *value* variable is copied (deep copy) by allocating memory. If multiple columns are bound by using the same memory buffer, **CCI_BIND_PTR** must not be configured for the *flag*.

T_CCI_A_TYPE is a C language type that is used in CCI application programs, and consists of primitive types such as int and float and user-defined types defined by CCI such as **T_CCI_BIT** and **T_CCI_DATE**. The identitier for each type is defined as shown in the table below.

a_type	**value Type**
CCI_A_TYPE_STR	char**
CCI_A_TYPE_INT	int*
CCI_A_TYPE_FLOAT	float*
CCI_A_TYPE_DOUBLE	double*
CCI_A_TYPE_BIT	**T_CCI_BIT***
CCI_A_TYPE_SET	**T_CCI_SET**
CCI_A_TYPE_DATE	**T_CCI_DATE***
CCI_A_TYPE_BIGINT	int64_t* (For Windows : __int64*)

CCI_A_TYPE_BLOB	**T_CCI_BLOC**
CCI_A_TYPE_CLOB	**T_CCI_CLOB**

T_CCI_U_TYPE is a type supported by the CUBRID database. For the definition of the identifier for each type, see the table below. These two types are used in the **cci_bind_param**() function to deliver the information required to convert the A type data that can be understood by the C language to the U type data that can be understood by the database. **T_CCI_A_TYPE** and **T_CCI_U_TYPE** enums are all defined in the **cas_cci.h file**.

u_type	**value Type**
CCI_U_TYPE_CHAR	char**
CCI_U_TYPE_STRING	char**
CCI_U_TYPE_NCHAR	char**
CCI_U_TYPE_VARNCHAR	char**
CCI_U_TYPE_BIT	T_CCI_BIT*
CCI_U_TYPE_VARBIT	T_CCI_BIT*
CCI_U_TYPE_NUMERIC	char**
CCI_U_TYPE_INT	int*
CCI_U_TYPE_SHORT	int*
CCI_U_TYPE_MONETARY	Double*
CCI_U_TYPE_FLOAT	float*
CCI_U_TYPE_DOUBLE	Double*
CCI_U_TYPE_DATE	T_CCI_DATE*
CCI_U_TYPE_TIME	T_CCI_DATE*
CCI_U_TYPE_TIMESTAMP	T_CCI_DATE*
CCI_U_TYPE_OBJECT	char**
CCI_U_TYPE_BIGINT	int64_t* (Windows : __int64*)
CCI_U_TYPE_DATETIME	T_CCI_DATE*

Syntax

```
int cci_bind_param(int req_handle, int index, T_CCI_A_TYPE a_type, void *value,
T_CCI_U_TYPE u_type, char flag)
```

- *req_handle* : (IN) Request handle of a prepared SQL statement
- *index* : (IN) One-based binding location it starts with 1.
- *a_type* : (IN) Data type of *value*
- *value* : (IN) Data value to be bound
- *u_type* : (IN) Data type to be applied to the database
- *flag* : (IN) bind_flag (**CCI_BIND_PTR**)

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_TYPE_CONVERSION**
- **CCI_ER_BIND_INDEX**
- **CCI_ER_ATYPE**
- **CCI_ER_NO_MORE_MEMORY**

cci_bind_param_array

Description

This function is used to bind a parameter array for a prepared req_handle. Subsequently, whenever cci_execute_array() occurs, data is sent to the server by the saved *value* pointer. If **cci_bind_param_array**() is called multiple times for the same *index*, the last configured value is used. If **NULL** is bound to the data, a non-zero value is configured to *null_ind*.

If *value* is a **NULL** pointer, or *u_type* is **CCI_U_TYPE_NULL**, all data are bound to **NULL** and the data buffer used by *value* cannot be reused.

For the data type of *value* for *a_type*, see the cci_bind_param() function description.

Syntax

```
int cci_bind_param_array(int req_handle, int index, T_CCI_A_TYPE a_type, void *value, int
*null_ind, T_CCI_U_TYPE u_type)
```

- *req_handle* : (IN) Request handle of a prepared SQL statement
- *index* : (IN) Binding location
- *a_type* : (IN) Data type of *value*
- *value* : (IN) Data value to be bound
- *null_ind* : (IN) **NULL** indicator array (0 : not **NULL**, 1 : **NULL**)
- *u_type* : (IN) Data type to be applied to the database.

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_TYPE_CONVERSION**
- **CCI_ER_BIND_INDEX**
- **CCI_ER_ATYPE**
- **CCI_ER_BIND_ARRAY_SIZE**

cci_bind_param_array_size

Description

This function is used to determine the size of the array to be used in cci_bind_param_array().
cci_bind_param_array_size() must be called first before cci_bind_prarm_array() is used.

Syntax

```
int cci_bind_param_array_size(int req_handle, int array_size)
```

- *req_handle* : (IN) Request handle of a prepared statement
- *array_size* : (IN) Binding array size

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_REQ_HANDLE**

cci_close_req_handle

Description

This function is used to close the request handle obtained by cci_prepare().

Syntax

```
int cci_close_req_handle(int req_handle)
```

- *req_handle* : (IN) Request handle

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_COMMUNICATION**

cci_col_get

Description

This function is used to get an attribute value of collection type. If the name of the class is C, and the domain of set_attr is set (multiset, sequence), the query looks like as follows:

```
SELECT a FROM C, TABLE(set_attr) AS t(a) WHERE C = oid;
```

That is, the number of members becomes the number of records.

Syntax

```
intcci_col_get (int conn_handle, char *oid_str, char *col_attr, int *col_size, int
*col_type, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) oid
- *col_attr* : (IN) Collection attribute name
- *col_size* : (OUT) Collection size (-1 : null)
- *col_type* : (OUT) Collection type (set, multiset, sequence : u_type)
- *err_buf* : (OUT) Database error buffer

Return Value

- Request handle

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_OBJECT**
- **CCI_ER_DBMS**

cci_col_seq_drop

Description

This function is used to drop the index-th (base:1) member of the sequence attribute values. The following is an example of dropping the first member of the sequence attribute values.

```
cci_col_seq_drop(con_id, oid_str, seq_attr, 1, err_buf);
```

Syntax

```
intcci_col_seq_drop (int conn_handle, char *oid_str, char *col_attr, int index,
T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) oid
- *col_attr* : (IN) Collection attribute name
- *index* : (IN) Index
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_OBJECT**
- **CCI_ER_DBMS**

cci_col_seq_insert

Description

This function is used to insert a member at the index-th (base:1) position of the sequence attribute values. The following is an example of inserting "a" at the first position of the sequence attribute values.

```
cci_col_seq_insert(con_id, oid_str, seq_attr, 1, "a", err_buf);
```

Syntax

```
intcci_col_seq_insert (int conn_handle, char *oid_str, char *col_attr, int index, char
*value, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) oid
- *col_attr* : (IN) Collection attribute name
- *index* : (IN) Index
- *value* : (IN) Sequential element (string)
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_OBJECT**

- **CCI_ER_DBMS**

cci_col_seq_put

Description

This function is used to replace the index-th (base:1) member of the sequence attribute values with a new value. The following is an example of replacing the first member of the sequence attributes values with "a".

```
cci_col_seq_put(con_id, oid_str, seq_attr, 1, "a", err_buf);
```

Syntax

```
intcci_col_seq_put (int conn_handle, char *oid_str, char *col_attr, int index, char *value,
T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) oid
- col_attr : (IN) Collection attribute name
- index : (IN) Index
- value : (IN) Sequential value
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code

Return Value

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_OBJECT**
- **CCI_ER_DBMS**

cci_col_set_add

Description

This function is used to add a member to the set attribute values. The following is an example of adding "a" to the set attribute values.

```
cci_col_set_add(con_id, oid_str, set_attr, "a", err_buf);
```

Syntax

```
intcci_col_set_add ( int conn_handle, char *oid_str, char *col_attr, char *value,
T_CCI_ERRROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) oid
- *col_attr* : (IN) Collection attribute name
- *value* : (IN) Set element
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**

- **CCI_ER_OBJECT**
- **CCI_ER_DBMS**

cci_col_set_drop

Description

This function is used to drop a member from the set attribute values. The following is an example of dropping "a" from the set attribute values.

```
cci_col_set_drop(con_id, oid_str, set_attr, "a", err_buf);
```

Syntax

```
intcci_col_set_drop (int conn_handle, char *oid_str, char *col_attr, char *value,
T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) oid
- *col_attr* : (IN) Collection attribute name
- *value* : (IN) Set element (string)
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code

cci_col_size

Description

This function is used to get the size of the set (seq) attribute.

Syntax

```
intcci_col_size (int conn_handle, char *oid_str, char *col_attr, int *col_size,
T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) oid
- *col_attr* : (IN) Collection attribute name
- *col_size* : (OUT) Collection size (-1 : NULL)
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_OBJECT**
- **CCI_ER_DBMS**

cci_connect

Description

A connection handle to the database server is assigned and it tries to connect to the server. If it has succeeded, the connection handle ID is returned; if fails, an error code is returned.

Syntax

```
int cci_connect(char *ip, int port, char *db_name, char *db_user, char *db_password)
```

- *ip* : (IN) A character string representing the IP address of the server (host name)
- *port* : (IN) Broker port (the port configured in the **$CUBRID/conf/cubrid_broker.conf** file)
- *db_name* : (IN) Database name
- *db_user* : (IN) Database user name
- *db_passwd* : (IN) Database user password

Return Value

- Success : Connection handle ID (int)
- Failure : Error code

Error Code

- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_HOSTNAME**
- **CCI_ER_CON_HANDLE**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_CONNECT**

cci_connect_with_url

Description

cci_connect_with_url tries to connect a database by using connection information passed with an url string argument. If the HA feature is enabled in CCI, you must specify the connection information of the standby server, which is used for failover when failure occurs, in the url string argument of this function. If it has succeeded, the ID of connection handle is returned; if it fails, an error code is returned.

Syntax

```
int cci_connect_with_url (char *url [, char *db_user, char *db_password ])

<url> ::=
cci:CUBRID:<host>:<db_name>:<db_user>:<db_password>:[?<properties>]

<properties> ::= <property> [&<property>]
<property> ::= althosts=<alternative_hosts> [&rctime=<time>]
<alternative_hosts> ::= <standby_broker1_host>:<port> [,<standby_broker2_host>:<port>]

<host> := HOSTNAME | IP_ADDR
<time> := SECOND
```

- *url* : (IN) A character string that contains server connection information
- *host* : A host name or IP address of the master database
- *db_name* : A name of the database
- *db_user* : A name of the database user
- *db_password* : A database user password

- **althosts** =*standby_broker1_host, standby_broker2_host,* . . . : Specifies the broker information of the standby server, which is used for failover when it is impossible to connect to the active server. You can specify multiple brokers for failover, and the connection to the brokers is attempted in the order listed in **alhosts**.
- **rctime** : An interval between the attempts to connect to the active broker in which failure occurred. After a failure occurs, the system connects to the broker specified by althosts (failover), terminates the transaction, and then attempts to connect to the active broker of the master database at every **rctime**. The default value is 600 seconds.
- *db_user* : (IN) A name of the database user
- *db_passwd* : (IN) A database user password

Return Value

- Success : Connection handle ID (int)
- Failure : Error code

Error Code

- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_HOSTNAME**
- **CCI_ER_INVALID_URL**
- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**

Example

```
--connection URL string when a property(althosts) specified for HA
URL=cci:CUBRID:127.0.0.1:31000:db1:::?althosts=127.0.0.2:31000,127.0.0.3:31000

--connection URL string when properties(althosts,rctime) specified for HA
URL=cci:CUBRID:127.0.0.1:31000:db1:::?althosts=127.0.0.2:31000,127.0.0.3:31000&rctime=600
```

cci_cursor

Description

This function is used to move the cursor specified in the request handle to access the specific record in the query result executed by cci_execute(). The position of cursor is moved by the values specified in the *origin* and *offset* values. If the position to be moved is not valid, **CCI_ER_NO_MORE_DATA is** returned.

Syntax

```
int cci_cursor(int req_handle, int offset, T_CCI_CURSOR_POS origin, T_CCI_ERROR *err_buf)
```

- *req_handle* : (IN) Request handle
- *offset* : (IN) Offset to be moved
- *origin* : (IN) Variable to represent a position. The type is **T_CCI_CURSOR_POS**. **T_CCI_CURSOR_POS** enum consists of **CCI_CURSOR_FIRST**, **CCI_CURSOR_CURRENT**, and **CCI_CURSOR_LAST**.
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_NO_MORE_DATA**
- **CCI_ER_COMMUNICATION**

Example

```
//the cursor moves to the first record
cci_cursor(req, 1, CCI_CURSOR_FIRST, &err_buf);

//the cursor moves to the next record
cci_cursor(req, 1, CCI_CURSOR_CURRENT, &err_buf);

//the cursor moves to the last record
cci_cursor(req, 1, CCI_CURSOR_LAST, &err_buf);

//the cursor moves to the previous record
cci_cursor(req, -1, CCI_CURSOR_CURRENT, &err_buf);
```

cci_cursor_update

Description

This function is used to update *cursor_pos* from the value of the *index* th column to *value* . If the database is updated to **NULL**, *value* becomes **NULL**. For update conditions, see cci_prepare(). The data type of *value* for *a_type* is shown in the table below.

a_type	**value Type**
CCI_A_TYPE_STR	char*
CCI_A_TYPE_INT	int*
CCI_A_TYPE_FLOAT	float*
CCI_A_TYPE_DOUBLE	double*
CCI_A_TYPE_BIT	**T_CCI_BIT***
CCI_A_TYPE_SET	**T_CCI_SET**
CCI_A_TYPE_DATE	**T_CCI_DATE***
CCI_A_TYPE_BIGINT	int64_t (For Windows : __int64)
CCI_A_TYPE_BLOB **T_CCI_BLOB**	
CCI_A_TYPE_CLOB **T_CCI_CLOB**	

Syntax

```
int cci_cursor_update(int req_handle, int cursor_pos, int index, T_CCI_A_TYPE a_type, void
*value, T_CCI_ERROR *err_buf)
```

- *req_handle* : (IN) Request handle
- *cursor_pos* : (IN) Cursor position
- *index* : (IN) Column index
- *a_type* : (IN) *value* Type
- *value* : (IN) A new value
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : no error)

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_TYPE_CONVERSION**
- **CCI_ER_ATYPE**

cci_disconnect

Description

This function is used to disconnect all request handles created for *conn_handle*. If a transaction is being performed, the handles are disconnected after cci_end_tran() is executed.

Syntax

```
int cci_disconnect(int conn_handle, T_CCI_ERROR * err_buf)
```

- *conn_handle* : (IN) Connection handle
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**

cci_end_tran

Description

This function is used to perform a commit or rollback on the current transaction. At this point, all open request handles are terminated and the connection to the database server is disabled. However, even after the connection to the server is disabled, the connection handle remains valid. This is the same state as one in which one connection handle has been assigned by the cci_connect() function. The transaction is committed if the type is set to **CCI_TRAN_COMMIT**; and is rolled back if it is set to **CCI_TRAN_ROLLBACK**.

Syntax

```
int cci_end_tran(int conn_handle, char type, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *type* : (IN) **CCI_TRAN_COMMIT** or **CCI_TRAN_ROLLBACK**
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_TRAN_TYPE**

Remark

Auto-commit mode is supported for SELECT statements. To apply this mode, you must add **SELECT_AUTO_COMMIT=ON** to the cubrid_broker.conf file. However, auto-commit is performed only at the point at which the result set for all n query statements is fetched from the server when there are n prepared statements. An example is as follows:

Example 1

```
$sql1 = "select * from db_user";
$sql2 = "select * from db_class where owner_name = ?";

$result = cubrid_execute($con, $sql1);  // 1 select handle. fetch completed - autocommit
if ($result) {
  while ($row = cubrid_fetch ($result))
    {
      echo ($row[0]);

      $req = cubrid_prepare ($con, $sql2);
      cubrid_bind ($req, 1, $row[0]);
      $res = cubrid_execute ($req);   // 1 select handle. fetch completed - autocommit
    }
}
```

Example 2

```
$sql1 = "select * from db_user";
$sql2 = "select * from db_class where owner_name = ?";

$req = cubrid_prepare ($con, $sql2);
$result = cubrid_execute($con, $sql1);  // 2 handle. fetch completed for only 1 hanlde - 
no autcommit
if ($result) {
  while ($row = cubrid_fetch ($result))
    {
      echo ($row[0]);

      cubrid_bind ($req, 1, $row[0]);
      $res = cubrid_execute ($req);   // fetch completed for all select handles - 
autocommit
    }
}
```

Example 3

```
$sql1 = "select * from db_user";
$sql2 = "insert into a values (?)";

$result = cubrid_execute($con, $sql1);  // 1 select handle. fetch completed - autocommit
if ($result) {
  while ($row = cubrid_fetch ($result))
    {
      echo ($row[0]);

      $req = cubrid_prepare ($con, $sql2);
      cubrid_bind ($req, 1, $row[0]);
      $res = cubrid_execute ($req);    // no autocommit for insert
    }
}
```

Example 4

```
$sql1 = "select * from db_user";
$sql2 = "insert into a values (?)";

$req = cubrid_prepare ($con, $sql2);
$result = cubrid_execute($con, $sql1);  // no autocommit for insert because no fetch
if ($result) {
  while ($row = cubrid_fetch ($result))
    {
      echo ($row[0]);

      cubrid_bind ($req, 1, $row[0]);
      $res = cubrid_execute ($req);   // no autocommit for insert
    }
}
```

cci_execute

Description

This function is used to execute the prepared SQL statement, which is executing cci_prepare(). A request handle, a *flag*, the maximum length of the column to be fetched and the address of the **T_CCI_ERROR** construct to contain the error information are specified as parameters for this function.

The function of retrieving the query result from the server through a *flag* can be classified as synchronous or asynchronous. If the flag is set to **CCI_EXEC_QUERY_ALL**, a synchronous mode (sync_mode) is used to retrieve query results immediately after executing prepared queries if it is set to **CCI_EXEC_ASYNC**, an asynchronous mode (async_mode) is used to retrieve the result immediately each time a query result is created. The *flag* is set to **CCI_EXEC_QUERY_ALL** by default, and in such cases the following rules are applied.

- The return value is the result of the first query.
- If an error occurs in any query, the execution is processed as a failure.
- For a query composed of in a query composed of q1 q2 q3 if an error occurs in q2 after q1 succeeds the execution, the result of q1 remains valid. That is, the previous successful query executions are not rolled back when an error occurs.
- If a query is executed successfully, the result of the second query can be obtained using cci_next_result().

max_col_size is a value that is used to determine the size of the column to be transferred to the client when the type of the column of the prepared query is **CHAR**, **VARCHAR**, **NCHAR**, **VARNCHAR**, **BIT** or **VARBIT**. If it is set to 0, all data is transferred.

Syntax

```
int cci_execute(int req_handle, char flag, int max_col_size, T_CCI_ERROR *err_buf)
```

- *req_handle* : (IN) Request handle of a prepared SQL statement
- *flag* : (IN) Exec flag (**CCI_EXEC_ASYNC** or **CCI_EXEC_QUERY_ALL**)
- *max_col_size* : (IN) The size of the column to be fetched
- *err_buf* : (OUT) Database error buffer

Return Value

- Success
  - **SELECT** : Returns the number of results in sync mode returns 0 in async mode.
  - **INSERT**, **UPDATE** : Returns the number of tuples reflected.
  - Others queries : 0
- Failure : Error code

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_BIND**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**

cci_execute_array

Description

If more than one value are bound to the prepared statement, this gets the values of the variables to be bound and executes the query by binding each value to the variable.

To bind the data, call the cci_bind_param_array_size() function to specify the size of the array, bind each value to the variable by using the cci_bind_param_array() function, and execute the query by calling the **cci_execute_array**() function.

You can get three execution results by calling the cci_execute() function. However, the **cci_execute_array**() function returns the number of queries executed by the query_result variable. You can use the following macro to get the information about the execution result. However, note that the validity check is not performed for each parameter entered in the macro. After using the query_result variable, you must delete the query_result by using the cci_query_result_free() function.

Marco	Return Type	Meaning
CCI_QUERY_RESULT_RESULT	int	the number of results
CCI_QUERY_RESULT_ERR_MSG	char*	error message about query
CCI_QUERY_RESULT_STMT_TYPE	int(T_CCI_CUBRID_STMT enum)	type of query statement

Syntax

```
int cci_execute_array(int req_handle, T_CCI_QUERY_RESULT **query_result, T_CCI_ERROR
*err_buf)
```

- *req_handle* : (IN) Request handle of a prepared SQL statement
- *query_result* : (OUT) Query results (the number of executed queries)
- *err_buf* : (OUT) Database error buffer

Return Value

- Success : The number of executed queries
- Failure : Negative number

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_BIND**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**

Example

```
char *query =
  "update participant set gold = ? where host_year = ? and nation_code = 'KOR'";
int gold[2];
char *host_year[2];
int null_ind[2];
T_CCI_QUERY_RESULT *result;
int n_executed;
...

req = cci_prepare (con, query, 0, &cci_error);
if (req < 0)
{
  printf ("prepare error: %d, %s\n", cci_error.err_code, cci_error.err_msg);
  goto handle_error;
}

gold[0] = 20;
host_year[0] = "2004";

gold[1] = 15;
host_year[1] = "2008";

null_ind[0] = null_ind[1] = 0;
error = cci_bind_param_array_size (req, 2);
if (error < 0)
```

```
{
  printf ("bind_param_array_size error: %d\n", error);
  goto handle_error;
}

error =
  cci_bind_param_array (req, 1, CCI_A_TYPE_INT, gold, null_ind, CCI_U_TYPE_INT);
if (error < 0)
{
  printf ("bind_param_array error: %d\n", error);
  goto handle_error;
}
error =
  cci_bind_param_array (req, 2, CCI_A_TYPE_STR, host_year, null_ind, CCI_U_TYPE_INT);
if (error < 0)
  {
  printf ("bind_param_array error: %d\n", error);
  goto handle_error;
}

n_executed = cci_execute_array (req, &result, &cci_error);
if (n_executed < 0)
{
  printf ("execute error: %d, %s\n", cci_error.err_code,
          cci_error.err_msg);
  goto handle_error;
}
for (i = 1; i <= n_executed; i++)
{
  printf ("query %d\n", i);
  printf ("result count = %d\n", CCI_QUERY_RESULT_RESULT (result, i));
  printf ("error message = %s\n", CCI_QUERY_RESULT_ERR_MSG (result, i));
  printf ("statement type = %d\n",
          CCI_QUERY_RESULT_STMT_TYPE (result, i));
}
error = cci_query_result_free (result, n_executed);
if (error < 0)
{
  printf ("query_result_free: %d\n", error);
  goto handle_error;
}
error = cci_end_tran(con, CCI_TRAN_COMMIT, &cci_error);
if (error < 0)
{
  printf ("end_tran: %d, %s\n", cci_error.err_code, cci_error.err_msg);
  goto handle_error;
}
```

cci_execute_batch

Description

In CCI, multiple jobs can be processed simultaneously when using DML queries such as **INSERT**/**UPDATE**/**DELETE**. cci_execute_arrary() and **cci_execute_batch**() functions can be used to execute such batch jobs. Note that prepared statements cannot be used in the **cci_execute_batch**() function.

Executes sql_stmt as many times as num_sql_stmt specified as a parameter and returns the number of queries executed with the query_result variable. You can use the macro (CCI_QUERY_RESULT_RESULT, CCI_QUERY_RESULT_ERR_MSG, CCI_QUERY_RESULT_STMT_TYPE) available in the cci_execute_array() function to get the information about the execution result.

However, note that the validity check is not performed for each parameter entered in the macro. After using the *query_result* variable, you must delete the query result by using the cci_query_result_free() function.

Syntax

```
int cci_execute_batch(int conn_handle, int num_sql_stmt, char **sql_stmt,
T_CCI_QUERY_RESULT **query_result, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *num_sql_stmt* : (IN) The number of *sql_stmt*s
- *sql_stmt* : (IN) SQL statement array
- *query_result* : (OUT) The results of *sql_stmt*
- *err_buf* : (OUT) Database error buffer

Return Value

- Success : The number of executed queries
- Failure : Negative number

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_CONNECT**

Example

```
char **queries;
  T_CCI_QUERY_RESULT *result;
  int n_queries, n_executed;
...

  count = 3;
  queries = (char **) malloc (count * sizeof (char *));
  queries[0] =
    "insert into athlete(name, gender, nation_code, event) values('Ji-sung Park', 'M',
'KOR', 'Soccer')";
  queries[1] =
    "insert into athlete(name, gender, nation_code, event) values('Joo-young Park', 'M',
'KOR', 'Soccer')";
  queries[2] =
    "select * from athlete order by code desc for orderby_num() < 3";
//calling cci_execute_batch()
  n_executed = cci_execute_batch (con, count, queries, &result, &cci_error);
  if (n_executed < 0)
    {
      printf ("execute_batch: %d, %s\n", cci_error.err_code,
              cci_error.err_msg);
      goto handle_error;
    }
  printf ("%d statements were executed.\n", n_executed);

  for (i = 1; i <= n_executed; i++)
    {
      printf ("query %d\n", i);
      printf ("result count = %d\n", CCI_QUERY_RESULT_RESULT (result, i));
      printf ("error message = %s\n", CCI_QUERY_RESULT_ERR_MSG (result, i));
      printf ("statement type = %d\n",
              CCI_QUERY_RESULT_STMT_TYPE (result, i));
    }

  error = cci_query_result_free (result, n_executed);
  if (error <
0)

    {

      printf ("query_result_free: %d\n", error);
      goto handle_error;
    }
```

cci_execute_result

Description

This function is used to get the execution results (e.g. statement type, result count) performed by cci_execute(). The results of each query are retrieved by CCI_QUERY_RESULT_STMT_TYPE and CCI_QUERY_RESULT_RESULT. The query results used must be deleted by cci_query_result_free.

Syntax

```
int cci_execute_result(int req_handle, T_CCI_QUERY_RESULT **query_result, T_CCI_ERROR
*err_buf)
```

- *req_handle* : (IN) Request handle of a prepared SQL statement
- *query_result* : (OUT) Query results
- *err_buf* : (OUT) Database error buffer

Return Value

- Suceess : The number of queries
- Failure : Negative number

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_COMMUNICATION**

Example

```
T_CCI_QUERY_RESULT *qr;
…

cci_execute( … );
res = cci_execute_result(req_h, &qr, &err_buf);
if (res < 0) {
  /* error */
}
else {
  for (i=1 ; i <= res ; i++) {
    result_count = CCI_QUERY_RESULT_RESULT(qr, i);
    stmt_type = CCI_QUERY_RESULT_STMT_TYPE(qr, i);
  }
  cci_query_result_free(qr, res);
}
```

cci_fetch

Description

Fetches the query result executed by cci_execute() from the server-side CAS and saves it to the client buffer. The cci_get_data() function can be used to identify the data of a specific column from the fetched query result.

Syntax

```
int cci_fetch(int req_handle, T_CCI_ERROR *err_buf)
```

- *req_handle* : (IN) Request handle
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : success)

cci_fetch_buffer_clear

Description

This function is used to clear the records temporarily saved in the client buffer.

Syntax

```
int cci_fetch_buffer_clear(int req_handle)
```

- *req_handle* : (IN) Request handle

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_REQ_HANDLE**

cci_fetch_sensitive

Description

This function is used to send changed values for sensitive columns when the results are sent to the client from the server. If the results by *req_handle* are not sensitive, they are same as the ones by cci_fetch(). The return value of **CCI_ER_DELETED_TUPLE** means that the given tuple has been deleted.

Syntax

```
int cci_fetch_sensitive(int req_handle, T_CCI_ERROR *err_buf)
```

- *req_handle* : (IN) Request handle
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_NO_MORE_DATA**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_DBMS**
- **CCI_ER_DELETED_TUPLE**

cci_fetch_size

Description

This function is used to determine the number of records sent by cci_fetch() from the server to the client.

Syntax

```
int cci_fetch_size(int req_handle, int fetch_size)
```

- *req_handle :* (IN) Request handle
- *fetch_size* : (IN) Fetch size

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_REQ_HANDLE**

cci_get_bind_num

Description

This function is used to get the number of input bindings. If the SQL statement used during preparation is composed of multiple queries, it represents the number of input bindings used in all queries.

Syntax

```
int cci_get_bind_num(int req_handle)
```

- *req_handle* : (IN) Request handle for a prepared SQL statement

Return Value

- The number of input bindings

Error Code

- **CCI_ER_REQ_HANDLE**

cci_get_class_num_objs

Description

This function is used to get the number of objects of the *class_name* class and the number of pages being used. If the flag is configured to 1, an approximate value is fetched; if it is configured to 0, an exact value is fetched.

Syntax

```
int cci_get_class_num_objs(int conn_handle, char *class_name, int flag, int *num_objs, int
*num_pages, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *class_name* : (IN) Class name
- *flag* : (IN) 0 or 1
- *num_objs* : (OUT) The number of objects
- *num_pages* : (OUT) The number of pages
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_CONNECT**

CCI_GET_COLLECTION_DOMAIN

Description

If *u_type* is set, multiset or sequence type, this macro gets the domain of the set, multiset or sequence. If *u_type* is not a set type, the return value is the same as *u_type*.

Syntax

```
#define CCI_GET_COLLECTION_DOMAIN(u_type)
```

Return Value

- Type (CCI_U_TYPE)

cci_get_cur_oid

Description

This function is used to get the OID of the currently fetched records if **CCI_INCLUDE_OID** is configured in execution. The OID is represented as a string for a page, slot or volume.

Syntax

```
intcci_get_cur_oid(int req_handle, char *oid_str_buf)
```

- *conn_handle :* (IN) Request handle
- *oid_str_buf* : (OUT) OID string

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_REQ_HANDLE**

cci_get_data

Description

Gets the *col_noth* value from the currently fetched result. The *type* of the *value* variable is determined according to the given *type* parameter, and the value or the pointer is copied to the value variable accordingly.

For a value to be copied, the memory for the address to be transferred to the *value* variable must have been previously assigned. Note that if a pointer is copied, a pointer in the application client library is returned, so the value becomes invalid next time the **cci_get_data**() function is called.

In addition, the pointer returned by the pointer copy must not be freed. However, if the type is **CCI_A_TYPE_SET**, the memory must be freed by using the cci_set_free() function after using the set because the set is returned after the **T_CCI_SET** type memory is allocated. The following table shows the summary of *type* parameters and data types of their corresponding *value*s.

type	**value Type**	**Meaning**
CCI_A_TYPE_STR	char**	pointer copy
CCI_A_TYPE_INT	int*	value copy
CCI_A_TYPE_FLOAT	float*	value copy
CCI_A_TYPE_DOUBLE	double*	value copy
CCI_A_TYPE_BIT	**T_CCI_BIT***	value copy (pointer copy for each member)
CCI_A_TYPE_SET	**T_CCI_SET***	memory alloc and value copy
CCI_A_TYPE_DATE	**T_CCI_DATE***	value copy
CCI_A_TYPE_BIGINT	int64_t* (For Windows : __int64*)	value copy
CCI_A_TYPE_BLOB	**T_CCI_BLOB**	memory alloc and value copy

CCI_A_TYPE_CLOB	**T_CCI_CLOB**	memory alloc and value copy

Syntax

```
int cci_get_data(int req_handle, int col_no, int type, void *value, int *indicator)
```

- *req_handle* : (IN) Request handle
- *col_no* : (IN) One-based column index. It starts with 1.
- *type* : (IN) Data type (defined in the **T_CCI_A_TYPE**) of *value* variable
- *value* : (OUT) Variable address for data to be stored
- *indicator* : (OUT) **NULL** indicator (-1 : **NULL**)
- if *type* is **CCI_A_TYPE_STR** : -1 is returned in case of **NULL**; the length of character string stored in *value* is returned, otherwise.
- if *type* is **CCI_A_TYPE_STR** : -1 is returned in case of **NULL**, 0 is returned, otherwise.

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_TYPE_CONVERSION**
- **CCI_ER_COLUMN_INDEX**
- **CCI_ER_ATYPE**

cci_get_db_parameter

Description

This function is used to get a parameter specified in the database. The data type of *value* for *param_name* is shown in the table below.

param_name	value Type	note
CCI_PARAM_ISOLATION_LEVEL	int*	get/set
CCI_PARAM_LOCK_TIMEOUT	int*	get/set
CCI_PARAM_MAX_STRING_LENGTH	int*	get only

Syntax

```
int cci_get_db_parameter(int conn_handle, T_CCI_DB_PARAM param_name, void *value,
T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *param_name* : (IN) Database parameter name
- *value* : (OUT) Parameter value
- *err_buf* : (OUT) Database error buffer

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_PARAM_NAME**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_CONNECT**

cci_get_db_version

Description

This function is used to get the Database Management System (DBMS) version.

Syntax

```
int cci_get_db_version(int conn_handle, char *out_buf, int out_buf_size)
```

- *conn_handle* : (IN) Connection handle
- *out_buf* : (OUT) Result buffer
- *out_buf_size* : (IN) *oub_buf* size

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_CONNECT**

cci_get_result_info

Description

If the prepared statement is **SELECT**, the **T_CCI_COL_INFO** struct that stores the column information about the execution result can be obtained by using this function. If it is not **SELECT**, **NULL** is returned and the *num* value becomes 0.

You can access the **T_CCI_COL_INFO** struct directly to get the column information from the struct, but you can also use a macro to get the information, which is defined as follows. The address of the **T_CCI_COL_INFO** struct and the column index are specified as parameters for each macro. The macro can be called only for the **SELECT** query. Note that the validity check is not performed for each parameter entered in each macro. If the return type of the macro is char*, do not free the memory pointer.

Macro	Return Type	Meaning
CCI_GET_RESULT_INFO_TYPE	T_CCI_U_TYPE	column type
CCI_GET_RESULT_INFO_SCALE	short	column scale
CCI_GET_RESULT_INFO_PRECISION	int	column precision
CCI_GET_RESULT_INFO_NAME	char*	column name
CCI_GET_RESULT_INFO_ATTR_NAME	char*	column attribute name
CCI_GET_RESULT_INFO_CLASS_NAME	char*	column class name
CCI_GET_RESULT_INFO_IN_NON_NULL	char(0 or 1)	whether a column is NULL

Syntax

```
T_CCI_COL_INFO* cci_get_result_info(int req_handle, T_CCI_SQLX_CMD *cmd_type, int *num)
```

- *req_handle* : (IN) Request handle for a prepared SQL statement
- *cmd_type* : (OUT) Command type
- *num* : (OUT) The number of columns in the **SELECT** statement (if *cmd_type* is **SQLX_CMD_SELECT**)

Return Value

- Success : Result info pointer
- Failure : **NULL**

Example

```
col_info = cci_get_result_info (req, &cmd_type, &col_count);
  if (col_info == NULL)
    {
      printf ("get_result_info error: %d, %s\n", cci_error.err_code,
              cci_error.err_msg);
      goto handle_error;
    }
  for (i = 1; i <= col_count; i++)
    {
      printf ("%-12s = %d\n", "type", CCI_GET_RESULT_INFO_TYPE (col_info, i));
      printf ("%-12s = %d\n", "scale",
              CCI_GET_RESULT_INFO_SCALE (col_info, i));
      printf ("%-12s = %d\n", "precision",
              CCI_GET_RESULT_INFO_PRECISION (col_info, i));
      printf ("%-12s = %s\n", "name", CCI_GET_RESULT_INFO_NAME (col_info, i));
      printf ("%-12s = %s\n", "attr_name",
              CCI_GET_RESULT_INFO_ATTR_NAME (col_info, i));
      printf ("%-12s = %s\n", "class_name",
              CCI_GET_RESULT_INFO_CLASS_NAME (col_info, i));
      printf ("%-12s = %s\n", "is_non_null",
              CCI_GET_RESULT_INFO_IS_NON_NULL (col_info,i) ? "true" : "false");
```

CCI_GET_RESULT_INFO_ATTR_NAME

Description

This macro is used to get the actual attribute name of the *index* -th column of a prepared **SELECT** statement. If there is no name for the attribute (constant, function, etc), " " (empty string) is returned. It does not check whether the specified argument, *res_info*, is **NULL** and whether *index* is valid. You cannot delete the returned memory pointer with **free**().

Syntax

```
#define CCI_GET_RESULT_INFO_ATTR_NAME(T_CCI_COL_INFO* res_info, int index)
```

- *res_info* : (IN) pointer to the column information fetched by cci_get_result_info
- *index* : (IN) Column index

Return Value

- Attribute name (char*)

CCI_GET_RESULT_INFO_CLASS_NAME

Description

This macro is used to get the *index* -th class name of a prepared **SELECT** statement. It does not check whether the specified argument, *res_info*, is **NULL** and whether *index* is valid. You cannot delete the returned memory pointer with **free**(). The returned value can be **NULL**.

Syntax

```
#define CCI_GET_RESULT_INFO_CLASS_NAME(T_CCI_COL_INFO* res_info, int index)
```

- *res_info* : (IN) Column info pointer by cci_get_result_info
- *index* : (IN) Column index

Return Value

- Class name (char*)

CCI_GET_RESULT_INFO_IS_NON_NULL

Description

This macro is used to get a value indicating whether the *index* -th column of a prepared **SELECT** statement is nullable. It does not check whether the specified argument, *res_info*, is **NULL** and whether *index* is valid.

Syntax

```
#define CCI_GET_RESULT_INFO_IS_NON_NULL(T_CCI_COL_INFO* res_info, int index)
```

- *res_info* : (IN) Column info pointer by cci_get_result_info
- *index* : (IN) Column index

Return Value

- 0 : nullable
- 1 : non **NULL**

CCI_GET_RESULT_INFO_NAME

Description

This macro is used to get the *index* -th column name of a prepared **SELECT** statement. It does not check whether the specified argument, *res_info*, is **NULL** and whether *index* is valid. You cannot delete the returned memory pointer with **free**().

Syntax

```
#define CCI_GET_RESULT_INFO_NAME(T_CCI_COL_INFO* res_info, int index)
```

- *res_info* : (IN) Column info pointer to cci_get_result_info
- *index* : (IN) Column index

Return Value

- Column name (char*)

CCI_GET_RESULT_INFO_PRECISION

Description

This macro is used to get the *index* -th precision of a prepared **SELECT** statement. It does not check whether the specified argument, *res_info*, is **NULL** and whether *index* is valid.

Syntax

```
#define CCI_GET_RESULT_INFO_PRECISION(T_CCI_COL_INFO* res_info, int index)
```

- *res_info* : (IN) Column info pointer by cci_get_result_info
- *index* : (IN) Column index

Return Value

- Precision (int)

CCI_GET_RESULT_INFO_SCALE

Description

This macro is used to get the *index* -th column's scale of a prepared **SELECT** statement. It does not check whether the specified argument, *res_info*, is **NULL** and whether *index* is valid.

Syntax

```
#define CCI_GET_RESULT_INFO_SCALE(T_CCI_COL_INFO* res_info, int index)
```

- *res_info* : (IN) Column info pointer by cci_get_result_info
- *index* : (IN) Column index

Return Value

- Scale (int)

CCI_GET_RESULT_INFO_TYPE

Description

This macro is used to get the *index* -th column type of a prepared **SELECT** statement. It does not check whether the specified argument, *res_info*, is **NULL** and whether *index* is valid.

Syntax

```
#define CCI_GET_RESULT_INFO_TYPE(T_CCI_COL_INFO* res_info, int index)
```

- *res_info* : (IN) pointer to the column information fetched by cci_get_result_info
- *index* : (IN) Column index

Return Value

- Column type (**T_CCI_U_TYPE**)

cci_blob_new

Description

This function creates an empty file where LOB data is stored and returns Locator referring to the data to blob structure.

Syntax

```
int cci_blob_new(int conn_handle, T_CCI_BLOB* blob, T_CCI_ERROR* error_buf)
```

- *conn_handle* : (IN) Connection handle
- *blob* : (OUT) LOB Locator
- *error_buf* : (OUT) Error buffer

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_CONNECT**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_DBMS**

cci_blob_size

Description

This function returns data file size that is specified in *blob*.

Syntax

```
long long cci_blob_size (T_CCI_BLOB blob)
```

- *blob* : (IN) LOB Locator

Return Value

- Size of BLOB/CLOB data file

cci_blob_write

Description

This function reads data as long as the value of *length* from *buf* and then stores the value from *start_pos* in **LOB** data file.

Syntax

```
int cci_blob_write(int conn_handle, T_CCI_BLOB blob, long start_pos, int length, const
char *buf, T_CCI_ERROR* error_buf)
```

- *conn_handle* : (IN) Connection handle
- *blob* : (IN) LOB Locator
- *start_pos* : (IN) Index location of LOB data file
- *length* : (IN) Data length from buffer
- *error_buf* : (OUT) Error buffer

Return Value

- Size of written value (> = : success)
- Error code (0 : success)

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_DBMS**

cci_blob_write

Description

This function reads **LOB** data as long as the value of *length* from *start_pos* in **LOB** data file, stores the value in *buf*, and then returns it.

Syntax

```
int cci_blob_read(int conn_handle, T_CCI_BLOB blob, long start_pos, int length, const char
*buf, T_CCI_ERROR* error_buf)
```

- *conn_handle* : (IN) Connection handle
- *blob* : (IN) LOB Locator

- *start_pos* : (IN) Index location of LOB data file
- *length* : (IN) Data length from buffer
- *error_buf* : (OUT) Error buffer

Return Value

- Size of read value (> = : success)
- Error code (0 : success)

Error Code

- **CCI_ER_INVALID_LOB_READ_POS**
- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_DBMS**

cci_blob_free

Description

This function frees memory of BLOC structure.

Syntax

```
int cci_blob_free (T_CCI_BLOB blob);
```

Return Value

- None

CCI_IS_SET_TYPE, CCI_IS_MULTISET_TYPE, CCI_IS_SEQUENCE_TYPE, CCI_IS_COLLECTION_TYPE

Description

This macro is used to check whether *u_type* is set, multiset or sequence type.

Syntax

```
#define CCI_IS_SET_TYPE(u_type)
#define CCI_IS_MULTISET_TYPE(u_type)
#define CCI_IS_SEQUENCE_TYPE(u_type)
#define CCI_IS_COLLECTION_TYPE(u_type)
```

Return Value

- **CCI_IS_SET_TYPE**
  - 1 : set
  - 0 : not set
- **CCI_IS_MULTISET_TYPE**
  - 1 : multiset
  - 0 : not multiset
- **CCI_IS_SEQUENCE_TYPE**
  - 1 : sequence
  - 0 : not sequence

- **CCI_IS_SET_TYPE**
  - 1 : collection (set, multiset, sequence)
  - 0 : not collection

cci_is_updatable

Description

It is used to check whether the SQL statement, which executed cci_prepare(), is updatable. If it is updatable, 1 is returned.

Syntax

```
int cci_is_updatable(int req_handle)
```

- *req_handle* : (IN) Request handle for a prepared SQL statement

Return Value

- 1 : updatable
- 0 : not updatable

Error Code

- **CCI_ER_REQ_HANDLE**

cci_next_result

Description

The function is used to get results of next query if **CCI_EXEC_QUERY_ALL** *flag* is set upon cci_execute(). The information about the query fetched by next_result can be obtained with cci_get_result_info. If next_result is executed successfully, the database is updated with the information of the current query.

The error code **CAS_ER_NO_MORE_RESULT_SET** means that no more result set exists.

Syntax

```
int cci_next_result(int req_handle, T_CCI_ERROR *err_buf)
```

- *req_handle* : (IN) Request handle of a prepared statement
- *err_buf* : (OUT) Database error buffer

Return Value

- Success
- **SELECT** (sync mode) : the number of results, (async mode) : 0
- **INSERT**, **UPDATE** : the number of records reflected
- Others : 0
- Failure : Error code

Error Code

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**

cci_oid

Description

CCI_OID_DROP : Deletes the given oid.

CCI_OID_IS_INSTANCE : Checks whether the given oid is an instance oid.

CCI_OID_LOCK_READ : Sets a read lock on the given oid.

CCI_OID_LOCK_WRITE : Sets a write lock on the given oid.

Syntax

```
intcci_oid(int conn_handle, T_CCI_OID_CMD cmd, char *oid_str, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *cmd* : (IN) CCI_OID_DROP, CCI_OID_IS_INSTANCE, CCI_OID_LOCK_READ, CCI_OID_LOCK_WRITE
- *oid_str* : (IN) oid
- *err_buf* : (OUT) Database error buffer

Return Value

- CCI_OID_IS_INSTANCE
- 0 : non-instance
- 1 : instance
- < 0 : error
- CCI_OID_DROP, CCI_OID_LOCK_READ, CCI_OID_LOCK_WRITE
- Error code (0 : success, negative : failure)

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_OID_CMD**
- **CCI_ER_OBJECT**
- **CCI_ER_DBMS**

cci_oid_get

Description

This function is used to get the attribute values of the given oid. *attr_name* is an array of the attributes, and it must end with **NULL**. If *attr_name* is NULL, the information of all attributes is fetched. The request handle has the same form as when the SQL statement "SELECT attr_name FROM oid_class WHERE oid_class = oid" is executed.

Syntax

```
int cci_oid_get(int conn_handle, char *oid_str, char **attr_name, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) oid
- *attr_name* : (IN) A list of attributes
- *err_buf* : (OUT) Database error buffer

Return Value

- Success : Request handle
- Failure : Error code

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_CONNECT**

cci_oid_get_class_name

Description

This function is used to get the class name of the given oid.

Syntax

```
intcci_oid_get_class_name(int conn_handle, char *oid_str, char *out_buf, int out_buf_len,
T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) oid
- *out_buf* : (OUT) Out buffer
- *out_buf_len* : (IN) *out_buf* length
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_OBJECT**
- **CCI_ER_DBMS**

cci_oid_put

Description

This function is used to configure the *attr_name* attribute values of the given oid to *new_val_str*. The last value of *attr_name* must be **NULL**. Any value of any type must be represented as a string. The value represented as a string is applied to the database after being converted depending on the attribute type on the server. To insert a **NULL** value, configure the value of *new_val_str*[i] to **NULL**.

Syntax

```
intcci_oid_put(int conn_handle, char *oid_str, char **attr_name, char **new_val_str,
T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) oid
- *attr_name* : (IN) A list of attribute names
- *new_val_str* : (IN) A list of new values
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**

cci_oid_put2

Description

This function is used to set the *attr_name* attribute values of the given oid to *new_val*. The last value of *attr_name* must be **NULL**. To insert a **NULL** value, set the value of *new_val*[i] to **NULL**.

The type of *new_val*[i] for *a_type* is shown in the table below.

Type of new_val[i] for a_type

Type	value type
CCI_A_TYPE_STR	char*
CCI_A_TYPE_INT	int*
CCI_A_TYPE_FLOAT	float*
CCI_A_TYPE_DOUBLE	double*
CCI_A_TYPE_BIT	**T_CCI_BIT***
CCI_A_TYPE_SET	**T_CCI_SET**
CCI_A_TYPE_DATE	**T_CCI_DATE***
CCI_A_TYPE_BIGINT	int64_t (For Windows : __int64)

Syntax

```
intcci_oid_put2(int conn_handle, char *oidstr, char **attr_name, void **new_val, int
*a_type, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) oid
- *attr_name* : (IN) A list of attribute names
- *new_val* : (IN) A new value array
- *a_type* : (IN) *new_val* type array
- *err_buf :* (OUT) Database error buffer

Return Value

- Error code (0 : success, negative number : failure)

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**

Example

```
char *attr_name[array_size]
void *attr_val[array_size]
int a_type[array_size]
int int_val

…

attr_name[0] = "attr_name0"
attr_val[0] = &int_val
a_type[0] = CCI_A_TYPE_INT
attr_name[1] = "attr_name1"
```

```
attr_val[1] = "attr_val1"
a_type[1] = CCI_A_TYPE_STR

…
attr_name[num_attr] = NULL

res = cci_put2(con_h, oid_str, attr_name, attr_val, a_type, &error)
```

cci_prepare

Description

This function is used to prepare SQL execution by acquiring request handle for SQL statements. If a SQL statement consists of multiple queries, the preparation is performed only for the first query. With the parameter of this function, an address to **T_CCI_ERROR** where connection handle, SQL statement, *flag*, and error information are saved.

CCI_PREPARE_UPDATABLE or **CCI_PREPARE_INCLUDE_OID** can be configured in *flag*. If **CCI_PREPARE_UPDATABLE** is configured, updatable result set is created and **CCI_PREPARE_INCLUDE_OID** is automatically configured. However, not all updatable result sets are created even though **CCI_PREPARE_UPDATABLE** is configured. So you need to check if the results are updatable by using cci_is_updatable after preparation.

The conditions of updatable queries are as follows:

- A query must be **SELECT**.
- OID must be contained in the query result.
- The column to be updated must be the one that belongs to the table specified in the **FROM** clause.

Syntax

```
int cci_prepare(int conn_handle, char *sql_stmt, char flag,T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *sql_stmt* : (IN) SQL statement
- *flag* : (IN) prepare flag (**CCI_PREPARE_INCLUDE_OID** or **CCI_PREPARE_UPDATABLE**)
- *err_buf* : (OUT) Database error buffer

Return Value

- Success : Request handle ID (int)
- Failure : Error code (negative)

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_STR_PARAM**
- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_CONNECT**

CCI_QUERY_RESULT_ERR_MSG

Description

This macro is used to get error messages for the cci_execute_batch query. If there is no error message, " " (empty string) is returned. It does not check whether the specified argument, *query_result*, is **NULL**, and whether *index* is valid.

Syntax

```
#define CCI_QUERY_RESULT_ERR_MSG(T_CCI_QUERY_RESULT* query_result, int index)
```

- *query_result* : (IN) Query results of cci_execute_batch
- *index* : (IN) Column index (base : 1)

Return Value

- Error message

cci_query_result_free

Description

This function is used to delete query result.

Syntax

```
int cci_query_result_free(T_CCI_QUERY_RESULT* query_result, int num_query)
```

- *query_result* : (IN) Query results of cci_execute_batch
- *num_query* : (IN) The number of arrays in *query_result*

Return Value

- Error code (0 : success, negative number : failure)

Example

```
T_CCI_QUERY_RESULT *qr;
char **sql_stmt;


res = cci_execute_array(conn, &qr, &err_buf);


cci_query_result_free(qr, res);
```

CCI_QUERY_RESULT_RESULT

Description

This macro is used to get the result count of the cci_execute_batch query. It does not check whether the specified argument, *query_result*, is **NULL** and whether *index* is valid.

Syntax

```
#define CCI_QUERY_RESULT_RESULT(T_CCI_QUERY_RESULT* query_result, int index)
```

- *query_result* : (IN) Query results of cci_execute_batch
- *index* : (IN) Column index (base : 1)

Return Value

- Result count

CCI_QUERY_RESULT_STMT_TYPE

Description

This macro is used to get the statement type of the cci_execute_batch query. It does not check whether the specified argument, *query_result*, is **NULL** and whether *index* is valid.

Syntax

```
#define CCI_QUERY_RESULT_STMT_TYPE(T_CCI_QUERY_RESULT* query_result, int index)
```

- *query_result* : (IN) Query results of cci_execute_batch
- *index* : (IN) Column index (base : 1)

Return Value

- Statement type (**T_CCI_SQLX_CMD**)

cci_savepoint

Description

This function is used to configure a savepoint or performs transaction rollback to a specified savepoint. Sets a savepoint if cmd is set to **CCI_SP_SET**. If it is set to **CCI_SP_ROLLBACK**, the transaction is rolled back to the specified savepoint.

Syntax

```
intcci_savepoint(int conn_handle, T_CCI_SAVEPOINT_CMD cmd, char* savepoint_name,
T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *cmd* : (IN) CCI_SP_SET or CCI_SP_ROLLBACK
- savepoint_name : (IN) Savepoint name
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code

Example

```
con = cci_connect( ...);
.../* query execute */

/* sets a savepoint named "savepoint1"
cci_savepoint(con, CCI_SP_SET, "savepoint1", err_buf);

... /* query execute */

/* rolls back the set savepoint to "savepoint1" */
cci_savepoint(con, CCI_SP_ROLLBACK, "savepoint1", err_buf);
```

cci_schema_info

Description

This function is used to get schema information. If it is performed successfully, the results are managed by the request handle and can be fetched by fetch and getdata. If you want to retrieve a *class_name* of attr_name by pattern matching, configure the *flag*.

Two flags, **CCI_CLASS_NAME_PATTERN_MATCH** and **CCI_ATTR_NAME_PATTERN_MATCH**, are used for pattern matching. You can configure these two *flag*s by using the OR operator (|). Performance may significantly decrease if pattern matching is used.

The following table shows records composition of each *type*.

Record Composition of Each Type

Type	Column Order	Column Name	Column Type
CCI_SCH_CLASS	1	NAME	char*
	2	TYPE	short 0 : system

			class 1 : vclass 2 : class 3 : proxy
CCI_SCH_VCLASS	1	NAME	char*
	2	TYPE	short 1 : vclass 3 : proxy
CCI_SCH_ATTRIBUTE	1	NAME	char*
	2	DOMAIN	int
	3	SCALE	int
	4	PRECISION	int
	5	INDEXED	int 1 : indexed
	6	NON_NULL	int 1 : non null
	7	SHARED	int 1 : shared
	8	UNIQUE	int 1 : unique
	9	DEFAULT	void*
	10	ATTR_ORDER	int base : 1
	11	CLASS_NAME	char*
	12	SOURCE_CLASS	char*
	13	IS_KEY	short 1 : key
CCI_SCH_CLASS_METHOD	1	NAME	char*
	2	RET_DOMAIN	int
	3	ARG_DOMAIN	char*
CCI_SCH_METHOD_FILE	1	METHOD_FILE	char*
CCI_SCH_super class	1	CLASS_NAME	char*
	2	TYPE	short
CCI_SCH_SUBCLASS	1	CLASS_NAME	char*
	2	TYPE	short
CCI_SCH_CONSTRAINT	1	TYPE 0 : unique 1 : index 2 : reverse unique 3 : reverse index	int
	2	NAME	char*
	3	ATTR_NAME	char*
	4	NUM_PAGES	int
	5	NUM_KEYS	int
	6	PRIMARY_KEY 1 : primary key	short

	7	KEY_ORDER	short base : 1
CCI_SCH_TRIGGER	1	NAME	char*
	2	STATUS	char*
	3	EVENT	char*
	4	TARGET_CLASS	char*
	5	TARGET_ATTR	char*
	6	ACTION_TIME	char*
	7	ACTION	char*
	8	PRIORITY	float
	9	CONDITION_TIME	char*
	10	CONDITION	char*
CCI_SCH_CLASS_PRIVILEGE	1	CLASS_NAME	char*
	2	PRIVELEGE	char*
	3	GRANTABLE	char*
CCI_SCH_ATTR_PRIVILEGE	1	ATTR_NAME	char*
	2	PRIVILEGE	char*
	3	GRANTABLE	char*
CCI_SCH_PRIMARY_KEY	1	CLASS_NAME	char*
	2	ATTR_NAME	char*
	3	KEY_SEQ	short base : 1
	4	KEY_NAME	char*
CCI_SCH_IMPORTED_KEY Used to retrieve primary key columns that are referred by a foreign key column in a given table. The results are sorted by PKTABLE_NAME and KEY_SEQ. If this type is specified as a parameter, a foreign key table is specified for *class_name*, and NULL is specified for *attr_name*.	1	PKTABLE_NAME	char**
	2	PKCOLUMN_NAME	char**
	3	FKTABLE_NAME	char**
	4	FKCOLUMN_NAME	char**
	5	KEY_SEQ	char**
	6	UPDATE_ACTION -cascade=0 -restrict=1 -no action=2 -set null=3	Int*
	7	DELETE_ACTION -cascade=0 -restrict=1 -no action=2 -set null=3	Int*
	8	FK_NAME	char**
	9	PK_NAME	char**
CCI_SCH_EXPORTED_KEYS Used to retrieve primary key columns that are referred by all foreign key columns. The results are sorted by FKTABLE_NAME and KEY_SEQ.	1	PKTABLE_NAME	char**
	2	PKCOLUMN_NAME	char**
	3	FKTABLE_NAME	char**

If this type is specified as a parameter, a primary key table is specified for *class_name*, and NULL is specified for *attr_name*.	4	FKCOLUMN_NAME	char**
	5	KEY_SEQ	char**
	6	UPDATE_ACTION -cascade=0 -restrict=1 -no action=2 -set null=3	Int*
	7	DELETE_ACTION -cascade=0 -restrict=1 -no action=2 -set null=3	Int*
	8	FK_NAME	char**
	9	PK_NAME	char**
CCI_SCH_CROSS_REFERENCE Used to retrieve foreign key information when primary keys and foreign keys in a given table are cross referenced. The results are sorted by FKTABLE_NAME and KEY_SEQ. If this type is specified as a parameter, a primary key is specified for *class_name*, and a foreign key table is specified for *attr_name*.	1	PKTABLE_NAME	char**
	2	PKCOLUMN_NAME	char**
	3	FKTABLE_NAME	char**
	4	FKCOLUMN_NAME	char**
	5	KEY_SEQ	char**
	6	UPDATE_ACTION -cascade=0 -restrict=1 -no action=2 -set null=3	Int*
	7	DELETE_ACTION -cascade=0 -restrict=1 -no action=2 -set null=3	Int*
	8	FK_NAME	char**
	9	PK_NAME	char**

Pattern match

CCI_SCH_TYPE	**Class name**	**ATTR_name**
CCI_SCH_CLASS (VCLASS)	O	none
CCI_SCH_ATTRIBUTE (CLASS ATTRIBUTE)	O	O
CCI_SCH_CLASS_PRIVILEGE	O	none
CCI_SCH_ATTR_PRIVILEGE	X	O
CCI_SCH_PRIMARY_KEY	O	none

If the pattern flag is not configured, exact string matching is used for the given class or attribute name. Therefore, there is no result if NULL is given. If the name of the class or attribute is NULL when the pattern flag is configured, the result is the same as when "%" is used.

Note TYPE column of CCI_SCH_CLASS and CCI_SCH_VCLASS : The proxy type is added. When used in OLEDB, ODBC or PHP, vclass is represented without distinguishing between proxy and vclass.

Syntax

```
int cci_schema_info(int conn_handle, T_CCI_SCHEMA_TYPE type, char *class_name, char
*attr_name, char flag, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *type* : (IN) Schema type
- *class_name* : (IN) Class name or NULL
- *attr_name* : (IN) Attribute name of NULL
- *flag* : (IN) Pattern matching flag (**CCI_CLASS_NAME_PATTERN_MACTH** or **CCI_CLASS_NAME_PATTERN_MATCH**)
- *err_buf* : (OUT) Database error buffer

Return Value

- Success : Request handle
- Failure : Error code

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_SCHEMA_TYPE**
- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_CONNECT**

cci_set_db_parameter

Description

This function is used to configure a database parameter. For the type of *value* for *param_name*, see cci_get_db_parameter().

Syntax

```
int cci_set_db_parameter(int conn_handle, T_CCI_DB_PARAM param_name, void* value,
T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *param_name* : (IN) Database parameter name
- *value* : (IN) Parameter value
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : success)

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_PARAM_NAME**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_CONNECT**

cci_set_element_type

Description

This function is used to get the element type for the set fetched by CCI_A_TYPE_SET with cci_get_data().

Syntax

```
intcci_set_element_type(T_CCI_SET set)
```

- *set* : (IN) cci set pointer

Return Value

- Type

cci_set_free

Description

This function is used to release the memory assigned to **T_CCI_SET gotten** by **CCI_A_TYPE_SET** with cci_get_data().

Syntax

```
voidcci_set_free(T_CCI_SET set)
```

- *set* : (IN) cci set pointer

Return Value

- None

cci_set_get

Description

This function is used to get the index-th data for the set fetched by **CCI_A_TYPE_SET** with cci_get_data(). The data type of *value* for *a_type* is shown in the table below.

a_type	***value* Type**
CCI_A_TYPE_STR	char**
CCI_A_TYPE_INT	int*
CCI_A_TYPE_FLOAT	float*
CCI_A_TYPE_DOUBLE	double*
CCI_A_TYPE_BIT	**T_CCI_BIT***
CCI_A_TYPE_DATE	**T_CCI_DATE***
CCI_A_TYPE_BIGINT	int64_t* (For Windows : __int64*)

Syntax

```
intcci_set_get(T_CCI_SET set, int index, T_CCI_A_TYPE a_type, void *value, int *indicator)
```

- *set* : (IN) cci set pointer
- *index* : (IN) Set index (base : 1)
- *a_type* : (IN) Type
- *value* : (OUT) Result buffer
- *indicator* : (OUT) Null indicator

Return Value

- Error code

cci_set_isolation_level

Description

This function is used to set the transaction isolation level of connections. All further transactions for the given connections work as *new_isolation_level*.

Note If the transaction isolation level is set by cci_set_db_parameter(), only the current transaction is affected. When the transaction is complete, the transaction isolation level returns to the one set by CAS. You must use **cci_set_isolation_level**() to set the isolation level for the entire connection.

Syntax

```
intcci_set_isolation_level(int conn_handle, T_CCI_TRAN_ISOLATION  new_isolation_level,
T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *new_isolation_level* : (IN) Transaction isolation level
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code

Error Code

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_ISOLATION_LEVEL**
- **CCI_ER_DBMS**

cci_set_make

Description

This function is used to make a set of a new **CCI_A_TYPE_SET** type. The created set is sent to the server as **CCI_A_TYPE_SET** by cci_bind_param(). The memory for the set created by **cci_set_make()** must be freed by **cci_set_free()**. The type of *value* for *u_type* is shown in the table below.

Syntax

```
intcci_set_make(T_CCI_SET *set, T_CCI_U_TYPE u_type, int size, void *value, int *indicator)
```

- set : (IN) cci set pointer
- *u_type* : (IN) Element type
- *size* : (IN) Set size
- *value* : (IN) Set element
- *indicator* : (IN) Null indicator array

Return Value

- Error code

cci_set_max_row

Description

This function is used to configure the maximum number of records for the results of the **SELECT** statement executed by cci_execute. If the *max* value is 0, it is the same as not setting the value.

Syntax

```
intcci_set_max_row(int req_handle, int max)
```

- *req_handle* : (IN) Connection handle
- *max* : (IN) The maximum number of rows

Return Value

- Error code

Example

```
req = cci_prepare( … );
cci_set_max_row(req, 1);
cci_execute( … );
```

cci_set_size

Description

This function is used to get the number of elements for the set fetched by **CCI_A_TYPE_SET** with cci_get_data().

Syntax

```
intcci_set_size(T_CCI_SET set)
```

- *set* : (IN) cci set pointer

Return Value

- Size

CUBRID Manager

This chapter explains how to use the CUBRID Manager, a GUI-based database management and query tool. The CUBRID Manager facilitates various management tasks and provides the "Query Editor," allowing users to execute SQL statements against the connected database.

CUBRID Manager consists of a database server, a manager server running on a host where the Broker is installed, and a GUI client. The CUBRID Manager client utility is written in Java and can be run in any environment that supports Java.

This chapter covers the following topics:

- Introduction to the CUBRID Manager
- Running CUBRID Manager
- Client Features of the CUBRID Manager

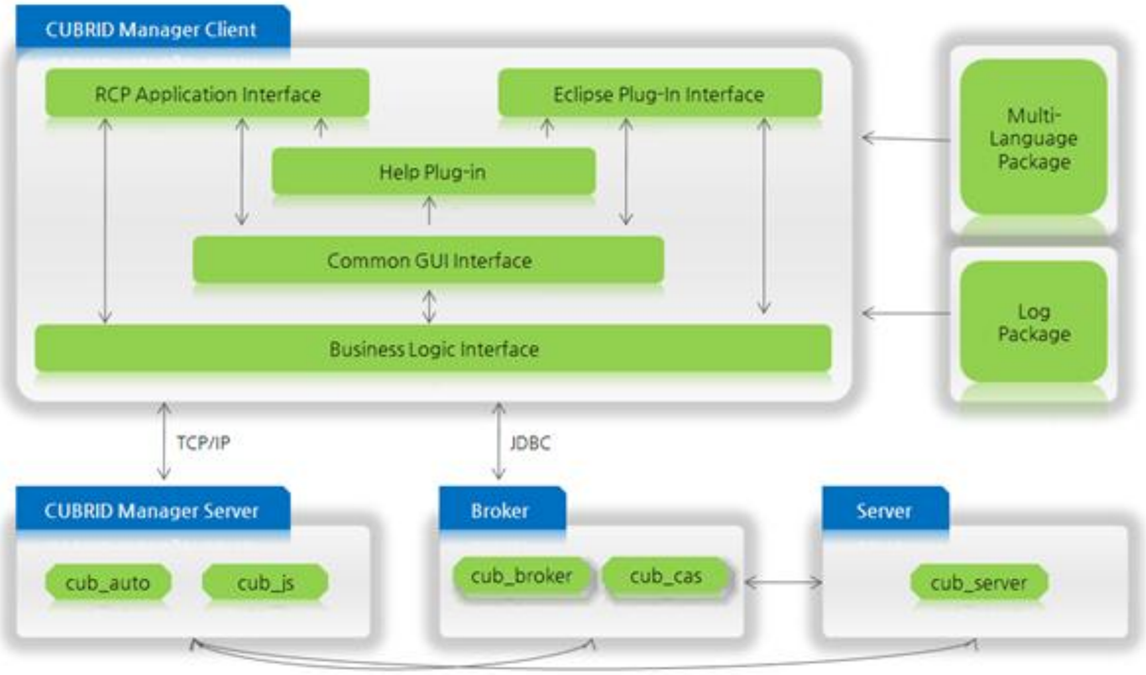
Introduction to CUBRID Manager

CUBRID Manager Architecture

The figure below shows the architecture of the CUBRID Manager. The CUBRID Manager server runs on a host installed with the Database Server and Broker. The CUBRID Manager client is connected to this CUBRID Manager server.

The CUBRID Manager server consists of a **cub_auto** process and a **cub_js** process. To connect to the CUBRID Manager server from the CUBRID Manager client, the CUBRID Manager server must be running, and the network port (TCP/IP) that corresponds to the **cub_auto** process and the **cub_js** process must be set. To perform a query, the Broker to which JDBC connects also must be running.

- The **cub_auto** process authenticates the CUBRID Manager client users, performs periodic automations, and collects analysis information.
- The **cub_js** process executes user requests received from the CUBRID Manager client.



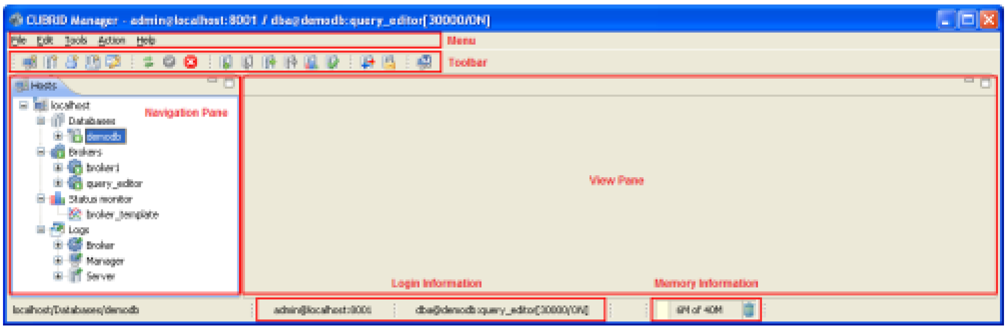
For more information on running the server, see Configuring CUBRID Manager Server.

CUBRID Manager Client

RCP Application

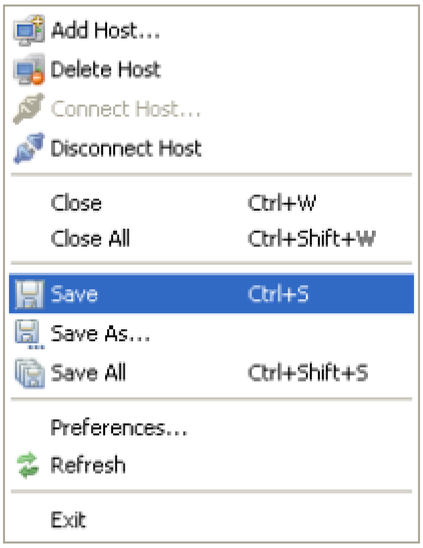
Query Editor Layout

The Client consists of 4 areas - the menu, search, view, and status bar. Each area provides unique information.



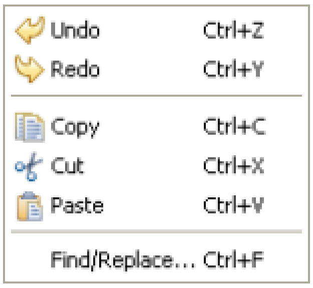
File Menu

The [File] menu consists of host menu, file menu, CUBRID Manager preference menu and others.



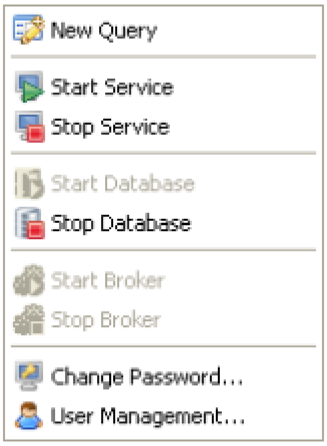
Edit Menu

The [Edit] menu consists of options used in the CUBRID Manager for editing.



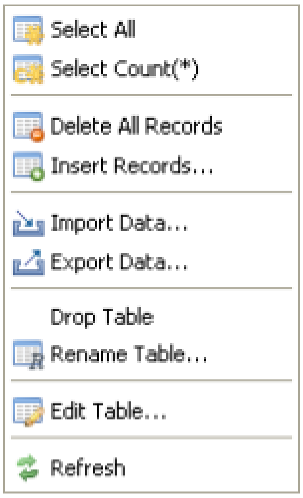
Tools

The [Tools] menu is the main menu of the CUBRID Manager, and consists of service control menu such as Start/Stop Service, and User Management.



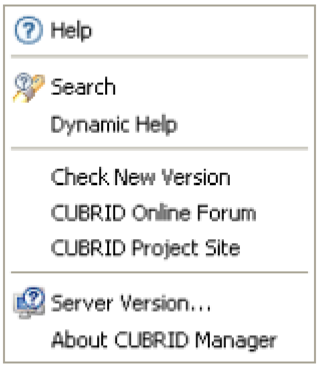
Action Menu

The [Action] menu is a dynamic menu, showing available actions by a right-click in the navigation pane. The following shows the options of the [Action] menu, displayed by a right-click of a table.



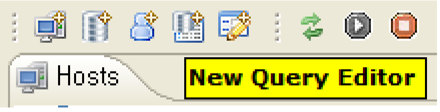
Help Menu

The [Help] menu consists of Help, which allows you to browse CUBRID Database Manual, Dynamic Help, and Search options. To check CUBRID tips or development news, select [CUBRID Online Forum] or [CUBRID Project Site].

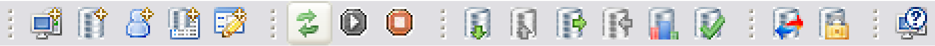


Toolbar

The toolbar houses frequently used functionalities of the CUBRID Manager. The functionalities automatically activate or deactivate depending on their availability. The figure below shows a tooltip that appears when hovering the mouse pointer on an icon on the toolbar.



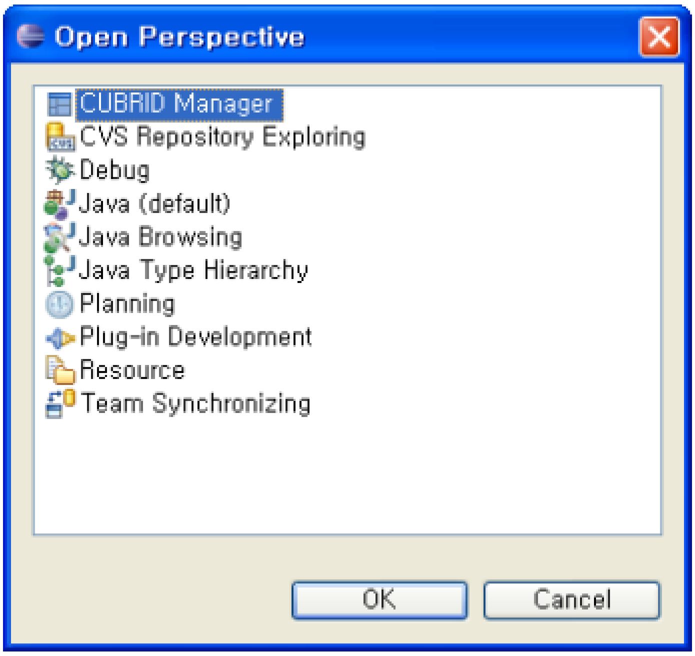
The toolbar provides the following functions.

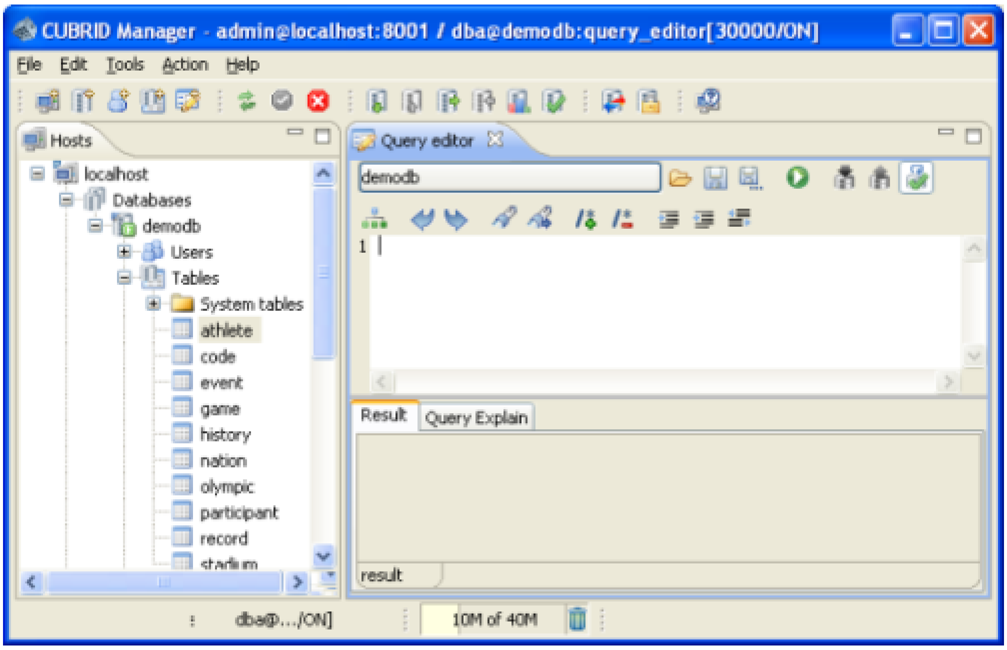


Add Host	Unload Database
Create Database	Load Database
Create User	Backup Database
Create Table	Restore Database
New Query	Optimize Database
Refresh	Check Database
Start	Transaction Info
Stop	Lock Information
	Server Version

Eclipse Plug-in

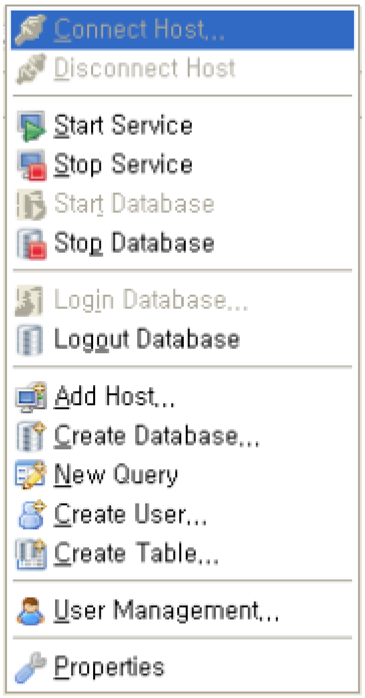
When you connect to the CUBRID Manager client with Eclipse plug-in, the [CUBRID] menu provides the basic menu and the same tree pane for the RCP application. To run the CUBRID Manager plug-in, select [Window] > [Open Perspective] > [Other] in Eclipse, then select [CUBRID Manager].





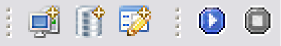
CUBRID Menu

The [CUBRID] menu only provides the basic menu for running, setting, and adding. To use other functionalities, right-click in the tree pane just like in the RCP application.



Toolbar

The CUBRID Manager Eclipse plug-in toolbar provides functionalities such as Add Host, Create Database, New Query, and Start/Stop Service.



Running the CUBRID Manager

Configuring CUBRID Manager Server

The configuration file for the CUBRID Manager server is **cm.conf** and located at **$CUBRID/conf**.

In the CUBRID Manager configuration file, where parameter names and values are stored, comments are prefaced by "#." Parameter names and values are separated by blank or equals sign (=). The following are parameters that can be set in the **cm.conf** file.

cm_port

A parameter that sets the connection port for the connection between the CUBRID Manager server and the Client. The default value is **8001**. **cm_port** is a port used by **cub_auto**, and **cm_js** automatically uses the value set by **cm_port** plus 1. For example, if **cm_port** is set to 8001, **cub_auto** uses the port 8001, and **cub_js** uses 8002. Therefore, to run the CUBRID Manager in an environment where a firewall is installed, you must set both ports that are actually used to open.

monitor_interval

A parameter that sets the monitoring interval of **cub_auto** in seconds. The default value is **5** seconds.

allow_user_multi_connection

A parameter that allows multiple Client connections to the CUBRID Manager server. The default value is **YES**. That is, more than one CUBRID Manager client can connect to the CUBRID Manager server, even with the same user name.

server_long_query_time

A parameter that sets a reference time (in seconds) determined by **slow_query**, a diagnosis operation performed by the server. The default value is **10**. If the execution time of the query performed on the server exceeds this parameter value, the number of the **slow_query** parameters increases.

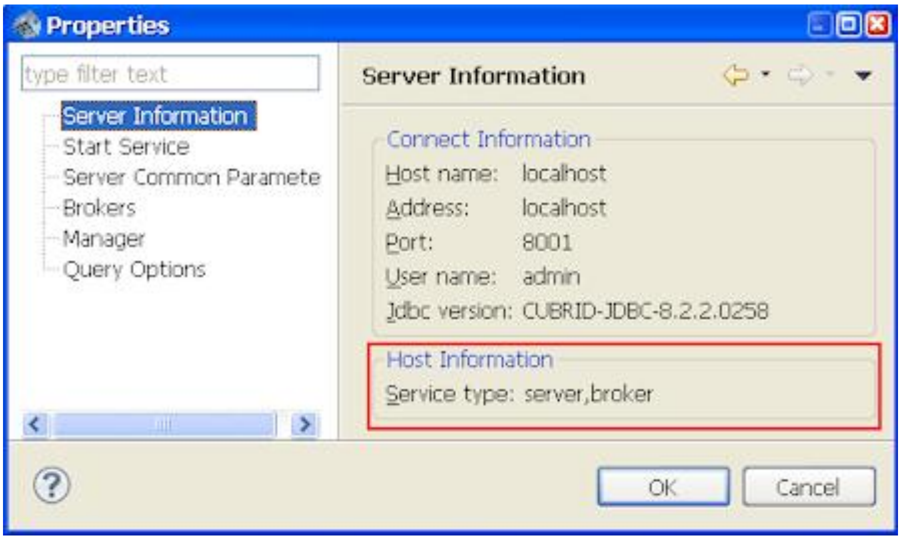
cm_target

A parameter that displays appropriate menu items of the Manager depending on the service being provided in a configuration where the Broker and the Database Server are separated. The default value means the environment where both the Broker and the Database Server are installed. The following settings are possible.

- **cm_target broker, server** : Both the Broker and the Database Server exist.
- **cm_target broker** : Only the Broker exists.
- **cm_target server** : Only the Database Server exists.

If you set only for the Broker, only Broker-related menu items are displayed; If you set only for the Database Server, only server-related menu items are displayed.

If you right-click the host in the navigation tree and then select [Properties], you can check the setting information under [Host Information].



Server Information

Connect Information

Host name: localhost

Address: localhost

Port: 8001

User name: admin

Jdbc version: CUBRID-JDBC-8.2.2.0258

Host Information

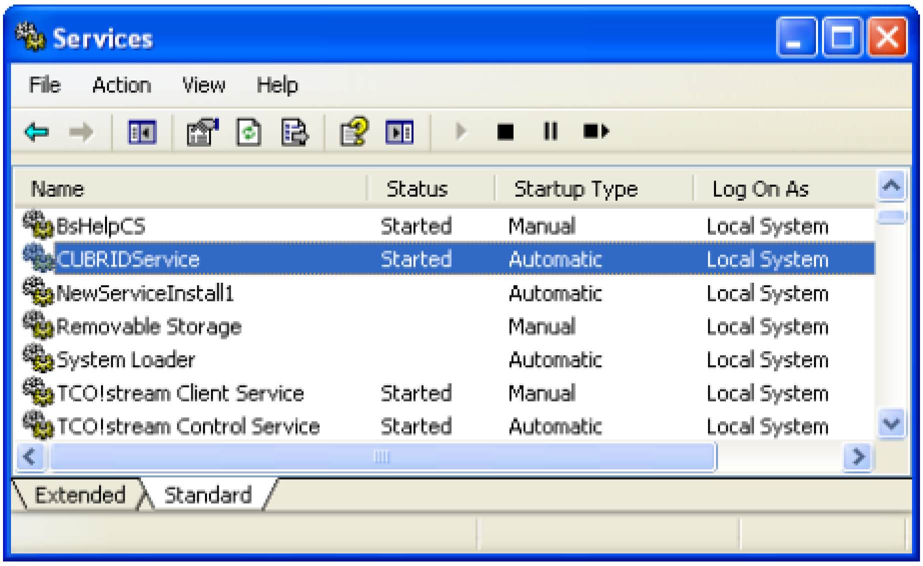
Service type: server,broker

Running CUBRID Manager Server

The CUBRID Manager consists of a server and a client. You can run the CUBRID Manager client only when the CUBRID Manager server is running.

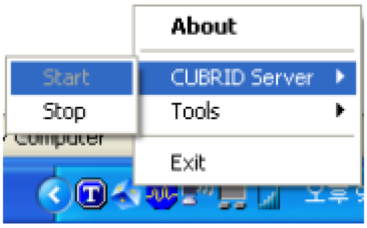
Running in Windows

- To start the CUBRID Server, go to [Control Panel] > [Performance and Maintenance] > [Administrative Tools] > [Services] and double-click CUBRIDService.



Name	Status	Startup Type	Log On As
BsHelpCS	Started	Manual	Local System
CUBRIDService	Started	Automatic	Local System
NewServiceInstall1		Automatic	Local System
Removable Storage		Manual	Local System
System Loader		Automatic	Local System
TCO!stream Client Service	Started	Manual	Local System
TCO!stream Control Service	Started	Automatic	Local System

- Click the CUBRID Service Tray and then select [CUBRID Server] > [Start].



Note Remember that if you select [Exit] on the CUBRID Service Tray while CUBRID is running, all services and processes running on the server will be stopped.

Running in Linux

Enter one of the following commands on the shell to start the CUBRID Manager server.

- %cubrid service start
- %cubrid manager start

CUBRID Manager Server Log

Logs about tasks performed on the CUBRID Manager server and errors are recorded in **$CUBRID/log/manager**, and PID of the running server is recorded in **$CUBRID/var/manager**. The following files are recorded:

- **cub_js.access.log** : Records a task processed by **cub_js**.
- **cub_js.error.log** : Records an error that occurred during processing a task by **cub_js**.
- **cub_auto.access.log** : Records information of the client logged into the server.
- **cub_auto.error.log** : Records information of the client that failed to log into the server.
- **connlist** : Records the list of clients currently connected to the CUBRID Manager server.

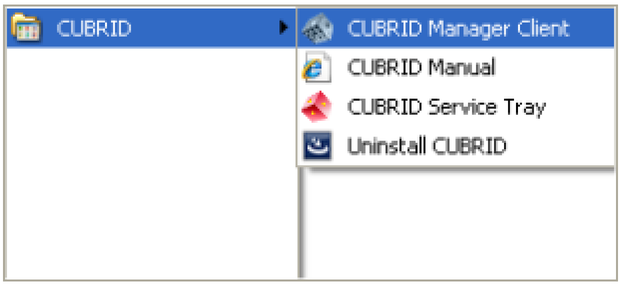
Running CUBRID Manager Client

The CUBRID Manager consists of a server and a client. You can run the CUBRID Manager client only when the CUBRID Manager server is running.

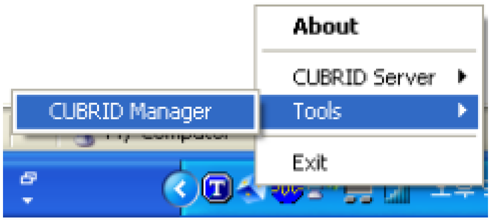
The CUBRID Manager client is a Java application program that runs only on JRE/JDK 1.6 or higher. If the version is below 1.6, an error message appears: "Unsupported JRE version. You must use JRE 1.6 or a later version."

Running in Windows

- Select [Start] > [All Programs] > [CUBRID] > [CUBRID Manager Client].



- Click the CUBRID Service Tray and then select [Tools] > [CUBRID Manager].



Running in Linux

You can start the CUBRID Manager client by entering one of the following commands on the shell.

- %cd $CUBRID/cubridmanager
- %./cubridmanager

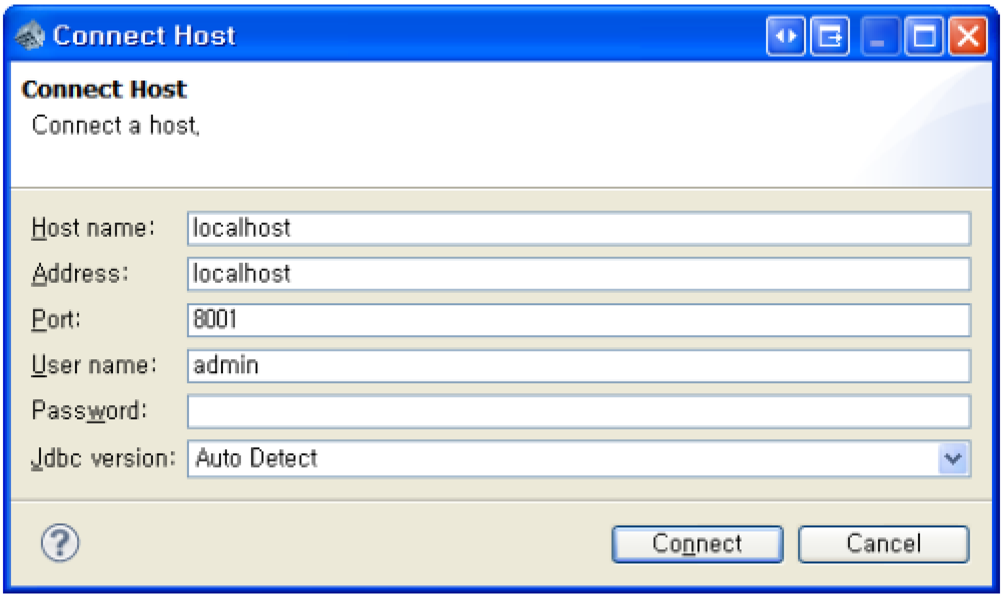
CUBRID Manager Client Log

Error logs that occurred while the CUBRID Manager client is running are created as the **cubridmanager.log** file in **$CUBRID/cubridmanager/logs**. When analyzing CUBRID Manager improvements and errors, you can participate in efforts to improve the CUBRID Manager by registering the error log and the symptom as an issue at the CUBRID Manager development site.

Host Management

Default Host Information

When you start the CUBRID Manager for the first time after installation, a host named "localhost" has been set by default in the host navigation tree. This setting has been made under the assumption that the CUBRID Manager server is installed on your PC where the CUBRID Manager client is currently running, and shows the basic information about the CUBRID Manager connection.



Connect Host

Connect a host.

Host name: localhost

Address: localhost

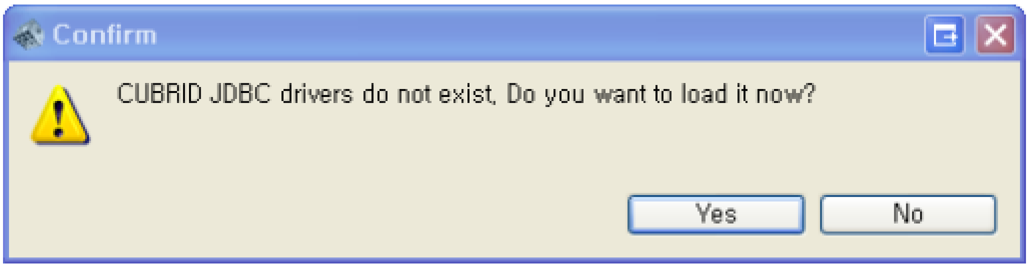
Port: 8001

User name: admin

Password:

Jdbc version: Auto Detect

- **Host name** : Host name is an identifier that identifies a host to be managed in the CUBRID Manager. A host name must be unique. Internally managed host identifier has the format of [Host name + Connection port]. A host name can contain only alphanumeric character. Do not use spaces. The length of a host name must be between 4 and 32 characters.
- **Host address** : A host address cannot be a space or contain spaces.
- **Connection port** : Only an integer value between 1024 and 65535 can be entered for the connection port. The default port value is **8001**. If the connection fails, check the **cm_port** value in the **$CUBRID/conf/cm.conf** file.
- **User name** : The user who has the administrator authorization of the CUBRID Manager is **admin**. You must connect with the **admin** account when you use the CUBRID Manager for the first time after installation. A user name can contain only alphanumeric character. Do not use spaces. The length of a user name must be between 4 and 32 characters.
- **Password** : The initial password for the **admin** user, who has the administrator authorization of the CUBRID Manager, is "admin." You cannot connect if you do not modify the password when you make the first connection with the **admin** account. Be careful not to lose the modified password of the **admin** user. The password cannot be reset to "admin." A password cannot contain spaces. The length of a password must be between 4 and 32 characters.
- **Driver version** : You can select a CUBRID JDBC driver which will be used in the CUBRID Manager. The default value is **Auto Detect;** a driver is automatically detected if any version of driver which can be used in the CUBRID Manager exists; if exists, automatic connection is made. If not, the following message will appear.



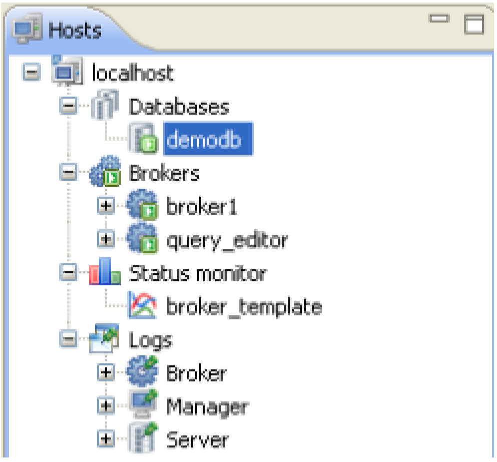
CUBRID JDBC drivers do not exist. Do you want to load it now?

Connect/Disconnect Host

Connect Host

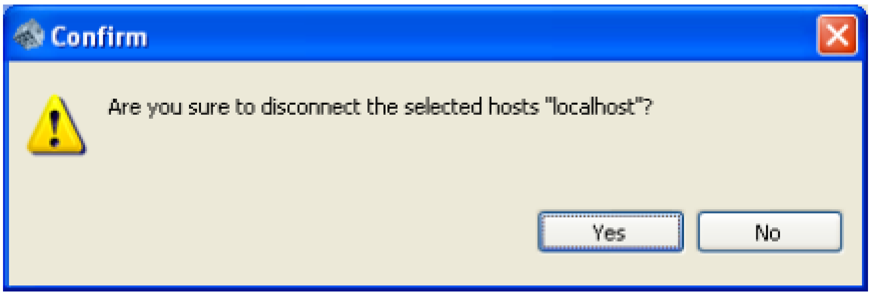
A connection to a specific host is made when you double-click the host registered in the host navigation tree. Or you can right-click a host, select [Connect Host], check the user name, and then click [Connect] to connect to the host.

If the connection is successful, the host icon changes from to . The initial layout of the host pane after the connection is shown in the figure below:



Disconnect Host

Right-click the host in the host navigation tree, and then select [Disconnect Host].



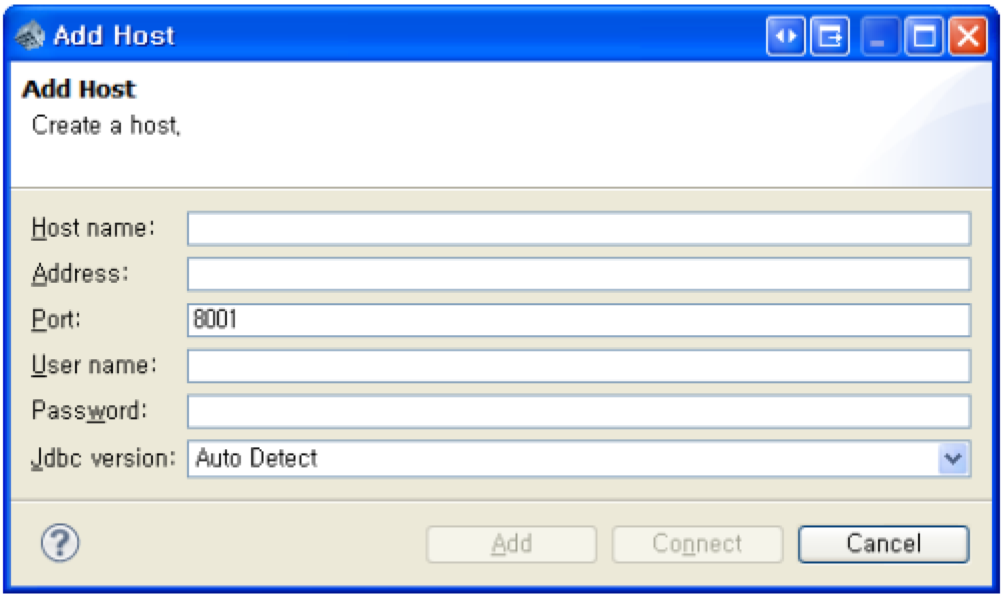
Are you sure to disconnect the selected hosts "localhost"?

Add/Delete Host

Add Host

You can manage databases through multiple hosts in a single manager by adding a host. To add a host, perform one of the following:

- Click [Add Host] from the toolbar.
- Select [File] > [Add Host] on the menu.
- Right-click in the navigation tree, and then select [Add Host].



Add Host

Add Host

Create a host.

Host name:

Address:

Port: 8001

User name:

Password:

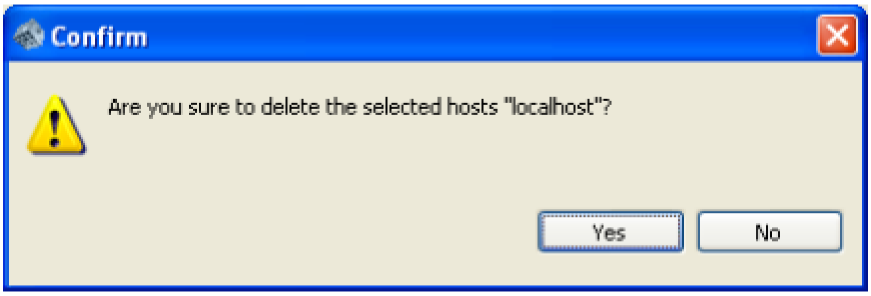
Jdbc version: Auto Detect

For more information about host name, host address, connection port, user name and password, see Default Host Information.

If you enter the information and then click the [Add] button, the host is added to the host navigation tree, but not connected to. If you click [Connect], however, the host is added to the navigation tree and connected to as well.

Delete Host

Deletes the information of the host registered in the host navigation tree. When a host is deleted, all information in the sub-nodes is deleted as well. However, only the CUBRID Manager client configuration, not the CUBRID Manager server configuration, is deleted.

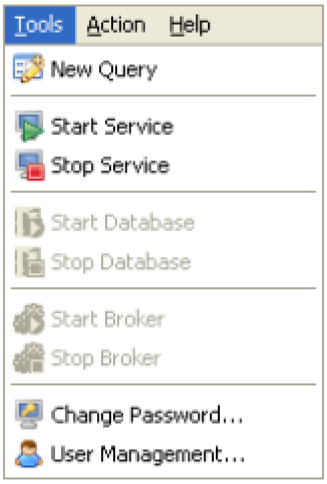


Confirm

Are you sure to delete the selected hosts "localhost"?

Managing Service

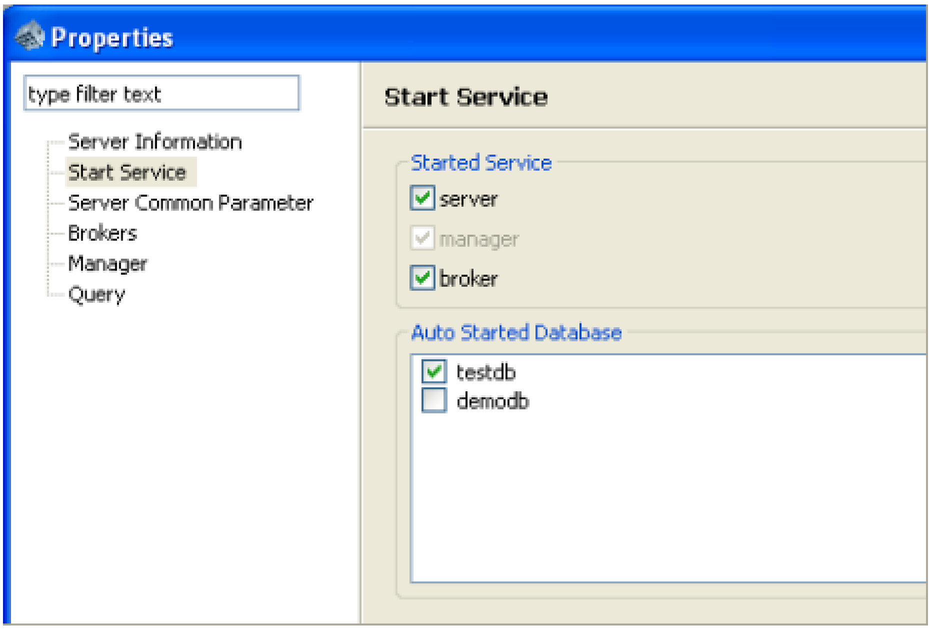
You can start or stop CUBRID related services with the CUBRID Manager. This functionality works the same as using 'cubrid,' a management utility used in the CUBRID database. However, you cannot start or stop the CUBRID Manager server with the CUBRID Manager client.



Start Service

If you select [Tools] > [Start Service] in the menu, the service and database selected from [Start Service] start. To select a service and database to be started, right-click a host and then select [Properties] > [Start Service].

[Automatic Start Service] is activated only if the service from [Start Service] is selected.



Start Service

Started Service

☑ server

☑ manager

☑ broker

Auto Started Database

☑ testdb

□ demodb

Stop Service

If you select [Tools] > [Stop Service] in the menu, the service and database selected from [Start Service] stop.

Start/Stop Database

To start or stop the selected database, perform one of the following:

- Click [Start ▶] or [Stop ■] from the toolbar.
- Right-click a database and then select [Start Database] or [Stop Database].
- Select [Action] > [Start Database], or [Action] > [Stop Database] on the menu.

Start/Stop Broker

To start or stop selected broker, perform one of the following:

- Click [Start ▶] or [Stop ■] from the toolbar.
- Right-click a Broker and then select [Start Broker] or [Stop Broker].
- Select [Action] > [Start Broker], or [Action] > [Stop Broker] on the menu.

User Management

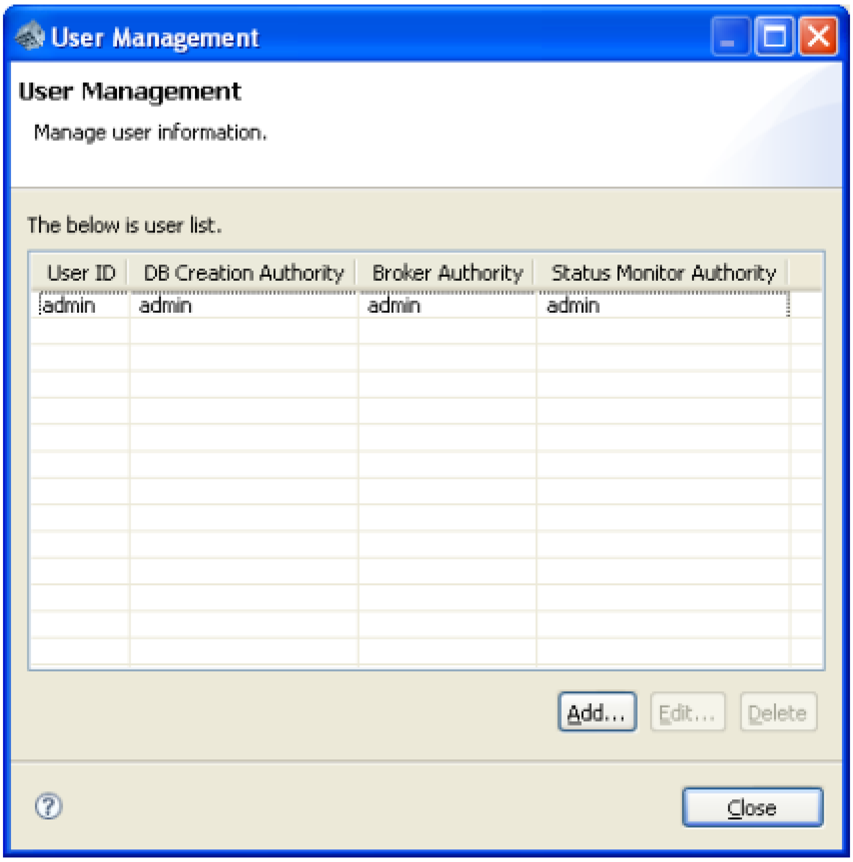
Setting User Authorization

Like registering multiple users and managing their authorization in a single database, multiple users can be registered and their authorization can be managed in a single Manager server.

Information on the CUBRID Manager users is stored in the CUBRID Manager server. Therefore, to access the host, you must acquire the authorization to use the CUBRID Manager before connecting to the host.

Only the **admin** account and general user accounts registered by **admin** can connect to the CUBRID Manager. Only **admin** can perform CUBRID Manager user management. To do this, select [Tools] > [User Management].

In [User Management], you can set each user's authority as follows:



User Management

Manage user information.

The below is user list.

User ID	DB Creation Authority	Broker Authority	Status Monitor Authority
admin	admin	admin	admin

DB Creation Authority

- **admin** : Authorization to create a new database. Only the **admin** user is granted with this authorization.
- **none** : No authorization.

Broker Authority

- **admin** : Authorization to start/stop, add, edit, delete the Broker.
- **monitor** : Authorization to monitor the progress of the Broker with the view status function.
- **none** : No authorization.

Status Monitor Authority

- **admin** : Authorization to perform, add, edit, and delete status monitoring.

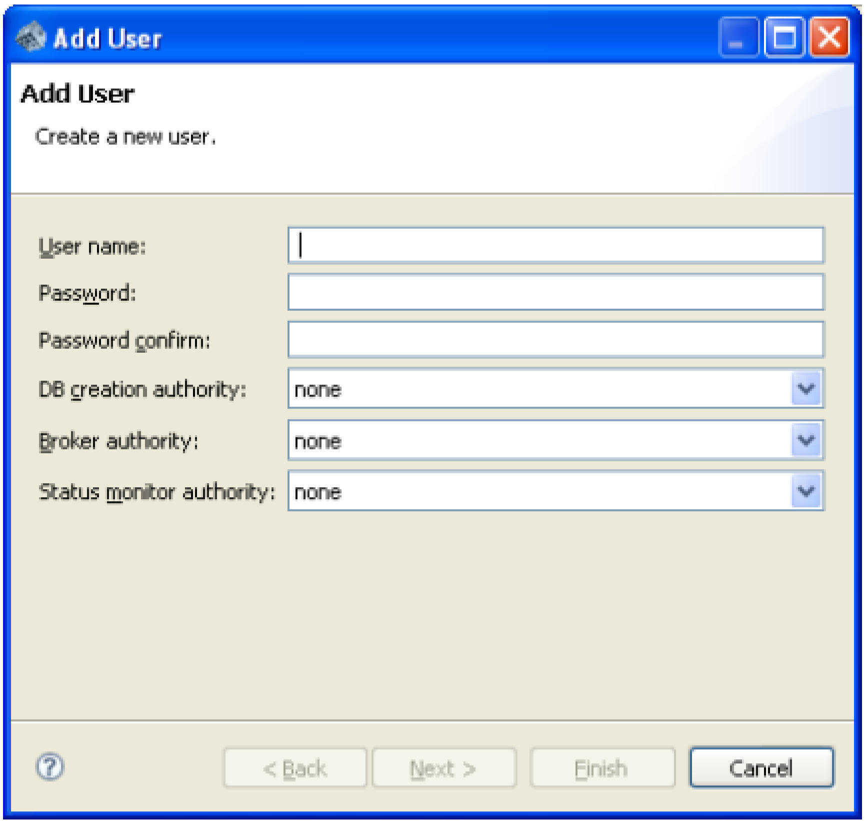
- **monitor** : Authorization to monitor the progress of a status monitor.
- **none** : No authorization.

Adding, Editing, and Deleting Users

Add User

To add a user, you need to specify the user account information and authorization, and database access authority.

Setting User Account and Authority



Add User

Add User

Create a new user.

User name:

Password:

Password confirm:

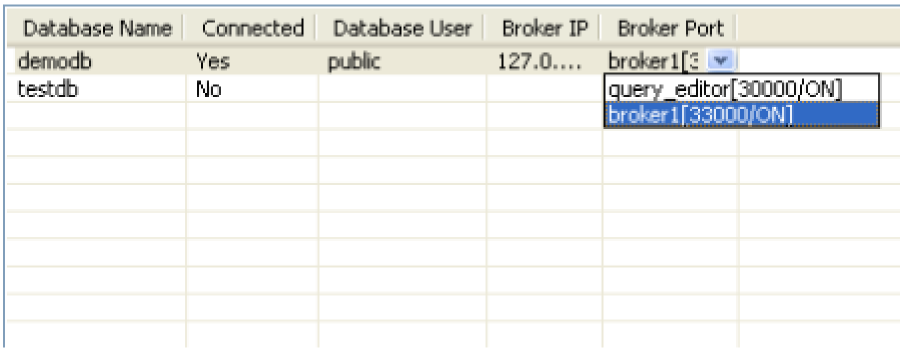
DB creation authority: none

Broker authority: none

Status monitor authority: none

- **User name** : The length of a user name must be between 4 and 32 characters. A user name can contain only alphanumeric character. Do not use spaces. "admin" cannot be used as a user name, and it must be unique in the host.
- **Password** : The length of a password must be between 4 and 32 characters. Do not use spaces. "admin" cannot be used as a password.

Setting Database Authorization



Database Name	Connected	Database User	Broker IP	Broker Port
demodb	Yes	public	127.0....	broker1[3
testdb	No			

query_editor[30000/ON]
broker1[33000/ON]

- **Connected** : Select databases that the CUBRID Manager user being added can access. Only the databases with the [Yes] option selected are displayed in the host's navigation tree.
- **Database User** : Enter database account information that is used when the CUBRID Manager user being added accesses the database. You can enter values such as "dba" or "public."
- **Broker IP** : Enter the Broker IP that is used when connecting to the database. The default value is set to the address of the database server. If there is a separate Broker server, this value can be edited to provide access information.
- **Broker Port** : Specify a Broker port that is used when the CUBRID Manager user being added accesses the database. You can check the port information of the Broker through the Broker properties. The Broker information consists of "Broker name[Port/Status]."

Edit User

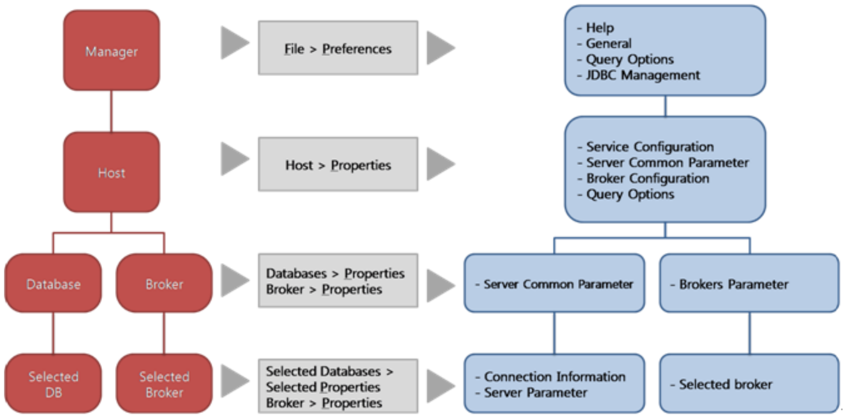
You can select a user from the user list and edit it as you add a user. However, editing the **admin** account is limited to changing its password. Other authorization for the **admin** account cannot be edited.

Delete User

To delete a user, select the user from the user list and then click the [Delete] button. Note that the **admin** account cannot be deleted.

Properties Management

With the CUBRID Manager, you can set the operation environment of the service, the database, the Broker, the Manager server and the Query editor. Configuration can be set in steps and consists of the following structures:



Help Setting

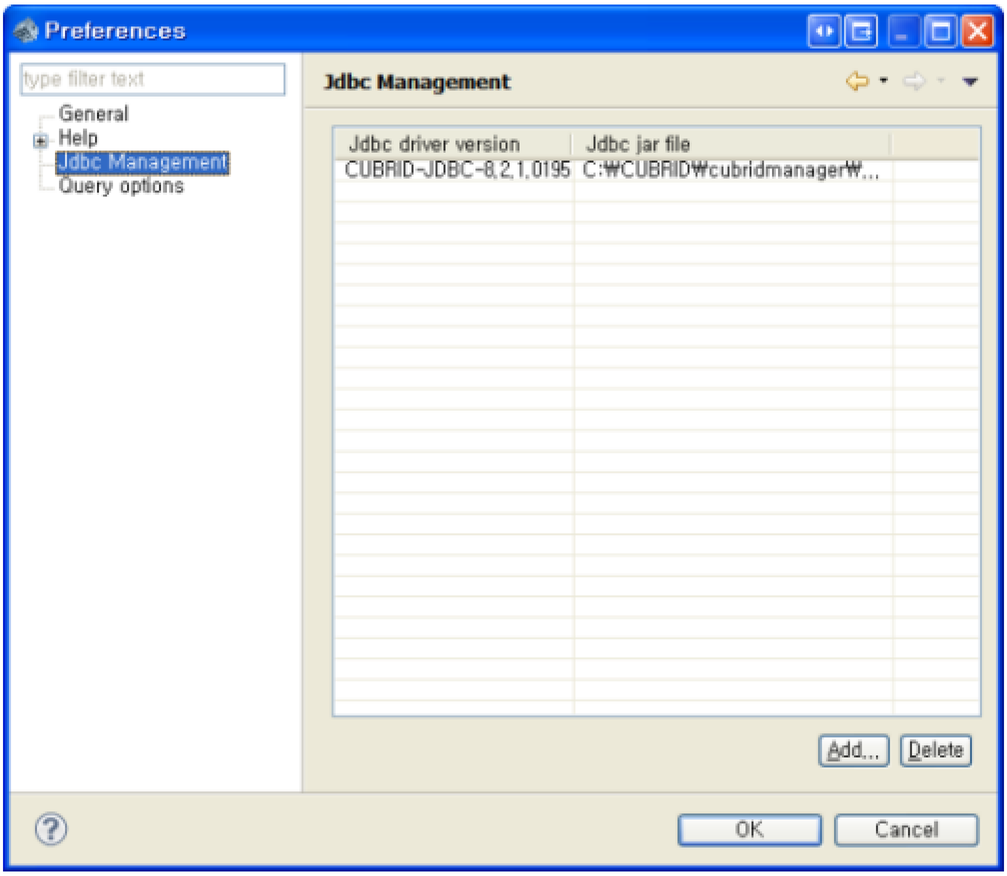
You can set whether the Help is displayed through the internal Help window or the external browser. You can also select the open mode when the Help is executed as an Eclipse plug-in.

Basic Information Setting

You can set the following through basic information setting:

- Whether the CUBRID Manager window is open in the maximum size or the previous size
- Whether to show or not CUBRID News on startup(available when system locale is set to KR)
- Whether the information window opens with a single or double click in the navigation tree. (The default value is double click.)

JDBC Driver Setting



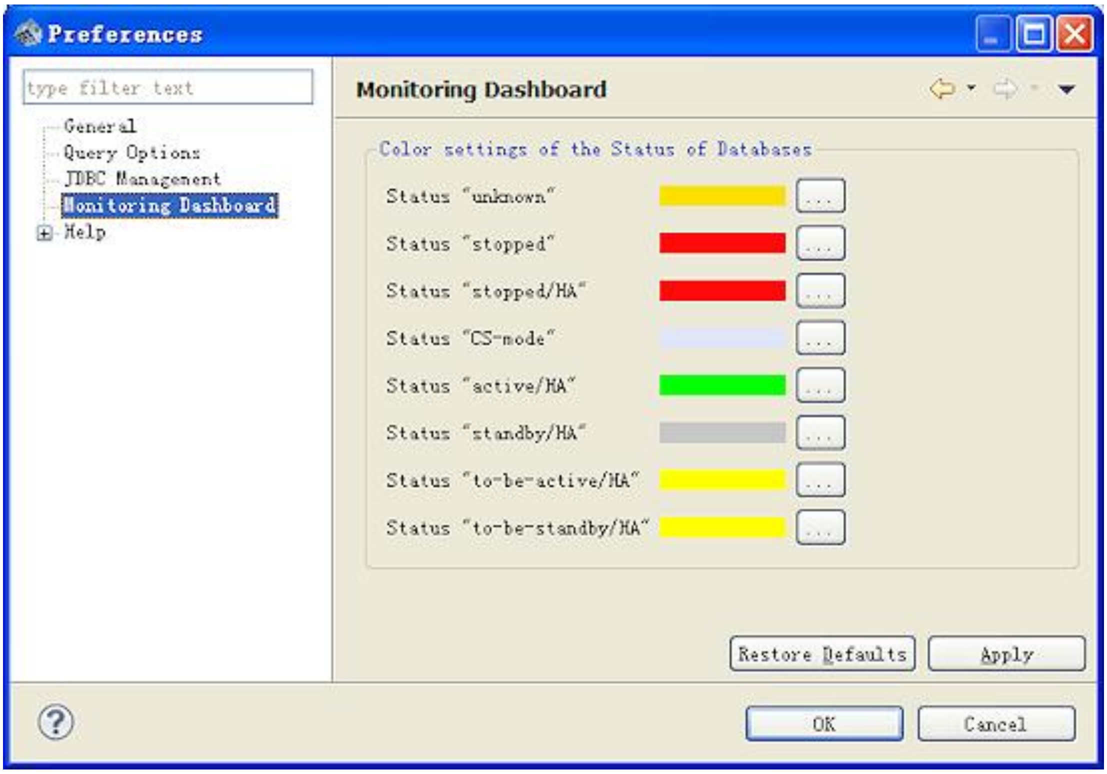
Jdbc driver version	Jdbc jar file
CUBRID-JDBC-8.2.1.0195	C:₩CUBRID₩cubridmanager₩...

Multiple JDBC drivers are supported in CUBRID 2008 R2.1 or later. To access database in version earlier than 2.1, you can configure additional version of CUBRID JDBC driver. The CUBRID JDBC drivers are contained in **cubridmanager/plugins**. In CUBRID Manager R2.1, in general, you can access the corresponding version of database server by using CUBRID-JDBC-8.2.1.x. As multiple JDBC drivers are supported, you can use access the earlier version of database than the version of CUBRID Manager to execute queries. To do it, go to [Default Configuration] > [JDBC Driver Configuration] and configure additional JDBC driver that you will use.

However, depending on the version of the database server that is being connected to, there might be some restricted functions. For example, while the functions in the left tree structure work normally in the database of CUBRID 2008 R2.0 or higher, they do not work in earlier versions of the database. In addition, query editing or executing will work normally in the database of CUBRID 2008 R1.4 or higher, but will not work in earlier versions.

Monitoring Dashboard

You can set the color for each status of hosts, database servers, or brokers in the monitoring dashboard.



Monitoring Dashboard

Color settings of the Status of Databases

Status "unknown"	🟨
Status "stopped"	🟥
Status "stopped/HA"	🟥
Status "CS-mode"	⬜
Status "active/HA"	🟩
Status "standby/HA"	⬜
Status "to-be-active/HA"	🟨
Status "to-be-standby/HA"	🟨

Query Editor Option

Sets the parameters that are to be applied when a connection to the database is made through the Query editor. For more information, see Query Editor Option.

Service Operation Setting

Sets the options related to the service operation. You may set automatic database startup.

Server-Common/-Specific Parameter Setting

Sets database common parameters. For more information, see Database Configuration.

Specific Broker Variable Setting

Sets broker information. For more information, see Broker Menu.

Connection Information Setting

You can set the port for the database connection and the character set through connection information setting.

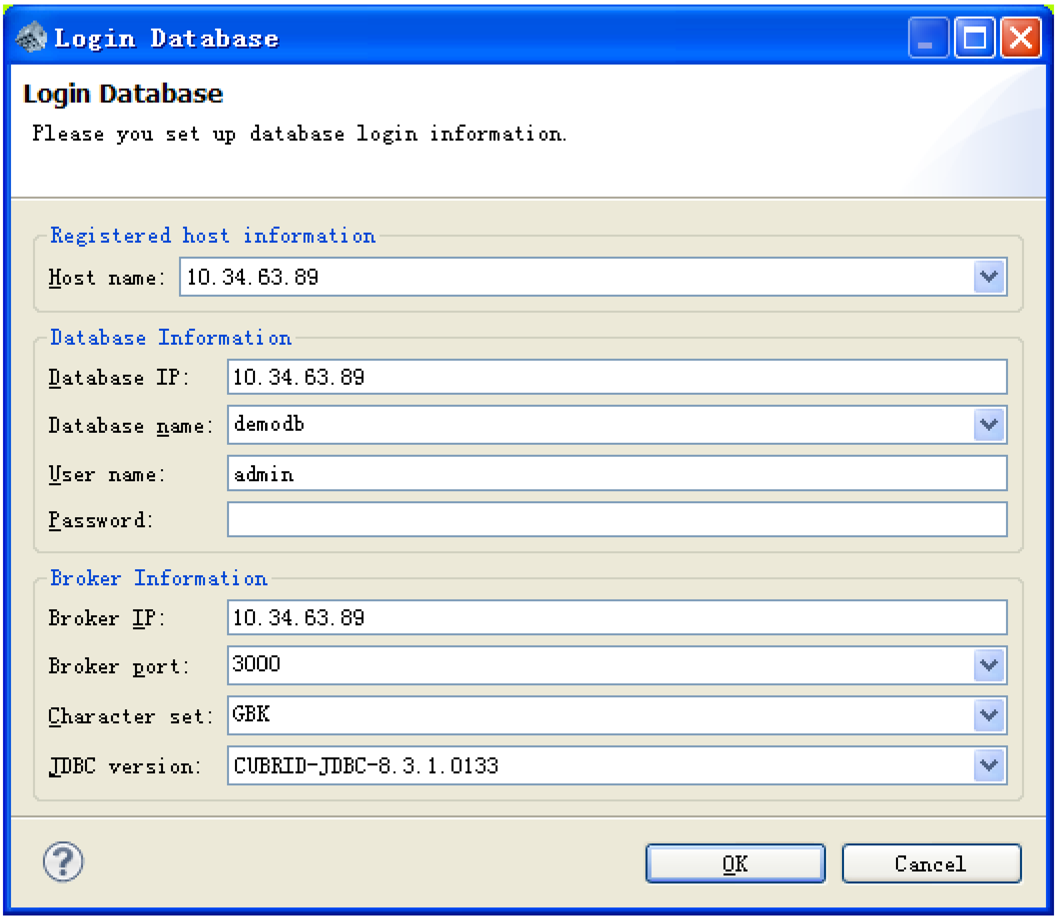
Query Editor

Query Editor Structure

The CUBRID Manager's Query Editor is a query tool that supports execution of all **DML**, **DDL**, **DCL** statements, allowing users to edit and execute queries more easily.

To run the Query Editor, select [Tools] > [New Query] or select [New Query 📝] from the toolbar. You can also right-click a specific database and then select [New Query].

If you select [New Query] by right clicking on a database, the default query editor is run with the basic information provided upon login to the corresponding database. However, if you select it on the menu or toolbar, you can specify your login information by yourself to connect the database. The character set displayed on the screen is the same one that is specified for the database connection. You can log in setting the information of the database login, character set, and JDBC driver version. Note that separate configuration for broker connection is enabled in case a Broker is running in another server.



Login Database

Login Database

Please you set up database login information.

Registered host information

Host name: 10.34.63.89

Database Information

Database IP: 10.34.63.89

Database name: demodb

User name: admin

Password:

Broker Information

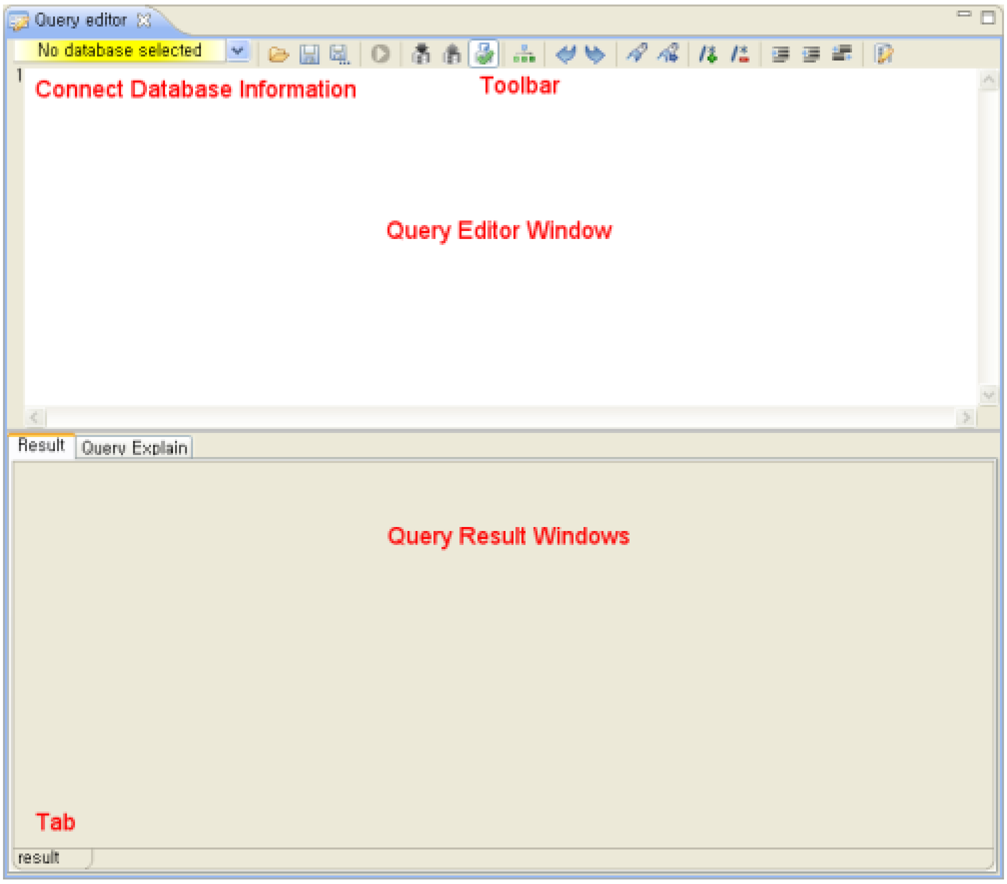
Broker IP: 10.34.63.89

Broker port: 3000

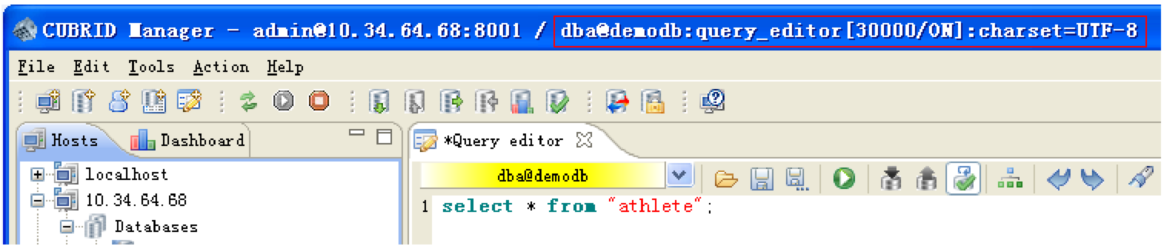
Character set: GBK

JDBC version: CUBRID-JDBC-8.3.1.0133

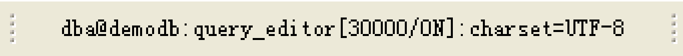
The Query Editor window is divided into a query edit pane at the top and a query results pane at the bottom. In the query edit pane, you can type and edit queries to execute with a toolbar that contains icons for frequently used functions in the Query Editor. In the query results pane, you can see the query results in a tab format and check the query execution time.



When you use multiple editors at the same time, it could more convenient to check the connection information in the title bar (at top) and in the status bar (at bottom) of the CUBRID Manager. Information that will be displayed is as follows: database, user name, broker port, and charset.


CUBRID Manager - admin@10.34.64.68:8001 / dba@demodb:query_editor[30000/ON]:charset=UTF-8

```
1 select * from "athlete";
```

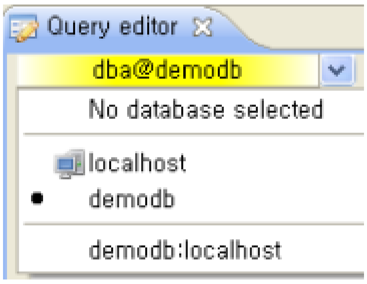

dba@demodb:query_editor[30000/ON]:charset=UTF-8

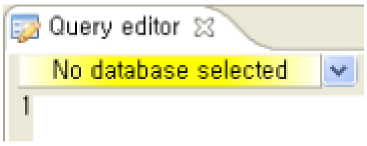
Toolbar Options

Select Database

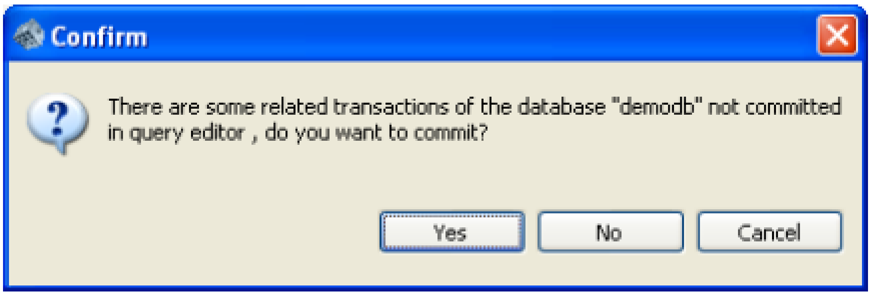
With this option, a Query Editor to access multiple databases from multiple hosts.

It also displays the currently accessed database. You can also selectively access a database from a different host that are currently connected to the CUBRID Manager. In the figures below, yellow highlighted sections mean the followings: the left one is displayed when it is accessed by **DBA**; the right one is displayed when no database is selected. The former makes you to check out which account is currently used; the latter allows you to select a database to connect when no database connection is made.





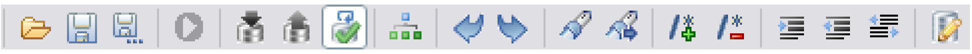
If you select a different database without stopping transactions in the currently accessed database, you are prompted whether to commit or rollback transactions in progress as below.



There are some related transactions of the database "demodb" not committed in query editor , do you want to commit?

Toolbar Options

The edit function of the Query Editor is synchronized with [Edit] on the menu. The following table shows the options available on the Query Editor's toolbar: Major options are provided with a shortcut key.



Option	Icon	Shortcut Key	Description
Open	📂		Opens a text-based SQL and displays it in the query edit pane.
Save	💾		Saves the contents of the query edit pane.
Save As	💾		Saves the contents of the query edit pane in a different name.
Run	▶	F5	Executes all the queries in the query edit pane. Alternatively, executes queries selected as a block. This icon is deactivated when the execution is unavailable (▶).
Commit	⬇		Keeps deactivated in autocommit mode (⬇); while in non-autocommit mode, allows you to commit (⬇) explicitly when a transaction occurs.
Rollback	⬆		Keeps deactivated in autocommit mode (⬆); while in non-autocommit mode, allows you to rollback (⬆) explicitly when a transaction occurs.
auto commit	✓		Automatically commits queries executed in the query edit pane. You can toggle the icon. When auto commit is on, the icon is displayed as ☑, and when it is off, the icon is displayed as ✓. Note that the setting is only applied to the corresponding Query Editor; that is the default value of the option is not changed.
Display query plan	品	F6	Displays the execution plan of the selected query. For more information, see Query Execution Plan.
Undo	↶	Ctrl+Z	Cancels the most recent edit action.
Redo	↷	Ctrl+Y	Re-executes the most recent edit action that is canceled by Undo.

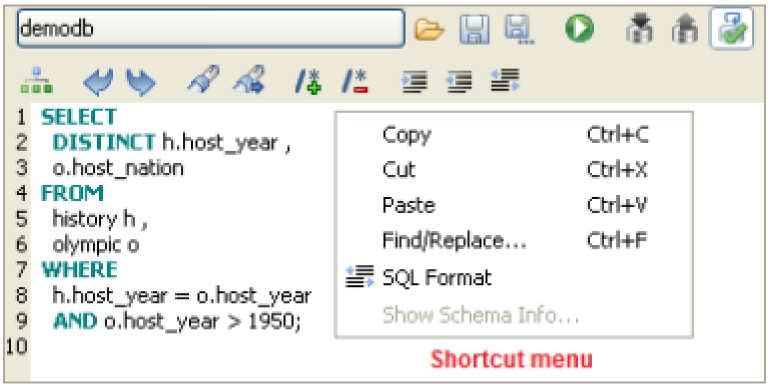
Find/Replace		Ctrl+F	Provides search and replace function that can be used in the query edit pane.
Find next		F3	Provides further search for items that are already searched once.
Convert to comment		Ctrl+/	Adds the comments to the selection area or to the line where the cursor is located in the query edit pane. Comment strings are inserted with a double dash (--).
Delete comment string		Ctrl+/	Deletes the comments from the selection area or from the line where the cursor is located in the query edit pane.
Insert tab		Tab	Indents the selection area in the query edit pane.
Delete tab		Shift+Tab	Un-indents the selection area in the query edit pane.
Format SQL		Ctrl+Shift+F	Formats the selected SQL strings in the query edit pane.
Get OID info			A toggle button available on the toolbar in the query edit pane for [Get OID info], which can be found in the Query Editor options. That is, by selecting this button on the toolbar when [Get OID info], which can be found in the Query Editor options, is OFF, you can directly modify/delete data in the query results pane for subsequent queries. In this button is selected on the toolbar, however, this only applies to the corresponding Query Editor. The values of the Query Editor options for the corresponding database won't be changed.
Set defined SQL parameter			To specify a parameter whenever the statement is executed, write Prepared Statement, select the written statement, and click the icon.

Query Edit Pane

In the query edit pane, you can enter and edit queries to manipulate the database. You can also use all functions available on the toolbar. The query edit pane provides functions such as automatic statement completion, editing via the pop-up menu and viewing schema information.

Shortcut Menu

If you right-click on the query edit pane, you can select Copy, Cut, Paste, Find/Replace, SQL Format and Show Schema Info. The editing function is synchronized with [Edit] of the menu.

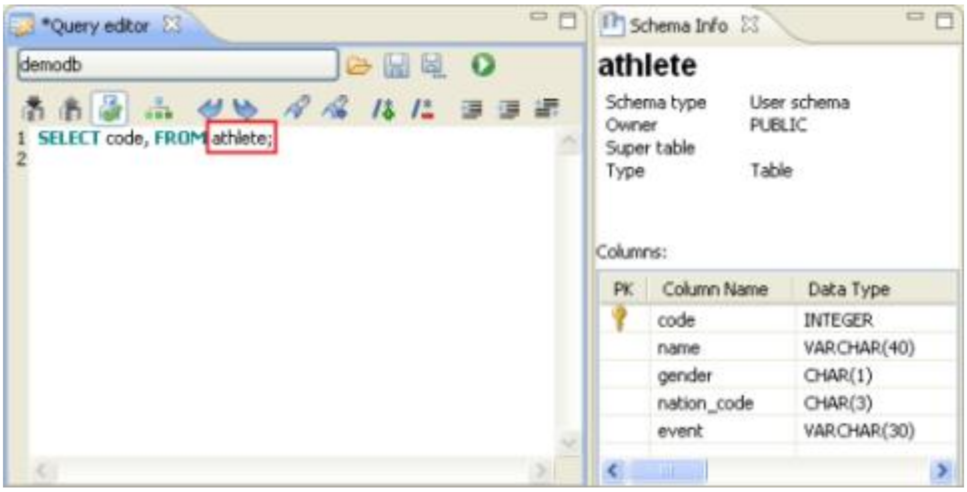


demodb

```
SELECT
  DISTINCT h.host_year ,
  o.host_nation
FROM
  history h ,
  olympic o
WHERE
  h.host_year = o.host_year
  AND o.host_year > 1950;
```

Show Schema Info

If you select a table name and then execute [Show Schema Info], you can create a query while seeing the Query Editor view and the Schema Info view on the same window. Note that you can view the schema information only when executing Query Editor in the Explorer (toolbar not supported).



```
SELECT code, FROM athlete;
```

athlete

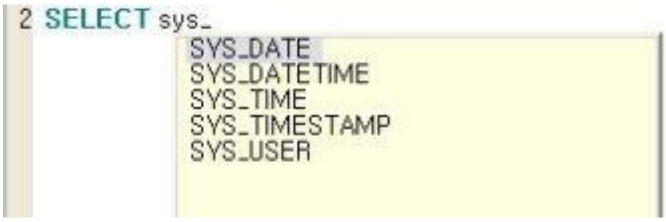
Schema type	User schema
Owner	PUBLIC
Super table	
Type	Table

Columns:

PK	Column Name	Data Type
🔑	code	INTEGER
	name	VARCHAR(40)
	gender	CHAR(1)
	nation_code	CHAR(3)
	event	VARCHAR(30)

Automatic Statement Completion

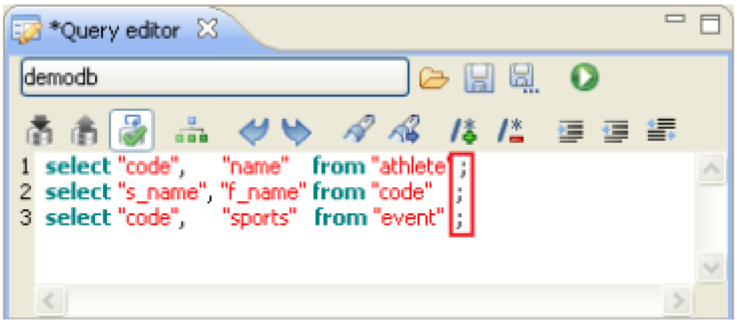
If you enter a CUBRID database keyword, CUBRID automatically finds and completes the statement, increasing usability and user accessibility.



```
SELECT sys_
```

Executing Multiple Queries

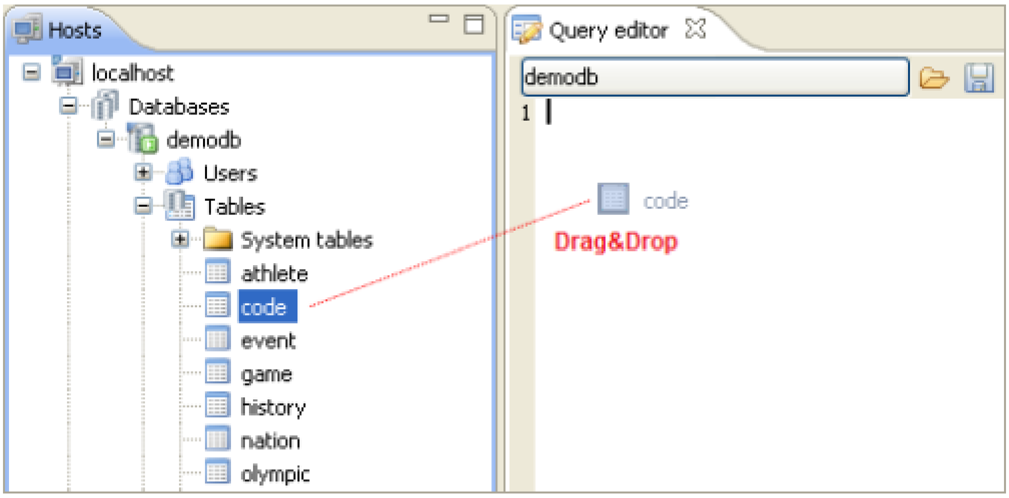
Enter a semicolon at the end of the query statement to specify the end of one query and the start of the next. If there are multiple queries, they are executed sequentially. Each query creates a corresponding tab in the query results pane. If you execute multiple queries without separating them with semi-colons, only the first query is executed, with the rest ignored.



```
select "code",    "name"   from "athlete";
select "s_name", "f_name" from "code"   ;
select "code",    "sports"  from "event"  ;
```

Drag and Drop

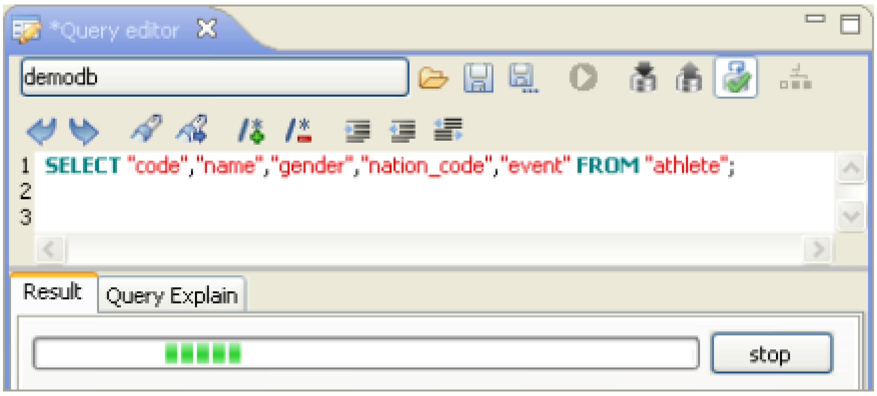
If you drag and drop a table to retrieve from the host navigation tree into the query edit pane of the Query Editor, a **SELECT** query statement is created automatically.



Cancel Query Function

A function that stops the currently running query. This can be divided into two functions.

- **When executing multiple queries** : If you click the Stop button when multiple queries are being executed with the auto commit button enabled, queries processed before the stop operation are reflected normally, and the currently canceled query and following ones are not reflected. If the auto commit button is not enabled, no executed queries are reflected.
- **When executing a long transaction** : If you click the Stop button when executing a long transaction, a query stop command is delivered to CUBRID Manager > JDBC > Broker > Server, and the query finally stops in the database server the query is actually canceled. For Windows, this function is supported in CUBRID 2008 R2.2 or later.

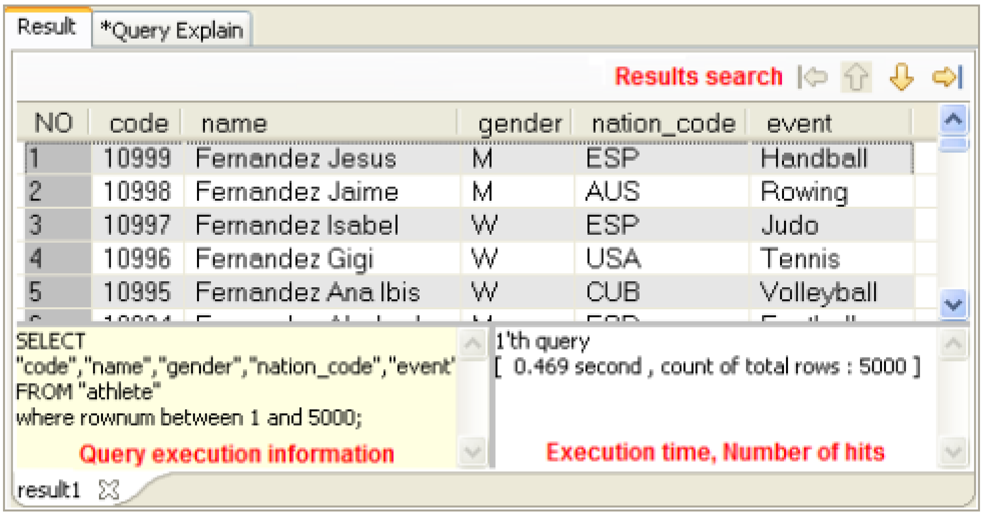


```
SELECT "code","name","gender","nation_code","event" FROM "athlete";
```

Query Results Pane

The query results pane displays the results of the query executed. If there are multiple queries executed, the results of each query are displayed in a separate tab. You can check the query results by selecting the corresponding tab.

The query results pane is divided into areas where you can navigate the results, view information of the executed query, and check execution time and the number of results returned by the query.

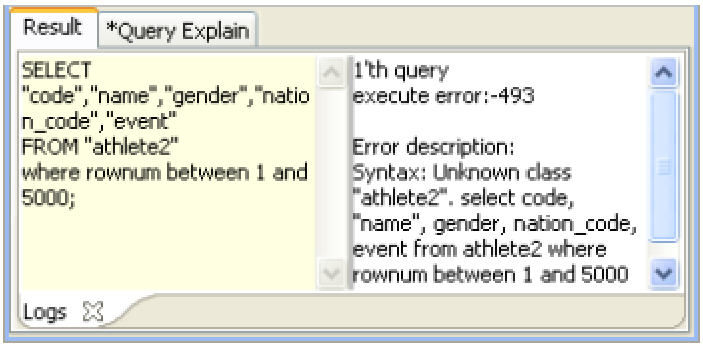


NO	code	name	gender	nation_code	event
1	10999	Fernandez Jesus	M	ESP	Handball
2	10998	Fernandez Jaime	M	AUS	Rowing
3	10997	Fernandez Isabel	W	ESP	Judo
4	10996	Fernandez Gigi	W	USA	Tennis
5	10995	Fernandez Ana Ibis	W	CUB	Volleyball

```
SELECT
"code","name","gender","nation_code","event"
FROM "athlete"
where rownum between 1 and 5000;
```

```
1'th query
[ 0.469 second , count of total rows : 5000 ]
```

If you execute an SQL statement other than **SELECT** or if there is an error in the **SELECT** statement executed, query execution information and error message or execution time information are displayed in the Logs tab.



```
SELECT
"code","name","gender","natio
n_code","event"
FROM "athlete2"
where rownum between 1 and
5000;
```

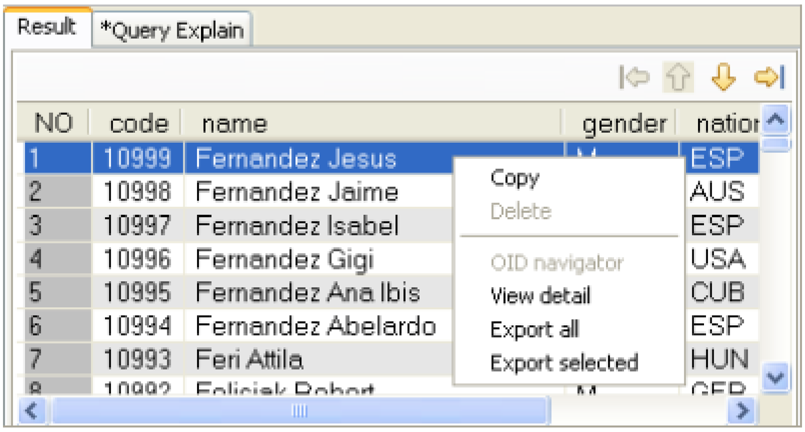
```
1'th query
execute error:-493

Error description:
Syntax: Unknown class
"athlete2". select code,
"name", gender, nation_code,
event from athlete2 where
rownum between 1 and 5000
```

Query Results Pane Structure

- **Results search** : You can navigate the entire search results while moving by the value set in [Page unit of result instances] of [Query options].
- **Query execution information** : Shows from which query the current result comes.
- **Execution time and Number of hits** : Provides information about the execution time on the server to get the current query results and total search count.

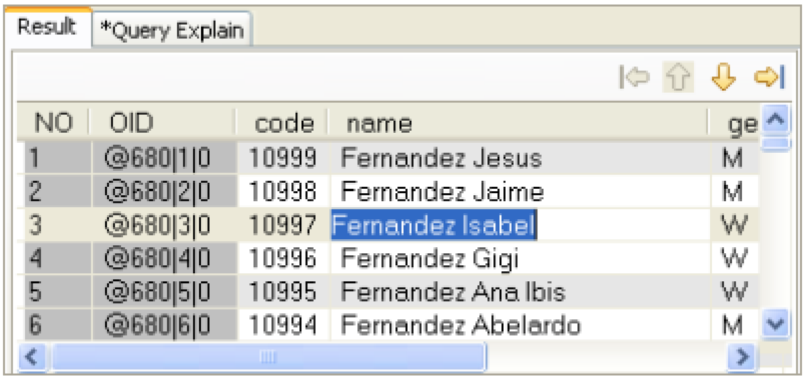
Options in the Query Results Pane

As shown in the figure below, the query results pane's shortcut menu contains options such as Copy, Modify, Delete, OID navigator, View detail, Export all and Export selected.



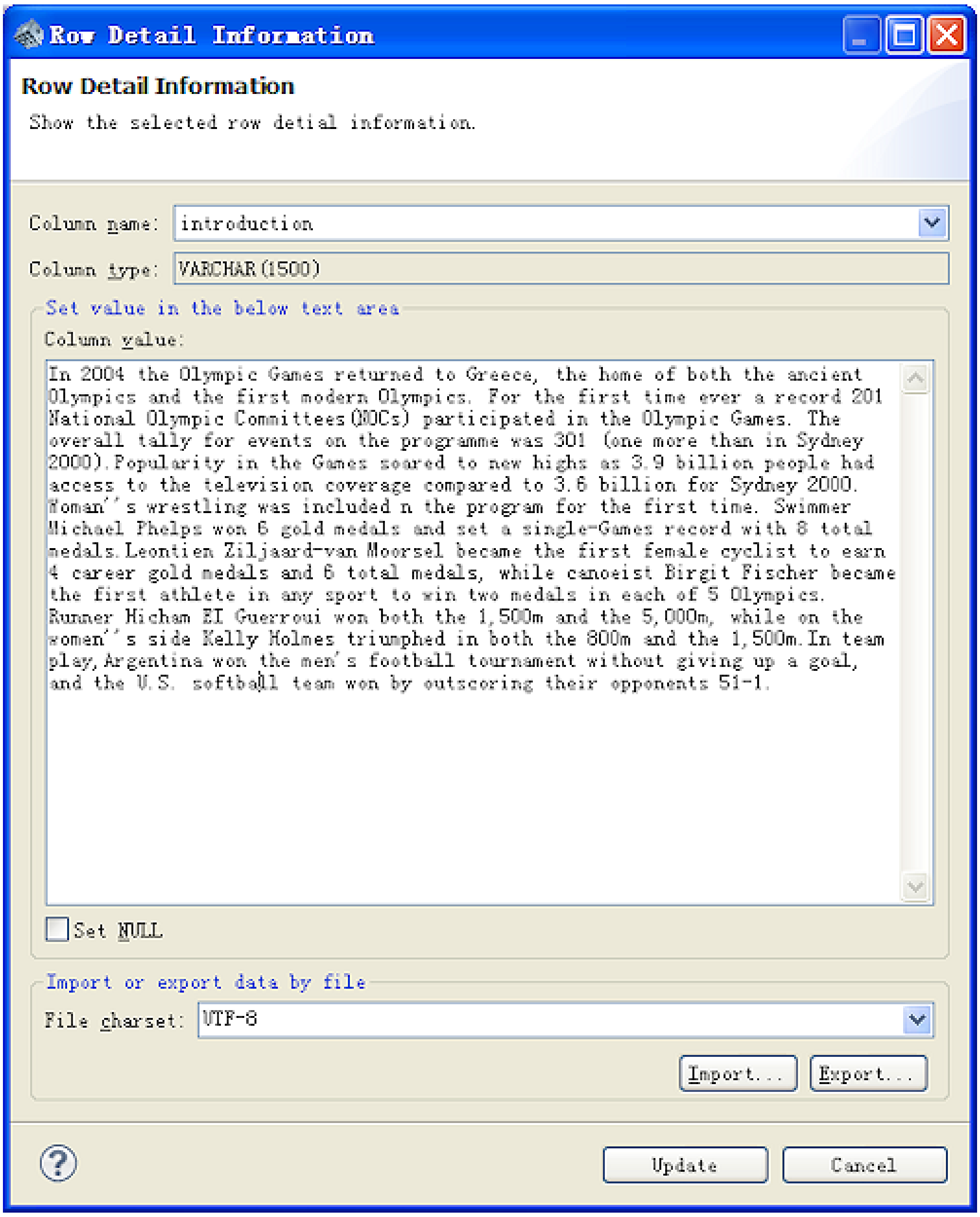
NO	code	name	gender	natior
1	10999	Fernandez Jesus		ESP
2	10998	Fernandez Jaime		AUS
3	10997	Fernandez Isabel		ESP
4	10996	Fernandez Gigi		USA
5	10995	Fernandez Ana Ibis		CUB
6	10994	Fernandez Abelardo		ESP
7	10993	Feri Attila		HUN

- **Copy** : Copies an entire row. To copy a specific column of the row only, select [View detail].
- **Modify** : The information can be modified directly from the query results pane. Change data to a modifiable state by double-clicking it, and then modify it. This option is available when [Get OID info] is set in the given host's Properties ( ) > [Query] dialog box, or [OID info ( )] is specified.



NO	OID	code	name	ge
1	@680|1|0	10999	Fernandez Jesus	M
2	@680|2|0	10998	Fernandez Jaime	M
3	@680|3|0	10997	Fernandez Isabel	W
4	@680|4|0	10996	Fernandez Gigi	W
5	@680|5|0	10995	Fernandez Ana Ibis	W
6	@680|6|0	10994	Fernandez Abelardo	M

- **Delete** : You can delete data directly in the query results pane. Right-click the row to delete and then select [Delete]. This option is available only when [Get OID info] is set as the modify function does.
- **OID navigator** : This option can be activated by right-clicking an OID data. It provides a function that allows you to navigate the selected OID directly.
- **View detail** : If there are too long data to be displayed in a single row, or if its size is too big, it is very difficult to see column values on the query results pane. In this case, you can see column values in detail by right-clicking the row and then selecting [View detail]. You can also modify data directly in the [Detail] dialog box if [Get OID info] option is set. For BLOB or CLOB type data, you can import and export it with a separate file.



Row Detail Information

Show the selected row detial information.

Column name: introduction

Column type: VARCHAR(1500)

Set value in the below text area

Column value:

In 2004 the Olympic Games returned to Greece, the home of both the ancient Olympics and the first modern Olympics. For the first time ever a record 201 National Olympic Committees(NOCs) participated in the Olympic Games. The overall tally for events on the programme was 301 (one more than in Sydney 2000).Popularity in the Games soared to new highs as 3.9 billion people had access to the television coverage compared to 3.6 billion for Sydney 2000. Woman''s wrestling was included n the program for the first time. Swimmer Michael Phelps won 6 gold medals and set a single-Games record with 8 total medals.Leontien Ziljaard-van Moorsel became the first female cyclist to earn 4 career gold medals and 6 total medals, while canoeist Birgit Fischer became the first athlete in any sport to win two medals in each of 5 Olympics. Runner Hicham EI Guerroui won both the 1,500m and the 5,000m, while on the women''s side Kelly Holmes triumphed in both the 800m and the 1,500m.In team play,Argentina won the men's football tournament without giving up a goal, and the U.S. softball team won by outscoring their opponents 51-1.

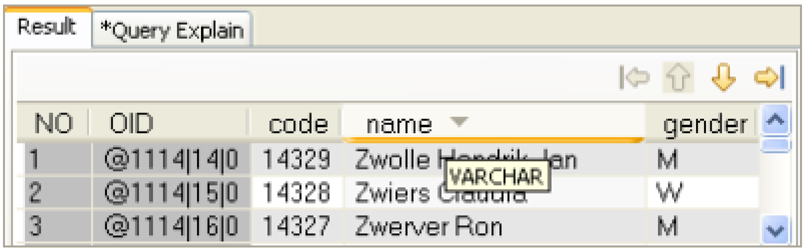
□ Set NULL

Import or export data by file

File charset: UTF-8

- **Export all** : Exports all data in the query results pane to an Excel or a CSV file.
- **Export selected** : Exports only the data in the row selected in the query results pane to an Excel or a CSV file. The charset for a file where exported data is saved can be specified.
- Export BLOC/CLOB data : Exporting/importing data for BLOB, CLOB, BIT VARYING (>100) cannot be executed, but importing detailed information in the result pane of the Query Editor is possible. Note that only type tags such as (BLOB), (CLOB), and (BIT) are displayed in the result pane, instead of displying real data.
- Import BLOB/CLOB data : Entering or editing data in the detailed information of the result pane is supported by opening the file where type data of STRING, CHAR, VARCHAR, NCHAR, NCHAR VARYING, BIT VARYING, BIT, BLOB, and CLOB is saved.

When you place the mouse pointer over a column name in the query results pane, you can see the size and data type of the column.



NO	OID	code	name	gender
1	@1114|14|0	14329	Zwolle Hendrik Jan	M
2	@1114|15|0	14328	Zwiers Claudia	W
3	@1114|16|0	14327	Zwerver Ron	M

Query Editor Options

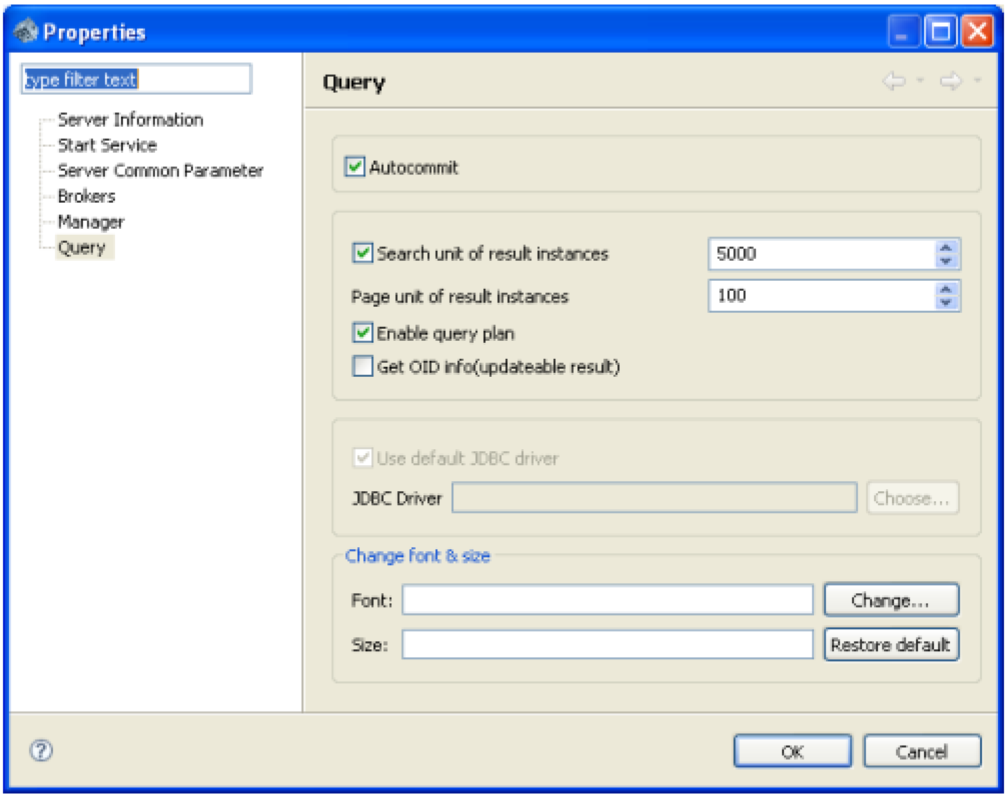
The Query Editor's options can be set in the following way. Options are applied according to the following priority.

How To	Priority	Description
[File] > [Preferences] > [Query options]	3rd	Basic information of the current CUBRID Manager. Default values that are applied last when there no values applied to each host and database.
[Host] > [Properties] > [Query]	2nd	It is applied prior to [File] > [Preferences] and commonly within the host.
[Database] > [Properties] > [Connection Information]	1st	It is applied prior to [Host] > [Properties]. Information such as the connection port and the character set can be specified. This information is applied to the Query Editor subsequently opened after the modification.

[File] > [Preferences] > [Query options]

Sets the basic information about the currently used CUBRID Manager. Provides the same items as the host setting.

[Host] > [Properties] > [Query]



☑ Autocommit

☑ Search unit of result instances: 5000

Page unit of result instances: 100

☑ Enable query plan

☐ Get OID info(updateable result)

☑ Use default JDBC driver

JDBC Driver:

Font:

Size:

In the [Query] dialog box, you can set the following options:

- **Autocommit** : You can set a default value so that autocommit is performed after a query is executed in the query edit pane. Even when [Autocommit] is selected, you can set ( )/clear ( ) the function from the toolbar in the Query Editor.
- **Search unit of result instances** : Sets the number of result rows to be fetched at once from the database after the execution of the query. That is, if the number of search result rows is 7,000 when the value is set to 5,000, you can select whether or not to continue to fetch more search results after 5,000 rows are fetched. In addition, if you execute queries that satisfy the following conditions, **ROWNUM** is automatically added based on the set value to prevent using excessive server resources.

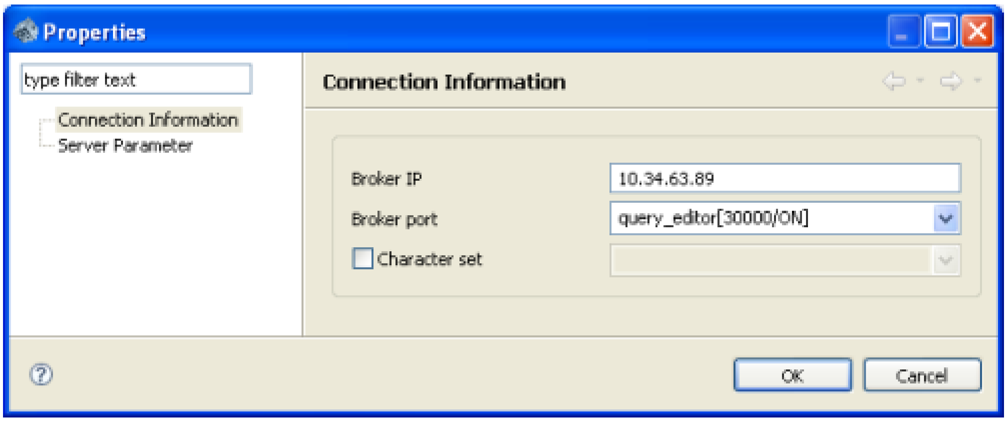
  WHERE condition is missing.

  GROUP BY is not used.

  ORDER BY is not used.

  Aggregate functions (**SUM**, **COUNT**, **MIN**, **MAX**, **AVG**, **STDDEV** and **VARIANCE**) are not used.
- Hierarchical Query is not used.
- **Page unit of result instances** : You can navigate in the query results pane by paging (|⇦ ⇧ ⇩ ⇨|) specified number of records.
- **Enable query plan** : You can check the query plan before and after its execution. Selecting this option may slightly affect the query execution time because plan information will be created in advance for queries to be executed.
- **Get OID info** : Fetches OID information during the query execution. This allows you to directly modify/delete data in the query results pane. However, “NONE” is displayed if the OID cannot be fetched as with Join queries. Selecting this option may extend the query execution time.
- **Character set** : You can set the default character set which is used in the host. The default is one used in the system where the CUBRID Manager is running.
- **Change font & size** : You can change fonts and their sizes to be used in the Query Editor.

[Database] > [Properties] > [Connection Information]



Broker IP 10.34.63.89

Broker port query_editor[30000/ON]

□ Character set

- **Broker IP** : By default, the Broker address for the database connection is set to the IP address of the database server. If the Broker server is not connected, you can connect to the Broker by modifying this value. The Broker address can be modified only by the CUBRID Manager **admin** user. Other users can connect only through the Broker address specified by the **admin** user.
- **Broker port** : A Broker port to be used for database connection. The port is displayed by the CUBRID Manager **admin** user. The Broker port can be modified only by the **admin** user. Other users can connect only through the Broker port specified by the admin user.
- **Character set** : You can specify the character set for each database. It is recommended to use only one character set for a database. The default is one used in the system where the CUBRID Manager is running.

Query Execution Plan

If [Display query plan] is selected in the Query Editor's options, [Display query plan] becomes activated in the Query Editor toolbar.

You can check how the selected query will be executed, even without executing it, by clicking [Display query plan] from the toolbar or by clicking < Ctrl+L> . You can also check query execution plan that have already been executed.

Since query plans are displayed all the time at the bottom of the Query Editor window, you can check the existing query plans by opening query plan history files without connecting to the database.

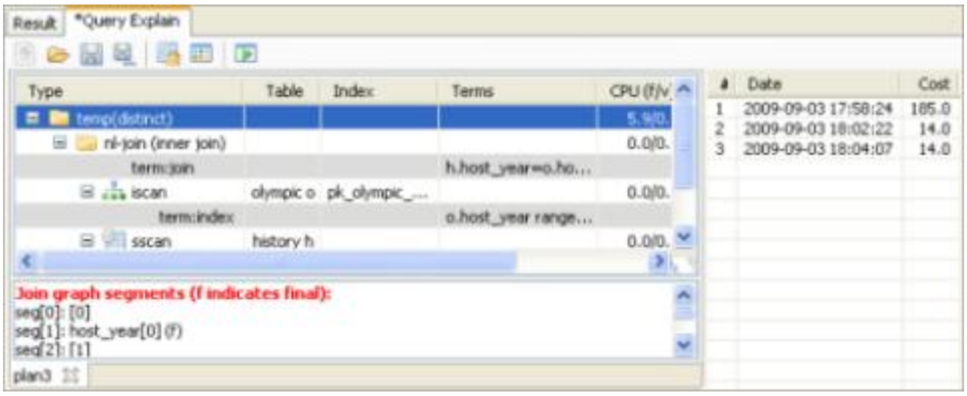
The query plan function retrieves the SQL execution plans it is used not for a one-time purpose but for a collection purpose to continuously manage and retrieve them. Every time you retrieve a query plan, the query plan history is accumulated. You can save this accumulated data to an .xml file. When you open the saved .xml file, you can check the original query plan and executed SQL statements. If it is a one-time retrieval, select [Disable to collecting histories] to view the current query plan without recording history.

The [Query Explain] tab consists of a toolbar, query plan display pane, original statement display pane and query plan history pane.

Query Explain tab

If you select a query and then click [Display query plan], the query plan is displayed in the [Query Explain] tab. The [Query Explain] tab shows the query plan summarized in the tree structure.

The [Query Explain] tab is located to the right of the [Result] tab. You can switch to the [Query Explain] tab while viewing the query result.



Type	Table	Index	Terms	CPU (f/v
temp(distinct)				5.9/0.
nl-join (inner join)				0.0/0.
term:join			h.host_year=o.ho...	
iscan	olympic o	pk_olympic_...		0.0/0.
term:index			o.host_year range...	
sscan	history h			0.0/0.

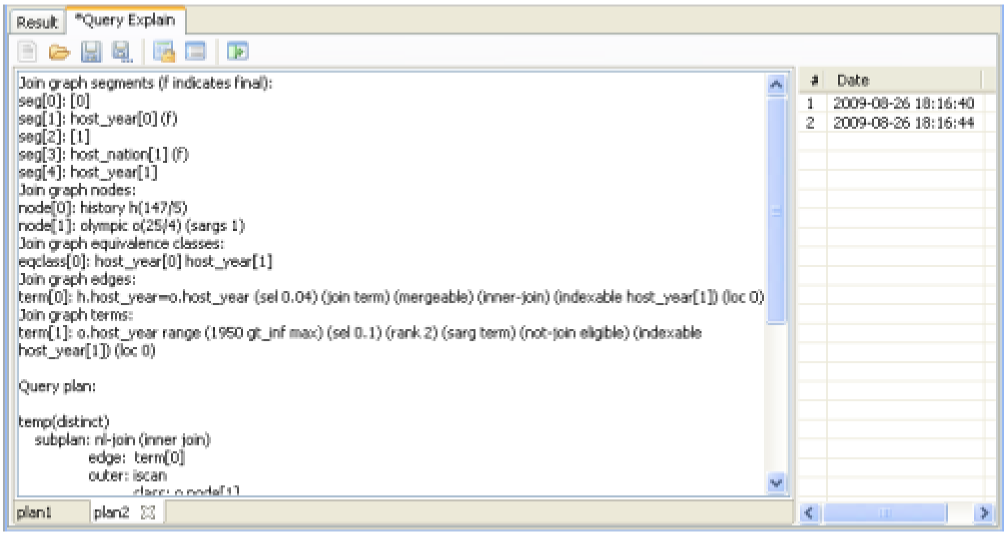
```
Join graph segments (f indicates final):
seg[0]: [0]
seg[1]: host_year[0] (f)
seg[2]: [1]
```

#	Date	Cost
1	2009-09-03 17:58:24	185.0
2	2009-09-03 18:02:22	14.0
3	2009-09-03 18:04:07	14.0

Query Explain Toolbar

The Query Explain toolbar has the following functions:

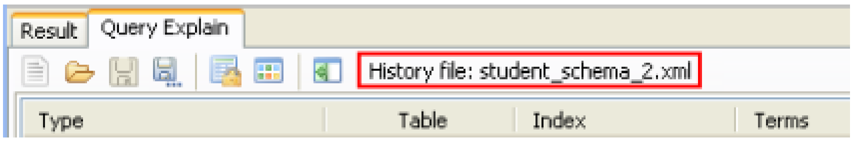
- **New** : Initializes all query plan histories retrieved so far and begins collecting them again. This function is used to initialize the existing job.
- **Open a query explain file** : Imports a previously saved query plan history file. If you open the .xml file, you can view the original query plan.
- **Save a query explain file** , **Save a query explain file as** : Saves the collected query plan to an external file. The file extension is **xml**.
- **Disable to collecting histories** : Histories are added to the query plan history pane whenever you retrieve the execution plan by using [Display query plan] in the Query Editor. To retrieve a query plan temporarily without recording history, click [Disable to collecting histories]. The name of the tab below the query plan display pane is displayed as "plan."
- **Query plan type** : By toggling the icon, the type can be switched between text and tree . You can view the unprocessed query plan source created by the database in a text form.



```
Join graph segments (f indicates final):
seg[0]: [0]
seg[1]: host_year[0] (f)
seg[2]: [1]
seg[3]: host_nation[1] (f)
seg[4]: host_year[1]
Join graph nodes:
node[0]: history h(147/5)
node[1]: olympic o(25/4) (sargs 1)
Join graph equivalence classes:
eqclass[0]: host_year[0] host_year[1]
Join graph edges:
term[0]: h.host_year=o.host_year (sel 0.04) (join term) (mergeable) (inner-join) (indexable host_year[1]) (loc 0)
Join graph terms:
term[1]: o.host_year range (1950 gt_inf max) (sel 0.1) (rank 2) (sarg term) (not-join eligible) (indexable
host_year[1]) (loc 0)

Query plan:

temp(distinct)
   subplan: nl-join (inner join)
            edge:  term[0]
            outer: iscan
                   class: o node[1]
```

#	Date
1	2009-08-26 18:16:40
2	2009-08-26 18:16:44

- **Show or Hide a query history pane** ◀ ▶ : Shows or hides the collection history pane on the right side.
- **Query plan history file** : If a query plan history is saved as a file, or an existing file is opened, the name of the currently used query plan history file is displayed.

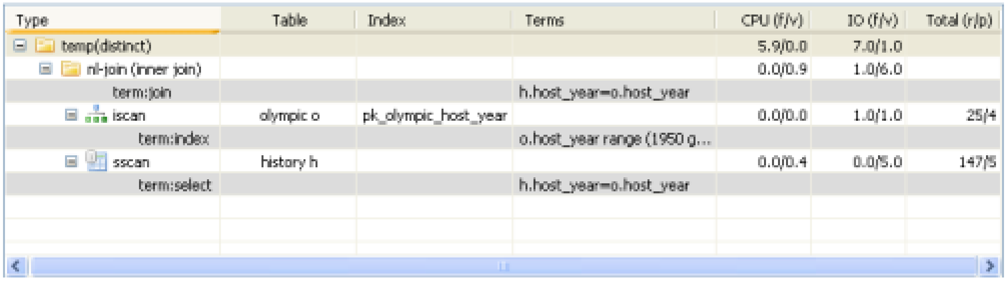


Query Plan Display Pane

In the query plan display pane, the query plan executed in each step is displayed in the tree structure.

Each item in the vertical axis is called a node. Each node contains different data. You can view the tree moving from the top to the bottom.

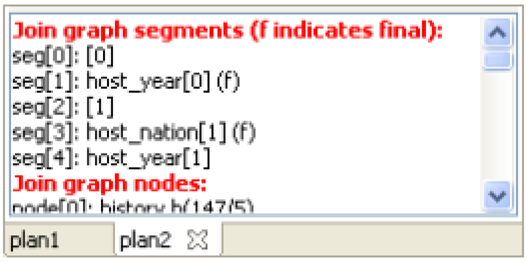
The horizontal axis is called an item and contains Type, Table, Index, Terms, CPU I/O cost, Disk I/O cost, and Total (ROW/PAGE).



Type	Table	Index	Terms	CPU (f/v)	IO (f/v)	Total (r/p)
⊟ temp(distinct)				5.9/0.0	7.0/1.0	
⊟ nl-join (inner join)				0.0/0.9	1.0/6.0	
term:join			h.host_year=o.host_year			
⊟ iscan	olympic o	pk_olympic_host_year		0.0/0.0	1.0/1.0	25/4
term:index			o.host_year range (1950 g...			
⊟ sscan	history h			0.0/0.4	0.0/5.0	147/5
term:select			h.host_year=o.host_year			

- **Type** : Indicates the scan type or join method (such as **sscan**, **iscan** and **idx-join**).
- **Table** : The table (class), view (virtual class) and alias, which are referred to when the node is executed, are displayed altogether.
- **Index** : The name of the index used is displayed if the type is **iscan**.
- **Terms** : Join and filter conditions are displayed. The contents are hidden for readability. If you click +, sub-nodes are extended to show details. In addition, different colors are used for different search conditions.
- **Cost** : Displays CPU and Disk I/O cost of the query plan. Fixed and variable costs are displayed separately.
- **Total (r/p)** : Displays the total number of rows and the number of pages to be used to fetch data.

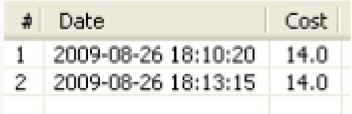
The original statements of the query plan selected in the query plan pane are displayed below the query plan display pane.



```
Join graph segments (f indicates final):
seg[0]: [0]
seg[1]: host_year[0] (f)
seg[2]: [1]
seg[3]: host_nation[1] (f)
seg[4]: host_year[1]
Join graph nodes:
node[0]: history h(147/5)
```

Query Plan History Pane

Histories are displayed accumulatively in the query plan history pane every time a query plan is executed. # is the accumulation order and corresponds to the tab number below the query plan display pane. Date indicates the date when a query plan is executed, and Cost is the sum of CPU and Disk I/O costs. If you double-click an item in the query plan history pane, you can view the query plan again in the query plan display pane.

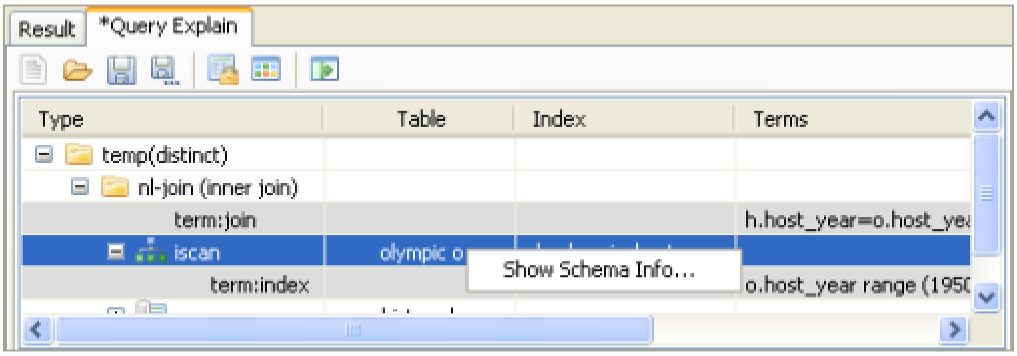


#	Date	Cost
1	2009-08-26 18:10:20	14.0
2	2009-08-26 18:13:15	14.0

Using Query Plans

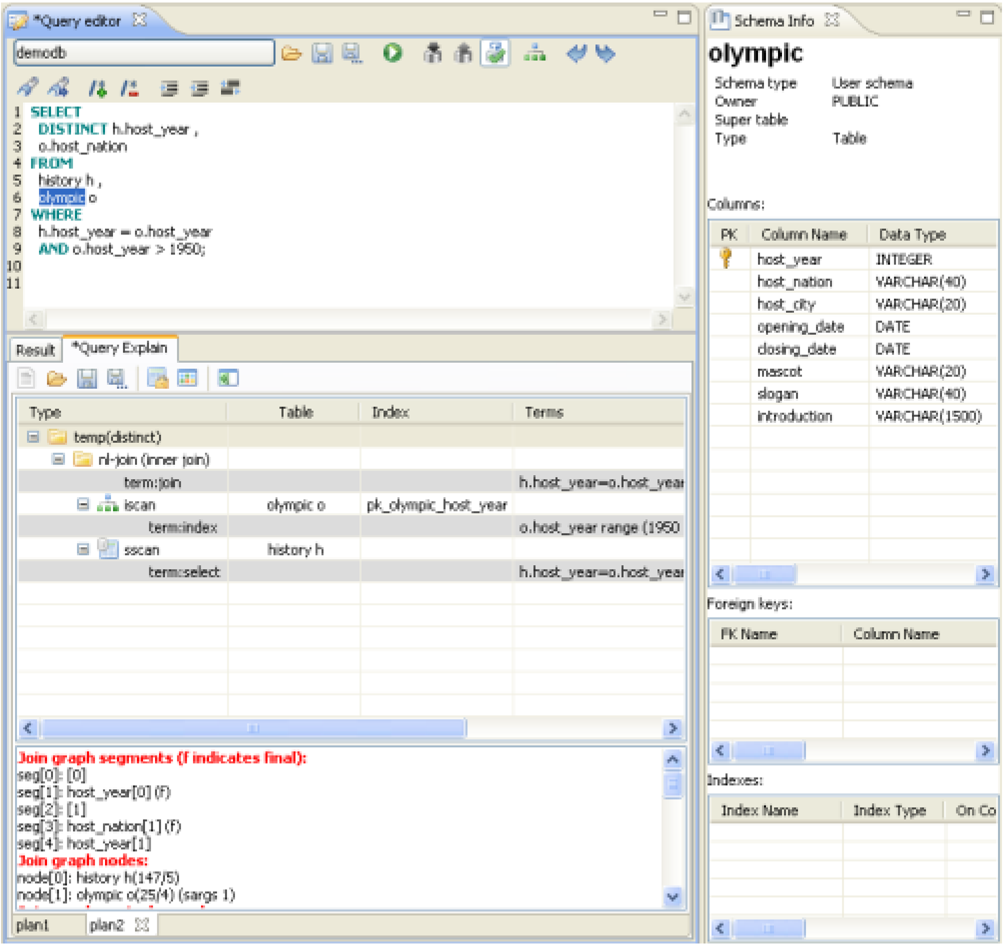
By using the query plan function, you can analyze data while viewing the query plan and the schema info of the corresponding table.

If you right-click a row where a table is located in the query plan display pane and then select [Show Schema Info], you can open and view the schema information of the table as well.



Type	Table	Index	Terms
temp(distinct)			
nl-join (inner join)			
term:join			h.host_year=o.host_yea
iscan	olympic o		
term:index			o.host_year range (1950

You can also view the information in a separate window by dragging the Schema Info pane out of the CUBRID Manager. This can be useful in an environment using multiple monitors.



```
SELECT
 DISTINCT h.host_year ,
 o.host_nation
FROM
 history h ,
 olympic o
WHERE
 h.host_year = o.host_year
 AND o.host_year > 1950;
```

Type	Table	Index	Terms
temp(distinct)			
nl-join (inner join)			
term:join			h.host_year=o.host_year
iscan	olympic o	pk_olympic_host_year	
term:index			o.host_year range (1950
sscan	history h		
term:select			h.host_year=o.host_year

```
Join graph segments (f indicates final):
seg[0]: [0]
seg[1]: host_year[0] (f)
seg[2]: [1]
seg[3]: host_nation[1] (f)
seg[4]: host_year[1]
Join graph nodes:
node[0]: history h(147/5)
node[1]: olympic o(25/4) (sargs 1)
```

olympic

Schema type User schema
Owner PUBLIC
Super table
Type Table

Columns:

PK	Column Name	Data Type
🔑	host_year	INTEGER
	host_nation	VARCHAR(40)
	host_city	VARCHAR(20)
	opening_date	DATE
	closing_date	DATE
	mascot	VARCHAR(20)
	slogan	VARCHAR(40)
	introduction	VARCHAR(1500)

Foreign keys:

FK Name	Column Name

Indexes:

Index Name	Index Type	On Co

Database

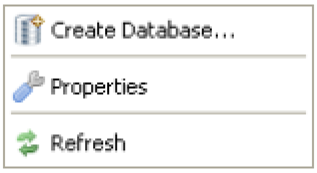
Database Functional Structure

The CUBRID Manager has functions for managing database, which can be largely divided into overall management and individual database management/development functions.

Overall Management Functions

To run these functions, right-click the database in the host navigation tree.

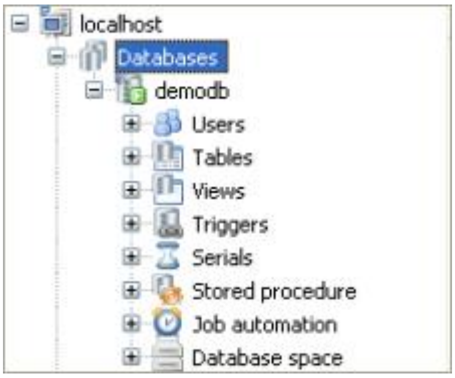
You can create databases, and set common database parameters that affect the entire host by selecting [Properties].



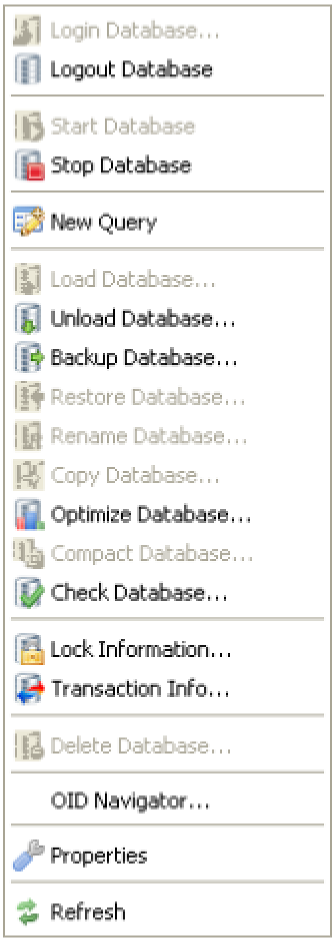
Individual Database Management/Development Function

Individual database management functions are provided in the navigation tree and via the shortcut menu.

Functions related objects in the database and job automation are provided in the navigation tree. These functions in the navigation tree are available only when the database is running.

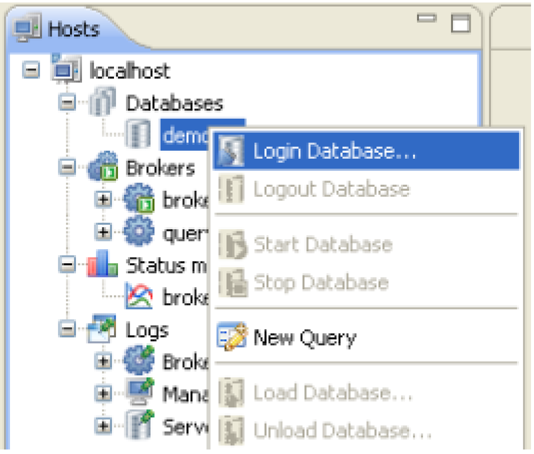


Shortcut menus provide all functions related to the operation. All functions except for the OID navigation function can be used even when the database is not running.

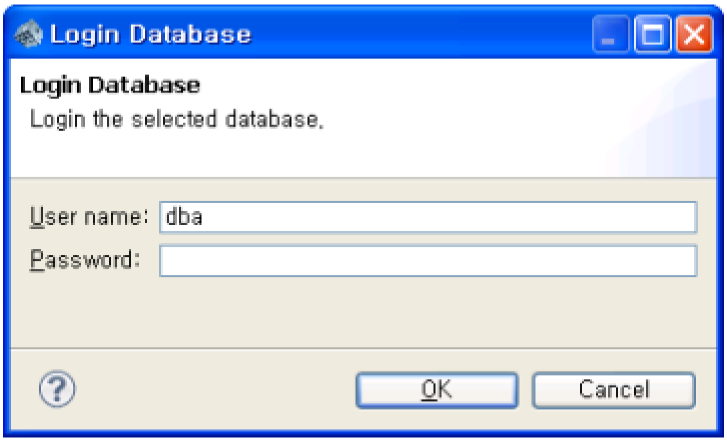


Connecting to Database

To run a database, you need to log into the database first. In the host navigation tree, right-click a database and select [Login Database] or double-click the database.



Enter a user name and password in the [Login Database] window. The default user name is **dba**; no password is required.



Login Database

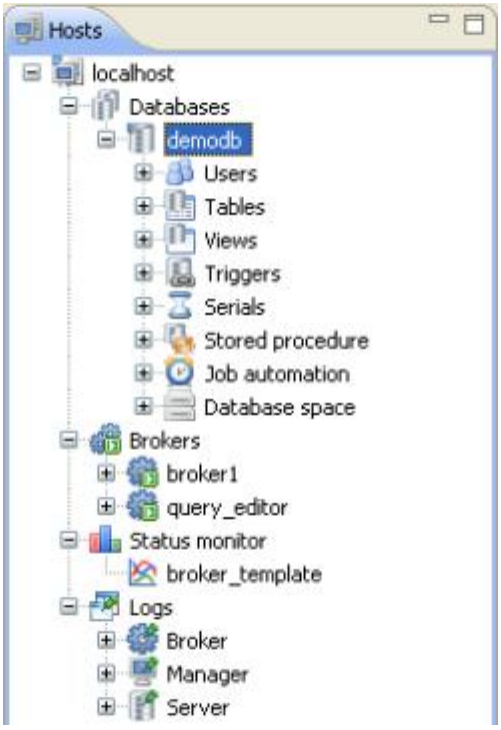
Login Database

Login the selected database.

User name: dba

Password:

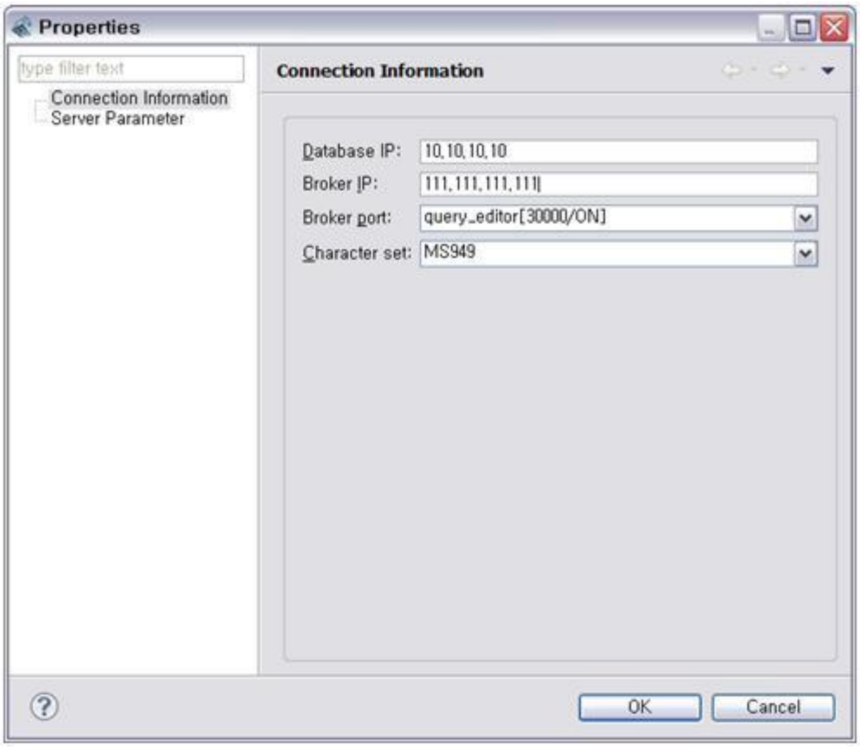
After user authorization is verified in the database login process, a tree corresponding to the user's authorization is displayed. To start the database, right-click the database in the left navigation tree and then sent [Start Database]. While the database is running, you can add a user or change the current password by right-clicking the [Users] node.



In the CUBRID Manager is not normally running, the following error message will be displayed: Cannot connect to a server. Please check the configuration environment of the CUBRID Manager server and other connection. In this case, you solve the problem by checking the following:

- Check whether the CUBRID Manager server is running.
- Open the configuration file of the CUBRID Manager server, and make sure that the value of the **cm_port** parameter is identical to the registered connection port. See Configuration the CUBRID Manager Server.
- If a firewall is installed on the system where the Manager client is running, allow all connection ports to be accessed to the Manager client connection (**cm_port**, **cm_port**+1). For example, if **cm_port** is 8001, the port 8002 must also be open.
- When the same operation is already being executed by the server, the message "Cannot execute the current operation because the previous operation is already running." is displayed. Then, retry the operation.
- When connecting Broker and a database in the CUBRID Manager client, the IP network between Broker and a database may be seperated. In that case, you must configure different addresses for the database and Broker in the

Connection Information of the Database Property. To change connection information, right-click in the navigation tree and then select [Properties].



Database IP: 10.10.10.10

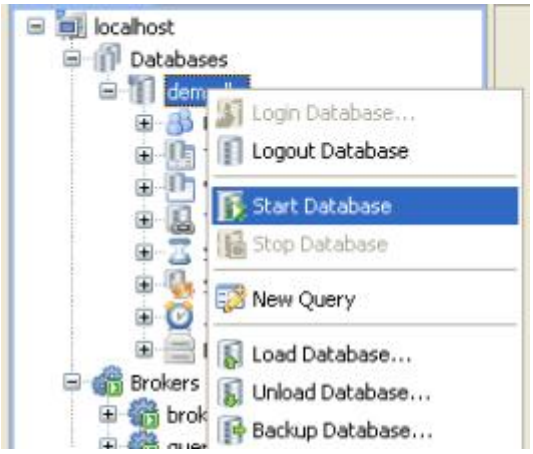
Broker IP: 111.111.111.111

Broker port: query_editor[30000/ON]

Character set: MS949

Starting Database

To execute a query in the CUBRID Manager, the database must start or be running. To start a database, select the database to start and click [Start Database] in the toolbar.



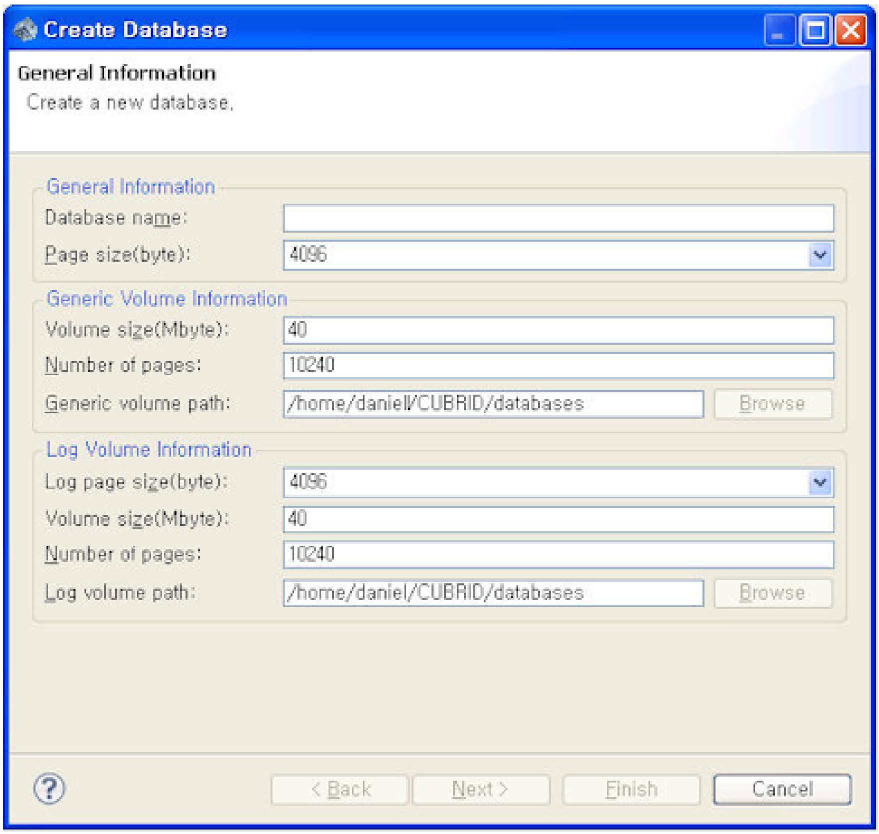
Creating Database

You can create a database by using the shortcut menu, or by clicking [Create Database] from the toolbar at the top. The Create Database wizard consists of 5 steps.

Only the CUBRID Manager **admin** account can create a database. For more information, see Setting User Authorization.

Step 1 : General Information

Enter the basic information such as database name, the generic and log volumes.



Create Database

General Information

Create a new database.

General Information

Database name:

Page size(byte): 4096

Generic Volume Information

Volume size(Mbyte): 40

Number of pages: 10240

Generic volume path: /home/daniel/CUBRID/databases

Log Volume Information

Log page size(byte): 4096

Volume size(Mbyte): 40

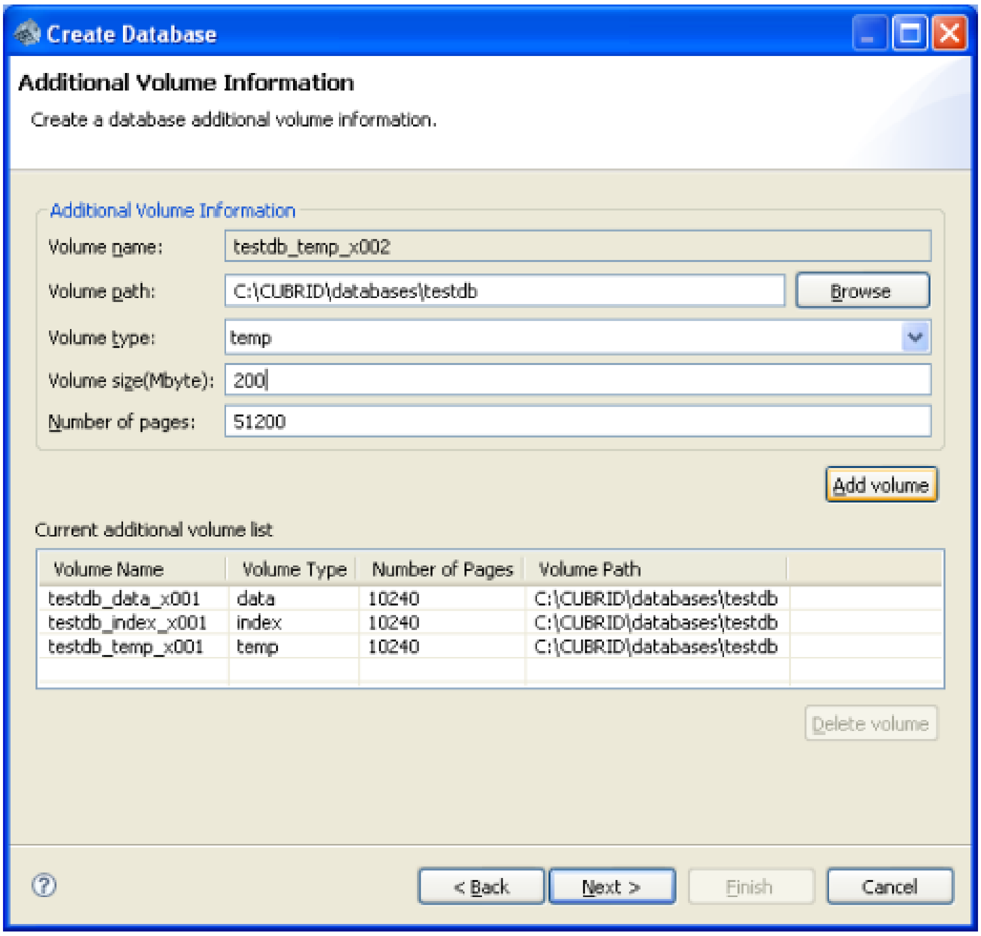
Number of pages: 10240

Log volume path: /home/daniel/CUBRID/databases

- **Database name** : The name of a database must be unique in the host. A warning message is displayed when the name already exists. The following rules apply to database names:
- Only alphanumeric are allowed for the name of the database. Its maximum length is 16 characters.
- Special characters that cannot be used in file names on Linux/UNIX are not allowed: space, *, &, %, $, |, ^, /, ~, \
- "." and ".." cannot be used in the name of the database.
- **Page size** : Select one of the following sizes: 1024, 2048, 4096, 8192, 16384. The default value is **4096**. Select an appropriate size for the purpose of the database since the size cannot be changed once the database is created. It is recommended to use the default value unless it is an exceptional case.
- **Generic Volume Information** : When the size of generic volume is entered in Mbytes, the number of pages for the volume is automatically calculated and displayed. The default value of a generic volume path is specified in the database location file (**$CUBRID_DATABASES/databases.txt**). When the database is being created on the server as the CUBRID Manager server, you can select a directory by clicking [Browse].
- **Log Volume Information** : Enter the size and path of the log volume. The default log volume path is the same as generic volume path.

Step 2 : Additional Volume Information

Enter information of additional volume by type such as generic, data, index, temp. In order to use the Automation of adding volume function for each volume type, you must configure the data volume and index volume.



Create Database

Additional Volume Information

Create a database additional volume information.

Additional Volume Information

Volume name:	testdb_temp_x002
Volume path:	C:\CUBRID\databases\testdb
Volume type:	temp
Volume size(Mbyte):	200
Number of pages:	51200

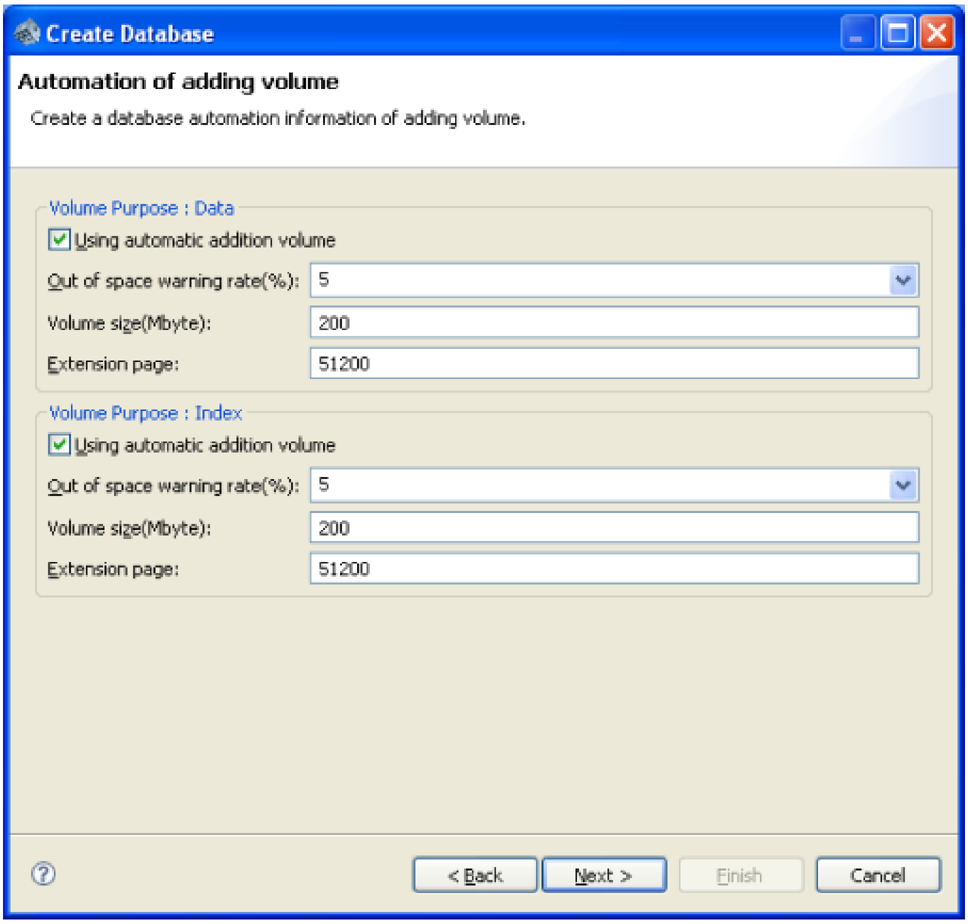
Current additional volume list

Volume Name	Volume Type	Number of Pages	Volume Path
testdb_data_x001	data	10240	C:\CUBRID\databases\testdb
testdb_index_x001	index	10240	C:\CUBRID\databases\testdb
testdb_temp_x001	temp	10240	C:\CUBRID\databases\testdb

- **Additional Volume Information** : If an additional volume is expected during the database creation, set information about the volume to be added in this step. If you click [Add volume] after entering the name, path, type and size of the volume to be added, the new volume is added accordingly. You can specify a volume type for each purpose, such as data, index, temp, and generic. You must additionally enter the data volume and index volume of an additional volume.
- **Add volume/Delete volume** : If you click [Add volume], the volume to be added is displayed on the list. If you click [Delete volume] after selecting a volume from the list, the volume is not created.
- **Current additional volume list** : Displays the list of volumes to be added during database creation.

Step 3 : Automation of Adding Volume

Enter information used for the automatic addition feature when data or index volumes do not have enough space. The automatic addition functionality can be used for a data or an index volume.



Automation of adding volume

Create a database automation information of adding volume.

Volume Purpose : Data

☑ Using automatic addition volume

Out of space warning rate(%): 5

Volume size(Mbyte): 200

Extension page: 51200

Volume Purpose : Index

☑ Using automatic addition volume

Out of space warning rate(%): 5

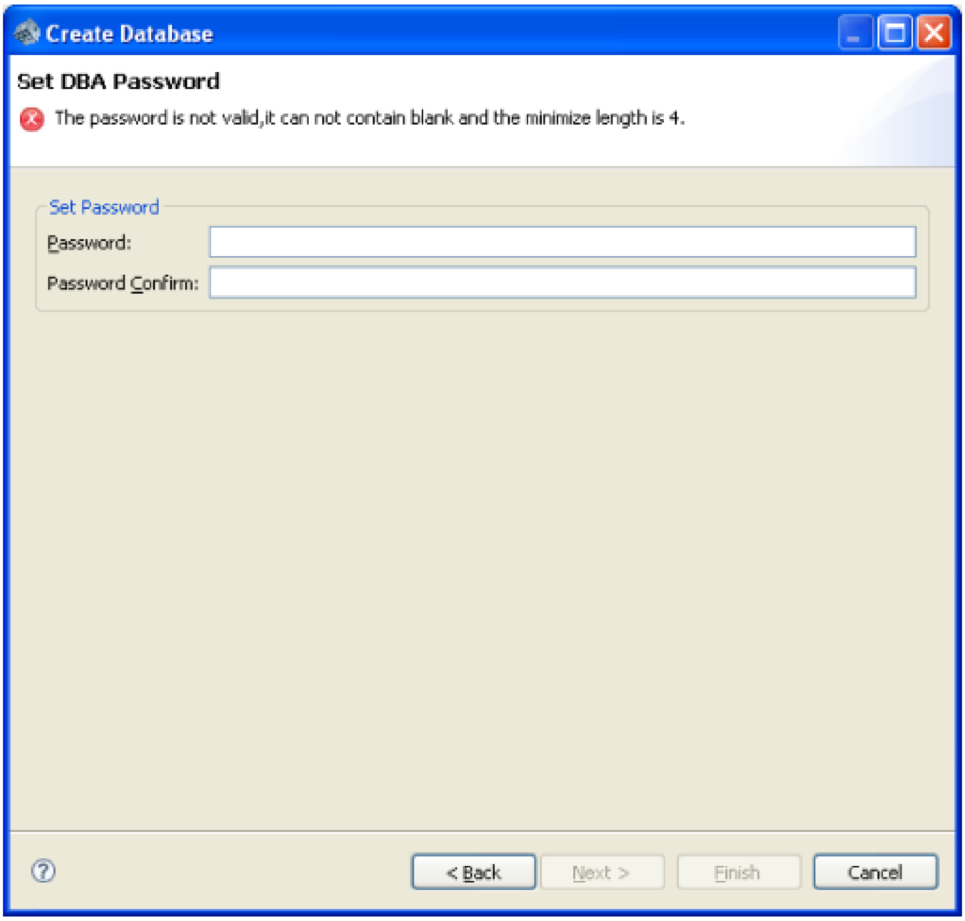
Volume size(Mbyte): 200

Extension page: 51200

- **Using automatic addition volume** : When it is selected, the automatic addition functionality is used for the selected volume type.
- **Out of space warning rate** : A volume is added automatically when the remaining volume equals to the value set by [Out of space warning rate]. For example, if this value is set to 5 % and the remaining space of the volume is 5 %, a data volume is added automatically. The minimum value is 5, and the maximum is 30.
- **Volume size** : Enter the size of the volume to be added automatically.

Step 4 : Set DBA Password

Enter the password for the DBA account of the database being created. The password cannot contain any spaces, and must be at least 4 characters.



Set DBA Password

The password is not valid,it can not contain blank and the minimize length is 4.

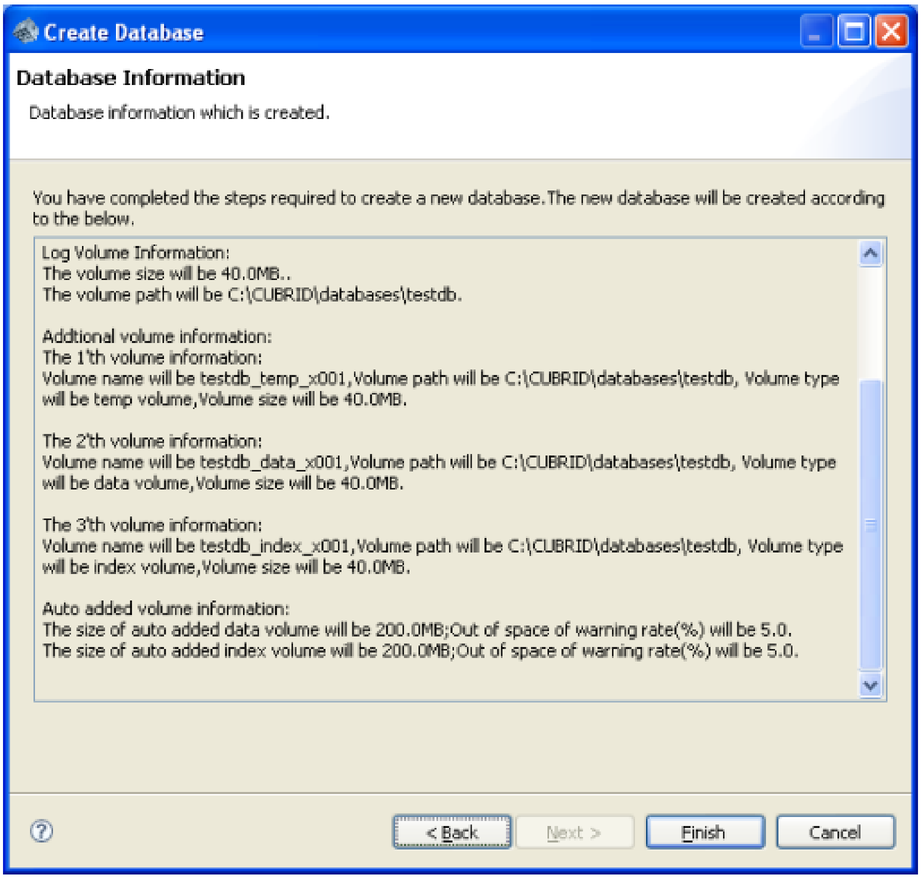
Set Password

Password:

Password Confirm:

Step 5 : Database Information

Confirm the information entered up to step 4 and create the database. If you need to make a change, click [Back] to go back to the previous step.



Database Information

Database information which is created.

You have completed the steps required to create a new database.The new database will be created according to the below.

Log Volume Information:
The volume size will be 40.0MB..
The volume path will be C:\CUBRID\databases\testdb.

Addtional volume information:
The 1'th volume information:
Volume name will be testdb_temp_x001,Volume path will be C:\CUBRID\databases\testdb, Volume type will be temp volume,Volume size will be 40.0MB.

The 2'th volume information:
Volume name will be testdb_data_x001,Volume path will be C:\CUBRID\databases\testdb, Volume type will be data volume,Volume size will be 40.0MB.

The 3'th volume information:
Volume name will be testdb_index_x001,Volume path will be C:\CUBRID\databases\testdb, Volume type will be index volume,Volume size will be 40.0MB.

Auto added volume information:
The size of auto added data volume will be 200.0MB;Out of space of warning rate(%) will be 5.0.
The size of auto added index volume will be 200.0MB;Out of space of warning rate(%) will be 5.0.

Configuring Database

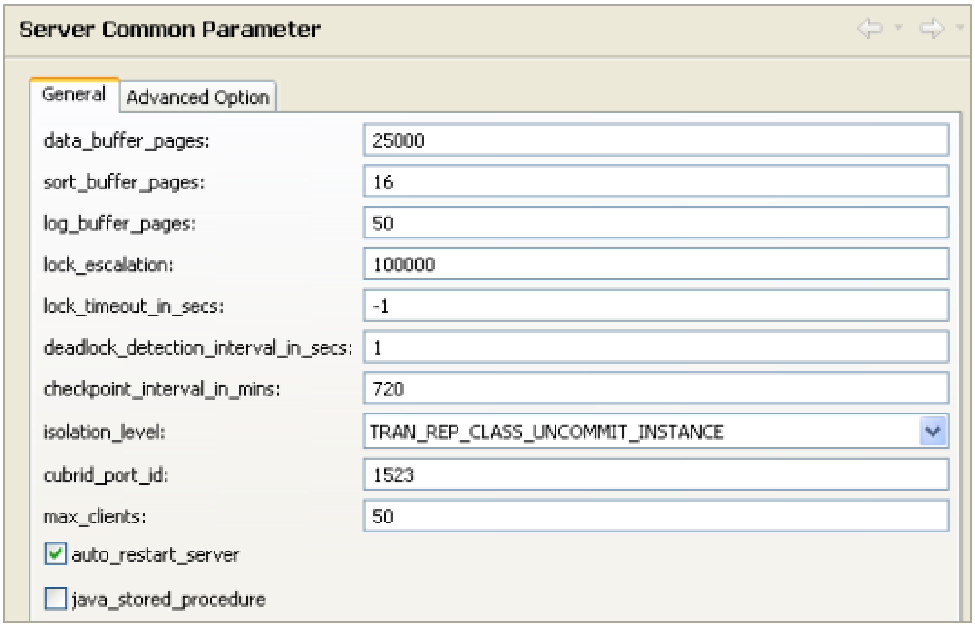
You can set database server parameters as follows:

- To set server parameters to be commonly applied to all databases in the host, right-click the host or [Databases], select [Properties], and then make settings in [Server Parameter].
- To set parameters of a certain database, right-click the database, select [Properties], and then make settings in [Server Parameter].

Server Parameter Setting

Set common parameters for the **[common]** section in the **$CUBRID/conf/cubrid.conf** file. You can set parameters that are included upon installation by default in the [General] tab, while all the other parameters can be set in the [Advanced Option] tab. For more information about each parameter, see Database Server Configuration.

- **General**



Server Common Parameter

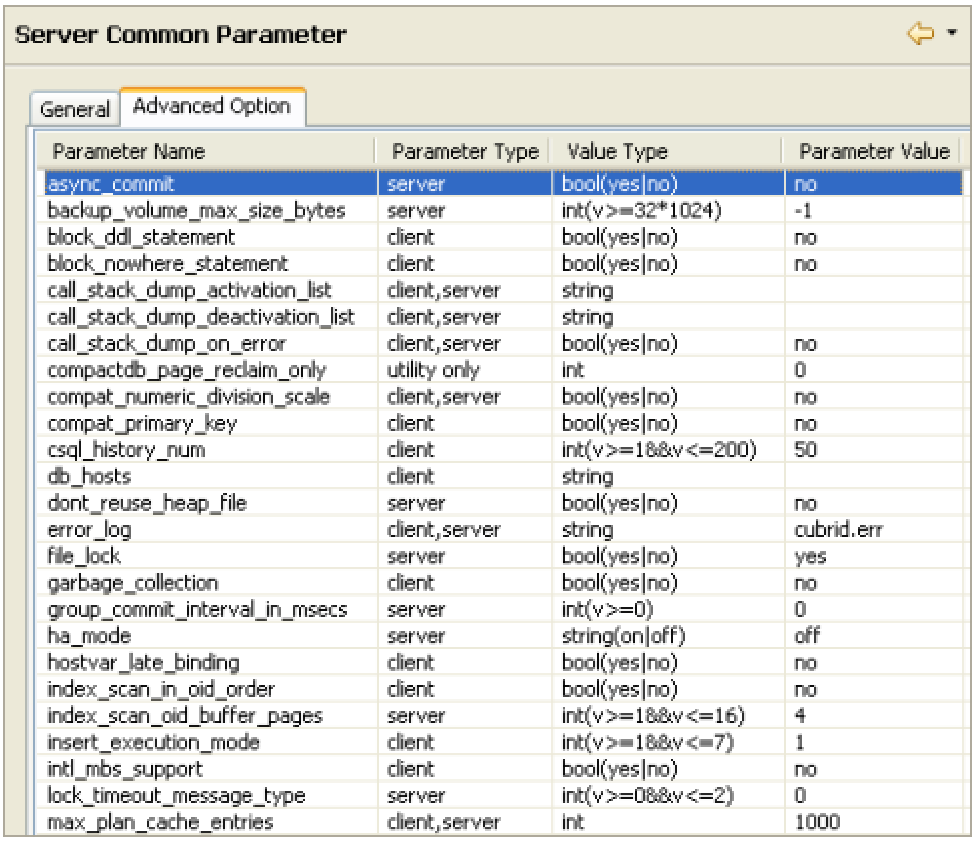
General | Advanced Option

data_buffer_pages:	25000
sort_buffer_pages:	16
log_buffer_pages:	50
lock_escalation:	100000
lock_timeout_in_secs:	-1
deadlock_detection_interval_in_secs:	1
checkpoint_interval_in_mins:	720
isolation_level:	TRAN_REP_CLASS_UNCOMMIT_INSTANCE
cubrid_port_id:	1523
max_clients:	50

☑ auto_restart_server

□ java_stored_procedure

- **Advanced Option**

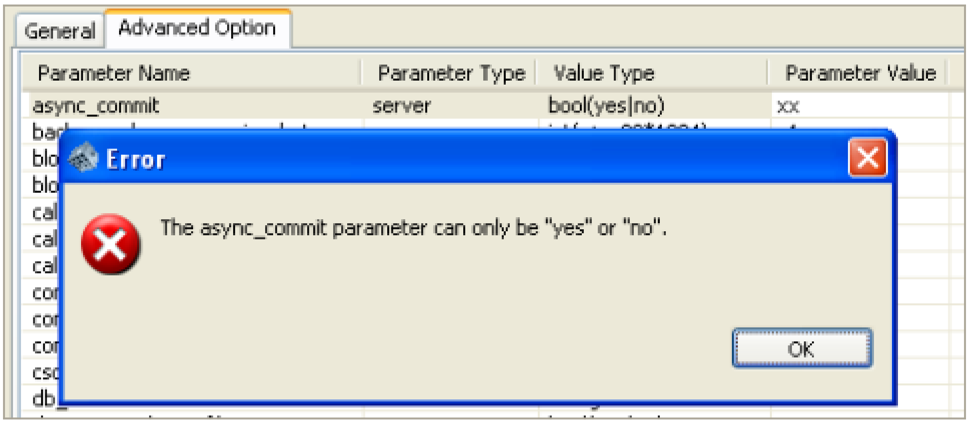


Server Common Parameter

General | Advanced Option

Parameter Name	Parameter Type	Value Type	Parameter Value
async_commit	server	bool(yes|no)	no
backup_volume_max_size_bytes	server	int(v>=32*1024)	-1
block_ddl_statement	client	bool(yes|no)	no
block_nowhere_statement	client	bool(yes|no)	no
call_stack_dump_activation_list	client,server	string	
call_stack_dump_deactivation_list	client,server	string	
call_stack_dump_on_error	client,server	bool(yes|no)	no
compactdb_page_reclaim_only	utility only	int	0
compat_numeric_division_scale	client,server	bool(yes|no)	no
compat_primary_key	client	bool(yes|no)	no
csql_history_num	client	int(v>=1&&v<=200)	50
db_hosts	client	string	
dont_reuse_heap_file	server	bool(yes|no)	no
error_log	client,server	string	cubrid.err
file_lock	server	bool(yes|no)	yes
garbage_collection	client	bool(yes|no)	no
group_commit_interval_in_msecs	server	int(v>=0)	0
ha_mode	server	string(on|off)	off
hostvar_late_binding	client	bool(yes|no)	no
index_scan_in_oid_order	client	bool(yes|no)	no
index_scan_oid_buffer_pages	server	int(v>=1&&v<=16)	4
insert_execution_mode	client	int(v>=1&&v<=7)	1
intl_mbs_support	client	bool(yes|no)	no
lock_timeout_message_type	server	int(v>=0&&v<=2)	0
max_plan_cache_entries	client,server	int	1000

A validation check is performed for all values set in [Server Parameter] immediately upon editing so that the user's mistake can be minimized.



Parameter Name	Parameter Type	Value Type	Parameter Value
async_commit	server	bool(yes|no)	xx

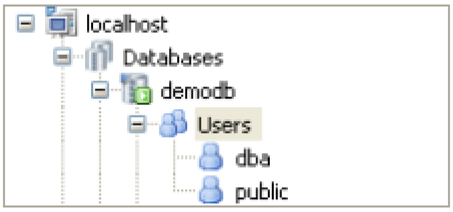
The async_commit parameter can only be "yes" or "no".

Server-Specific Parameter Setting

Server-specific parameter setting provides the same functionality as the server parameter setting. However, set values apply only to the given database and are added only to the **[@DBNAME]** section in the **$CUBRID/conf/cubrid.conf** file.

User

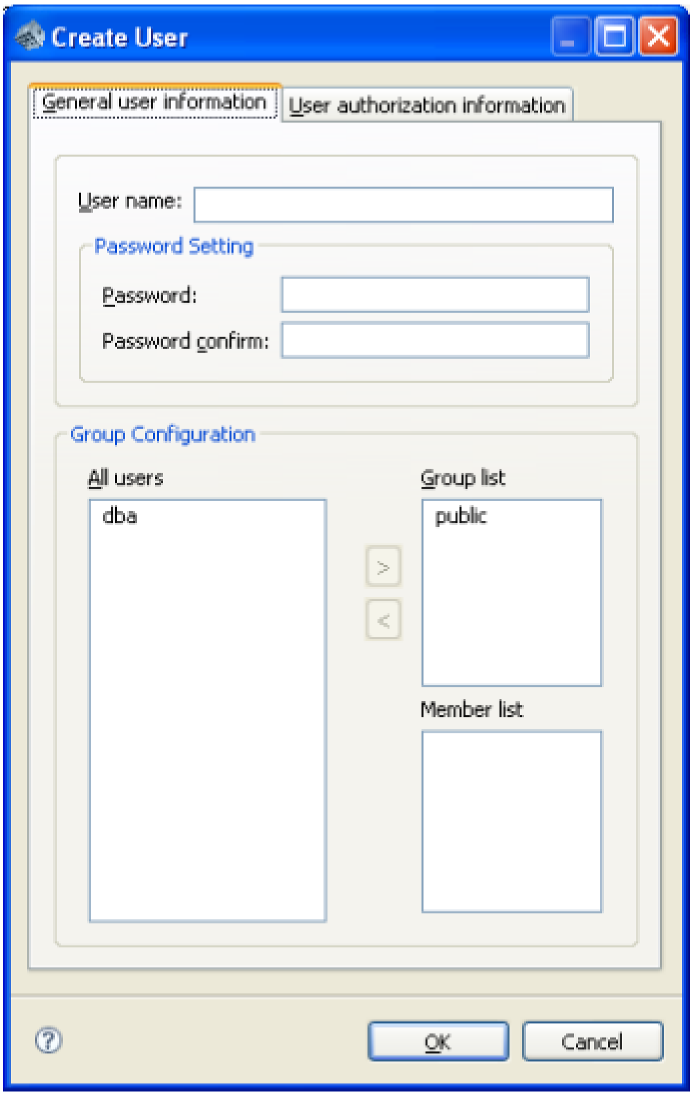
You can add, edit or drop users by right-clicking [Users] in the navigation tree after login to the database. All users of the current database are displayed in the sub-node of [Users].



Create User

You can add a user by right-clicking [Users] in the navigation tree and then selecting [Create User] or by clicking [Create User] from the toolbar menu.

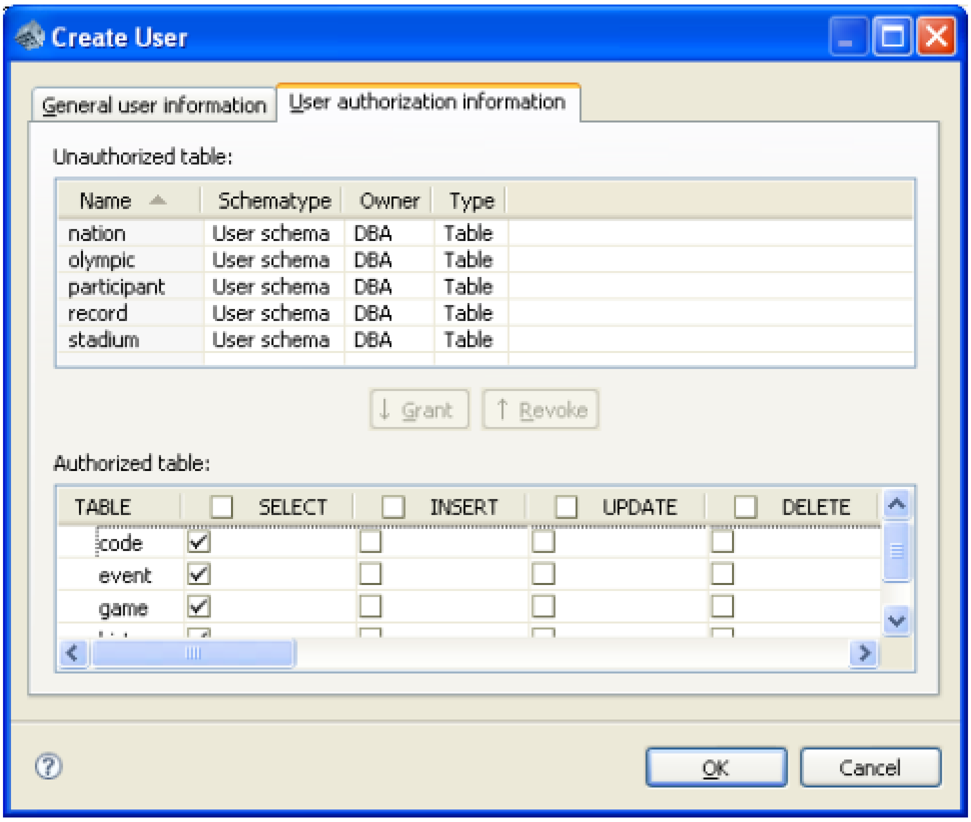
General User Information



- **User name/Password** : The name and password of the user to be added. The maximum length of the user name is 32 characters.
- **All users** : Lists the list of users that can be selected as the group for the user to be added.
- **Authorization of this user** : Displays the list of groups of users to be added. You can specify a group with which users to add from the [All users] list will join by using the arrow button. However, you cannot modify the authorization of **dba** and **public** accounts.
- **Users that have this user's authorization** : Displays the list of users that have this user's authorization.

User Authorization Information

In the [User authorization information] tab, you can grant or revoke authorization for each table. You can sort authorization grantees for each column and select multiple tables to grant authorization.



General user information | User authorization information

Unauthorized table:

Name	Schematype	Owner	Type
nation	User schema	DBA	Table
olympic	User schema	DBA	Table
participant	User schema	DBA	Table
record	User schema	DBA	Table
stadium	User schema	DBA	Table

Authorized table:

TABLE	☐ SELECT	☐ INSERT	☐ UPDATE	☐ DELETE
code	☑	☐	☐	☐
event	☑	☐	☐	☐
game	☑	☐	☐	☐

Edit User

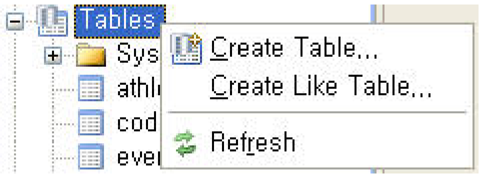
You can edit the settings entered in the [Add User] dialog box. However, the user name cannot be modified.

Drop User

You can drop the selected database user account.

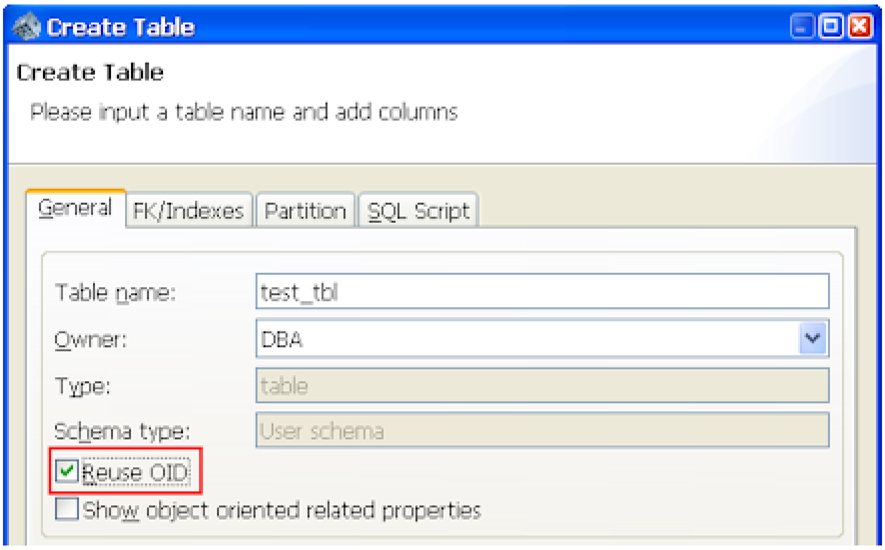
Table

When you login to the database, you can see accessible tables and system tables in the navigation tree. If you right-click the [Table] in the navigation tree, you can view the [Create Table], [Create Like Table], and [Refresh] menus.



Create Table

Right-click [Tables] in the navigation tree and then select [Create Table] or click [Create Table] from the toolbar. Then, a wizard which makes you create a new table will appear.



Create Table

Please input a table name and add columns

Table name: test_tbl

Owner: DBA

Type: table

Schema type: User schema

☑ Reuse OID

□ Show object oriented related properties

- **[General] tab** : You can define the name of the table to be added and add, edit and drop columns. You can also set the primary key (PK) and, before the table is created, adjust the position of the selected column by using [↑] and [↓]. Using the add column wizard, you can set the name, type, default value and constraints of the column to be added. You can also set to display a warning message when you enter something that is not grammatically correct, or to disable the selection of a grammatically incorrect entry.
- **Reuse_OID** : If this option is selected, a table is created with the option **REUSE_OID** applied. For this kind of table, using with OID is restricted. For more information on OID reuse table, see Table Options (REUSE_OID).
- **[FK/Indexes] tab** : You can set foreign keys and indexes.
- **Add Foreign Key** : Using the add foreign key wizard, you can set the name of the foreign key, the name and primary key of the reference table and the trigger actions to maintain referential integrity. The **ON UPDATE**, **ON DELETE** and **ON CACHE OBJECT** options are provided.
- **Add Index** : You can set index name, index type, a column to be indexed. Ascending (asc) and descending (desc) sorting can be selected within the supported range. For **REVERSE** indexes, only descending sorting is supported.
- **[Partition] Tab** : Supports partitioning setting and modification of the given table.
- **Add Partition** : Using the add partition wizard, you can set the partition type and expression. The **RANGE**, **LIST** and **HASH** partitions are supported.
- **Edit/Delete Partition** : You can edit or delete partition.
- **[SQL Statement] Tab** : You can check and copy SQL statements created according to the settings in **[General]**, **[Foreign Key/Index]** and **[Partition]** tabs.

Adding Object Oriented Tables

To add a table with object-oriented properties, select [Show object oriented related properties]. When you select [Show object oriented related properties], the Inheritance tab is added.

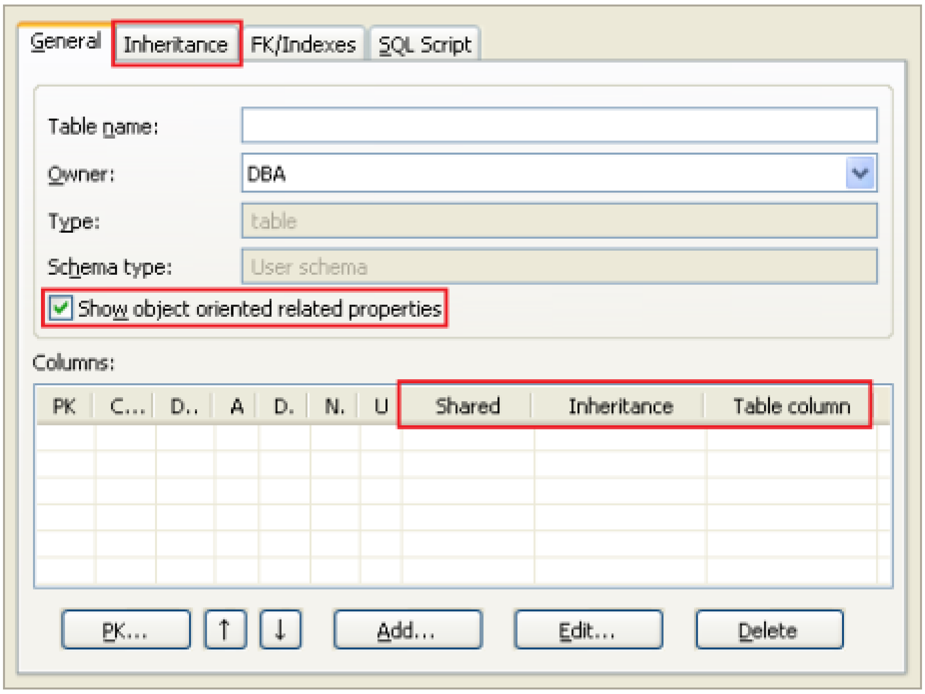


Table name:

Owner: DBA

Type: table

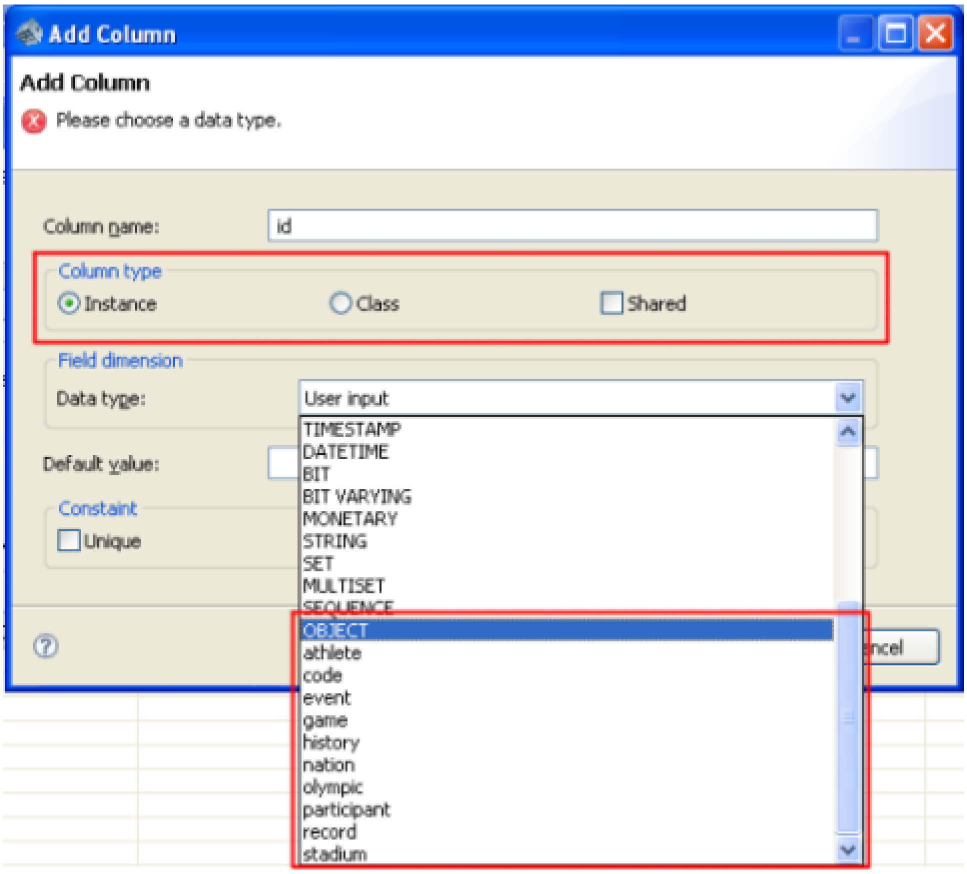
Schema type: User schema

☑ Show object oriented related properties

Columns:

PK	C...	D..	A	D.	N.	U	Shared	Inheritance	Table column

- **[General] tab** : When you select [Show object oriented related properties], Shared, Inheritance and Table column are added as a column in the table. When adding or editing columns, you can choose the column type as shown in the figure below. You can choose OBJECT or a table in the database as the data type.



Add Column

Please choose a data type.

Column name: id

Column type

● Instance ○ Class □ Shared

Field dimension

Data type: User input

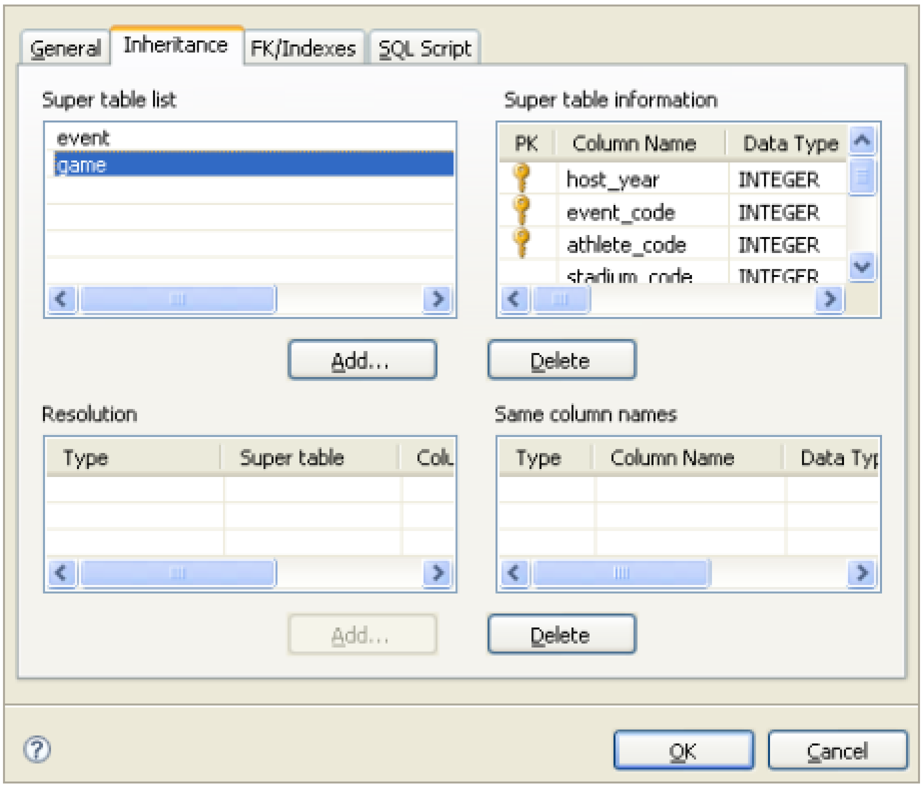
TIMESTAMP
DATETIME
BIT
BIT VARYING
MONETARY
STRING
SET
MULTISET
SEQUENCE
OBJECT
athlete
code
event
game
history
nation
olympic
participant
record
stadium

Default value:

Constraint

□ Unique

- **[Inheritance] tab** : You can define the super table to inherit from. If a column name conflict occurs, it can be adjusted.



PK	Column Name	Data Type
🔑	host_year	INTEGER
🔑	event_code	INTEGER
🔑	athlete_code	INTEGER
	stadium_code	INTEGER

Select All

Right-click a table in the navigation tree and select [Select All]. Or you can drag and drop the table into the editor results pane of the Query editor when it is open. Then, a new Query editor opens and retrieves the entire data.

Select Count

Retrieves a total data count of the table and performs the same functionality as the following syntax.

```
SELECT COUNT(*) FROM table_name
```

Delete All Records

Deletes all records from the table and performs the same functionality as the following syntax.

```
DELETE FROM table_name
```

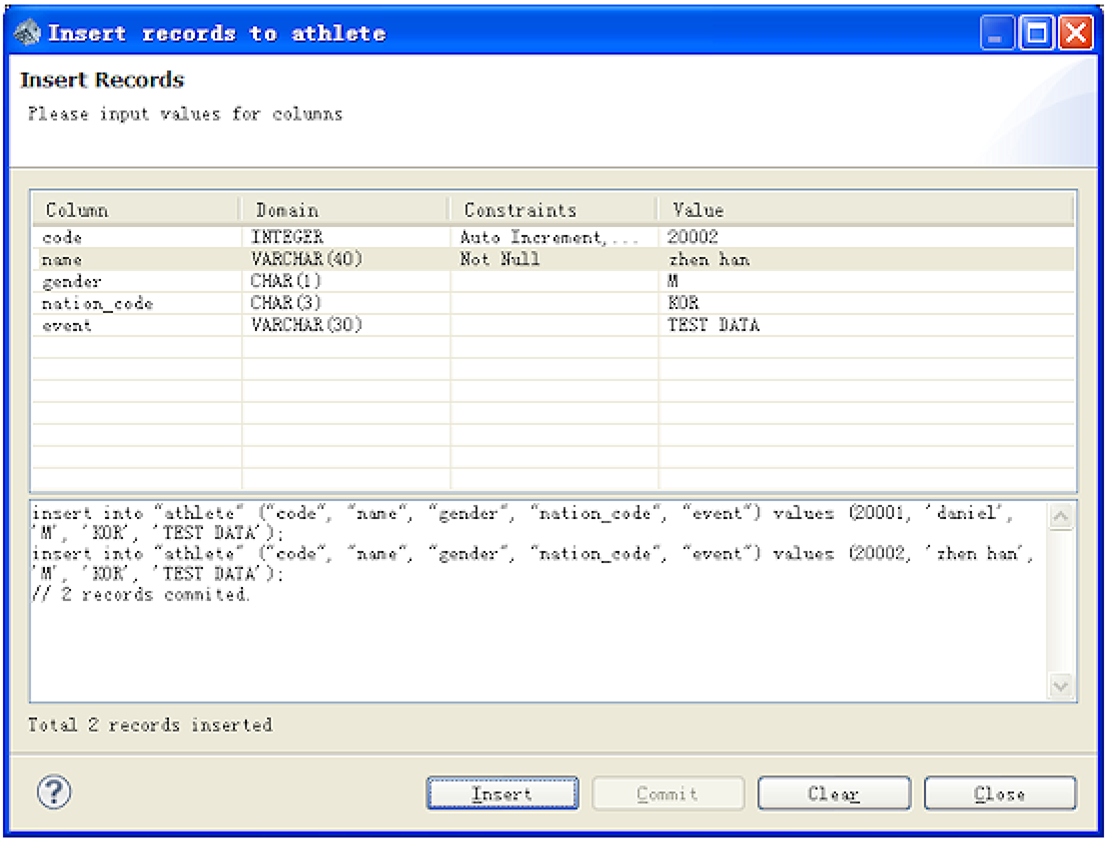
TRUNCATE TABLE

Deletes entire data in the table. It can delete all records including indexes and constraints in a table, so it is faster than [DELETE ALL]. The **ON DELETE** trigger is not activated when you use the [TRUNCATE TABLE]. It can perform the same functionality as the following syntax.

```
TRUNCATE TABLE table_name
```

Insert Records

You can insert values for each column while checking its type and constraints.



Column	Domain	Constraints	Value
code	INTEGER	Auto Increment,...	20002
name	VARCHAR(40)	Not Null	zhen han
gender	CHAR(1)		M
nation_code	CHAR(3)		KOR
event	VARCHAR(30)		TEST DATA

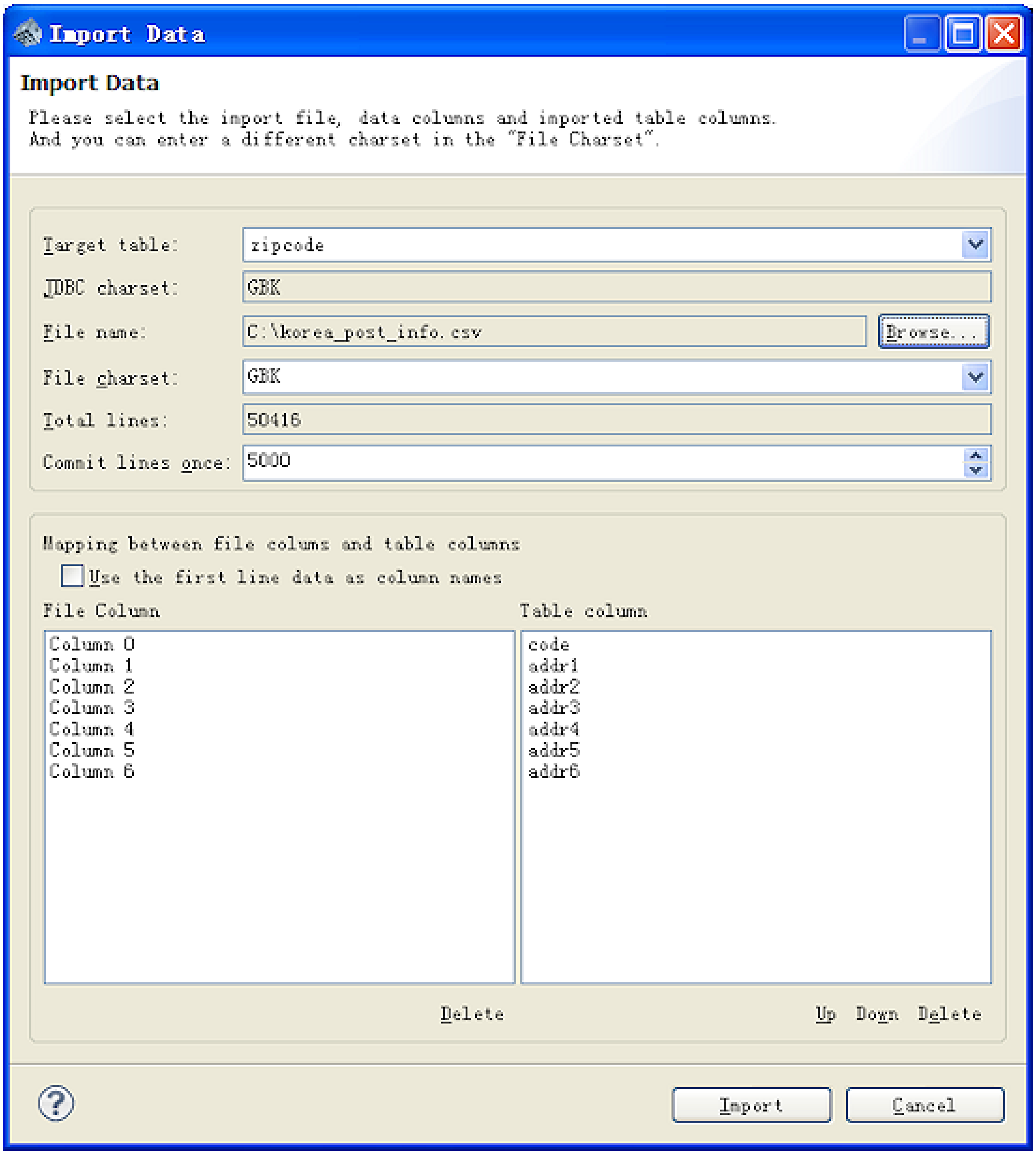
```
insert into "athlete" ("code", "name", "gender", "nation_code", "event") values (20001, 'daniel',
'M', 'KOR', 'TEST DATA');
insert into "athlete" ("code", "name", "gender", "nation_code", "event") values (20002, 'zhen han',
'M', 'KOR', 'TEST DATA');
// 2 records commited.
```

Total 2 records inserted

- When you add more than one instance, you can separate them by adding a line break character between each query and its result.
- You can move the cursor to the next field by pressing the [Enter] key when you enter a value for each field.
- When you click the [Clear] button, the value in the input box and the execution history are initialized.
- The execution history pane cannot be edited.
- For **DATE**, **TIME**, **TIMESTAMP** and **DATETIME** data types, you can enter different data for each type. For example, for a **DATE** type, you can enter data such as **SYSDATE**, **SYS_DATE**, **CURRENT_DATE** and **DATE**'*2009-07-05*'.

Import Data

You can import data from an Excel or CSV file into the database. You can use [File charset] to get data from a file, and use [JDBC charset] to specify charset for data to be stored in the database. To change the value for [JDBC charset], right-click the mouse and then select [Properties].



Import Data

Please select the import file, data columns and imported table columns.
And you can enter a different charset in the "File Charset".

Target table: zipcode

JDBC charset: GBK

File name: C:\korea_post_info.csv

File charset: GBK

Total lines: 50416

Commit lines once: 5000

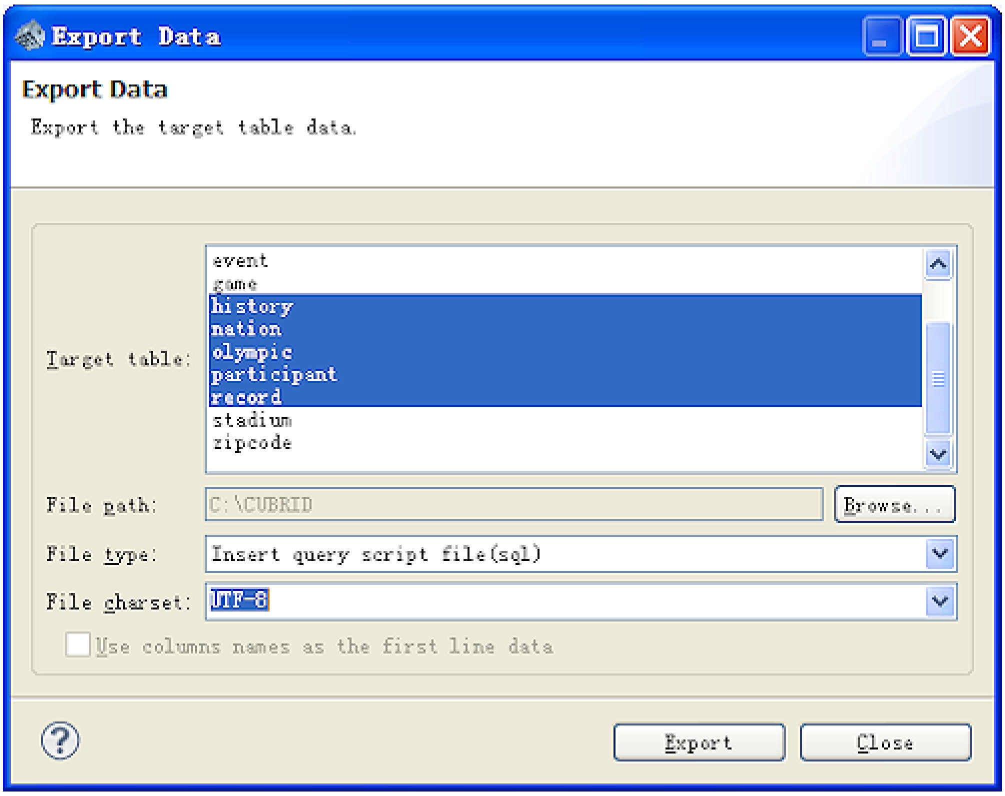
Mapping between file colums and table columns

□ Use the first line data as column names

File Column	Table column
Column 0	code
Column 1	addr1
Column 2	addr2
Column 3	addr3
Column 4	addr4
Column 5	addr5
Column 6	addr6

Export Data

You can export data (usually in one or more tables) in Excel (.xls), CSV, SQL, or CUBRID load (.obs) format. A file name is created with a table name and it is saved in the path specified in [File path]. You can choose [File charset] when exporting data.



Export Data

Export the target table data.

Target table: event, game, history, nation, olympic, participant, record, stadium, zipcode

File path: C:\CUBRID

File type: Insert query script file(sql)

File charset: UTF-8

□ Use columns names as the first line data

Drop Table

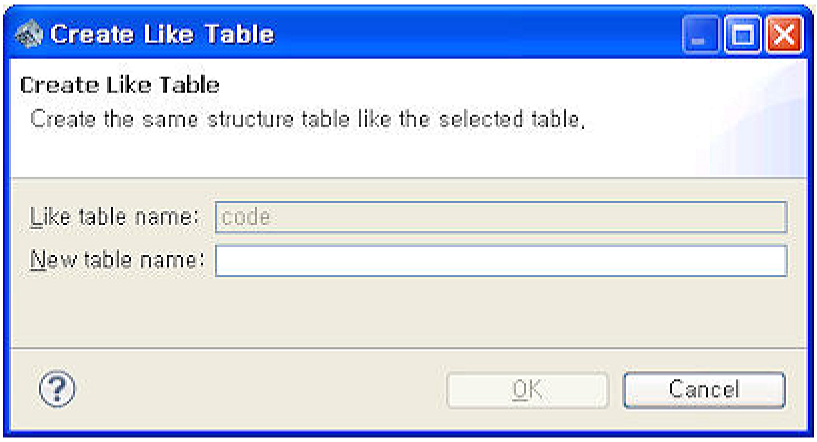
Drops the selected table. This is the same as the **DROP TABLE** statement.

Rename Table

You can change the name of the current table. It is the same as **RENAME TABLE** statement.

Create Like Table

Creates an empty table which has the same schema structure with an existing one. It works the same as **CREATE TABLE LIKE**. For more information, CREATE TABLE LIKE.



Create Like Table

Create the same structure table like the selected table.

Like table name: code

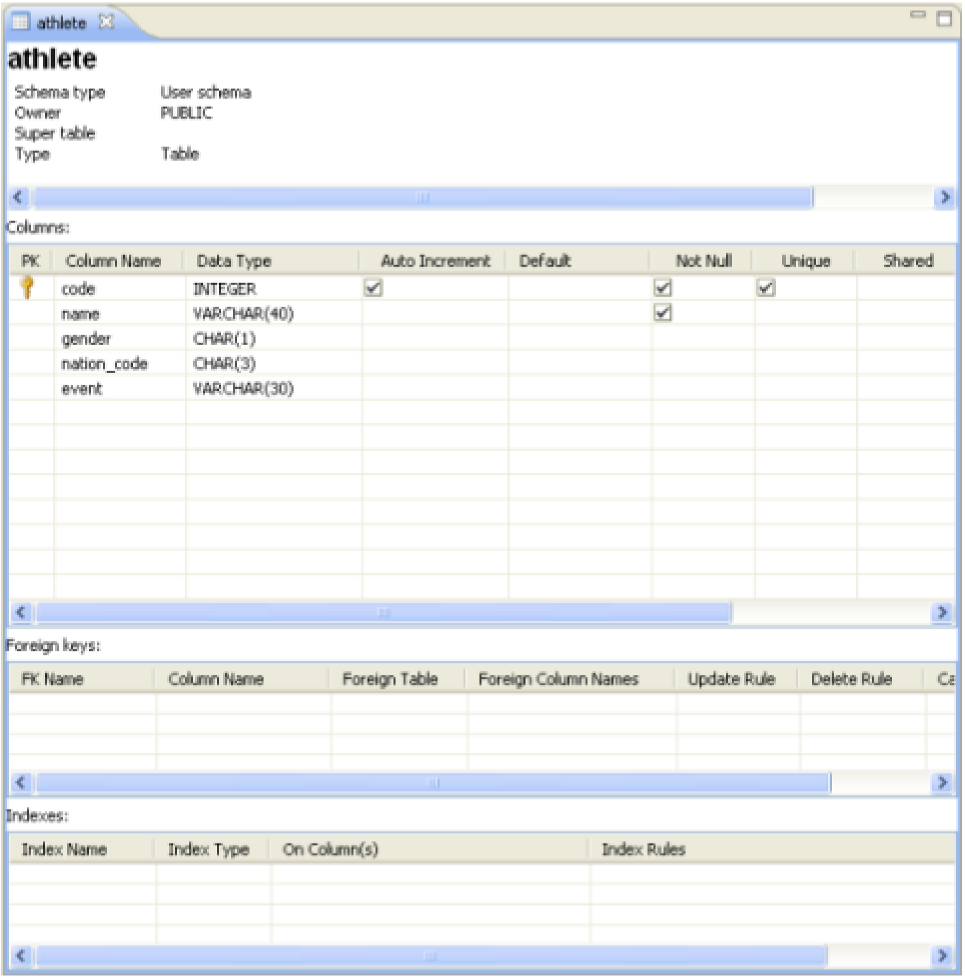
New table name:

Edit Table

You can use all functionalities of [Create Table] in [Edit Table] as well. However, you cannot adjust the order of columns in the table.

Table Information

You can check the schema information of the table by double-clicking it.



athlete

Schema type	User schema
Owner	PUBLIC
Super table	
Type	Table

Columns:

PK	Column Name	Data Type	Auto Increment	Default	Not Null	Unique	Shared
🔑	code	INTEGER	☑		☑	☑	
	name	VARCHAR(40)			☑		
	gender	CHAR(1)					
	nation_code	CHAR(3)					
	event	VARCHAR(30)					

Foreign keys:

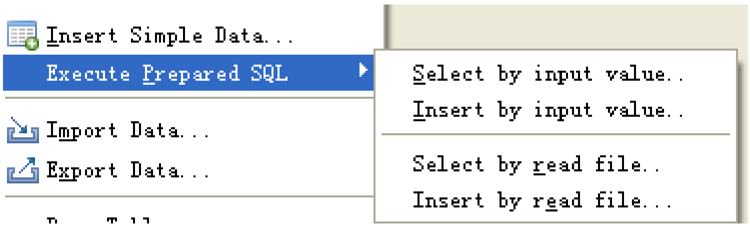
FK Name	Column Name	Foreign Table	Foreign Column Names	Update Rule	Delete Rule	Ca

Indexes:

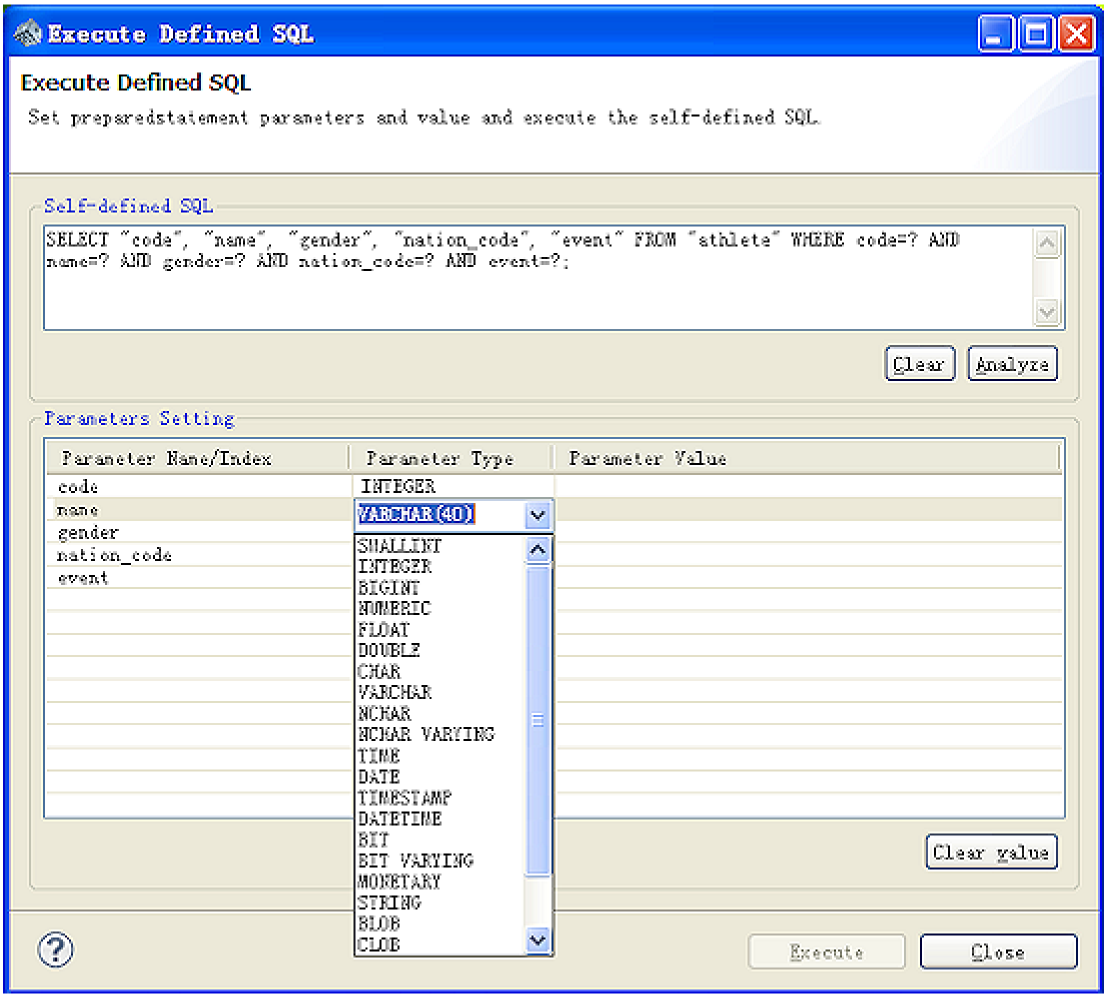
Index Name	Index Type	On Column(s)	Index Rules

Execute Defined SQL

With [Execute Defined SQL] menu, you can execute "prepared statement," pre-saving a specific query statement and specifying a new parameter value for the statement whenever the query is exeucted.



You can execute the **SELECT** statement which meets specified conditions by entering a parameter value of defined "prepared statement" by using [Select by input value].



Execute Defined SQL

Set preparedstatement parameters and value and execute the self-defined SQL.

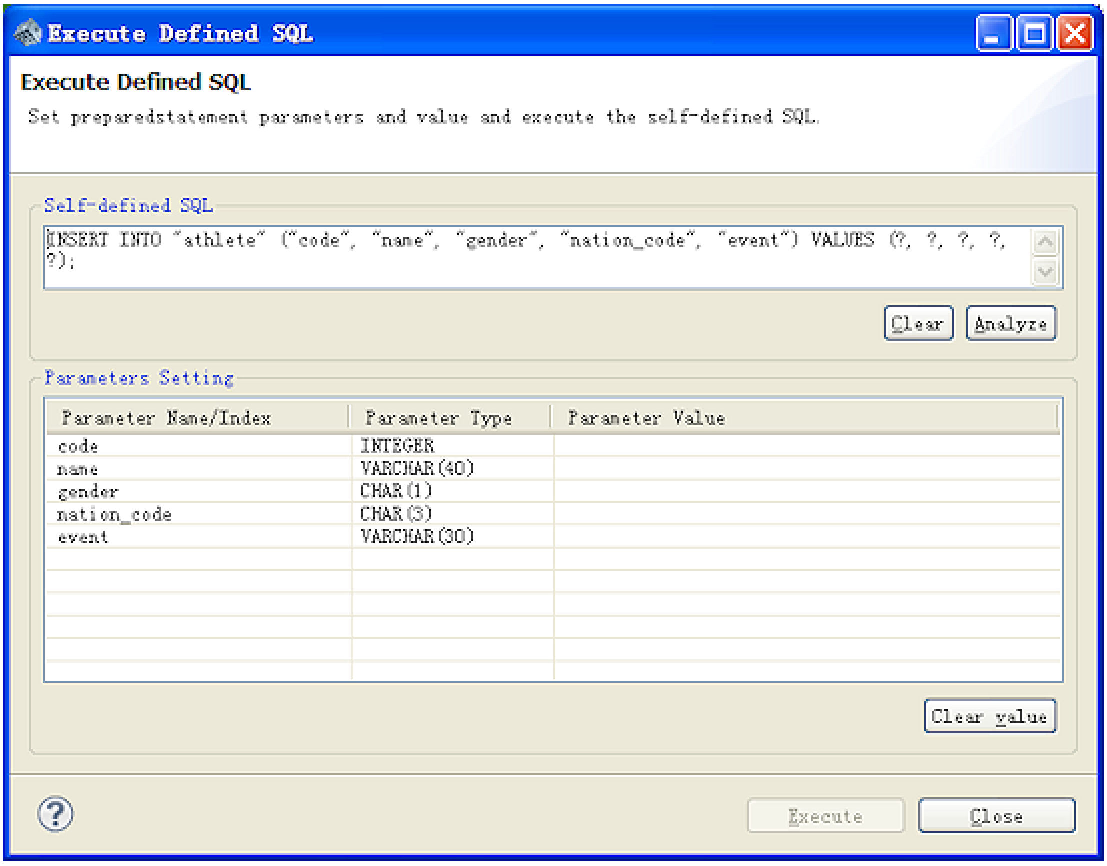
Self-defined SQL

```
SELECT "code", "name", "gender", "nation_code", "event" FROM "athlete" WHERE code=? AND
name=? AND gender=? AND nation_code=? AND event=?;
```

Parameters Setting

Parameter Name/Index	Parameter Type	Parameter Value
code	INTEGER	
name	VARCHAR(40)	
gender		
nation_code		
event		

You can execute the **INSERT** statement by entering a parameter value of defined "prepared statement" by using [Insert by input value].



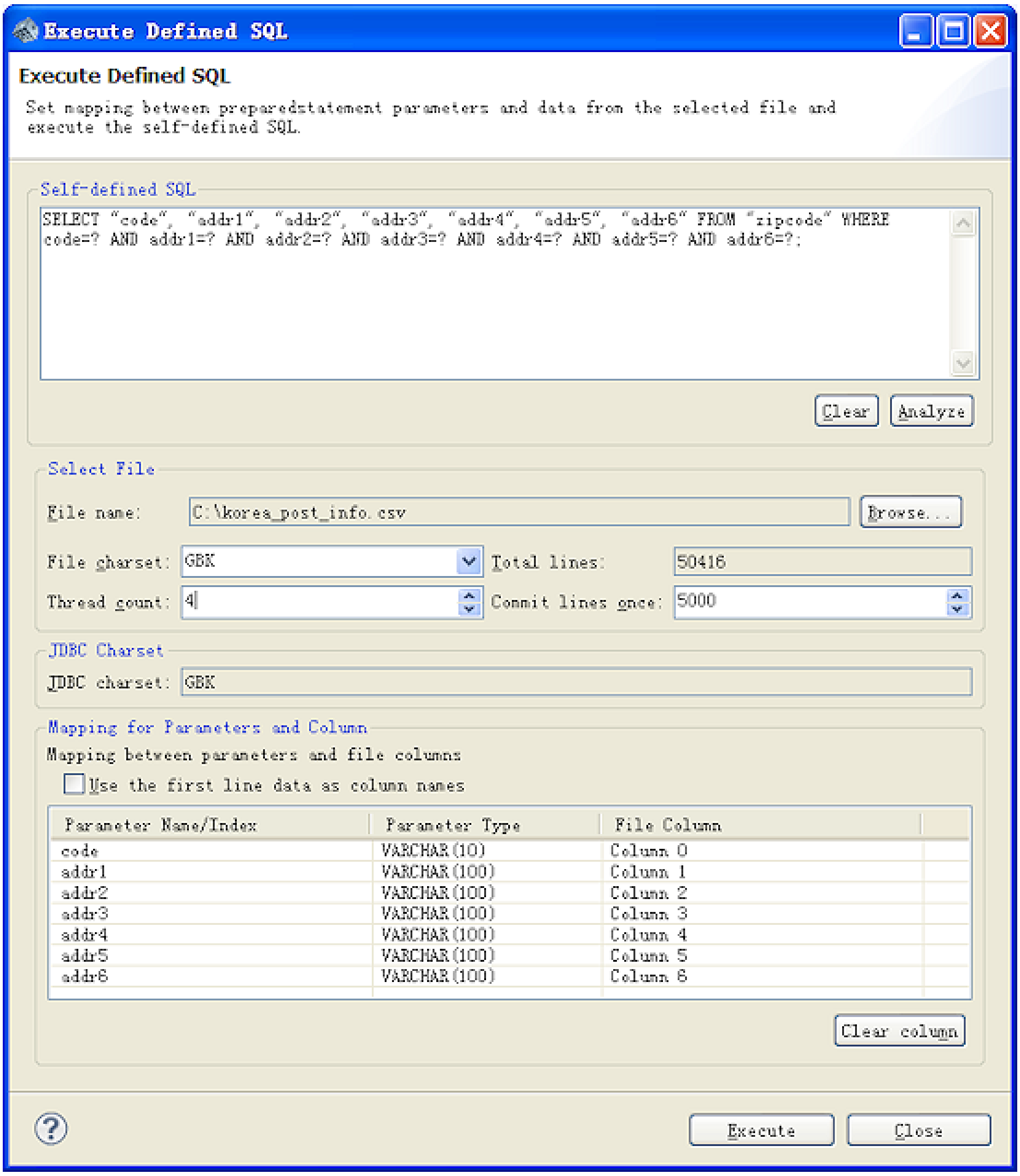
Execute Defined SQL

Set preparedstatement parameters and value and execute the self-defined SQL.

```
INSERT INTO "athlete" ("code", "name", "gender", "nation_code", "event") VALUES (?, ?, ?, ?, ?);
```

Parameter Name/Index	Parameter Type	Parameter Value
code	INTEGER	
name	VARCHAR(40)	
gender	CHAR(1)	
nation_code	CHAR(3)	
event	VARCHAR(30)	

You can use [Select by read file] when repeating the execution of **SELECT** statement by inputting several parameter values in the defined "prepared statement." The data is saved in Excel (.xls) or CSV format. You can configure a file charset where parameter values are saved, the number of concurrent threads to be executed, and commit cycle. To change the value for [JDBC charset], right-click the mouse in Explorer and then select [Properties].



Execute Defined SQL

Set mapping between preparedstatement parameters and data from the selected file and execute the self-defined SQL.

Self-defined SQL

```
SELECT "code", "addr1", "addr2", "addr3", "addr4", "addr5", "addr6" FROM "zipcode" WHERE
code=? AND addr1=? AND addr2=? AND addr3=? AND addr4=? AND addr5=? AND addr6=?;
```

Select File

File name: C:\korea_post_info.csv

File charset: GBK

Total lines: 50416

Thread count: 4

Commit lines once: 5000

JDBC Charset

JDBC charset: GBK

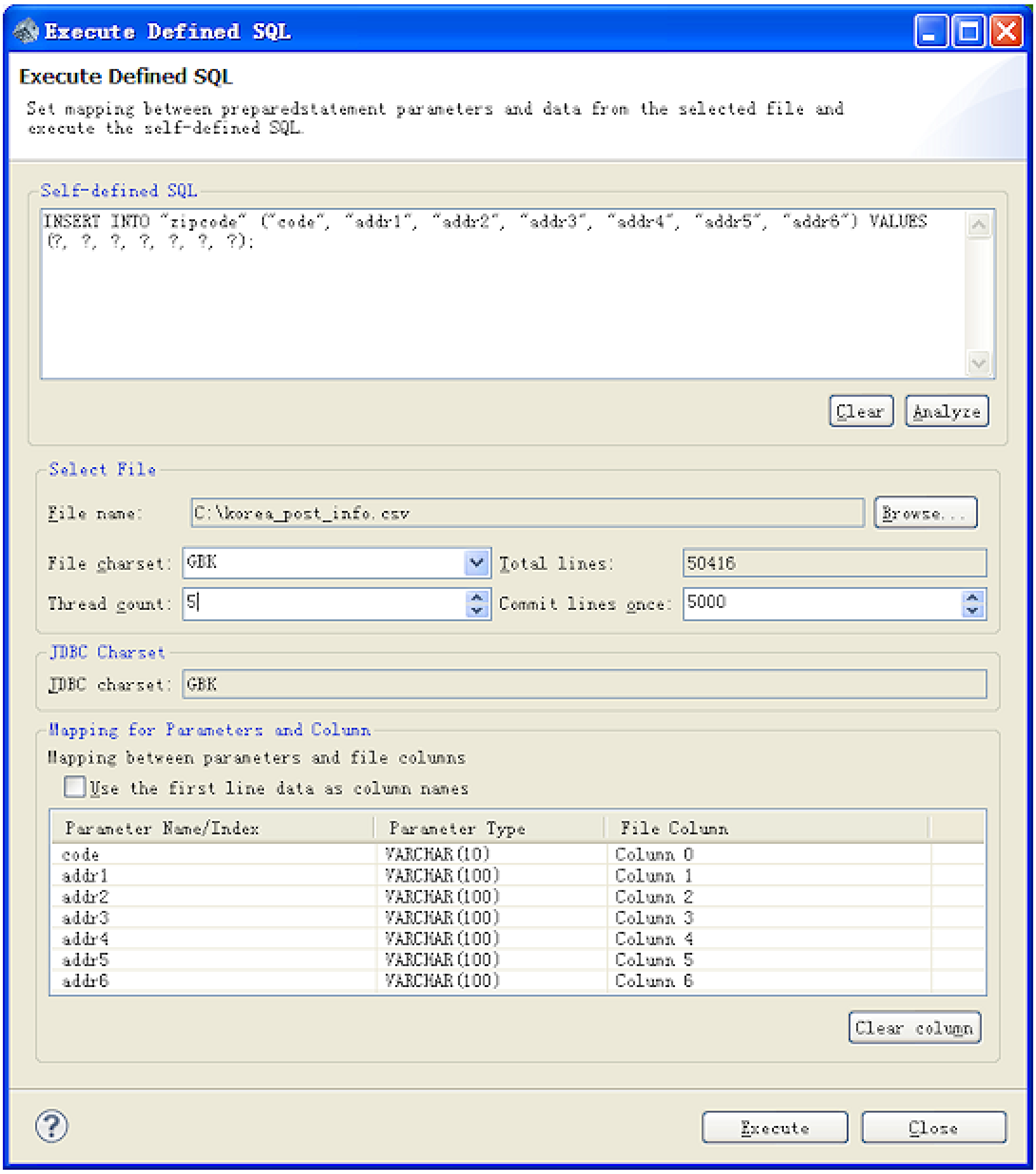
Mapping for Parameters and Column

Mapping between parameters and file columns

□ Use the first line data as column names

Parameter Name/Index	Parameter Type	File Column
code	VARCHAR(10)	Column 0
addr1	VARCHAR(100)	Column 1
addr2	VARCHAR(100)	Column 2
addr3	VARCHAR(100)	Column 3
addr4	VARCHAR(100)	Column 4
addr5	VARCHAR(100)	Column 5
addr6	VARCHAR(100)	Column 6

You can use [Insert by read file] when repeating the execution of **INSERT** statement by inputting several parameter values in the defined "prepared statement." The data is saved in Excel (.xls) or CSV format. You can configure a file charset where parameter values are saved, the number of concurrent threads to be executed, and commit cycle. To change the value for [JDBC charset], right-click the mouse in Explorer and then select [Properties].



Execute Defined SQL

Execute Defined SQL

Set mapping between preparedstatement parameters and data from the selected file and execute the self-defined SQL.

Self-defined SQL

```
INSERT INTO "zipcode" ("code", "addr1", "addr2", "addr3", "addr4", "addr5", "addr6") VALUES
(?, ?, ?, ?, ?, ?, ?);
```

Select File

File name: C:\korea_post_info.csv

File charset: GBK

Total lines: 50416

Thread count: 5

Commit lines once: 5000

JDBC Charset

JDBC charset: GBK

Mapping for Parameters and Column

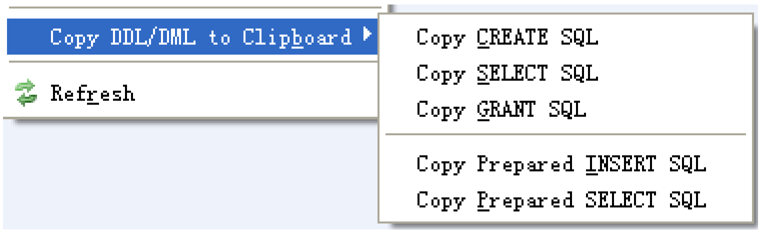
Mapping between parameters and file columns

□ Use the first line data as column names

Parameter Name/Index	Parameter Type	File Column
code	VARCHAR(10)	Column 0
addr1	VARCHAR(100)	Column 1
addr2	VARCHAR(100)	Column 2
addr3	VARCHAR(100)	Column 3
addr4	VARCHAR(100)	Column 4
addr5	VARCHAR(100)	Column 5
addr6	VARCHAR(100)	Column 6

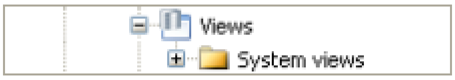
Copy as DDL/DML Clipboard

A function that allows you to view schema information for more than one table by copying DDL and DML in a selected table as a clipboard and then using it with another editor. **INSERT** and **SELECT** statements can be copied that are written with **CREATE** statement, **SELECT** statement, and **GRANT** statement, and prepared statement defined in the given table.



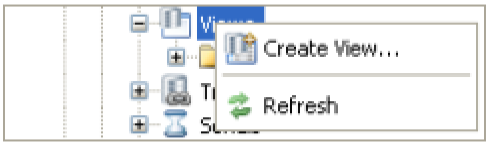
View

When you login to the database, you can see accessible views and system views in the navigation tree.

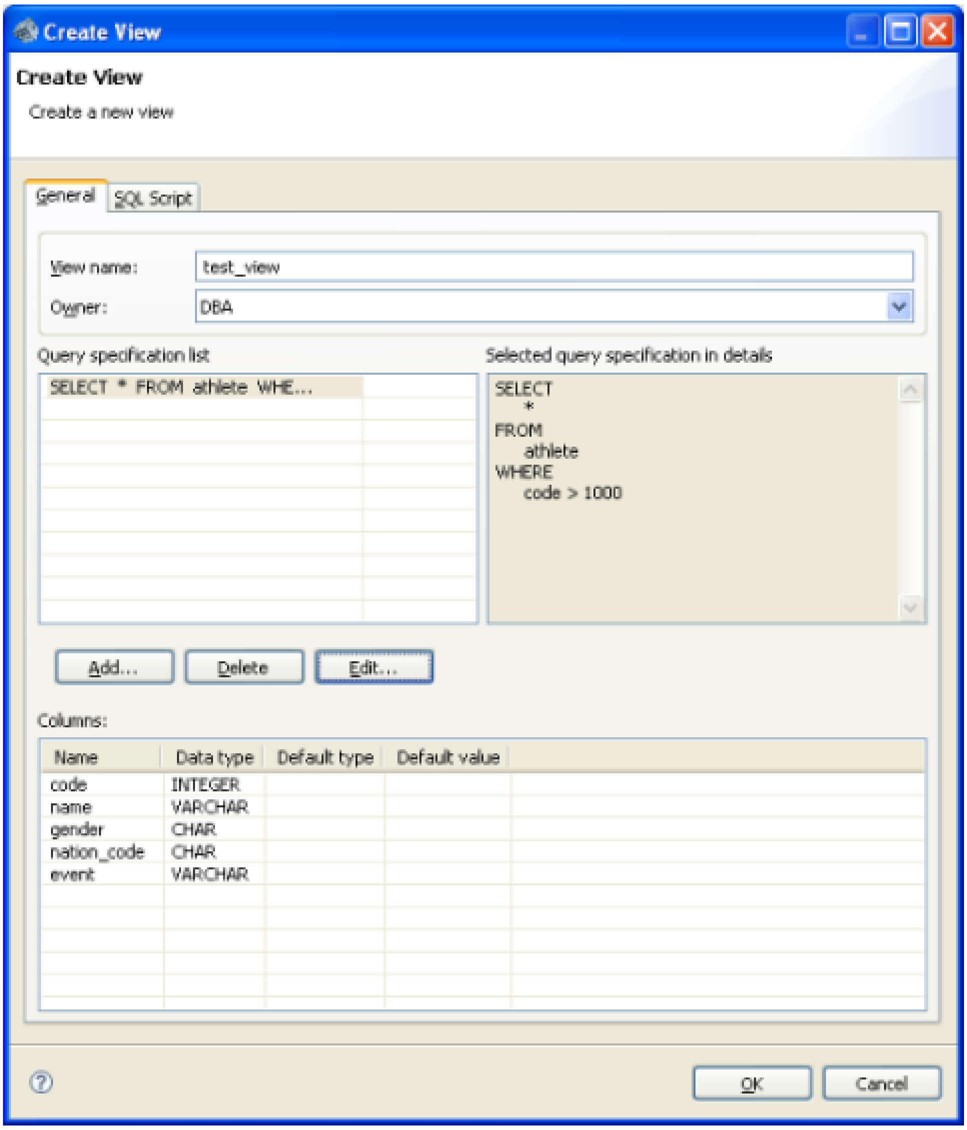


Create View

Right-click [Views] in the navigation tree and then select [Create View].



Specify the view name and the owner, click the [Add] button, and then enter the query for the view to be created. You can check the contents entered in the Create View wizard from the SQL statement in the [SQL Script] tab.



View name: test_view

Owner: DBA

Query specification list

```
SELECT * FROM athlete WHE...
```

Selected query specification in details

```
SELECT
    *
FROM
    athlete
WHERE
    code > 1000
```

Columns:

Name	Data type	Default type	Default value
code	INTEGER		
name	VARCHAR		
gender	CHAR		
nation_code	CHAR		
event	VARCHAR		

Select All

Right-click a view in the navigation tree and click [Select All]. Or you can drag and drop the view into the editor results pane of the Query editor when it is open. Then, a new Query editor opens and retrieves the entire data.

Select Count

Retrieves a total data count of the table and performs the same functionality as the following syntax.

```
SELECT COUNT(*) FROM view
```

Export Data

You can export all data of the view to an Excel, CSV, SQL or CUBRID load format.

Drop View

Drops the selected view. This is same as the **DROP VIEW** statement.

Rename View

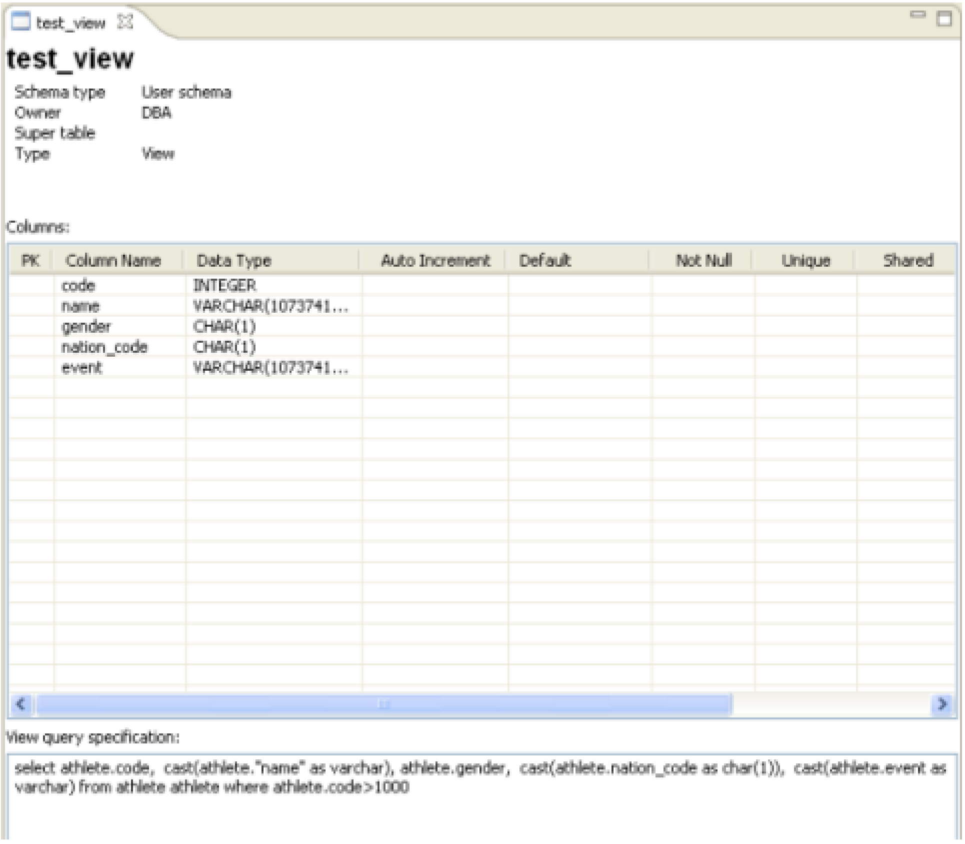
You can change the name of the current view. This is same as the **RENAME VIEW** statement.

Edit View

You can use all functionalities of [Create View] in [Edit View] as well.

View Information

You can check the schema information of the view by double-clicking it.



test_view

Schema type	User schema
Owner	DBA
Super table	
Type	View

Columns:

PK	Column Name	Data Type	Auto Increment	Default	Not Null	Unique	Shared
	code	INTEGER					
	name	VARCHAR(1073741...					
	gender	CHAR(1)					
	nation_code	CHAR(1)					
	event	VARCHAR(1073741...					

View query specification:

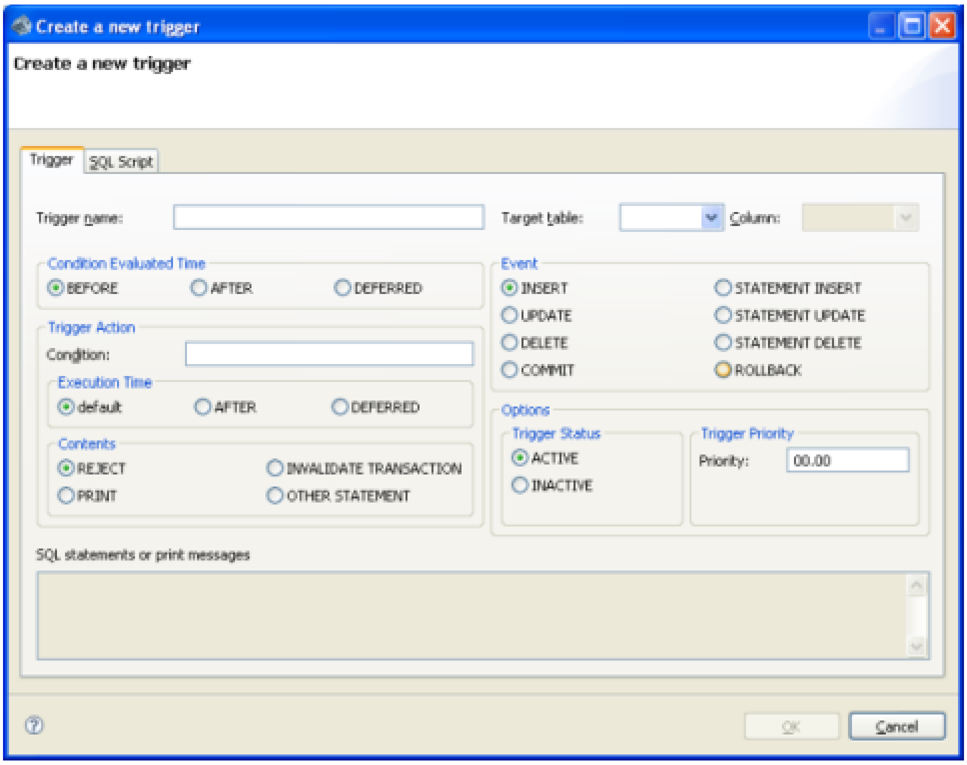
```
select athlete.code,  cast(athlete."name" as varchar), athlete.gender,  cast(athlete.nation_code as char(1)),  cast(athlete.event as varchar) from athlete athlete where athlete.code>1000
```

Trigger

When you login to the database, you can view accessible triggers in the navigation tree.

Create Trigger

Right-click [Triggers] in the navigation tree and then select [Create Trigger].



Create a new trigger

Create a new trigger

Trigger | SQL Script

Trigger name:

Target table:

Column:

Condition Evaluated Time: ● BEFORE ○ AFTER ○ DEFERRED

Event: ● INSERT ○ UPDATE ○ DELETE ○ COMMIT ○ STATEMENT INSERT ○ STATEMENT UPDATE ○ STATEMENT DELETE ○ ROLLBACK

Trigger Action

Condition:

Execution Time: ● default ○ AFTER ○ DEFERRED

Contents: ● REJECT ○ PRINT ○ INVALIDATE TRANSACTION ○ OTHER STATEMENT

Options

Trigger Status: ● ACTIVE ○ INACTIVE

Trigger Priority

Priority: 00.00

SQL statements or print messages

OK | Cancel

- **Trigger name** : Enter the name of the trigger to be added.
- **Condition Evaluated Time** : Select the point of time when the condition of the trigger is to be evaluated. You can select from **BEFORE**, **AFTER** and **DEFERRED**.
- **Event** : Select the type of the event to be occurred. Event types are **INSERT**, **DELETE**, **UPDATE**, **STATEMENT INSERT**, **STATEMENT DELETE**, **STATEMENT UPDATE**, **COMMIT** and **ROLLBACK**.
- **Target table/Column** : Enter the target table and column information.
- **Condition** : Enter the condition for the trigger action.
- **Execution Time** : Specify the point of time when the trigger is to be executed. If **default** is selected, the trigger fires based on the point of time when its condition is validated.
- **Contents** : Select the type of the trigger action. If the type of the trigger action is **PRINT**, **OTHER** or **STATEMENT**, you can enter additional information in the [SQL statements or print messages] below.
- **Trigger Status** : You can specify whether to activate or deactivate the trigger to be added.
- **Trigger Priority** : You can set the priority of the trigger. The priority value is a **FLOAT** and can be between 00.00 and 9999.99.

Drop Trigger

Right-click the trigger in the navigation tree and then select [Drop Trigger].

Edit Trigger

Right-click the trigger in the navigation tree and then select [Edit Trigger]. You can simply edit a value in the [Priority] field.

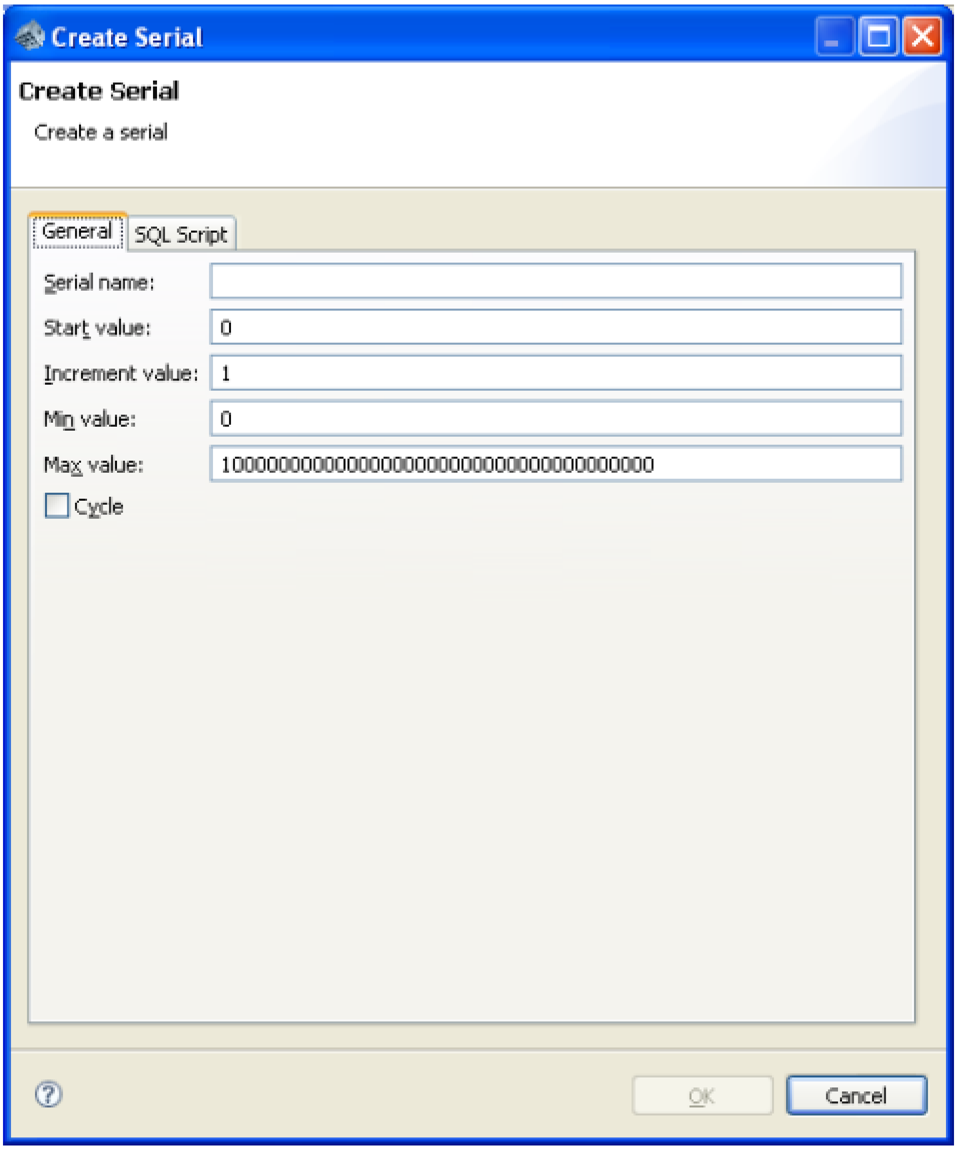
Serial

When you login to the database, you can view accessible serials in the navigation tree.

Create Serial

Right-click [Serials] in the navigation tree and then select [Create Serial].

The name of the serial must be unique in the database.



Create Serial

Create Serial

Create a serial

General | SQL Script

Serial name:

Start value: 0

Increment value: 1

Min value: 0

Max value: 10000000000000000000000000000000000000

□ Cycle

Drop Serial

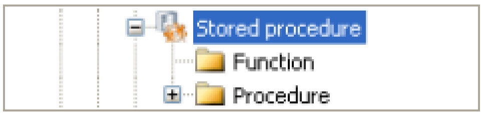
Right-click the serial in the navigation tree and then select [Drop Serial].

Edit Serial

Right-click the serial in the navigation tree and then select [Edit Serial].

Stored Procedure

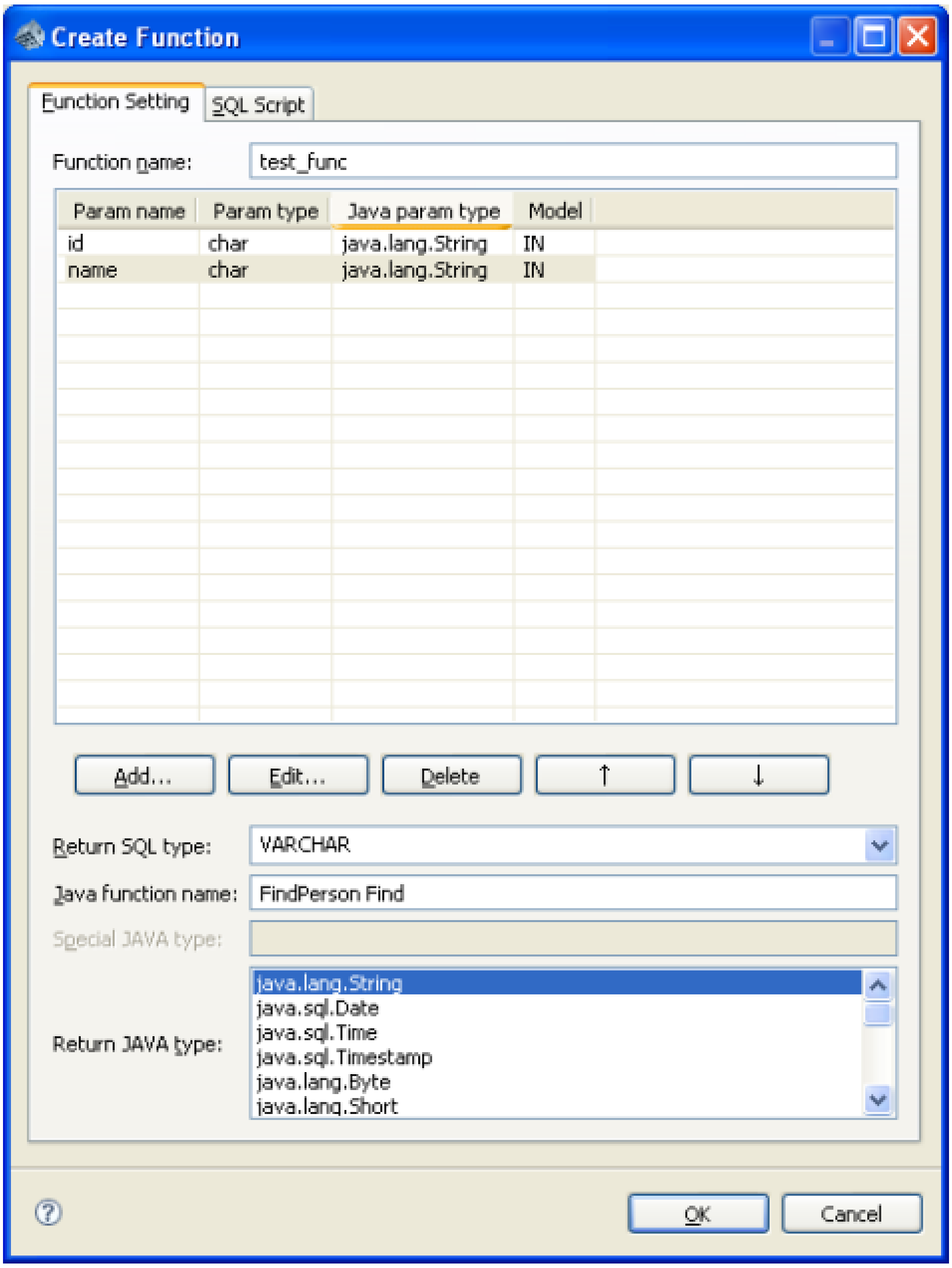
When you login to the database, you can view accessible stored procedures.



Create Function

A feature that registers a function written in Java in the database server with the **loadjava** command and adds a database function in order to use the given Java function.

Right-click [Stored procedure] in the navigation tree and then select [Create Function].



Function name: test_func

Param name	Param type	Java param type	Model
id	char	java.lang.String	IN
name	char	java.lang.String	IN

Return SQL type: VARCHAR

Java function name: FindPerson Find

Special JAVA type:

Return JAVA type:
- java.lang.String
- java.sql.Date
- java.sql.Time
- java.sql.Timestamp
- java.lang.Byte
- java.lang.Short

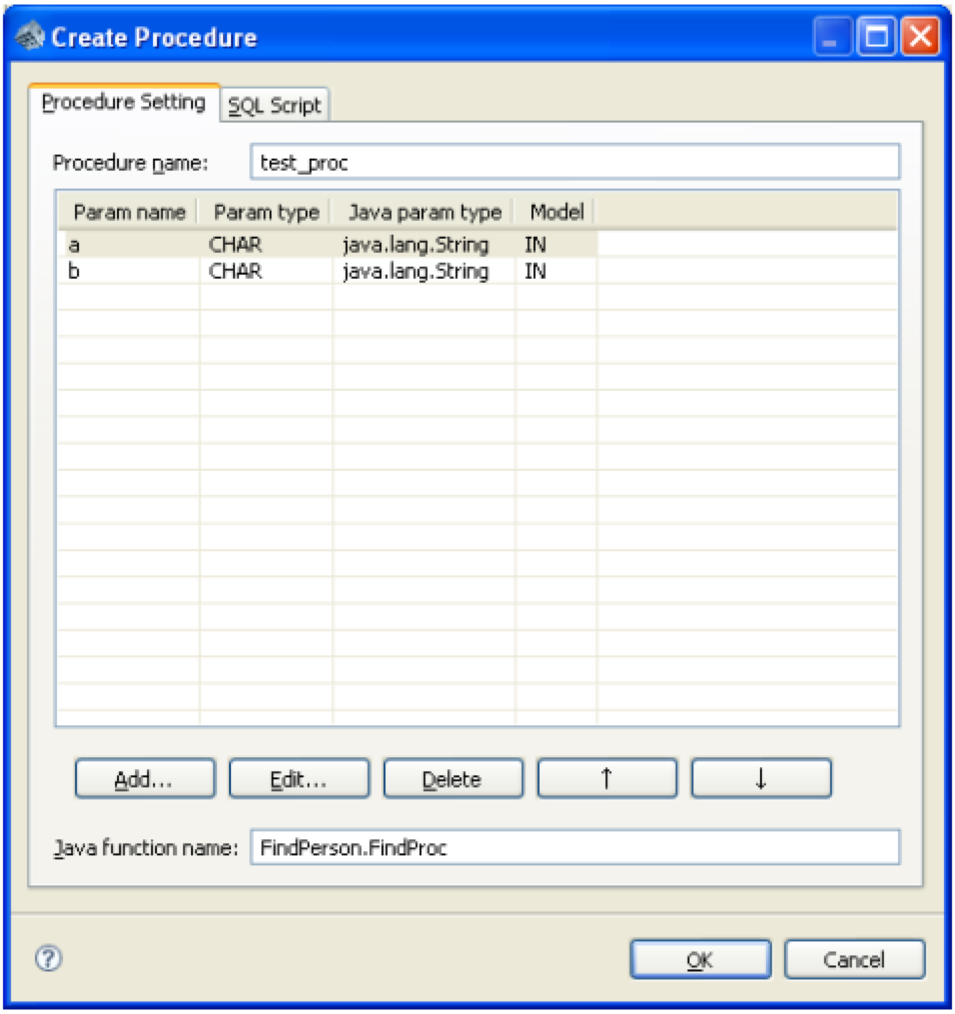
Only Java types compatible with the selected SQL type are displayed so that the user does not make a mistake with type mapping.

For more information, see Overview.

Create Procedure

A feature that registers a procedure written in Java in the database server with the **loadjava** command and creates a database procedure in order to use the given Java procedure.

Right-click [Stored procedure] in the navigation tree and then select [Create Procedure].



Procedure name: test_proc

Param name	Param type	Java param type	Model
a	CHAR	java.lang.String	IN
b	CHAR	java.lang.String	IN

Java function name: FindPerson.FindProc

Only Java types compatible with the selected SQL type are displayed so that the user does not make a mistake with type mapping.

For more information, see Overview.

The following is a statement created by using the Create Procedure wizard above.

```
CREATE PROCEDURE "test_proc"("a" CHAR,"b" CHAR)
AS LANGUAGE JAVA
NAME 'FindPerson.FindProc(java.lang.String,java.lang.Integer)'
```

Drop Function/Procedure

Right-click a certain function or a procedure and then select [Drop Function] or [Drop Procedure].

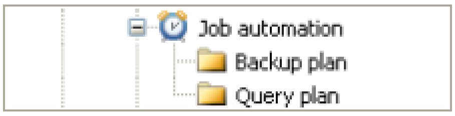
Edit Function/Procedure

Right-click a certain function or a procedure and then select [Edit Function] or [Edit Procedure]. The feature works the same as with Create Function/Procedure.

Automation

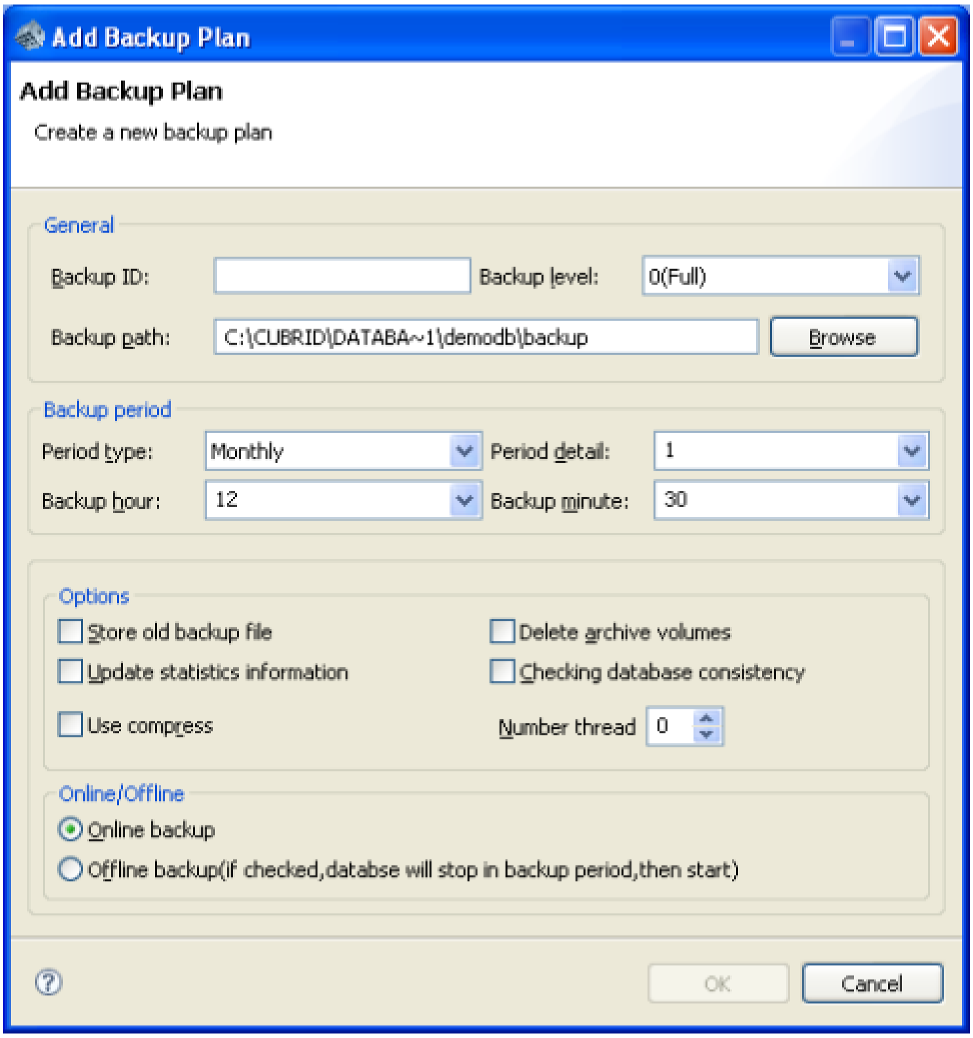
Backup Automation

When you login to the database, you can view [Backup plan] under [Job automation] in the navigation tree. The [Add Backup Plan] and [Auto Backup Logs] menus will appear when you right-click [Backup plan].



If you want to execute a backup periodically with the CUBRID Manager, configure the values of [Add Backup Plan]. The **DBA** can configure the backup automation while the Manager server is running. It is not affected by whether the database is running or not. That is, backup automation is executed only when the Manager server is running.

Add Backup Plan



Add Backup Plan

Add Backup Plan

Create a new backup plan

General

Backup ID:

Backup level: 0(Full)

Backup path: C:\CUBRID\DATABA~1\demodb\backup

Browse

Backup period

Period type: Monthly

Period detail: 1

Backup hour: 12

Backup minute: 30

Options

□ Store old backup file

□ Delete archive volumes

□ Update statistics information

□ Checking database consistency

□ Use compress

Number thread 0

Online/Offline

● Online backup

○ Offline backup(if checked,databse will stop in backup period,then start)

OK

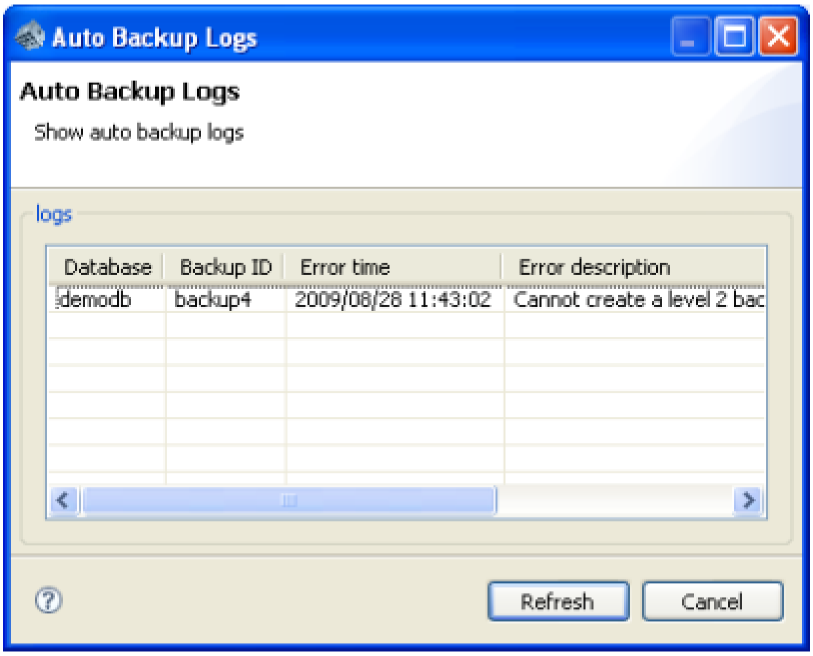
Cancel

- **Backup ID** : Enter the name of the backup job. The backup ID must be unique in the database because multiple backup plans may exist in the same database.
- **Backup level** : You can choose from 0, 1, and 2. Level0 is a full backup. Level1 is the first incremental backup that backs up changes made only after Level0 backup. Level2 is the second incremental backup that backs up changes made only after the Level1 backup.
- **Backup path** : Specifies the directory of the backup volume.
- **Period type** : You can select a backup period from options of Monthly, Weekly, Daily and A specific day.

- **Period detail** : You can set details for the period type you selected.
- **Backup hour** : Enter the time when the automated backup is to be executed. You must enter the time in hour and minute.
- **Options** : You can specify an option for backup. For more information, see Database Backup.
- **Store old backup file** : This option saves the current original backup volume file of the database in **database_directory/backupold** directory.
- **Delete archive volumes** : An option that deletes archive log volumes after backup. When this option is selected while the database is set to master server in the replication environment, volumes do not affect the replication will be deleted automatically.
- **Update statistics information** : Updates statistics information after backup.
- **Checking database consistency** : Checks database consistency during backup.
- **Use compress** : Uses compression during backup.
- **Number thread** : Specifies the number of threads to be used concurrently during backup. It is recommended to configure the maximum number of threads to be the same as the number of CPUs. The default value is **0**. If it is set to the default value, the number of threads is determined automatically by the system.
- **Online backup** : Automated backup is executed only when the database is running. If the database stopped, only error logs are recorded without backing up the database.
- **Offline backup** : Automated backup is executed only when the database stopped running. Forces the database to shut down if it is currently running, performs an automated backup, and then restarts the database.

Auto Backup Log

Provides the error logs created during backup automation.



Auto Backup Logs

Auto Backup Logs

Show auto backup logs

logs

Database	Backup ID	Error time	Error description
demodb	backup4	2009/08/28 11:43:02	Cannot create a level 2 bac

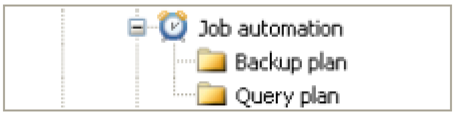
Edit/Delete Backup Plan

Edits backup automation jobs in the same way for adding backup automation, or deletes unnecessary backup automation jobs.

Right-click a desired backup automation in the search tree and then select [Edit Backup Plan], or [Delete Backup Plan].

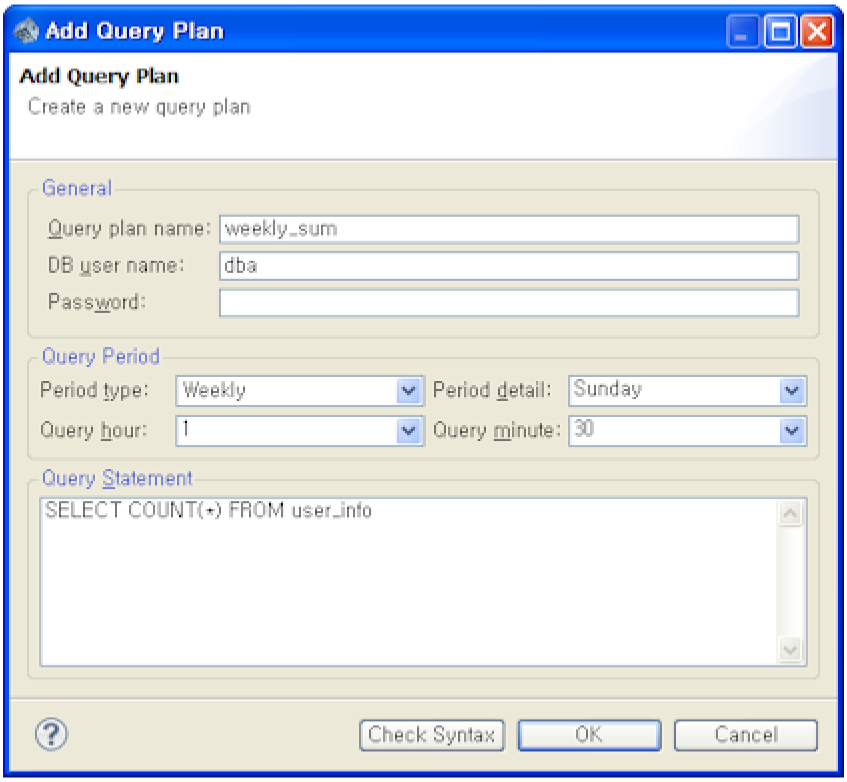
Query Automation

When you login to the database, you can view [Query plan] under [Job automation] in the navigation tree. The [Add Query Plan] and [Auto Query Logs] menus will appear when you right-click [Query plan].



If you want to execute a backup periodically with the CUBRID Manager, configure the values of [Add Query Plan]. The **DBA** can configure the backup automation while the Manager server is running. It is not affected by whether the database is running or not. That is, backup automation is executed only when the Manager server is running.

Add Query Plan



Query plan name: weekly_sum

DB user name: dba

Password:

Period type: Weekly

Period detail: Sunday

Query hour: 1

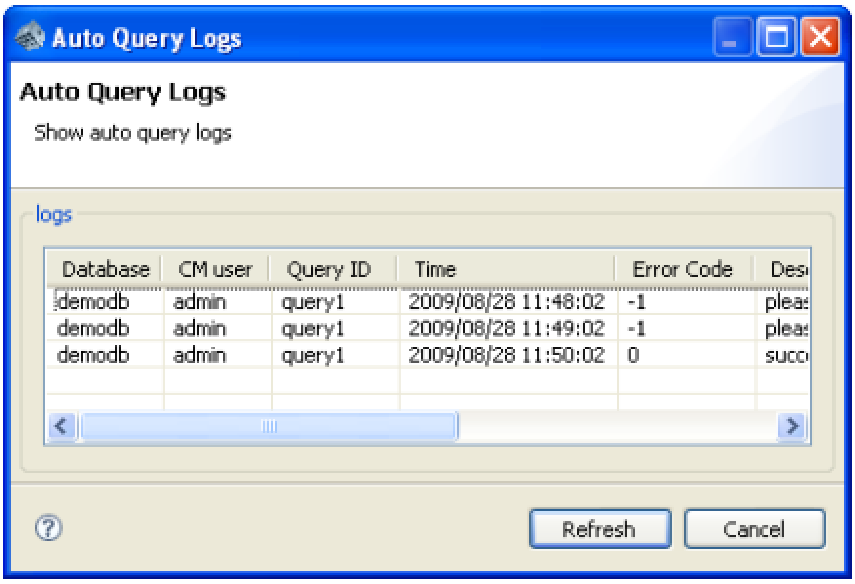
Query minute: 30

```
SELECT COUNT(*) FROM user_info
```

- **Query Plan ID** : Enter the name of the query job. The query plan ID must be unique in the database because multiple query plans may exist in the same database.
- **User name/password** : Enter the user name, which will execute the registered query automatically, and its password. If you enter wrong information, the query is not automatically executed. Therefore, you must change the user information in the [Edit Query Plan] in case of password change.
- **Period type** : You can select a query automation period from options of Monthly, Weekly, Daily and A specific day.
- **Period detail** : You can set details for the period type you selected.
- **Query hour** : Enter the time when the query is to be executed automatically. You must enter the time in hour and minute.
- **Query Statement** : Enter the query statement to be executed automatically. Note that the registered query is executed automatically at the specified time, but the execution results are not recorded.
- **Check Query** : Check query errors before registering the auto-executed query statement. It determines whether it is an error or not by creating query plan it does not actually execute the query. It is working like executing **optimization level 514** in the CSQL Interpreter.

Auto Query Log

You can view a log about queries that are automatically executed in the [Auto Query Logs]. Information such as a database, user ID, query ID, query hour, and error code (success : 0, failure : -1) is recorded.



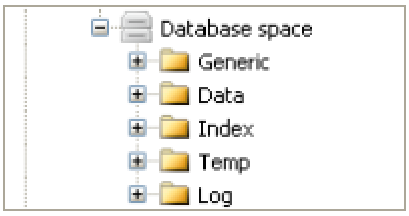
Database	CM user	Query ID	Time	Error Code	Des
demodb	admin	query1	2009/08/28 11:48:02	-1	pleas
demodb	admin	query1	2009/08/28 11:49:02	-1	pleas
demodb	admin	query1	2009/08/28 11:50:02	0	succ

Edit/Delete Query Plan

If you register a query job whose ID is every_mon_delete, the < every_mon_delete> item is added under the [Query plan]. You can view the [Edit Query Plan] and [Delete Query Plan] menus by right-clicking the item. With these menus, you can edit or delete the query plan.

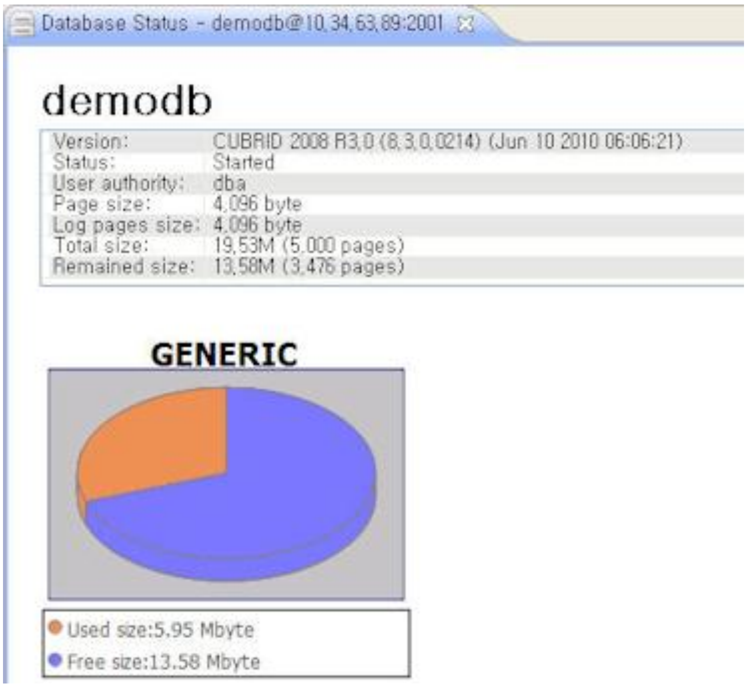
Database Space

When you login to the database, you can view [Database space] in the navigation tree. With this option, you can select shortcut menus, such as View Database, Set Auto Add Volume, or Add Volume.



View Database

If you double-click the name of a database in the navigation tree or right-click [Database space] then select [View Database], you can check space information of the database.



Database Status - demodb@10.34.63.89:2001

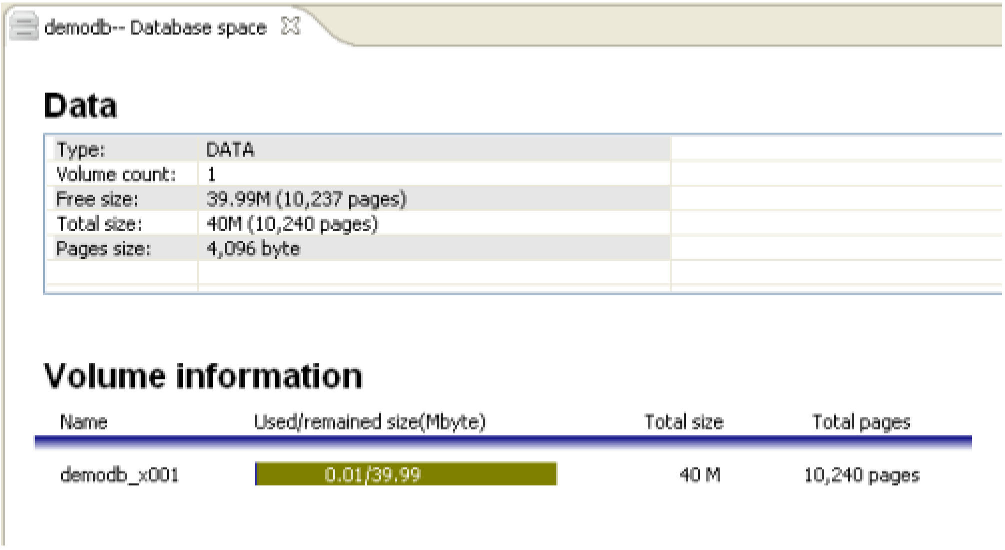
demodb

Version:	CUBRID 2008 R3.0 (8.3.0.0214) (Jun 10 2010 06:06:21)
Status:	Started
User authority:	dba
Page size:	4,096 byte
Log pages size:	4,096 byte
Total size:	19.53M (5,000 pages)
Remained size:	13.58M (3,476 pages)

● Used size:5.95 Mbyte

● Free size:13.58 Mbyte

If you double-click a sub-node of [Database space] in the navigation tree, the volume information appears.



demodb-- Database space

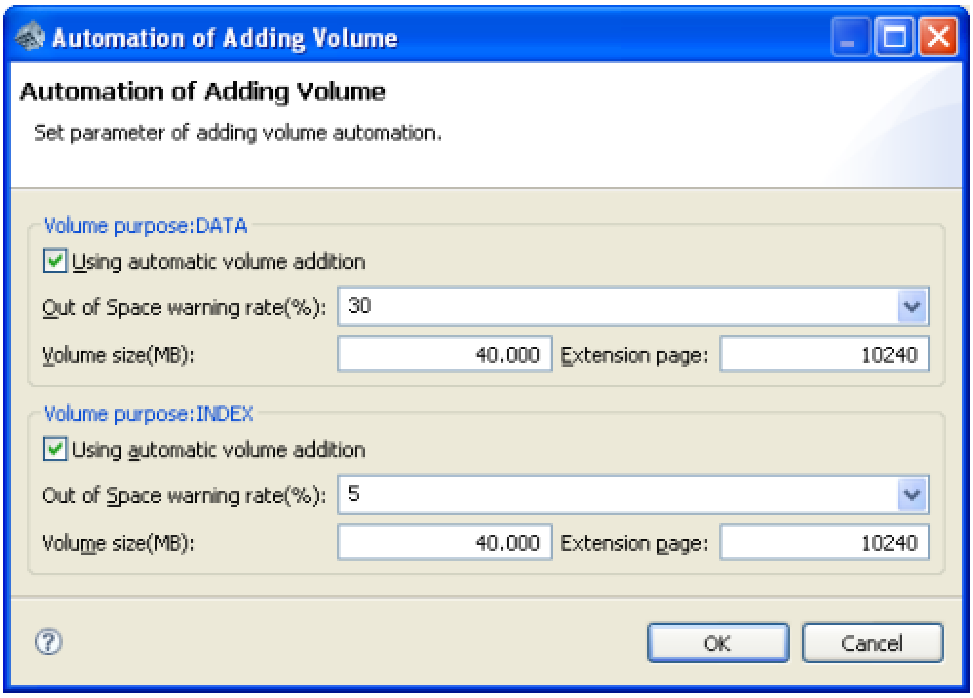
Data

Type:	DATA
Volume count:	1
Free size:	39.99M (10,237 pages)
Total size:	40M (10,240 pages)
Pages size:	4,096 byte

Volume information

Name	Used/remained size(Mbyte)	Total size	Total pages
demodb_x001	0.01/39.99	40 M	10,240 pages

Set Auto Add Volume



Automation of Adding Volume

Automation of Adding Volume

Set parameter of adding volume automation.

Volume purpose:DATA

☑ Using automatic volume addition

Out of Space warning rate(%): 30

Volume size(MB): 40.000 Extension page: 10240

Volume purpose:INDEX

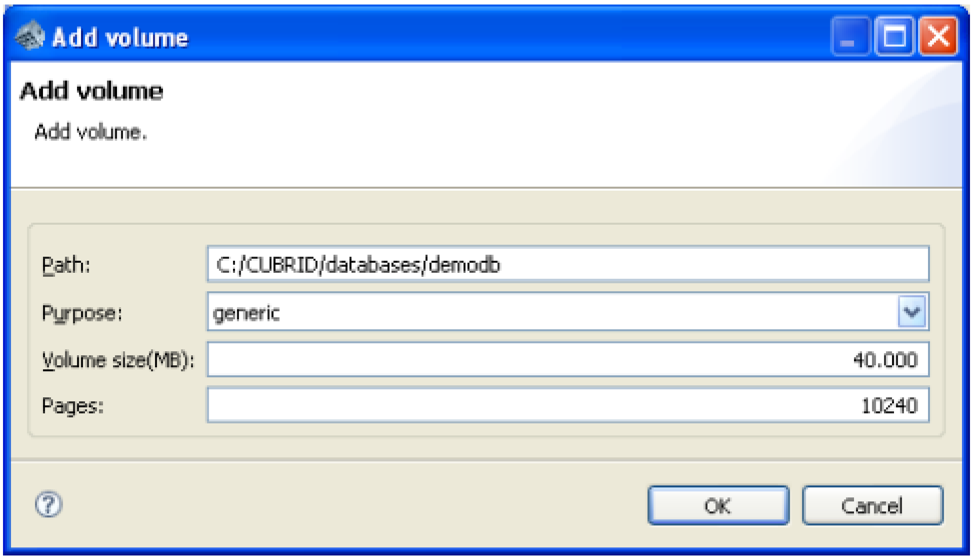
☑ Using automatic volume addition

Out of Space warning rate(%): 5

Volume size(MB): 40.000 Extension page: 10240

- **Volume purpose** : The automatic addition functionality can be used for a data or an index volume.
- **Using automatic volume addition** : Use the automatic volume addition functionality for the selected volume type.
- **Out of Space warning rate** : A volume is added automatically when the remaining volume equals to the value set by [Out of Space warning rate]. For example, if this value is set to 5 % and the remaining space of the volume is 5 %, a data volume is added automatically. The minimum value is 5, and the maximum is 30.
- **Volume size** : Enter the size of the volume to be added automatically.

Add Volume



Add volume

Add volume

Add volume.

Path: C:/CUBRID/databases/demodb

Purpose: generic

Volume size(MB): 40.000

Pages: 10240

- **Path** : Enter the directory where the added volume is to be saved. The default value is the directory where the database volume is created.
- **Purpose** : Specify the type of the volume to be added. You can select from data, generic, index and temp.
- **Volume size** : Enter the size of the volume to be added. Its unit is MB.

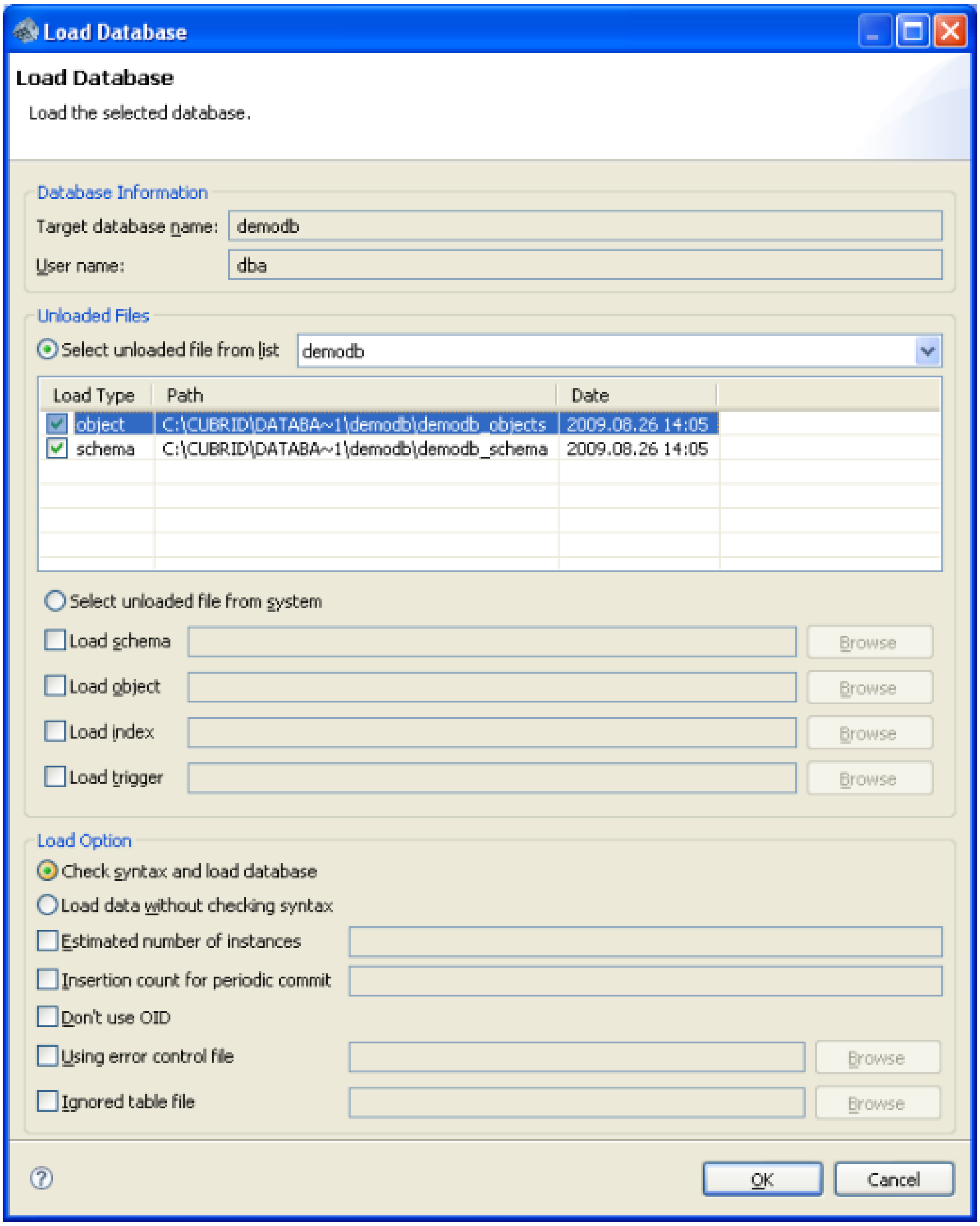
- **Pages** : When the size is entered in Volume size, the number of pages for the volume is automatically calculated and displayed.

Load Database

To load the unloaded data into the currently selected database, perform one of the following after login to the database.

- Click [Load Database] from the toolbar.
- Right-click the database and then select [Load Database].
- Select [Action] > [Load Database] on the menu.

The load database operation can be performed only when the database server is not running; The [Restore Database] menu is deactivated while the database is running.



Load Database

Load Database

Load the selected database.

Database Information

Target database name: demodb

User name: dba

Unloaded Files

● Select unloaded file from list: demodb

Load Type	Path	Date
☑ object	C:\CUBRID\DATABA~1\demodb\demodb_objects	2009.08.26 14:05
☑ schema	C:\CUBRID\DATABA~1\demodb\demodb_schema	2009.08.26 14:05

○ Select unloaded file from system

□ Load schema [Browse]

□ Load object [Browse]

□ Load index [Browse]

□ Load trigger [Browse]

Load Option

● Check syntax and load database

○ Load data without checking syntax

□ Estimated number of instances

□ Insertion count for periodic commit

□ Don't use OID

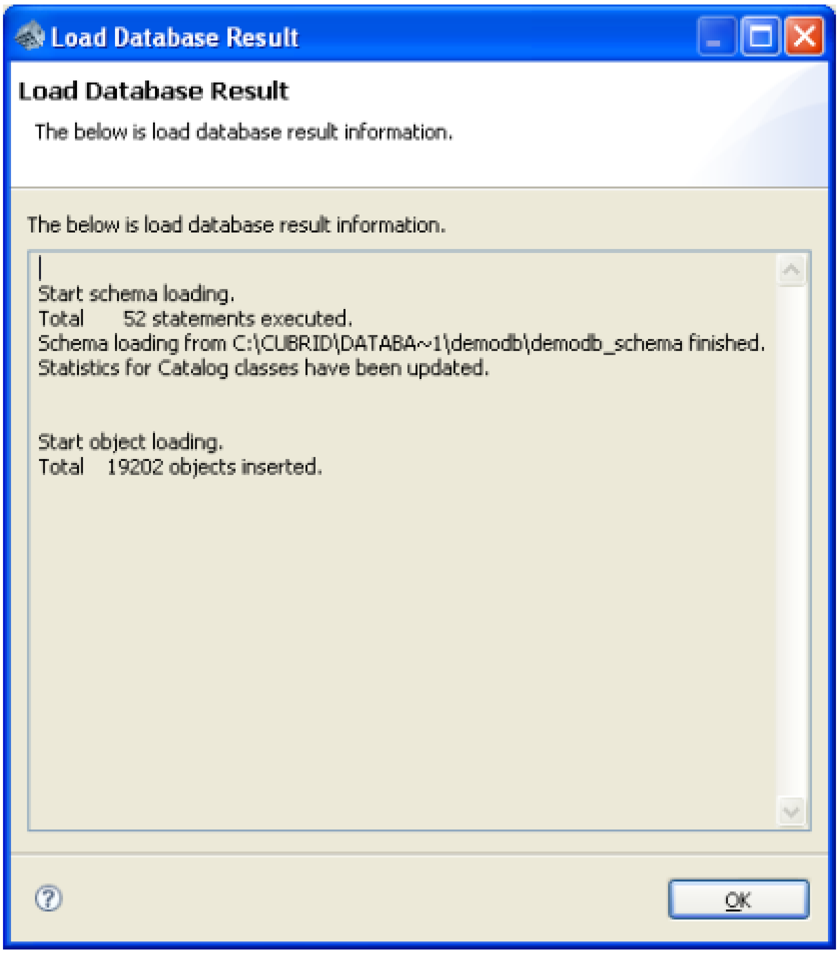
□ Using error control file [Browse]

□ Ignored table file [Browse]

[OK] [Cancel]

- **Database Information** : Displays the name of the target database into which the unloaded data is to be loaded, and the authority of the user who is logged in to the database. You must specify the name of currently logged-in user in the [User name] field.
- **Unloaded Target** : Enter information of the unloaded data. You can either select from the list or directly enter the path where the unloaded file is located.
- If the CUBRID Manager client and server are running on different systems, unloaded schema, object, index and trigger files must exist on the system running the CUBRID Manager server.
- If there is already any unloaded data from the CUBRID Manager, you can use unload information to load. Otherwise, manually enter unload information and then load it.
- **Load Option** : You can select whether to perform syntax checking before loading data, or perform syntax checking only. For more information, see Loading Database.

To check the results of the database load operation, set options and then click the [OK] button in the [Database load] dialog box.


Load Database Result
Load Database Result
The below is load database result information.
The below is load database result information.
OK

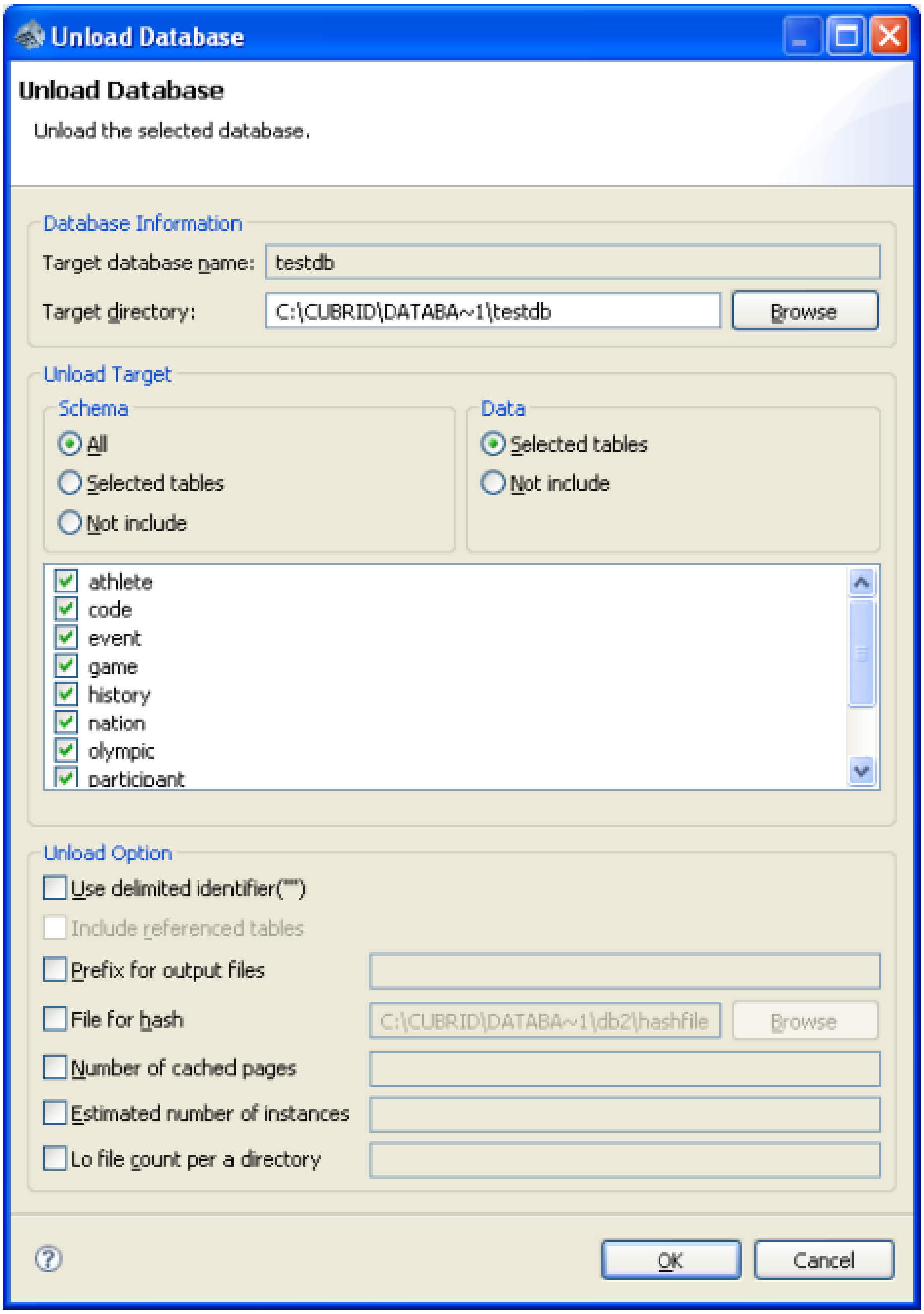
```
Start schema loading.
Total       52 statements executed.
Schema loading from C:\CUBRID\DATABA~1\demodb\demodb_schema finished.
Statistics for Catalog classes have been updated.


Start object loading.
Total    19202 objects inserted.
```

Unload Database

To unload data of the selected database, perform one of the following after login to the database.

- Click [Unload Database] from the toolbar.
- Right-click the database and then select [Unload Database].
- Select [Action] > [Unload Database] on the menu.

The unload operation can be performed even when the database server is running.



Unload Database

Unload Database

Unload the selected database.

Database Information

Target database name: testdb

Target directory: C:\CUBRID\DATABA~1\testdb

Browse

Unload Target

Schema

● All

○ Selected tables

○ Not include

Data

● Selected tables

○ Not include

☑ athlete

☑ code

☑ event

☑ game

☑ history

☑ nation

☑ olympic

☑ participant

Unload Option

□ Use delimited identifier("")

□ Include referenced tables

□ Prefix for output files

□ File for hash C:\CUBRID\DATABA~1\db2\hashfile Browse

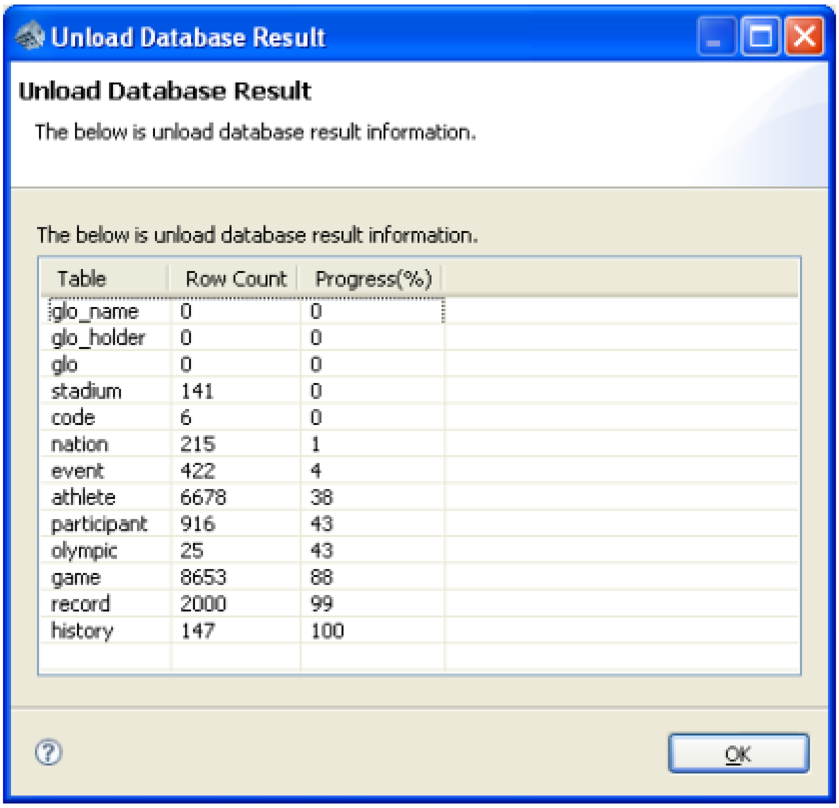
□ Number of cached pages

□ Estimated number of instances

□ Lo file count per a directory

- **Database Information** : The name of the database to be unloaded is displayed. Specify the directory in which a file is to be created after the unload operation.
- **Unload Target** : Select the schema and data of the database to be unloaded.
- **Unload Option** : You can specify whether to use delimited identifiers ("") or how many LO files are to be stored in a single directory. For more information, see Unloading Database.

To check the results of the database unload operation, set options and then click the [OK] button in the [Unload Database] dialog box.



Unload Database Result

The below is unload database result information.

The below is unload database result information.

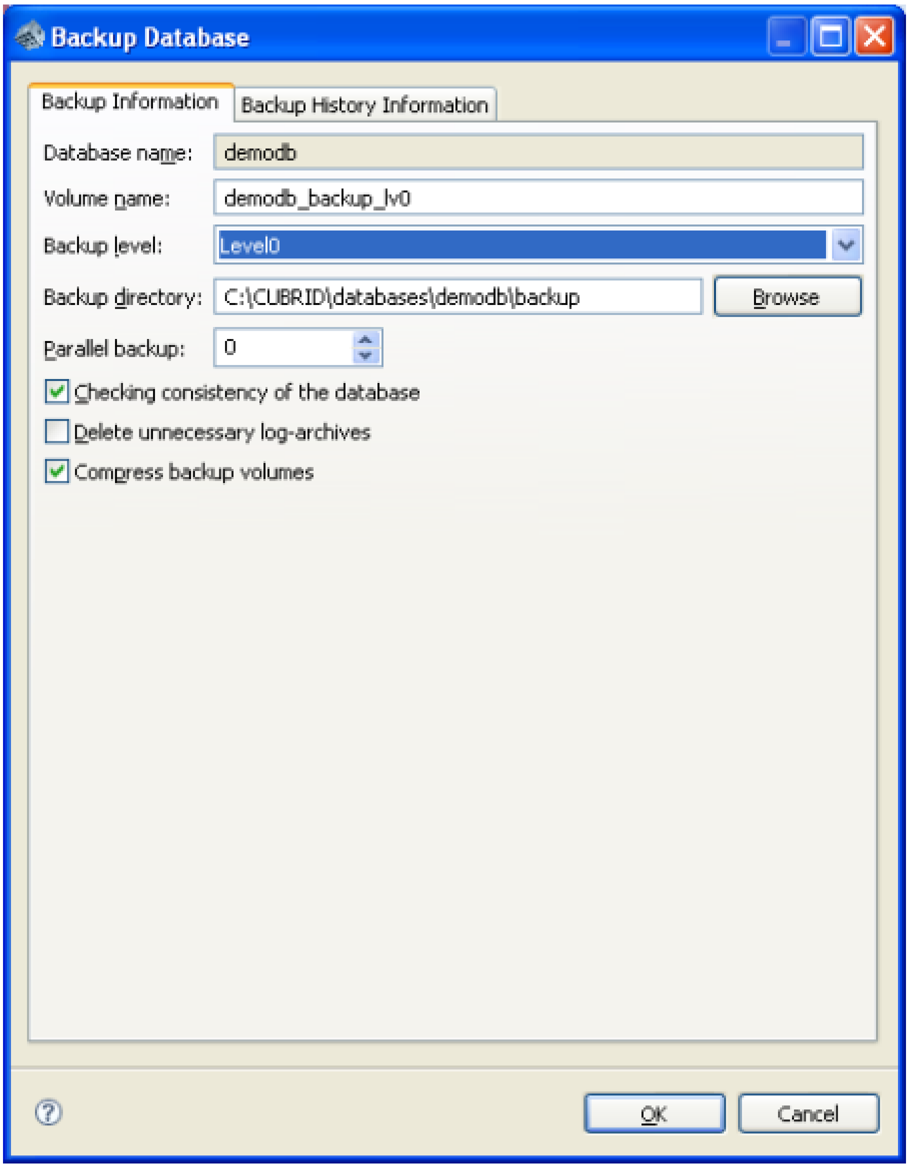
Table	Row Count	Progress(%)
glo_name	0	0
glo_holder	0	0
glo	0	0
stadium	141	0
code	6	0
nation	215	1
event	422	4
athlete	6678	38
participant	916	43
olympic	25	43
game	8653	88
record	2000	99
history	147	100

Backup Database

To backup a database, perform one of the following after login to the database.

- Click [Backup Database] from the toolbar.
- Right-click the database and then select [Backup Database].
- Select [Action] > [Backup Database] on the menu.

The backup operation can be performed even when the database server is running. For more information on database backup, see Database Backup.



Backup Database

Database name: demodb

Volume name: demodb_backup_lv0

Backup level: Level0

Backup directory: C:\CUBRID\databases\demodb\backup

Parallel backup: 0

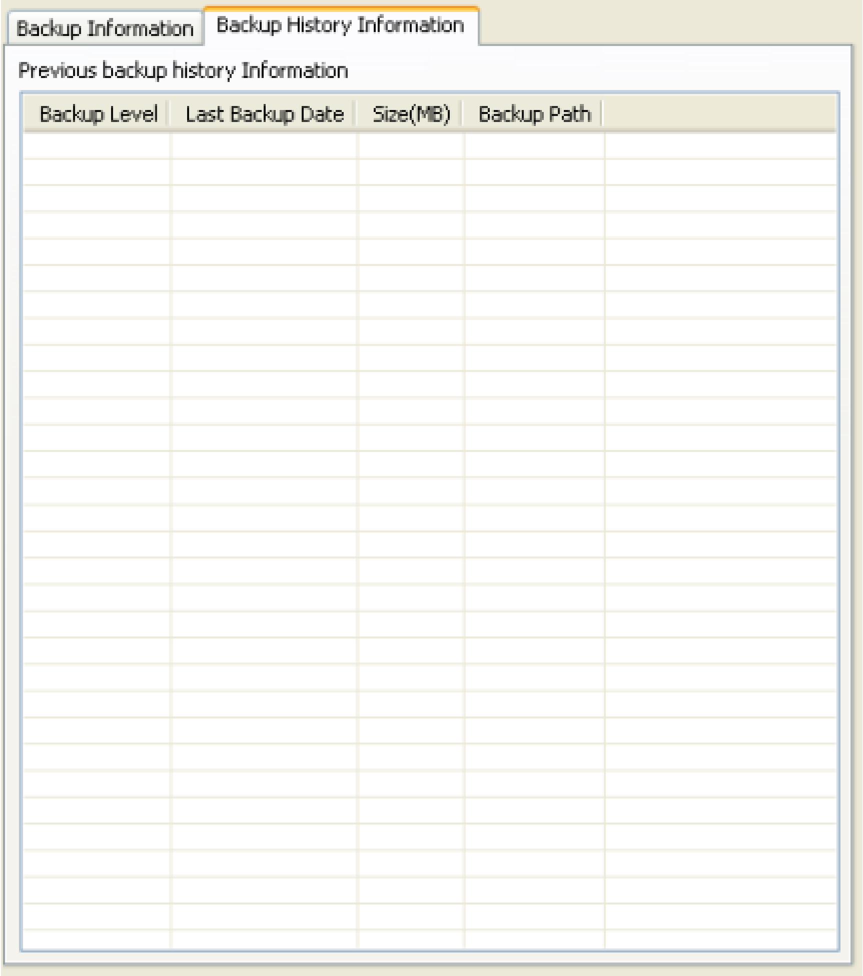
☑ Checking consistency of the database

□ Delete unnecessary log-archives

☑ Compress backup volumes

- **Database name** : The name of the database to backup.
- **Volume name** : Specifies the name of the backup volume. The default volume name is in the form of [database name_backup_backup level]. However, you can change the name.
- **Backup level** : Specifies the backup level. Level0 is a full backup, Level1 is a backup that saves changes made only after the Level0 backup. Level2 is a backup that saves changes made only after the level 1 backup.
  For example, if there is a backup volume displayed as Level0, the administrator can choose only Level0 or Level1. If there are no previously performed backups, the administrator can choose only Level0.
- **Backup directory** : Specifies the directory where backup files are to be saved. The default is set to the **$CUBRID/databases/[DBNAME]/backup** directory.
- **Checking consistency of the database** : Checks consistency of the database to backup before the backup process. It is recommended to select this option.
- **Delete unnecessary log-archives** : Deletes unnecessary archive log files while restoring a database. Note that archive log files might be required if database restore does not go well due to backup file errors. When this option is selected while the database is set to the master server in the replication environment, the Maintain replication log option is checked automatically.
- **Number thread** : Specifies the number of threads to be used concurrently during the database backup. It is recommended to set the maximum number of threads to be the same as the number of CPUs. The default value is 0, in which case the number of threads is automatically determined.
- **Compress backup volumes** : Compresses the database backup. It is recommended to select this option.

If a backup has been executed for the target database, you can check the backup history from the [Backup History Information] tab. You can also check information such as the backup level, last backup date, size of the backup file, backup file path as well as free space of the disk where the current database volume exists.



Previous backup history Information

Backup Level	Last Backup Date	Size(MB)	Backup Path	

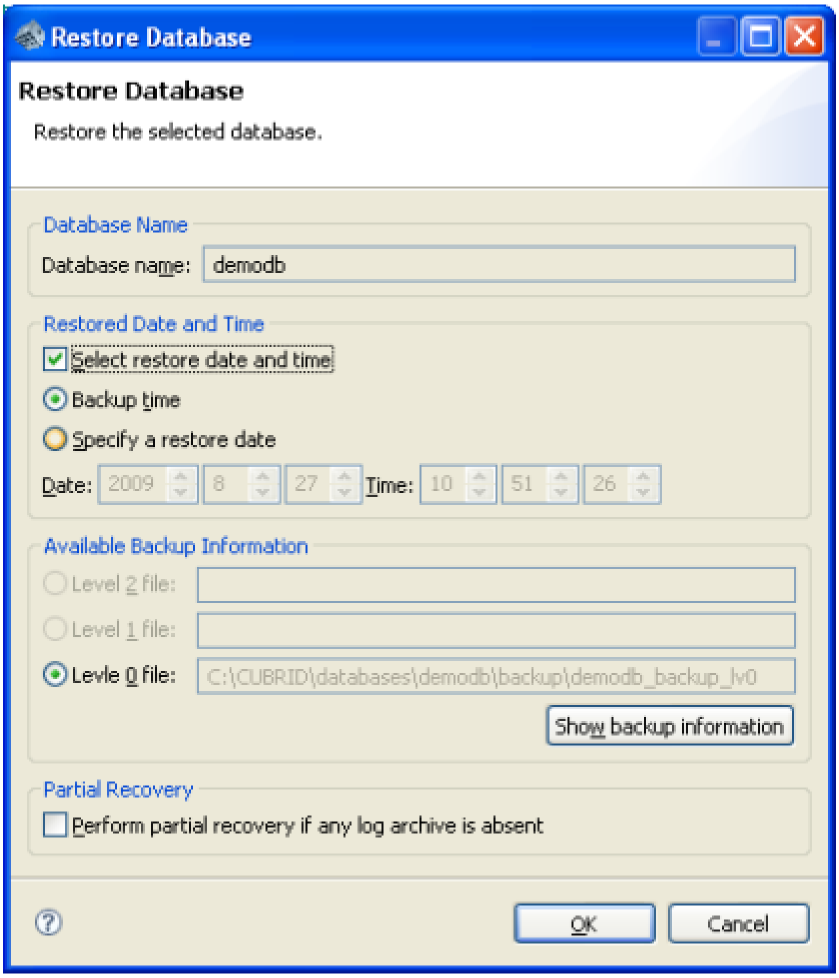
Restore Database

To restore a database, perform one of the following after login to the database.

- Click [Restore Database] from the toolbar.
- Right-click the database and then select [Restore Database].
- Select [Action] > [Restore Database] on the menu.

Loading data to restore database can be performed only when the database server is not running; The [Restore Database] menu is deactivated while the database is running.

For more information about restoring databases, see Database Backup.



Restore Database

Restore Database

Restore the selected database.

Database Name

Database name: demodb

Restored Date and Time

☑ Select restore date and time

● Backup time

○ Specify a restore date

Date: 2009 8 27 Time: 10 51 26

Available Backup Information

○ Level 2 file:

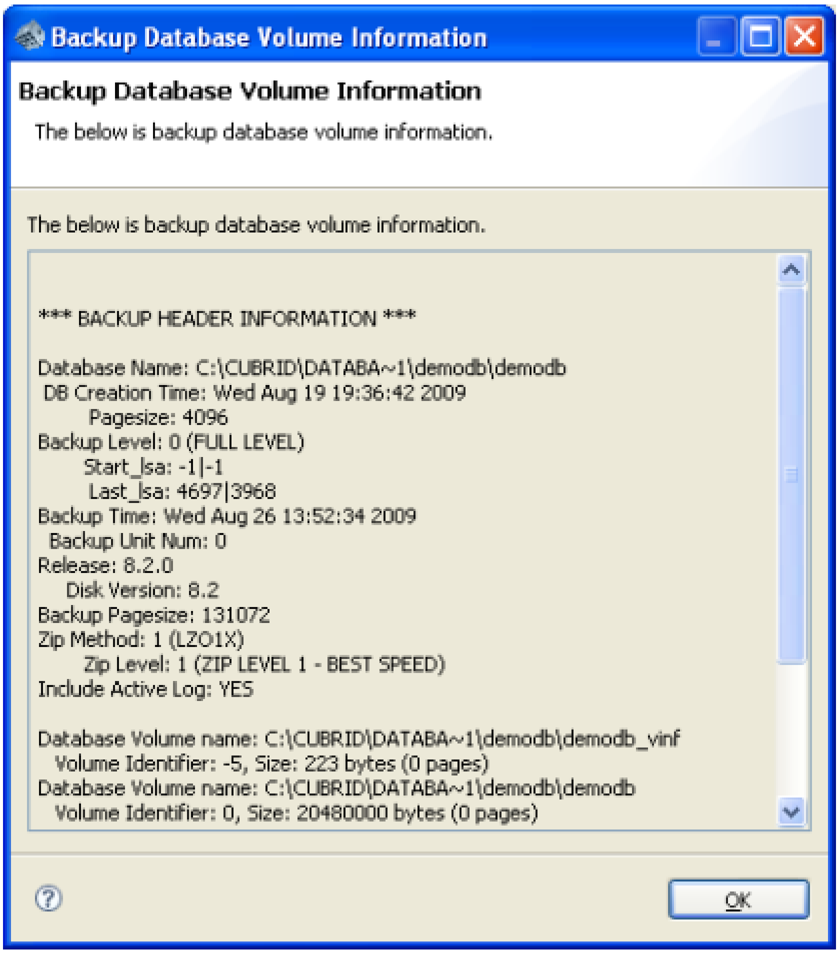
○ Level 1 file:

● Levle 0 file: C:\CUBRID\databases\demodb\backup\demodb_backup_lv0

Partial Recovery

□ Perform partial recovery if any log archive is absent

- **Database name** : The name of the target database to be restored.
- **Restored Date and Time** : Specifies to which point of time the database is to be restored back to. If you select [Backup time], a restore is performed with the **backuptime** keyword in the restore utility. This means that the database is restored to the point when the backup was complete. If you select [Specify a restore date], you can enter date and time you want.
- **Available Backup Information** : You can select the restore level after checking which levels of backing up have been performed on the target database. The specified file path is the path to the directory where the files for the back up for the selected level are located.
- **Perform partial restore if any log archive is absent** : Performs a partial restore if the case of incomplete log information. That is, database restore can be performed even without archive or active logs created after the backup point.
- **Restore to a path specified by the user** : Restores the database to the path specified in the database location file (**databases.txt**).
- **Show backup information** : Shows the information of the file backed up to the selected backup level.


Backup Database Volume Information
Backup Database Volume Information
The below is backup database volume information.
The below is backup database volume information.
OK

```
*** BACKUP HEADER INFORMATION ***

Database Name: C:\CUBRID\DATABA~1\demodb\demodb
 DB Creation Time: Wed Aug 19 19:36:42 2009
         Pagesize: 4096
Backup Level: 0 (FULL LEVEL)
        Start_lsa: -1|-1
         Last_lsa: 4697|3968
Backup Time: Wed Aug 26 13:52:34 2009
  Backup Unit Num: 0
Release: 8.2.0
     Disk Version: 8.2
Backup Pagesize: 131072
Zip Method: 1 (LZO1X)
        Zip Level: 1 (ZIP LEVEL 1 - BEST SPEED)
Include Active Log: YES

Database Volume name: C:\CUBRID\DATABA~1\demodb\demodb_vinf
   Volume Identifier: -5, Size: 223 bytes (0 pages)
Database Volume name: C:\CUBRID\DATABA~1\demodb\demodb
   Volume Identifier: 0, Size: 20480000 bytes (0 pages)
```

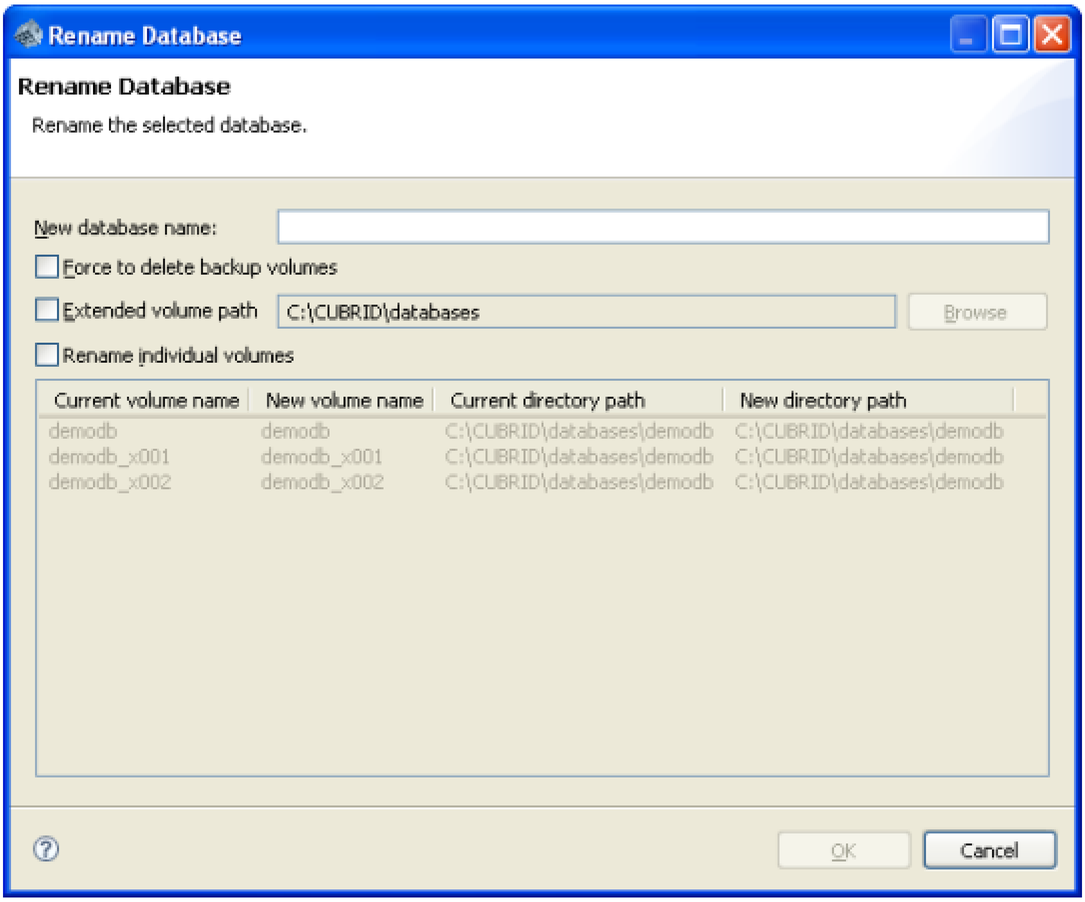
Rename Database

To rename a database, perform one of the following after login to the database.

- Right-click the database and then select [Rename Database].
- Select [Action] > [Rename Database] on the menu.

The rename database operation can be performed only when the database server is not running. Therefore, the [Rename Database] menu is deactivated while the database is running.

For more information about renaming databases, see Renaming Database.



Rename Database

Rename the selected database.

New database name:

□ Force to delete backup volumes

□ Extended volume path C:\CUBRID\databases

□ Rename individual volumes

Current volume name	New volume name	Current directory path	New directory path
demodb	demodb	C:\CUBRID\databases\demodb	C:\CUBRID\databases\demodb
demodb_x001	demodb_x001	C:\CUBRID\databases\demodb	C:\CUBRID\databases\demodb
demodb_x002	demodb_x002	C:\CUBRID\databases\demodb	C:\CUBRID\databases\demodb

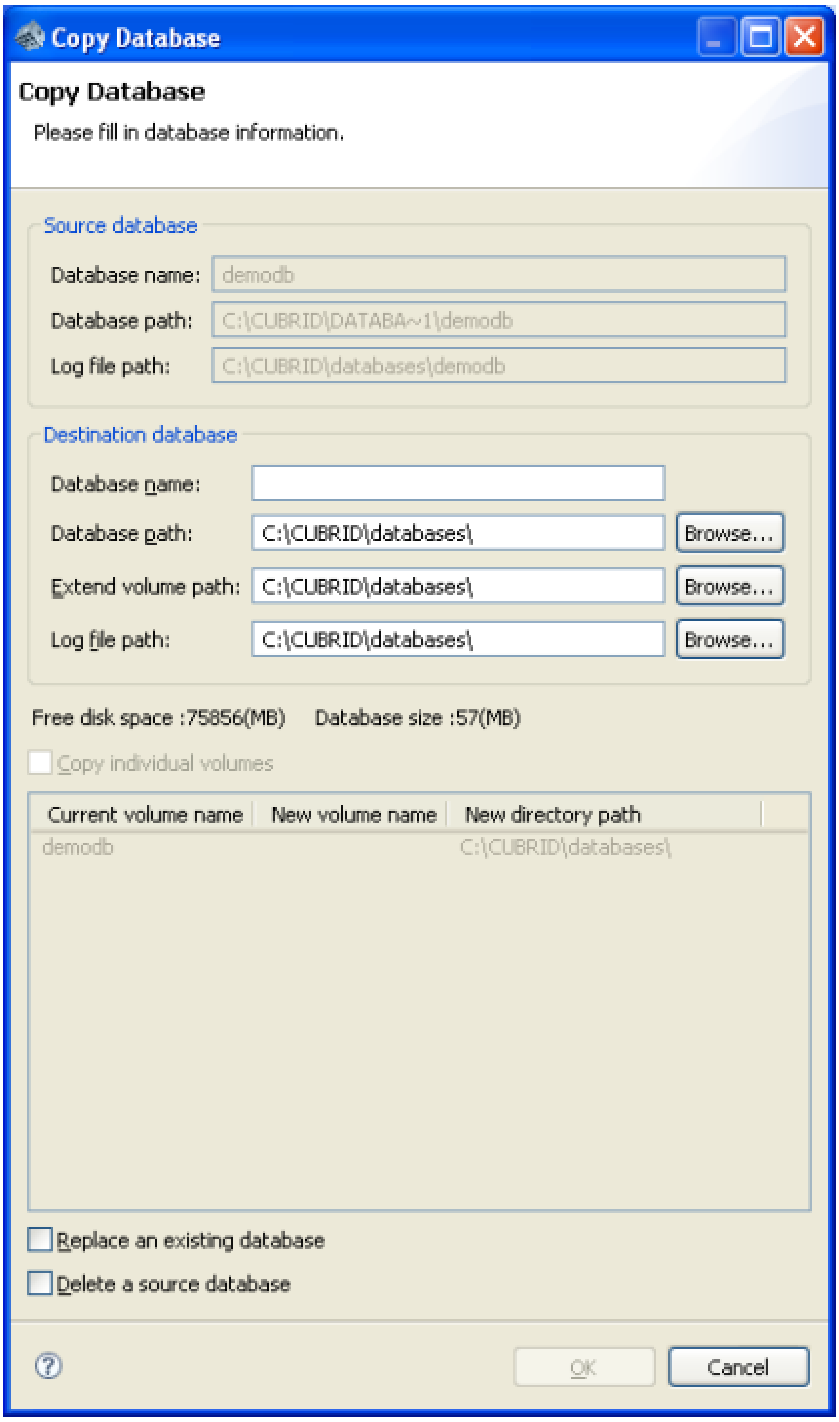
- **New database name** : Enter a new name for the database to be renamed.
- **Force to delete backup volumes** : Delete backup volumes of the database before the rename operation.
- **Extended volume path** : An option that specifies the path in which volumes to be added to the new database are stored.
- **Rename individual volumes** : If there are multiple volumes before renaming the database, you can rename individual database volumes and specify a new directory in which each database volume is stored.

Copy Database

To copy the database, perform one of the following after login to the database.

- Right-click the database and then select [Copy Database].
- Select [Action] > [Copy Database] on the menu.

The copy database operation can be performed only when the database server is not running. Therefore, the [Copy Database] menu is deactivated while the database is running.



Copy Database

Copy Database

Please fill in database information.

Source database

Database name: demodb

Database path: C:\CUBRID\DATABA~1\demodb

Log file path: C:\CUBRID\databases\demodb

Destination database

Database name:

Database path: C:\CUBRID\databases\

Extend volume path: C:\CUBRID\databases\

Log file path: C:\CUBRID\databases\

Free disk space :75856(MB)   Database size :57(MB)

□ Copy individual volumes

Current volume name	New volume name	New directory path
demodb		C:\CUBRID\databases\

□ Replace an existing database

□ Delete a source database

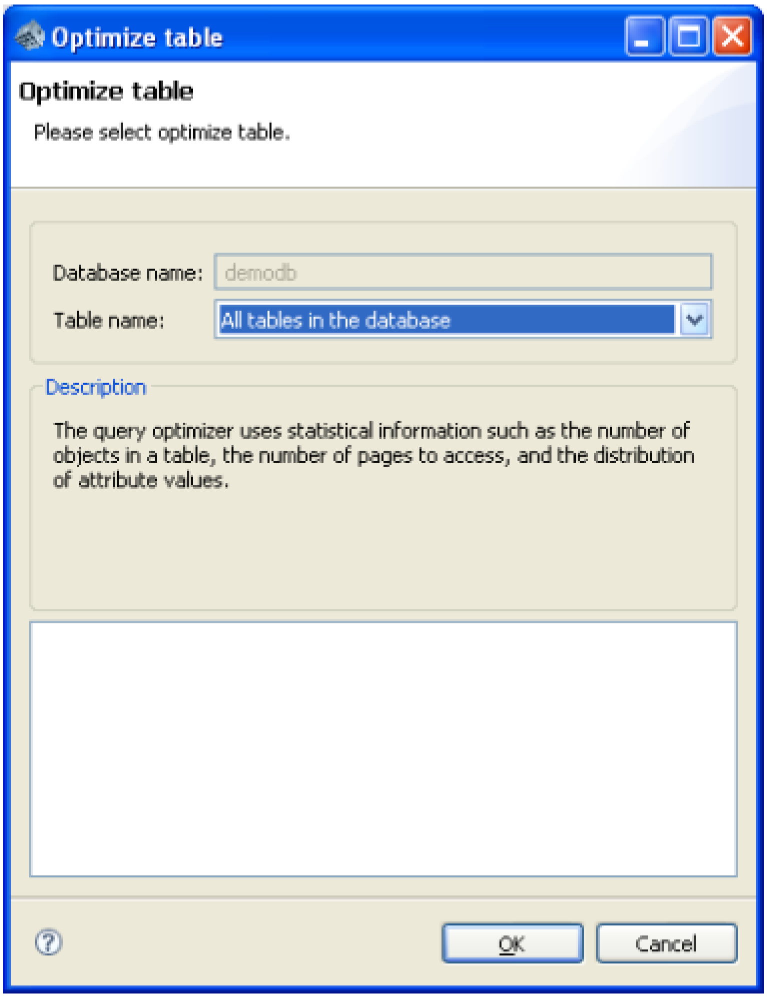
- **Source database** : A database name to be copied and a directory path where the volume and log file saved are shown.
- **Destination database** : Enter a database name to be created and a directory path where the volume, extended volume, and log file to be stored.
- **Copy individual volumes** : If there is more than one volume of the source database, you can rename each database volume and specify a new directory path to which each database volume is to be copied.
- **Replace an existing database** : Overwrites an existing database of the same name as the destination database.
- **Delete a source database** : Deletes the source database after copying.

Optimize Database

To optimize the database, perform one of the following after login to the database.

- Click [Optimize Database] from the toolbar.
- Right-click the database and then select [Optimize Database].
- Select [Action] > [Optimize Database] on the menu.

In the [Optimize table] dialog box, you can perform optimization for all or some tables in the database. Select the target to optimize and click [OK] to start database optimization. If you select [See Detailed Progress], detailed information is displayed when the compact database operation is completed.



Optimize table

Please select optimize table.

Database name: demodb

Table name: All tables in the database

Description

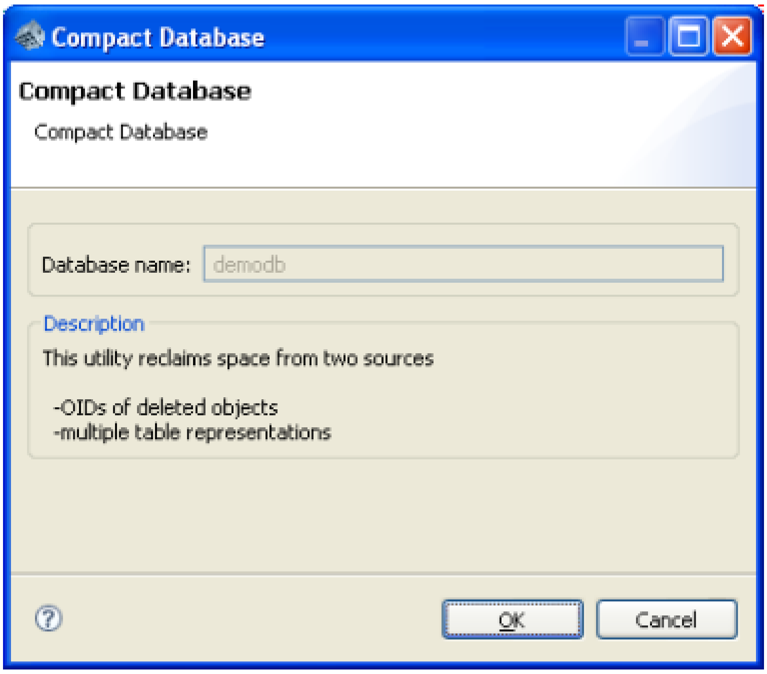
The query optimizer uses statistical information such as the number of objects in a table, the number of pages to access, and the distribution of attribute values.

Compact Database

To compact the database, perform one of the following after login to the database.

- Right-click the database and then select [Compact Database].
- Select [Action] > [Compact Database] on the menu.

The compact database operation can be performed only when the database server is not running; The [Compact Database] menu is deactivated while the database is running.



Database name: demodb

Description

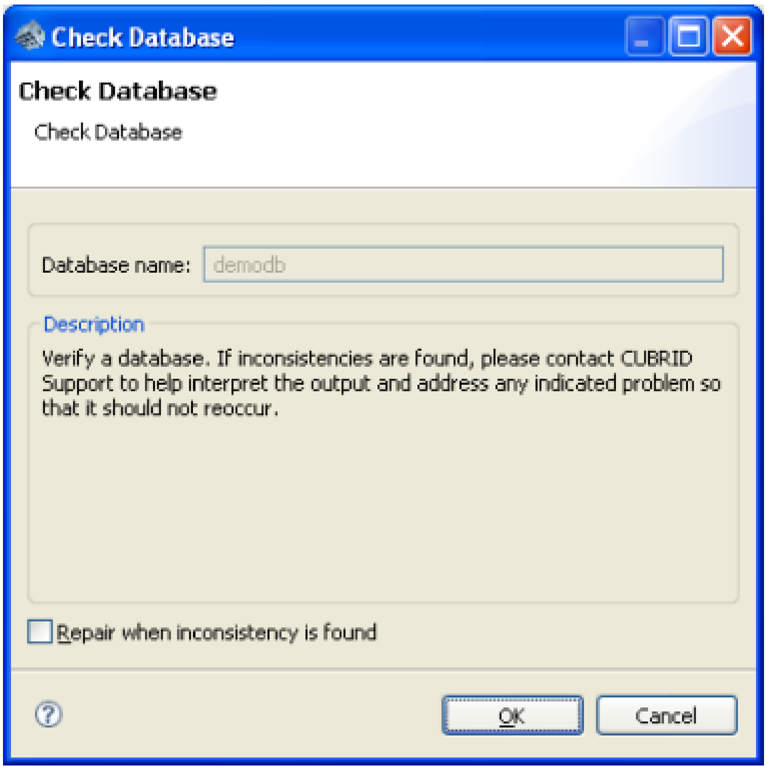
This utility reclaims space from two sources

-OIDs of deleted objects
-multiple table representations

Check Database

To check the database, perform one of the following after login to the database.

- Click [Check Database ✓] from the toolbar.
- Right-click the database and then select [Check Database].
- Select [Action] > [Check Database] on the menu.



Database name: demodb

Description

Verify a database. If inconsistencies are found, please contact CUBRID Support to help interpret the output and address any indicated problem so that it should not reoccur.

□ Repair when inconsistency is found

Database Lock Information

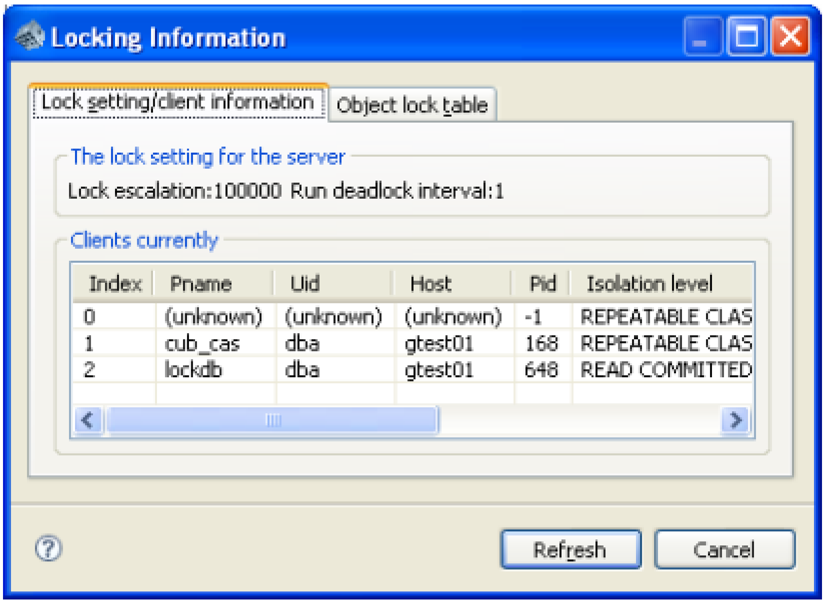
To view the lock information of the database, perform one of the following after login to the database.

- Click [Lock Information] from the toolbar.
- Right-click the database and then select [Lock Information].
- Select [Action] > [Lock Information] on the menu.

Note that this function is activated only while the database server is running; this is deactivated while the server is not running.

The [Lock Information] dialog box consists of two tabs.

The [Lock setting/client information] tab provides information of the clients currently connected to the database.



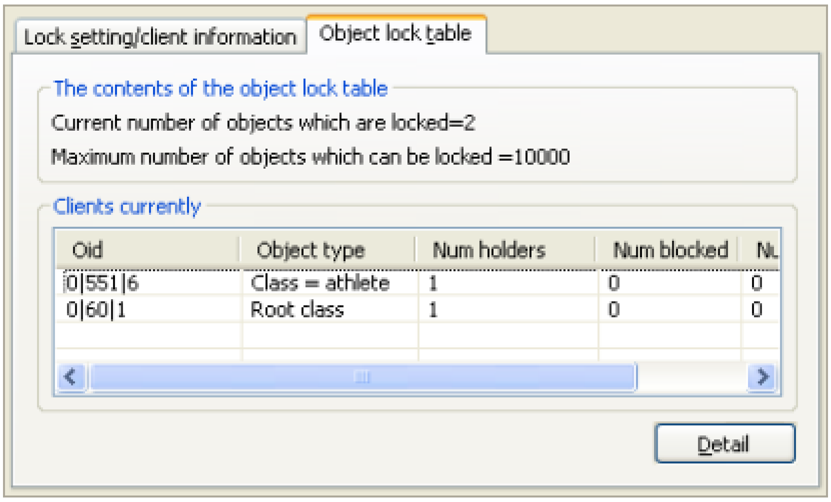
The lock setting for the server

Lock escalation:100000 Run deadlock interval:1

Clients currently

Index	Pname	Uid	Host	Pid	Isolation level
0	(unknown)	(unknown)	(unknown)	-1	REPEATABLE CLAS
1	cub_cas	dba	gtest01	168	REPEATABLE CLAS
2	lockdb	dba	gtest01	648	READ COMMITTED

In the [Object lock table] tab, you can check the lock information of the database objects.



The contents of the object lock table

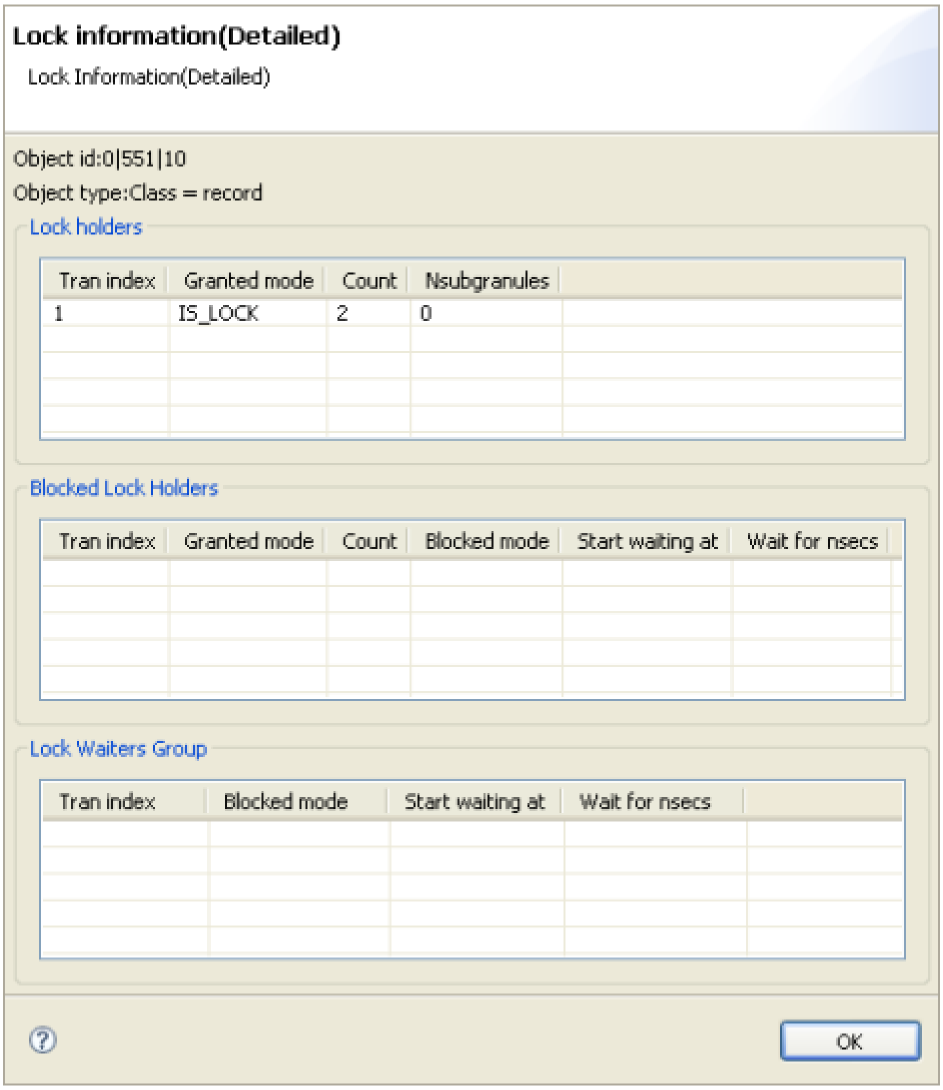
Current number of objects which are locked=2

Maximum number of objects which can be locked =10000

Clients currently

Oid	Object type	Num holders	Num blocked	Nu
0|551|6	Class = athlete	1	0	0
0|60|1	Root class	1	0	0

You can view detailed lock information by clicking [Detail] in the [Object lock table] tab.



Lock information(Detailed)

Lock Information(Detailed)

Object id:0|551|10

Object type:Class = record

Lock holders

Tran index	Granted mode	Count	Nsubgranules
1	IS_LOCK	2	0

Blocked Lock Holders

Tran index	Granted mode	Count	Blocked mode	Start waiting at	Wait for nsecs

Lock Waiters Group

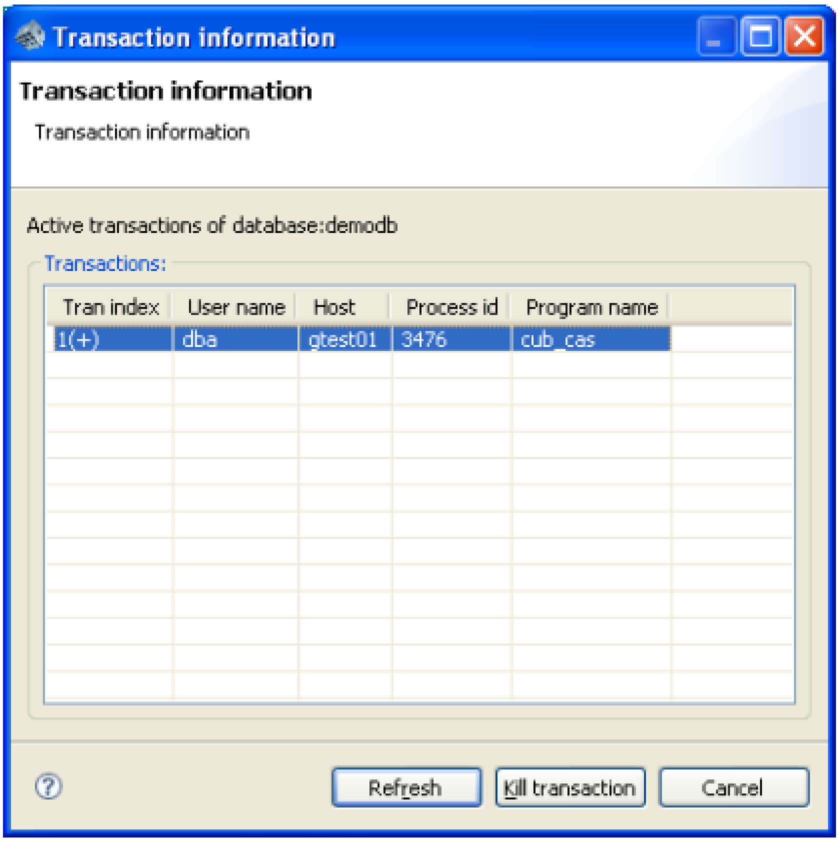
Tran index	Blocked mode	Start waiting at	Wait for nsecs

Database Transaction Information

To view the transaction information of the database, perform one of the following after login to the database.

- Click [Transaction Info] from the toolbar.
- Right-click the database and then select [Transaction Info].
- Select [Action] > [Transaction Info] on the menu.

Note that this function is activated only while the database server is running; this is deactivated while the server is not running.



Transaction information

Transaction information

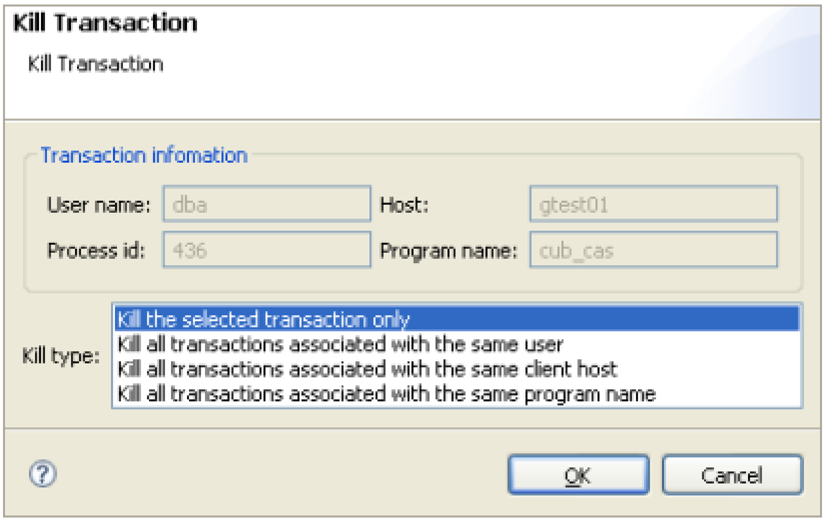
Transaction information

Active transactions of database:demodb

Transactions:

Tran index	User name	Host	Process id	Program name
1(+)	dba	gtest01	3476	cub_cas

- **Refresh** : Recollects and shows information of the transactions currently being performed by the CUBRID Manager server.
- **Kill transaction** : Forces termination of the transaction selected in the transaction list. One of the following four options can be selected.



Kill Transaction

Kill Transaction

Transaction infomation

User name: dba

Host: gtest01

Process id: 436

Program name: cub_cas

Kill type:

Kill the selected transaction only

Kill all transactions associated with the same user

Kill all transactions associated with the same client host

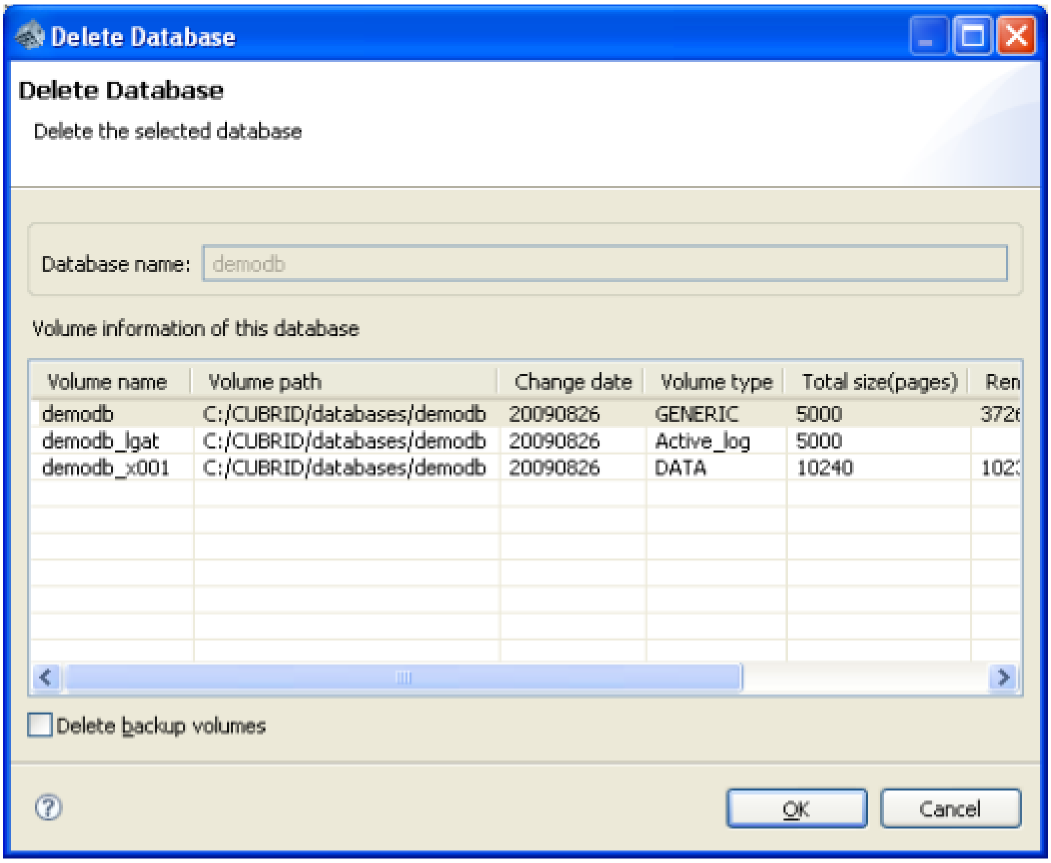
Kill all transactions associated with the same program name

Delete Database

To delete a database, perform one of the following after login to the database.

- Right-click the database and then select [Delete Database].
- Select [Action] > [Delete Database] on the menu.

The delete database operation can be performed only when the database server is not running; The [Delete Database] menu is deactivated while the database is running.



Delete Database

Delete the selected database

Database name: demodb

Volume information of this database

Volume name	Volume path	Change date	Volume type	Total size(pages)	Ren
demodb	C:/CUBRID/databases/demodb	20090826	GENERIC	5000	372
demodb_lgat	C:/CUBRID/databases/demodb	20090826	Active_log	5000	
demodb_x001	C:/CUBRID/databases/demodb	20090826	DATA	10240	102

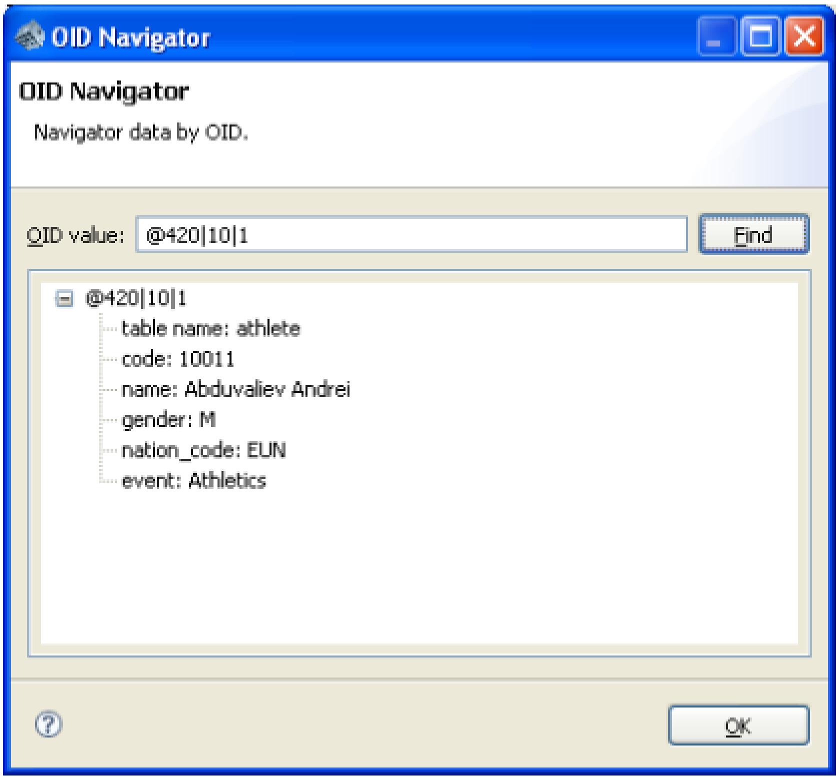
□ Delete backup volumes

If you click [OK], the database is deleted after checking the **dba** password. Only the **dba** user can delete a database.

OID Navigator

It is used to search data by OIDs.

You can run the OID Navigator by right-clicking a database or inside the Result pane of the Query editor and then selecting [OID Navigator]. Note that this function is activated only while the database server is running; this is deactivated while the server is not running.


OID Navigator

OID Navigator

Navigator data by OID.

OID value: @420|10|1

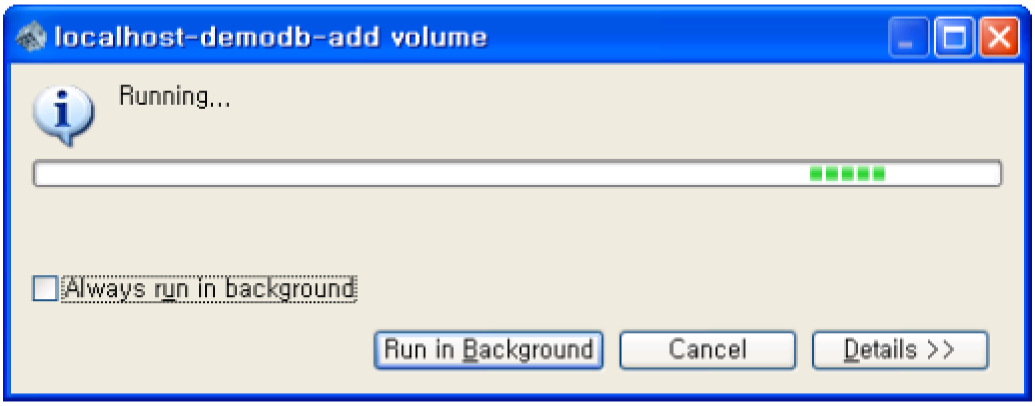
@420|10|1
- table name: athlete
- code: 10011
- name: Abduvaliev Andrei
- gender: M
- nation_code: EUN
- event: Athletics

Configuring Background Operations in Multiple-Host Environment

If you manage databases in several hosts by using [Add Host], databases can be configured for background operations. For example, if you want to execute a query to a database in host B while doing backup for a database in host A, the database backup operation, which generally takes a long time, can be configured to be executed as a background operation. The database operations that can be run in the background are as follows:

- **Database related** : Create Database, Unload/Load Database, Backup/Restore Database, Rename Database, Copy Database, Optimize Database, Compact Database, Check Database
- **Table related** : Import/Export, Delete All Records, Update data by changing constraints from NULL to NOT NULL
- **Volume** : Add Volume

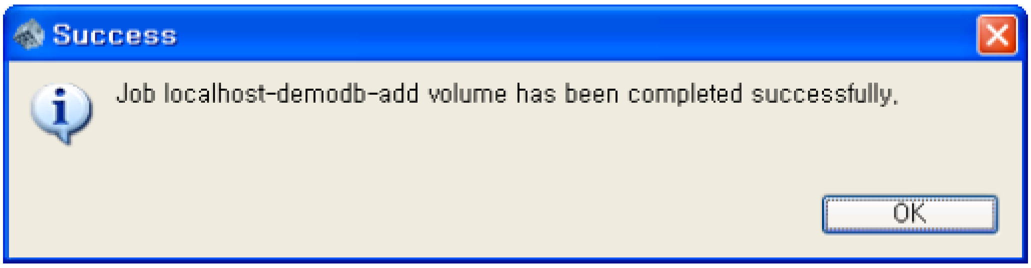
In the dialog shown below, you can select [Run in Background] when adding a volume.


localhost-demodb-add volume

Running...

□ Always run in background

When a background operation is complete, the dialog shown below appears.



Success

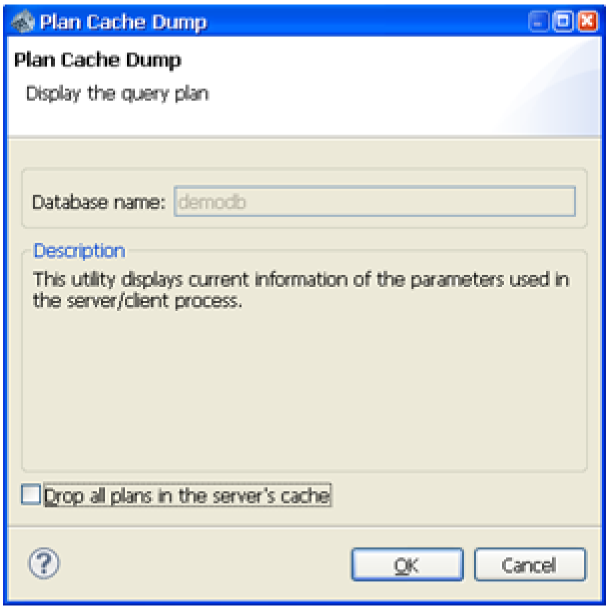
Job localhost-demodb-add volume has been completed successfully.

If an administrator stops the database or logs out from the database or host while a database operation is being executed as a background operation, a dialog indicating that a background operation is in progress appears.

Information on Parameters in Use

Displays information on the parameters used in the database. You can check the items to be displayed when the **cubrid paramdump** utility is running in the Manager.

For more information about this utility, see Outputting Parameters Used in Server/Client.



Plan Cache Dump

Plan Cache Dump

Display the query plan

Database name: demodb

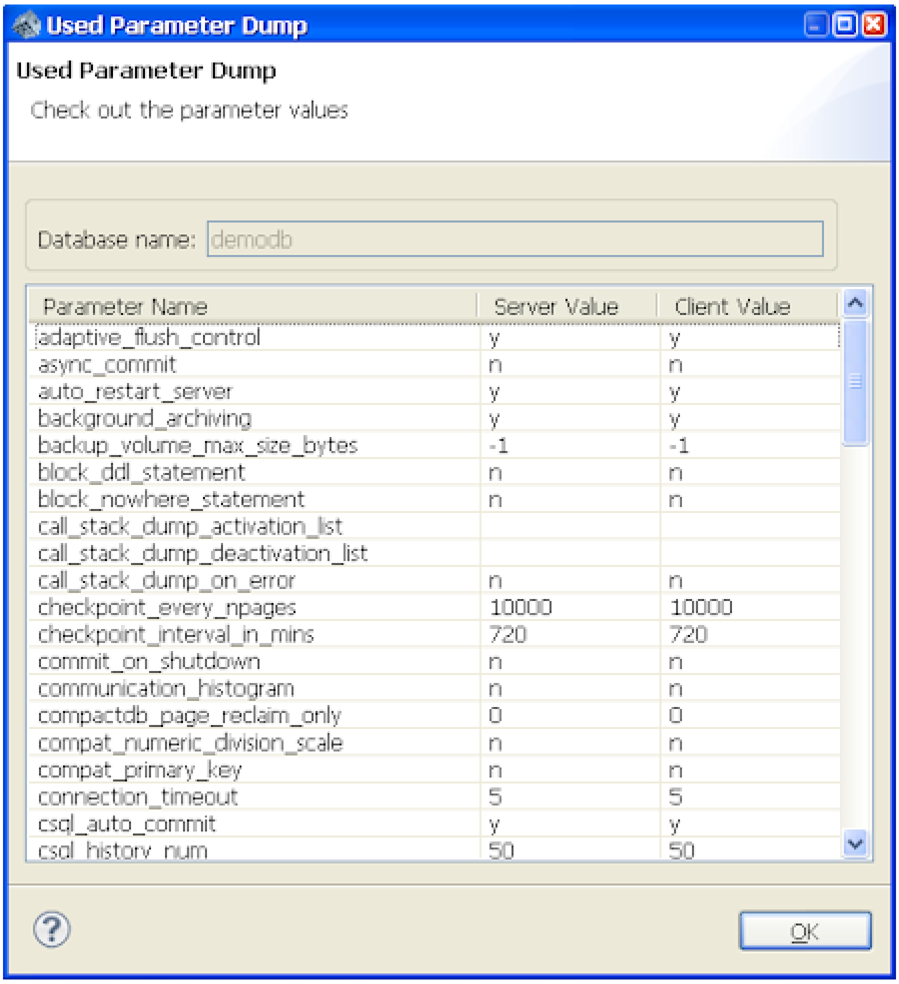
Description

This utility displays current information of the parameters used in the server/client process.

□ Drop all plans in the server's cache

If you select [Display server/client parameters information] option, you can check all parameter values applied to the server and the client.

If you deselect the option, you can check only the parameter values applied to the server. If you click [OK], you can check server/client parameter values, as follows:



Used Parameter Dump

Check out the parameter values

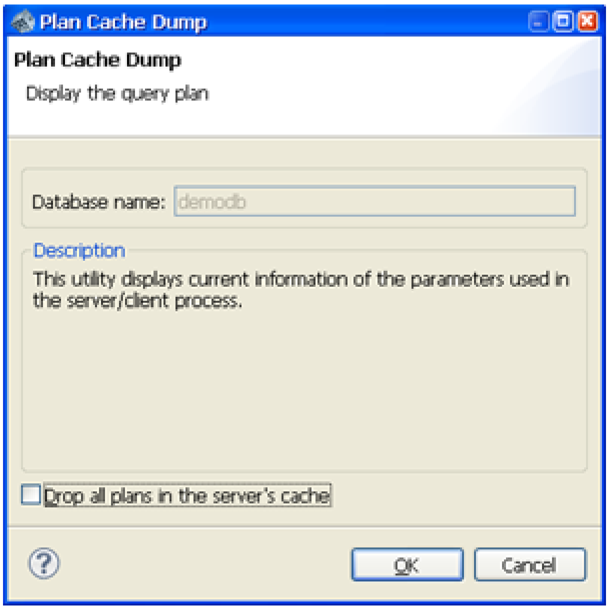
Database name: demodb

Parameter Name	Server Value	Client Value
adaptive_flush_control	y	y
async_commit	n	n
auto_restart_server	y	y
background_archiving	y	y
backup_volume_max_size_bytes	-1	-1
block_ddl_statement	n	n
block_nowhere_statement	n	n
call_stack_dump_activation_list		
call_stack_dump_deactivation_list		
call_stack_dump_on_error	n	n
checkpoint_every_npages	10000	10000
checkpoint_interval_in_mins	720	720
commit_on_shutdown	n	n
communication_histogram	n	n
compactdb_page_reclaim_only	0	0
compat_numeric_division_scale	n	n
compat_primary_key	n	n
connection_timeout	5	5
csql_auto_commit	y	y
csql_history_num	50	50

Query Plan Cache Information

Displays information on the query plan saved in the cache of the database server. You can check the items to be displayed when the **cubrid plandump** utility is running in the Manager. For more information about the utility, see Checking the Query Plan Cache.

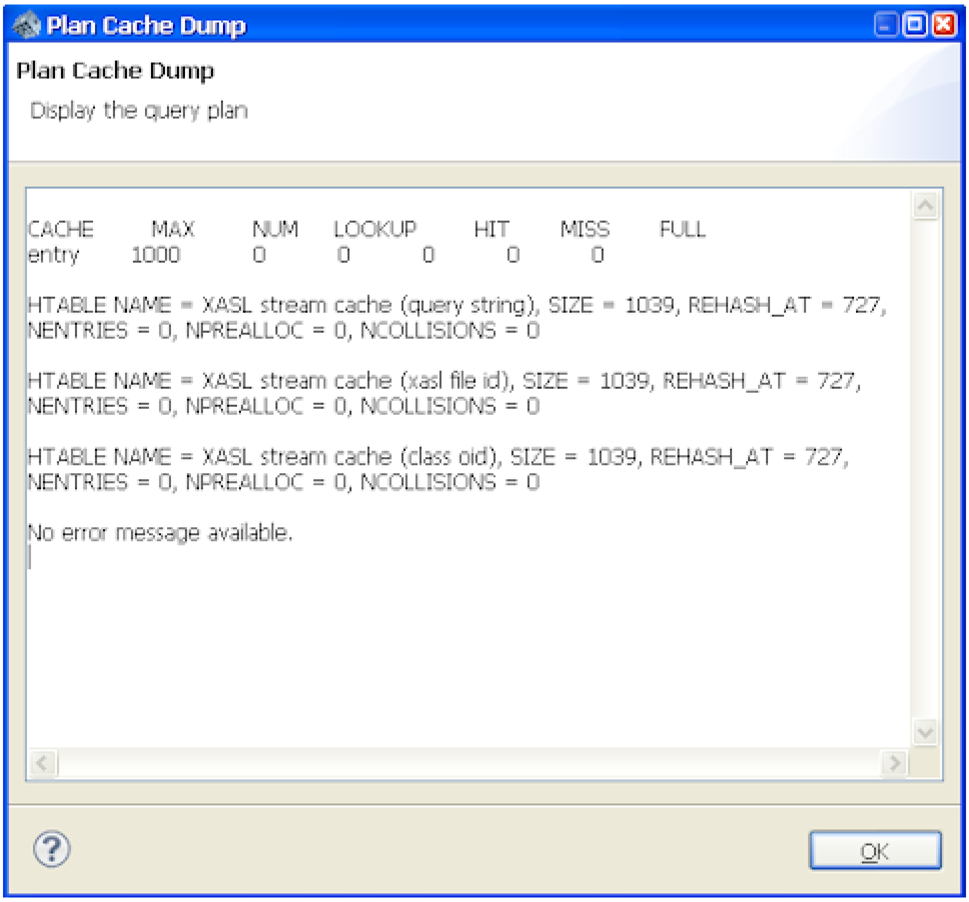
Since this feature can only be performed when the database server is running, the [Query Plan Cache Information] menu is activated only when the database server is running.



This utility displays current information of the parameters used in the server/client process.

□ Drop all plans in the server's cache

If you select the [Delete query plans saved in the cache] option, the collected database query plan cache information is displayed before it is initialized. If you click the [OK] button, you can check the query plan information saved in the cache, as follows:



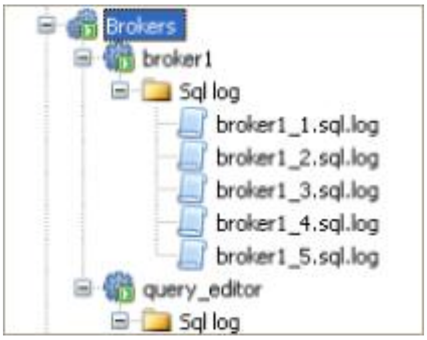
```
CACHE      MAX      NUM    LOOKUP     HIT     MISS     FULL
entry      1000     0      0          0       0        0

HTABLE NAME = XASL stream cache (query string), SIZE = 1039, REHASH_AT = 727,
NENTRIES = 0, NPREALLOC = 0, NCOLLISIONS = 0

HTABLE NAME = XASL stream cache (xasl file id), SIZE = 1039, REHASH_AT = 727,
NENTRIES = 0, NPREALLOC = 0, NCOLLISIONS = 0

HTABLE NAME = XASL stream cache (class oid), SIZE = 1039, REHASH_AT = 727,
NENTRIES = 0, NPREALLOC = 0, NCOLLISIONS = 0

No error message available.
```

Broker

Broker Structure

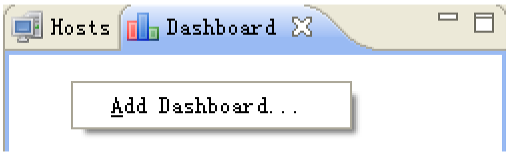
A Broker is a multi-function connector, enabling the connection between a database and many different interfaces such as ODBC, OLEDB, JDBC, PHP, etc. For more information, see Administrator's Guide.

A Broker consists of the names of individually configured Brokers and their SQL logs.

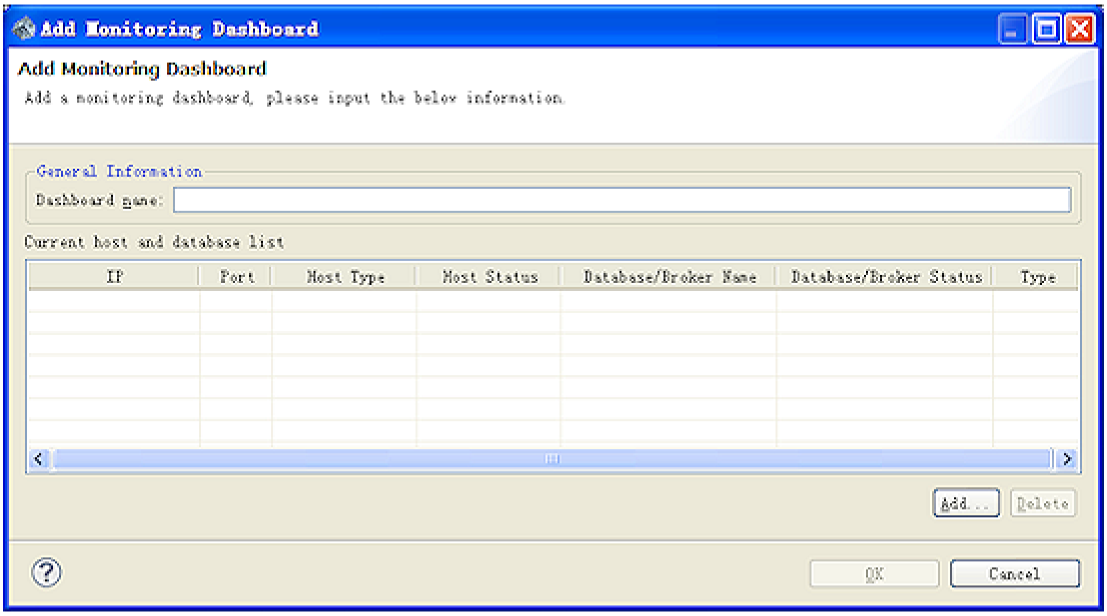


Add Dashboard

[Dashboard] tap appears next to [Hosts] tab when you select [Tool] > [Open Dashboard Explorer] in menu.

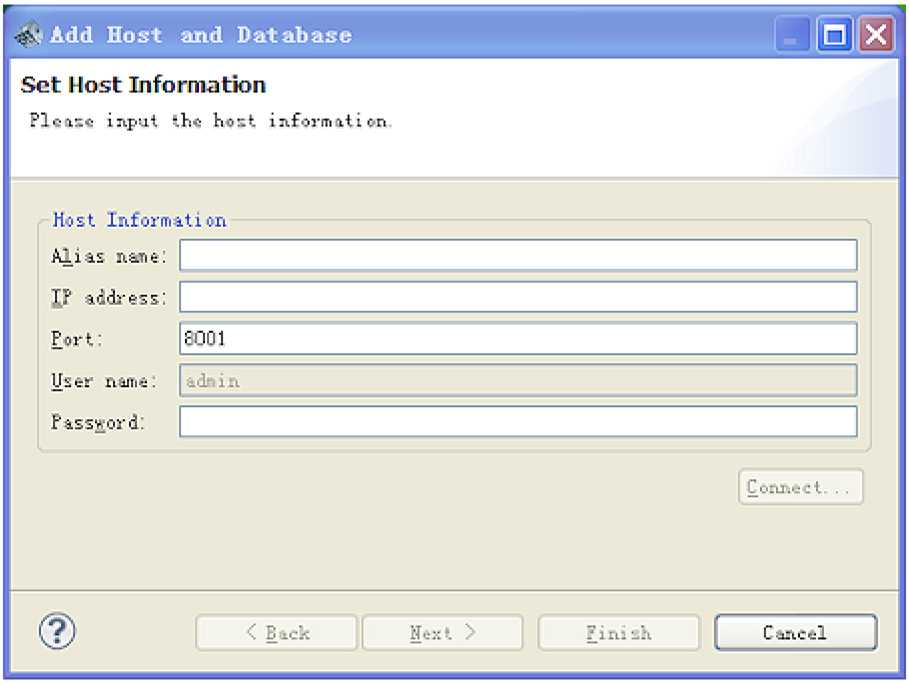


Right-click on the [Dashboard] tab and select [Add Dashboard]. Then, [Add Monitoring Dashboard] dialog will appear. Enter the dashboard name in [Dashboard name] field and click [Add] button.



Step 1: Set Host Information

Enter the host information of the database server you want to monitor. At this time, The CUBRID manager server must be running in the host.



Add Host and Database

Set Host Information

Please input the host information.

Host Information

Alias name:

IP address:

Port: 8001

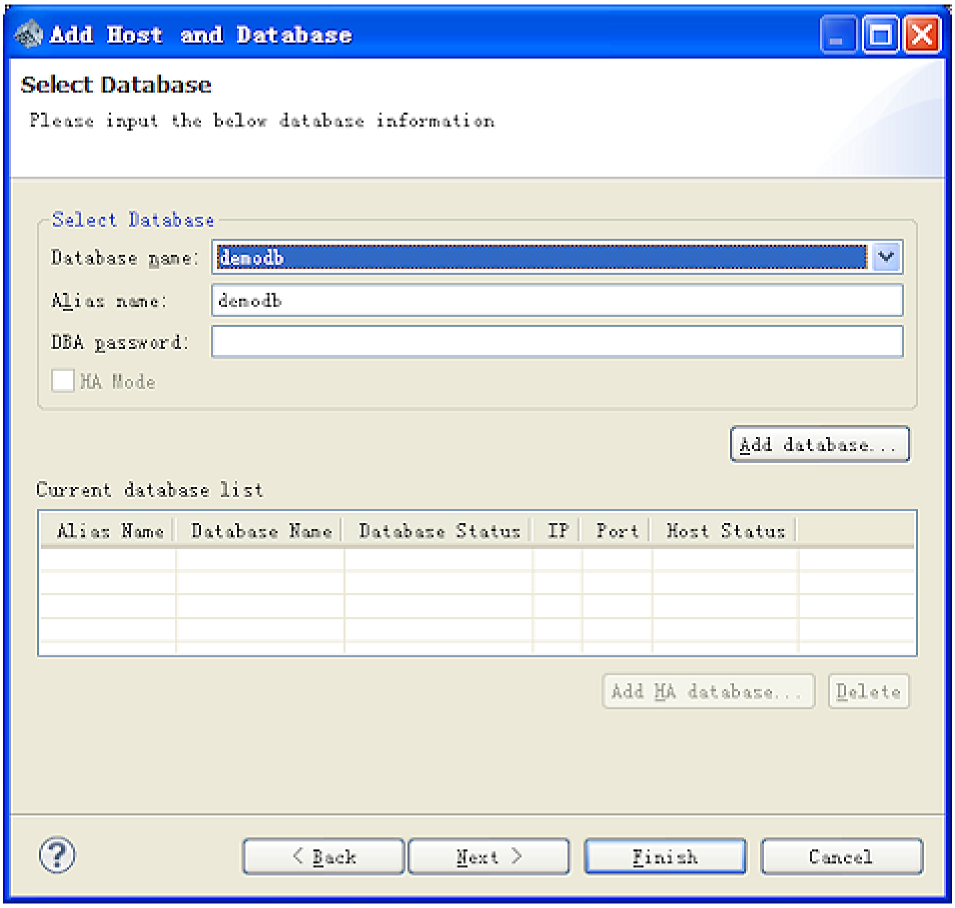
User name: admin

Password:

- **Alias name** : Enter an alias in the host information. If you skip this step, the host address is used.
- **IP address** : Enter the IP address of the host to monitor.
- **Port** : Enter the **cm_port** of CUBRID Manager server installed in the host. You can check **cm_port** value in **$CUBRID/conf/cm.conf** file.
- **Password** : Enter the password for a CUBRID Manager administrator.
- **Connect** : Verify whether it is possible to access with the entered host information. You can move to the next step only if the connection test succeeds.

Step 2: Select Database

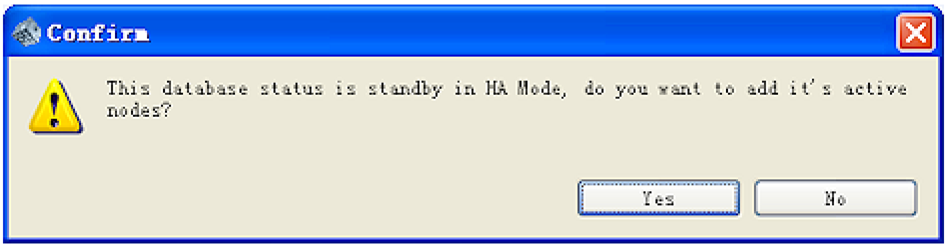
Select one of the databases you wish to monitor which are installed in the host.



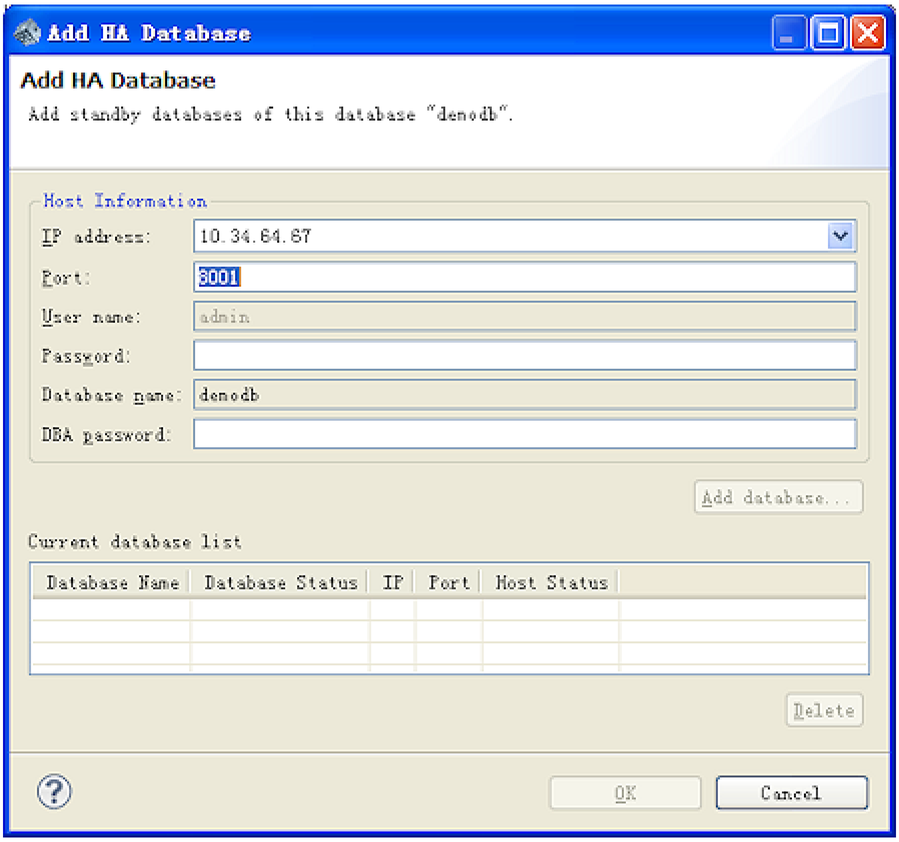
- **Alias name** : Enter an alias for the database information. If you skip this step, the database name is used.
- **DBA password** : Enter the DBA password of the database.
- **HA Mode** : It is automatically selected when the database is configured in HA mode (ha_mode=on).
- **Add database** : Verify whether the database information entered can access the database, and add it to the list of databases. If HA is configured in the database, a window that enables you to detect the rest of HA-configured database and add them will appear (see Step 2-1: Add HA Database).
- **Add HA database** : If HA is configured in the database, you can add the rest of HA-configured databases.
- **Delete** : Delete information registered in the list of databases.

Step 2-1: Add HA Database

If you click the [Add HA database] button in the step 2, you can add the rest of HA configured nodes. If you click [Yes] in the window that confirms whether to add standby node, you can add the host information and database information of the standby node in the [Add HA database] window.



This database status is standby in HA Mode, do you want to add it's active nodes?



Add HA Database

Add standby databases of this database "demodb".

Host Information

IP address: 10.34.64.67

Port: 3001

User name: admin

Password:

Database name: demodb

DBA password:

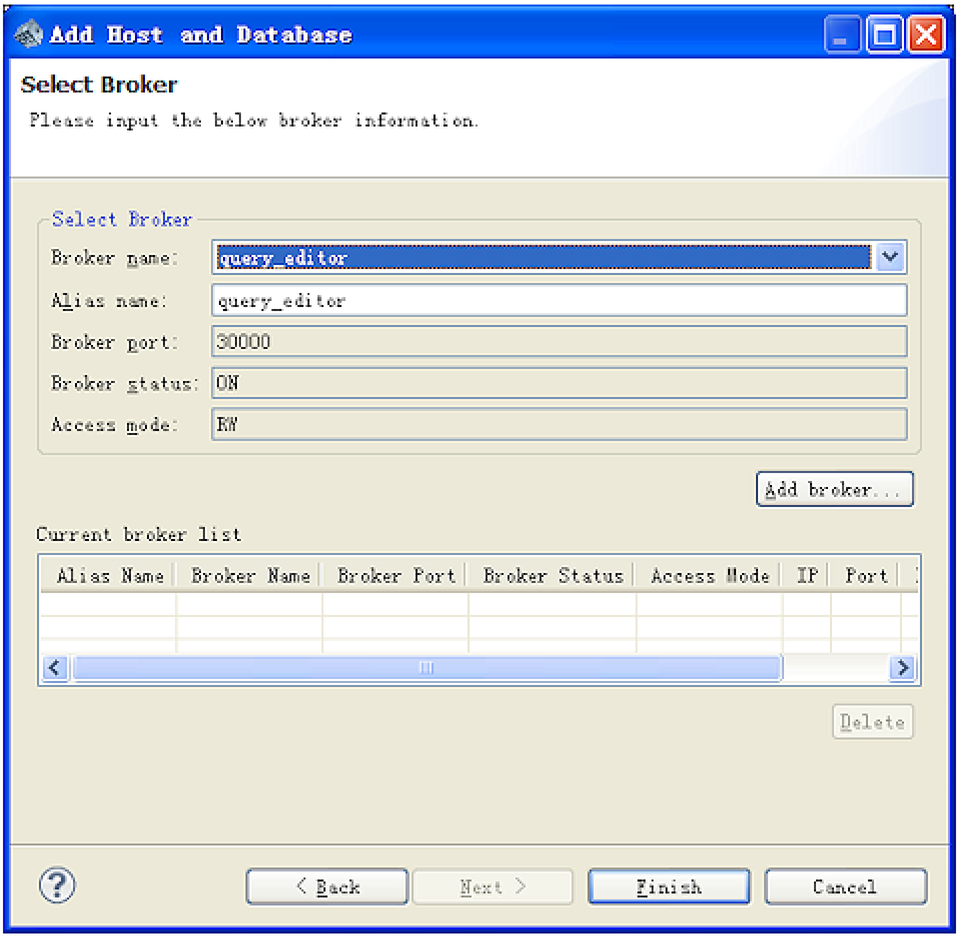
Current database list

Database Name	Database Status	IP	Port	Host Status

- **Port** : Enter the **cm_port** of CUBRID Manager server installed in the host. You can check **cm_port** in **$CUBRID/conf/cm.conf** file.
- **Password** : Enter the password for a CUBRID Manager administrator.
- **DBA password** : Enter the DBA password of the database.
- **Add database** : Verify whether the database information entered can access the database, and add it to the list of databases.
- **Delete** : Delete information registered in the list of databases.

Step 3: Select Broker

You can select the broker running in the host, and add it as a target to be monitored.



Add Host and Database

Select Broker

Please input the below broker information.

Select Broker

Broker name: query_editor

Alias name: query_editor

Broker port: 30000

Broker status: ON

Access mode: RW

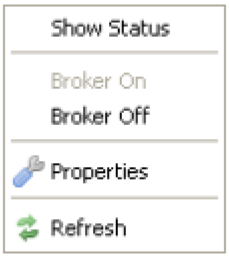
Alias Name	Broker Name	Broker Port	Broker Status	Access Mode	IP	Port

- **Alias name** : Enter an alias of the broker. If you skip this step, the broker name is used.
- **Add broker** : Add selected broker in the list of brokers.
- **Delete** : Delete the registered information in the list of brokers.

Note Dashboard is supported in CUBRID 2008 R3.1 or later.

Broker Function

A Broker may contain several individual Brokers. For each Broker, you need to set a unique name, port, and shared memory ID.

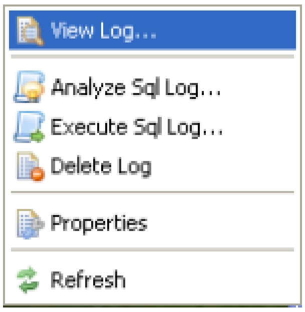


You can check the status, edit the properties, or start/stop each Broker.

If the Broker is running, it is displayed as ⚙ icon; if it has stopped, it is displayed as ⚙ icon in the navigation tree.

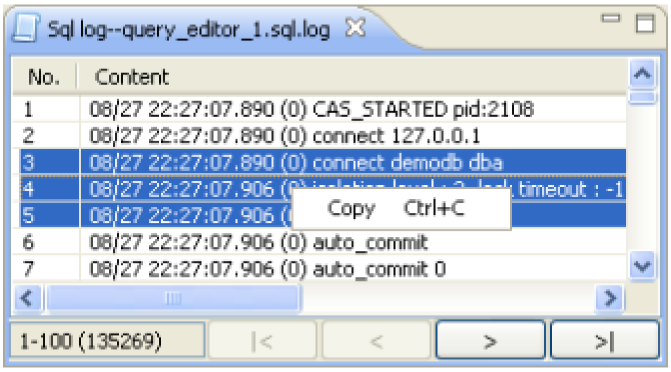
SQL Logs

All the executed queries are stored in the log file when the SQL_LOG parameter of the Broker is ON. This log file can be analyzed and re-executed with the CUBRID Manager.



View Log

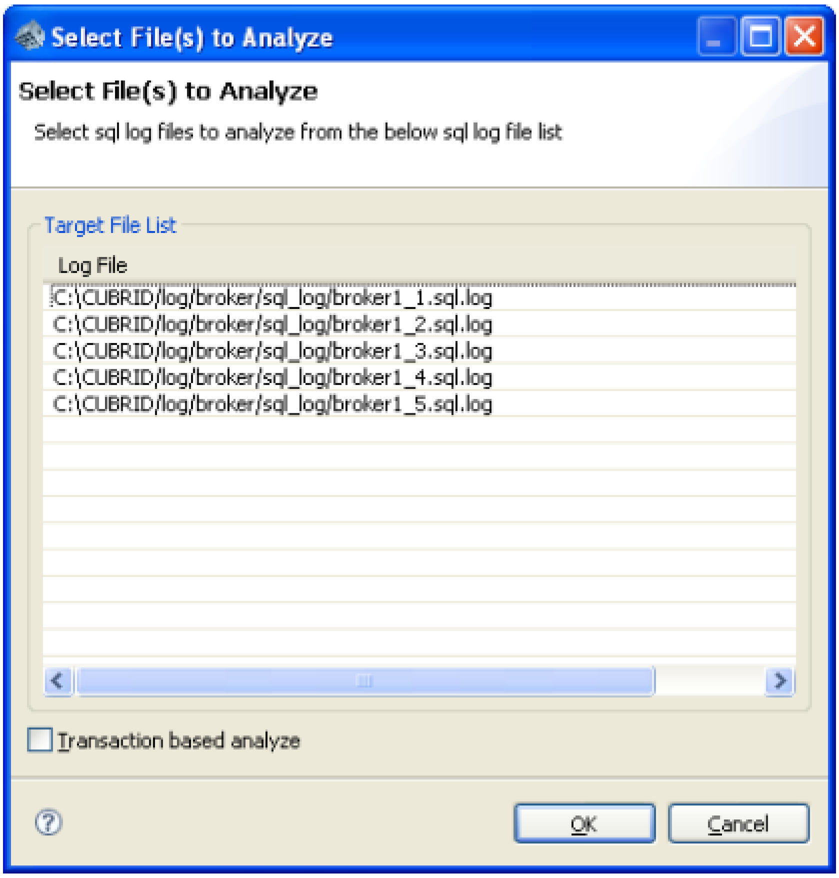
This function reads the SQL log stored in the selected SQL log file and displays 100 lines at a time. It also provides a function for selecting and copying a specific area of the log information.



No.	Content
1	08/27 22:27:07.890 (0) CAS_STARTED pid:2108
2	08/27 22:27:07.890 (0) connect 127.0.0.1
3	08/27 22:27:07.890 (0) connect demodb dba
4	08/27 22:27:07.906 (timeout : -1
5	08/27 22:27:07.906 (
6	08/27 22:27:07.906 (0) auto_commit
7	08/27 22:27:07.906 (0) auto_commit 0

Analyze Log

When [Analyze Sql Log] is selected, the [Select File(s) to Analyze] dialog box appears. You can select which Broker's SQL log is to be analyzed. When the [Transaction based analyze] check box is selected, the log is analyzed for each transaction; otherwise, they are analyzed for each query.



Select File(s) to Analyze

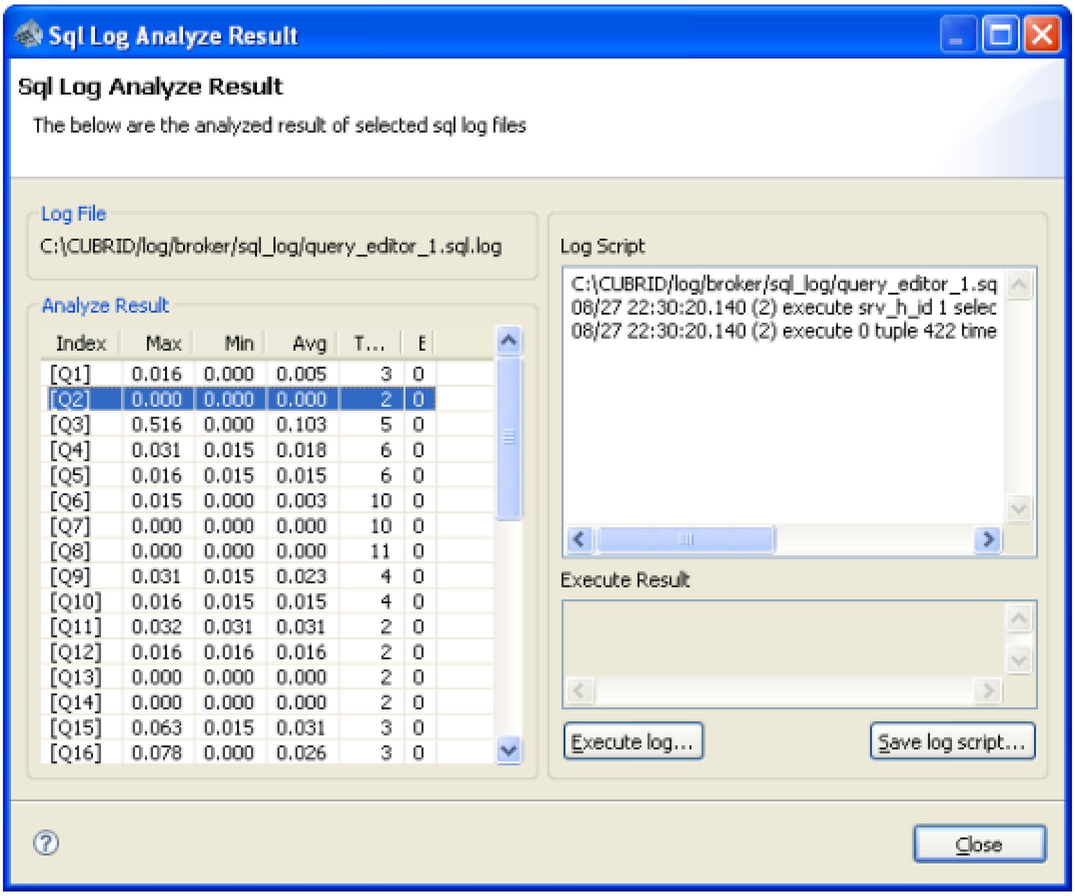
Select sql log files to analyze from the below sql log file list

Target File List

Log File
C:\CUBRID/log/broker/sql_log/broker1_1.sql.log
C:\CUBRID/log/broker/sql_log/broker1_2.sql.log
C:\CUBRID/log/broker/sql_log/broker1_3.sql.log
C:\CUBRID/log/broker/sql_log/broker1_4.sql.log
C:\CUBRID/log/broker/sql_log/broker1_5.sql.log

□ Transaction based analyze

If you select the SQL log file you want and then [OK] in the [Select File(s) to Analyze] dialog box, the [Sql Log Analyze Result] dialog box which shows the results appears.



Sql Log Analyze Result

The below are the analyzed result of selected sql log files

C:\CUBRID/log/broker/sql_log/query_editor_1.sql.log

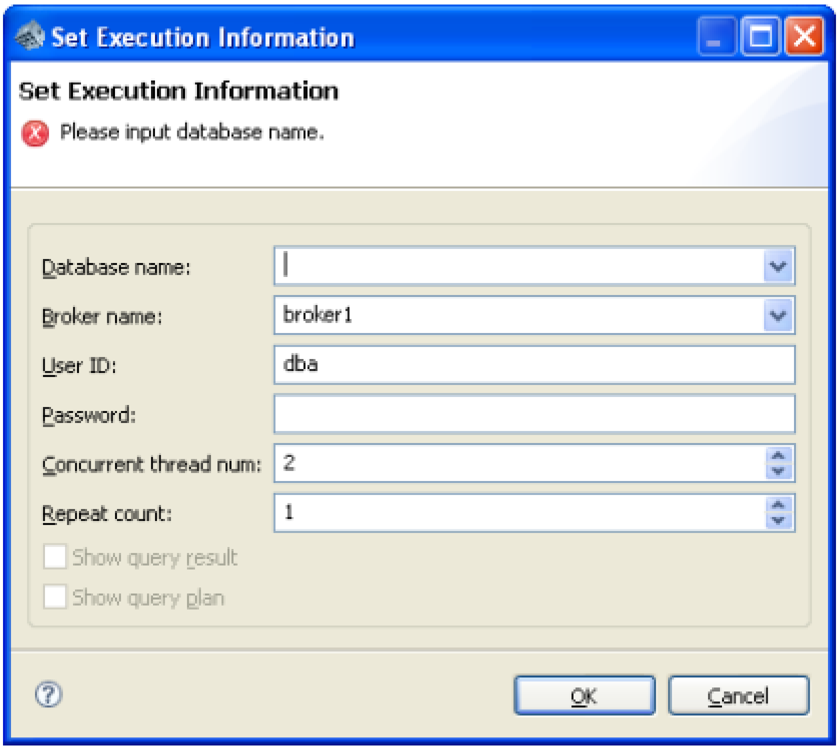
Index	Max	Min	Avg	T...	E
[Q1]	0.016	0.000	0.005	3	0
[Q2]	0.000	0.000	0.000	2	0
[Q3]	0.516	0.000	0.103	5	0
[Q4]	0.031	0.015	0.018	6	0
[Q5]	0.016	0.015	0.015	6	0
[Q6]	0.015	0.000	0.003	10	0
[Q7]	0.000	0.000	0.000	10	0
[Q8]	0.000	0.000	0.000	11	0
[Q9]	0.031	0.015	0.023	4	0
[Q10]	0.016	0.015	0.015	4	0
[Q11]	0.032	0.031	0.031	2	0
[Q12]	0.016	0.016	0.016	2	0
[Q13]	0.000	0.000	0.000	2	0
[Q14]	0.000	0.000	0.000	2	0
[Q15]	0.063	0.015	0.031	3	0
[Q16]	0.078	0.000	0.026	3	0

```
C:\CUBRID/log/broker/sql_log/query_editor_1.sq
08/27 22:30:20.140 (2) execute srv_h_id 1 selec
08/27 22:30:20.140 (2) execute 0 tuple 422 time
```

- **Log File** : Displays the file name and the directory path of the SQL log file of the target Broker.
- **Analyze Result** : Shows log analysis results. If [Transaction based analyze] is selected, each transaction's execution time is displayed; otherwise, analysis information (e.g. total number of executions, number of errors, maximum execution time, minimum execution time, average execution time) about each query is displayed. When you click a column in the analyze result section, the results are sorted by the value of that column.
- **Log Script** : Shows the log script for the analysis results.
- **Execute Result** : Shows the results of the log execution.
- **Execute log** : Re-executes the SQL log in the log script. You can tune queries and correct errors by modifying and re-executing log queries.
- **Save log script** : Saves the log script in a file.

Execute Log

If you select [Execute log], the [Set Execution Information] dialog box appears, in which you can configure the environment.



Set Execution Information

Please input database name.

Database name:	
Broker name:	broker1
User ID:	dba
Password:	
Concurrent thread num:	2
Repeat count:	1

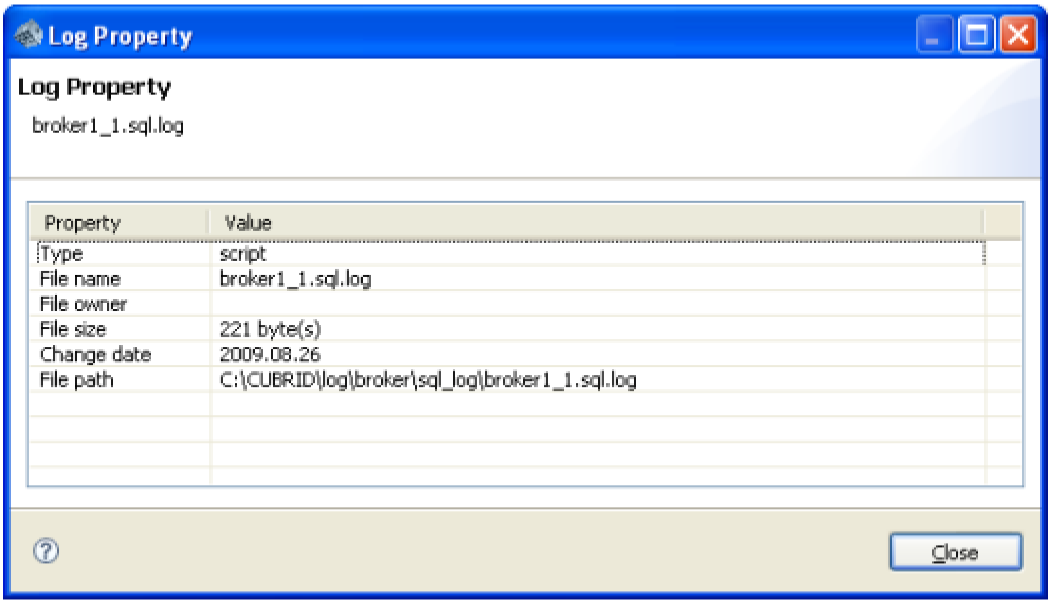
□ Show query result

□ Show query plan

- **Database name** : Select the database for which the log is to be re-executed.
- **Broker name** : Select the Broker for which the log is to be re-executed.
- **User ID/Password** : Enter the ID and password of the database user that re-executes the log.
- **Concurrent thread num** : Specify the number of times to execute the log query concurrently. When the log is re-executed, threads are created as many as this number, and the same query is executed concurrently. This function is useful when you want to check how a query is executed in a multi-user environment.
- **Repeat count** : Specify the number of times to execute a query repeatedly.
- **Show query result** : Shows the results of the query execution.
- **Show query plan** : This option is valid only when [Show query result] is selected.

Log Property

Provides information of the selected log file.



Log Property

broker1_1.sql.log

Property	Value
Type	script
File name	broker1_1.sql.log
File owner	
File size	221 byte(s)
Change date	2009.08.26
File path	C:\CUBRID\log\broker\sql_log\broker1_1.sql.log

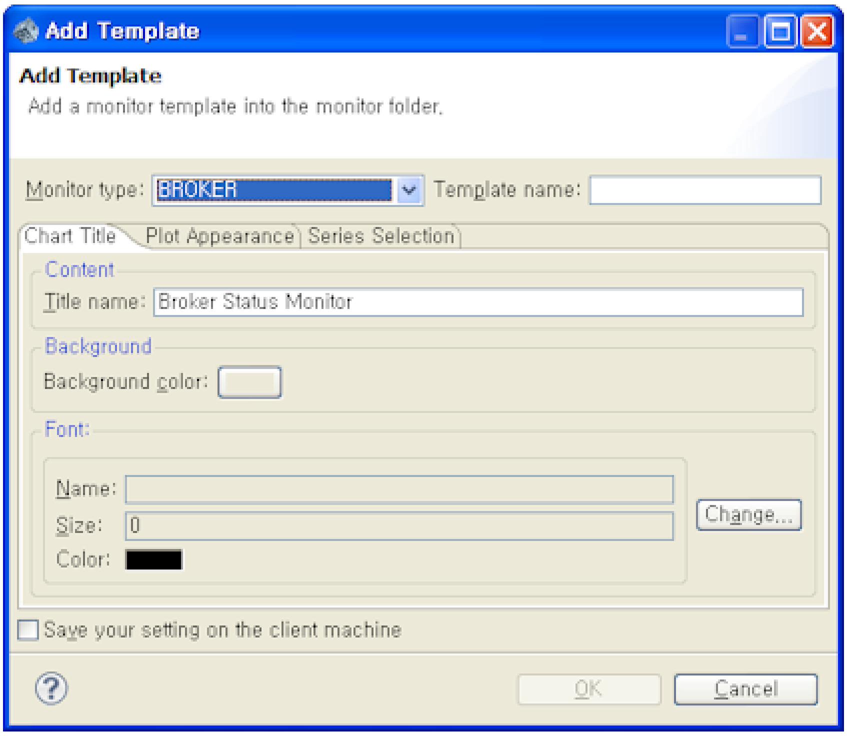
Status Monitor

The Status monitor provides the following functions:

Adding the Status Monitor

In the navigation tree, right-click the [Status monitor] and select [Add Template]; the [Add Template] dialog will appear. In the dialog, you can create a monitoring template by selecting monitoring target, item, or chart property etc. Note that the options in the [Add Template] may be different from the picture below if you are connected to the version which is earlier than 3.0.

- **[Monitor type]** : Select either **DATABASE** or **BROKER**.
- **[Template name]** : Enter the name of status monitoring template to be created.
- **[Chart Title] Tab** : Specify a title (to be displayed at the top of a chart), background color, and font.
- **[Plot Appearance] Tab** : Set colors for a chart title, horizontal/vertical asix, or grid line.
- **[Series Selection] Tab** : Select a target to be monitored.


Add Template

Add Template

Add a monitor template into the monitor folder.

Monitor type: BROKER

Template name:

Content

Title name: Broker Status Monitor

Background

Background color:

Font:

Name:

Size: 0

Color:

□ Save your setting on the client machine

Monitoring Database Status

If you select **DATABASE** in the [Monitor type], you can select a desired monitoring item of database in the [Series Selection] and then add it to a template.

Monitoring database shows statistics data executed by the CUBRID server by using the **cubrid statdump** utility. It monitors items such as log pages, indexes, queries, or transactions. For more information, see Outputting Statistics Information of Server.

Monitoring Broker Status

If you select **BROKER** in the [Monitor type], you can select a desired monitoring item of Broker in the [Serier Selection] and then add it to a template.

There are seven types of Broker monitoring item: **SESSION**, **ACTIVE_SESSION**, **ERR_Q**, **LONG_Q**, **LONG_T**, **QPS**, **RPS**, and **TPS**. For more information, see ecking the Broker Status. All other object monitoring values except for **ACTIVE_SESSION** of the Broker represent the number of occurrences that happen during the last sampling interval.

- **SESSION** : The number of applications (CAS) connected to the Broker. In other words, it means the number of applications, not IDLE status. This value cannot exceeds the value specified in the **MAX_NUM_APPL_SERVER** parameter.
- **ACTIVE_SESSION** : The number of active applications (CAS) with **BUSY** status. In other words, it means the number of transactions that are currently executed.
- **ERR_Q** : The number of errors occurred
- **LONG_Q** : The number of long queries
- **LONG_T** : The number of long transactions
- **QPS** : The number of queries processed per second by the Broker
- **RPS** : The number of requests per second received by the Broker
- **TPS** : The number of transactions processed per second by the Broker

Template Information

- **Name** : Enter the name of the Status monitoring template to be created.
- **Description** : Enter the description of the template to be created.
- **Sampling term (Second)** : Specify the interval (in seconds) at which the target object is to be monitored.
- **Target database** : Select the target database. It is activated when the monitoring target object is related to the Database Server.

Editing the Status Monitor

You can edit in the same interface as with [Add Template] by right-clicking the Status monitor template to edit and then selecting [Edit Template].

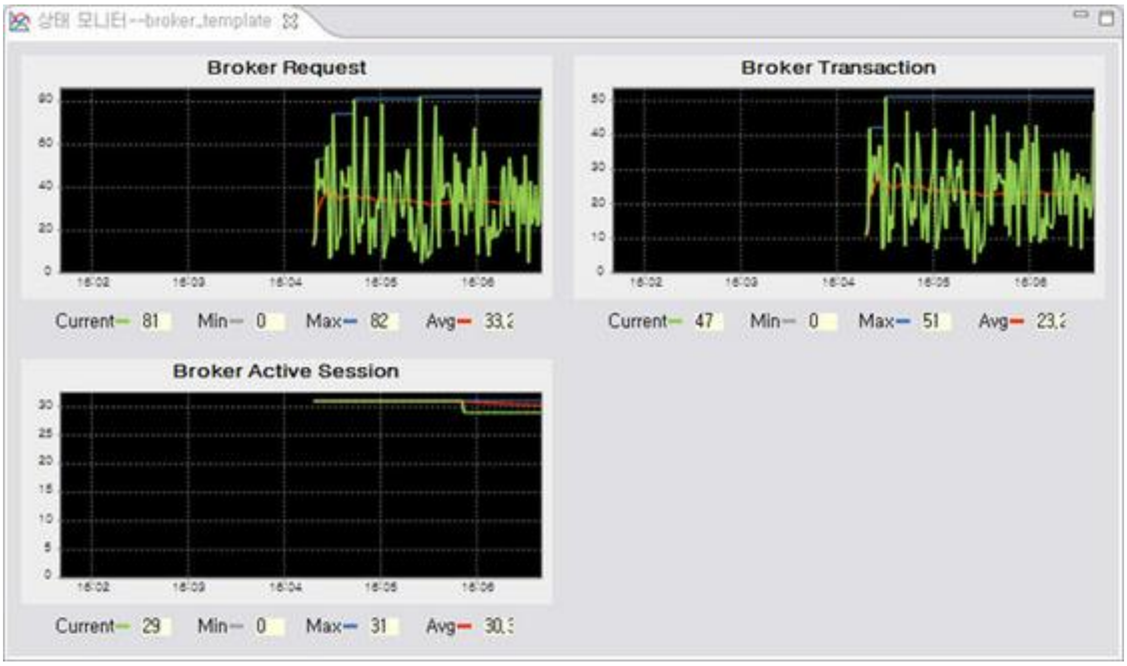
Deleting the Status Monitor

To delete a registered Status monitor template, right-click the template to delete and then select [Delete Template].

Executing the Status Monitor

If you right-click the Status Monitor Template to execute and then select [Status Monitor], the Status Monitor pane appears where you can monitor the Server and the Broker status depending on the selected template. The Status Monitor shows separate charts for each collected item, which consists of the current, minimum, maximum and average values.

The Status Monitor, represented in separate views, can be viewed together with other view interfaces. It can also be viewed outside the CUBRID Manager, which makes it convenient to perform jobs and monitoring at the same time in multi-monitor environment.

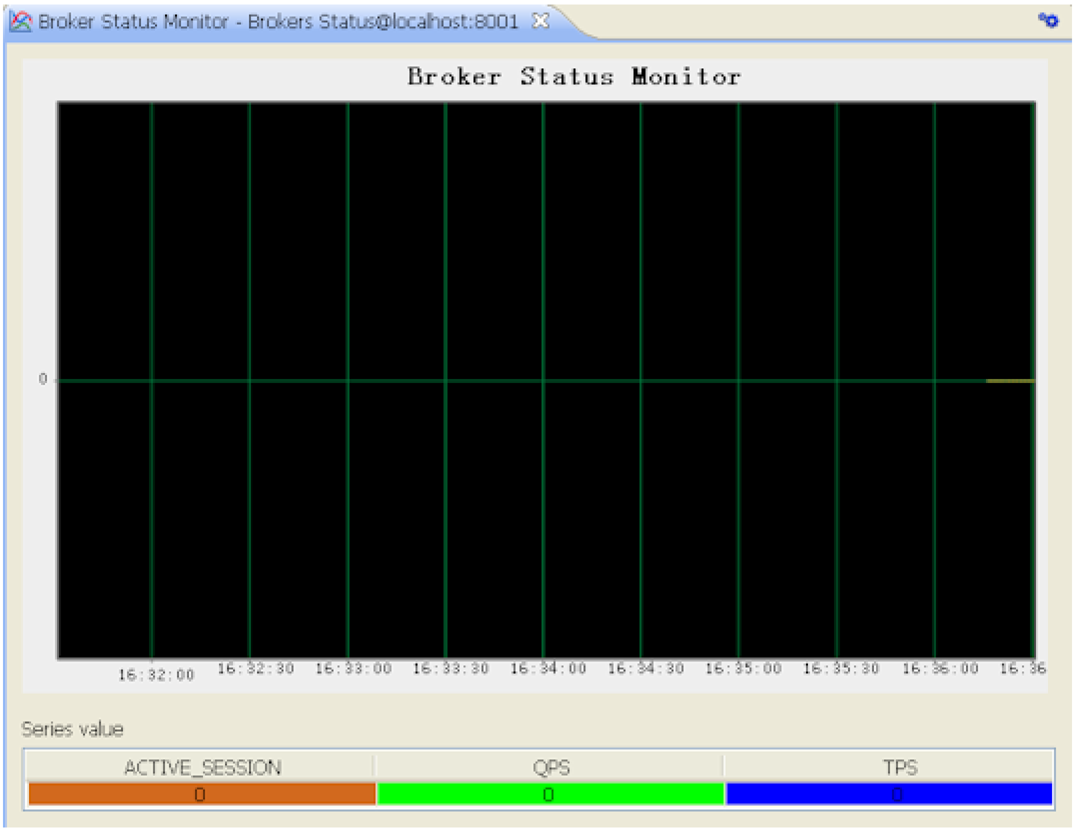


Integrated Status Monitoring

This feature is supported only when you connect to a database of CUBRID 2008 R2.2 or higher.

Broker Status Monitoring

You can monitor items related to the broker by combining them into a single chart. You can set the chart property by clicking the [Configuration] button in the bar at the top of the broker status monitoring pane.



Database Status Monitoring

This feature is supported only when you connect to a database of CUBRID 2008 R2.2 or higher. You can monitor items related to the database by combining them into a single chart.

The items that can be monitored are the same as the item displayed when the cubrid statdump utility is executed. You can set the chart property by clicking the [Configuration ⚙] button in the bar at the top of the database status monitoring pane.

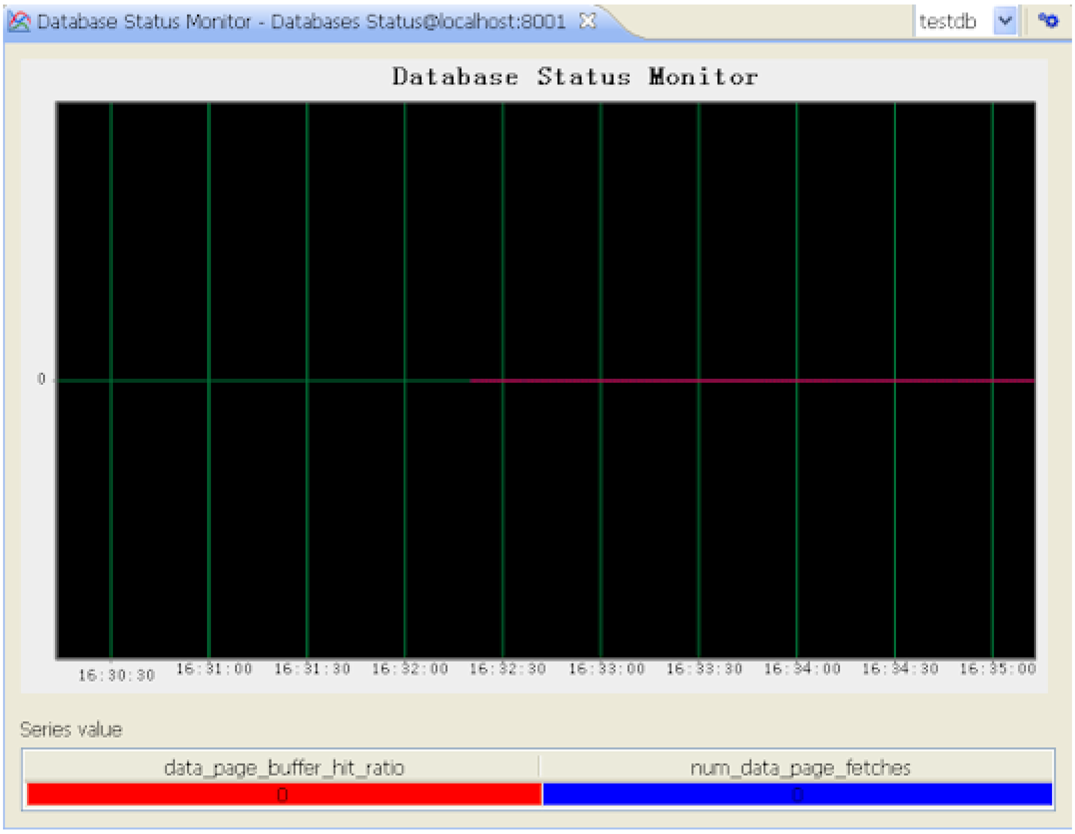


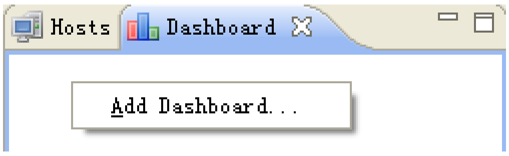
Chart Properties Setting

You can set the chart title, image and items to be monitored in the chart setting wizard.

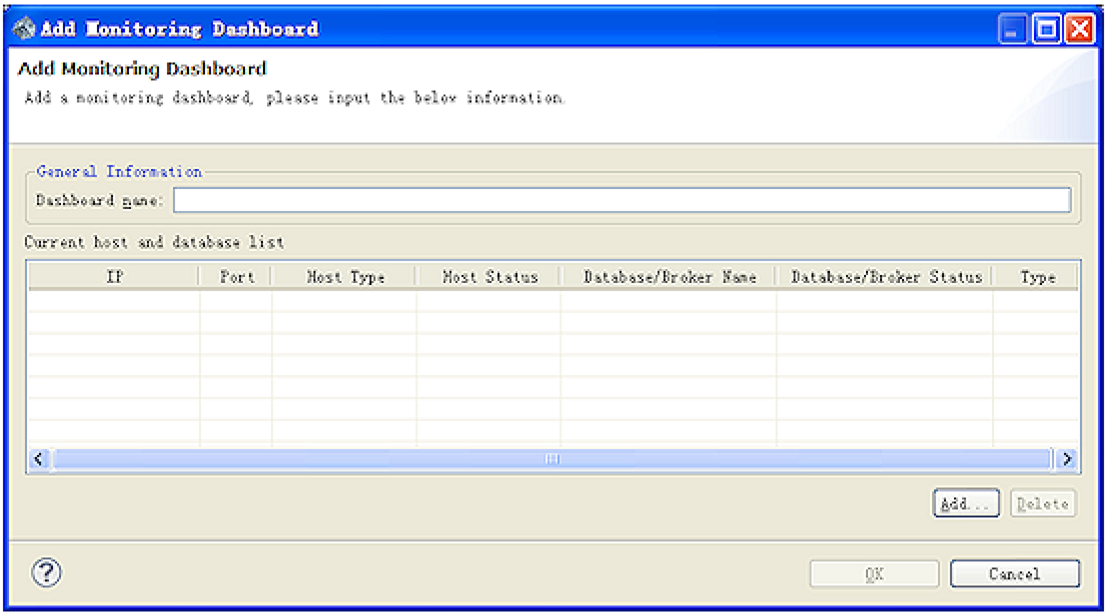
Dashboard

Add Dashboard

[Dashboard] tap appears next to [Hosts] tab when you select [Tool] > [Open Dashboard Explorer] in menu.



Right-click on the [Dashboard] tab and select [Add Dashboard]. Then, [Add Monitoring Dashboard] dialog will appear. Enter the dashboard name in [Dashboard name] field and click [Add] button.


Add Monitoring Dashboard

Add Monitoring Dashboard

Add a monitoring dashboard, please input the below information.

General Information

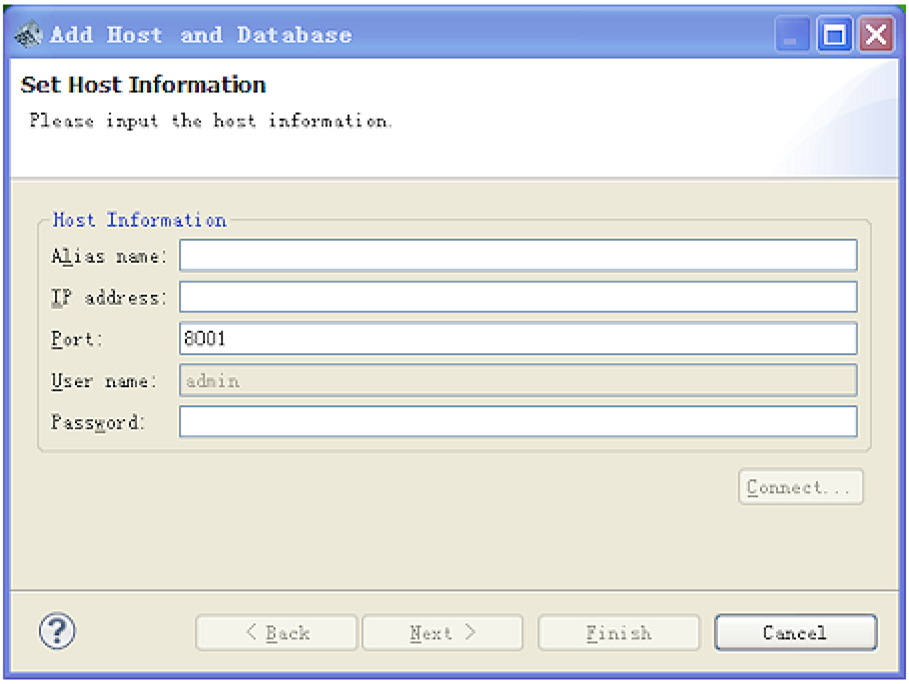
Dashboard name:

Current host and database list

IP	Port	Host Type	Host Status	Database/Broker Name	Database/Broker Status	Type

Step 1: Set Host Information

Enter the host information of the database server you want to monitor. At this time, The CUBRID manager server must be running in the host.



Add Host and Database

Set Host Information

Please input the host information.

Host Information

Alias name:

IP address:

Port: 8001

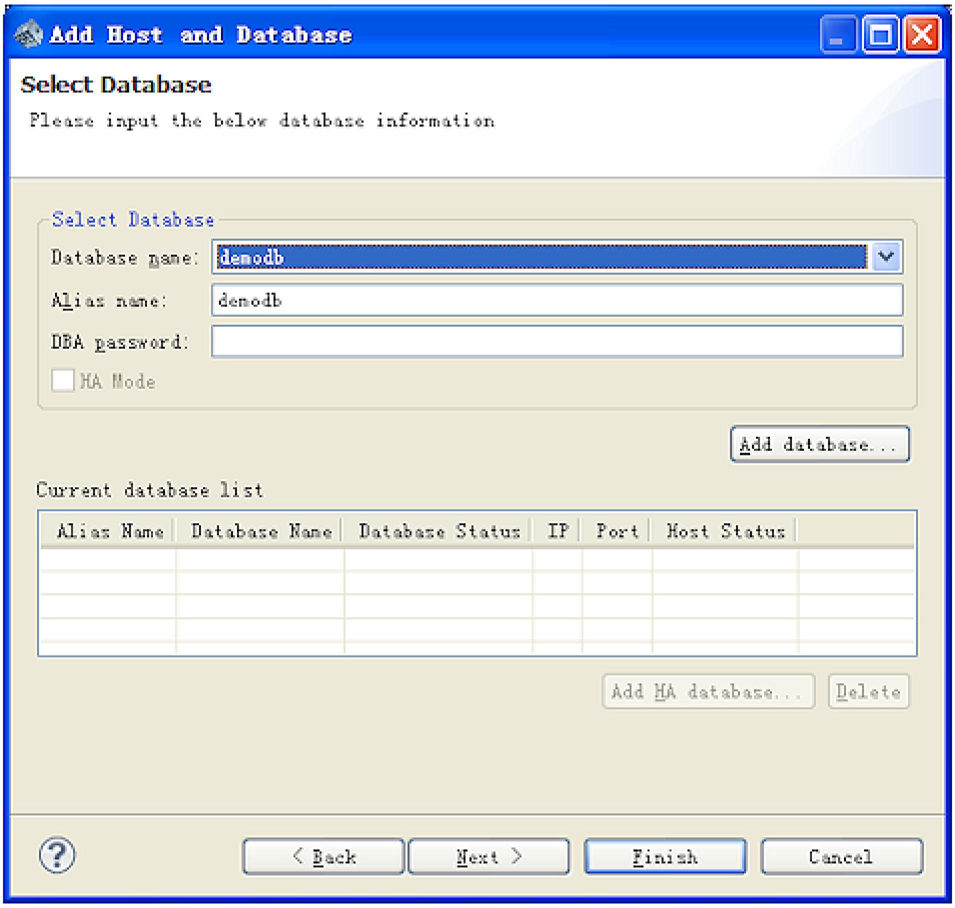
User name: admin

Password:

- **Alias name** : Enter an alias in the host information. If you skip this step, the host address is used.
- **IP address** : Enter the IP address of the host to monitor.
- **Port** : Enter the **cm_port** of CUBRID Manager server installed in the host. You can check **cm_port** value in **$CUBRID/conf/cm.conf** file.
- **Password** : Enter the password for a CUBRID Manager administrator.
- **Connect** : Verify whether it is possible to access with the entered host information. You can move to the next step only if the connection test succeeds.

Step 2: Select Database

Select one of the databases you wish to monitor which are installed in the host.



Database name: demodb

Alias name: demodb

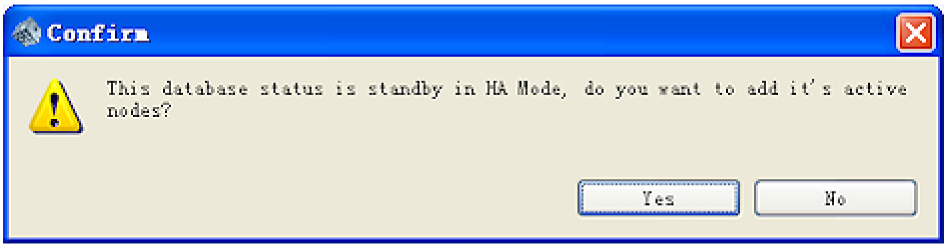
DBA password:

□ HA Mode

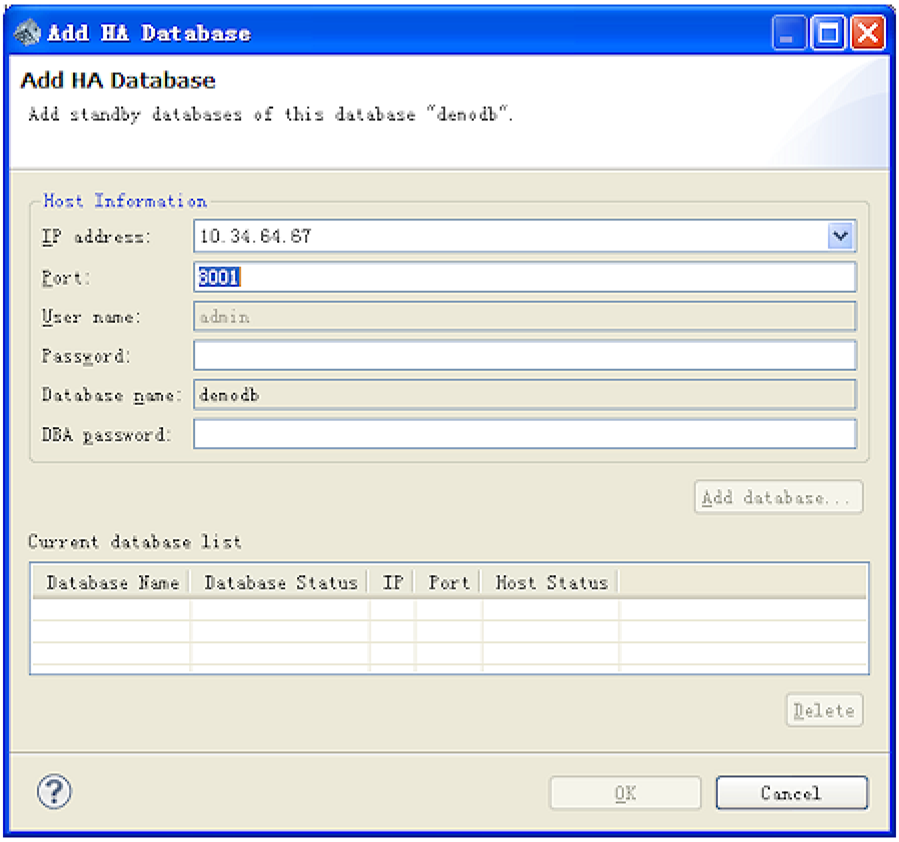
- **Alias name** : Enter an alias for the database information. If you skip this step, the database name is used.
- **DBA password** : Enter the DBA password of the database.
- **HA Mode** : It is automatically selected when the database is configured in HA mode (ha_mode=on).
- **Add database** : Verify whether the database information entered can access the database, and add it to the list of databases. If HA is configured in the database, a window that enables you to detect the rest of HA-configured database and add them will appear (see Step 2-1: Add HA Database).
- **Add HA database** : If HA is configured in the database, you can add the rest of HA-configured databases.
- **Delete** : Delete information registered in the list of databases.

Step 2-1: Add HA Database

If you click the [Add HA database] button in the step 2, you can add the rest of HA configured nodes. If you click [Yes] in the window that confirms whether to add standby node, you can add the host information and database information of the standby node in the [Add HA database] window.



This database status is standby in HA Mode, do you want to add it's active nodes?



Add HA Database

Add standby databases of this database "demodb".

Host Information

IP address: 10.34.64.67

Port: 8001

User name: admin

Password:

Database name: demodb

DBA password:

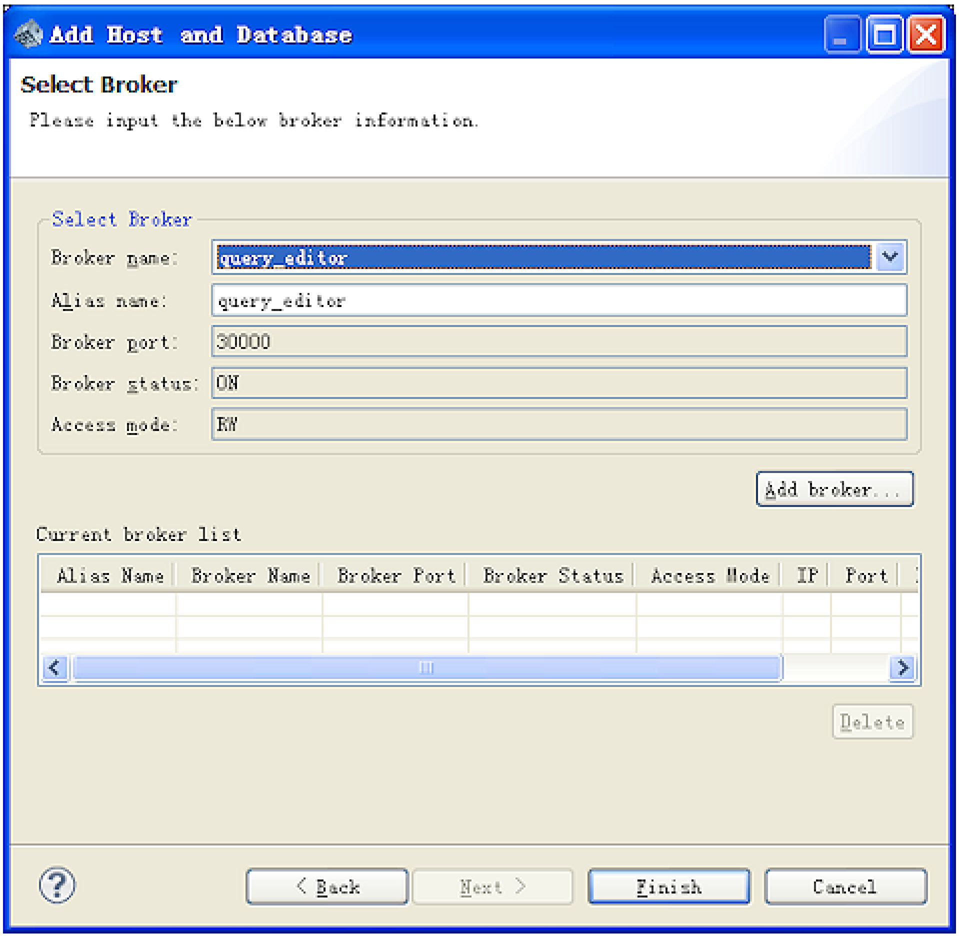
Current database list

Database Name	Database Status	IP	Port	Host Status

- **Port** : Enter the **cm_port** of CUBRID Manager server installed in the host. You can check **cm_port** in **$CUBRID/conf/cm.conf** file.
- **Password** : Enter the password for a CUBRID Manager administrator.
- **DBA password** : Enter the DBA password of the database.
- **Add database** : Verify whether the database information entered can access the database, and add it to the list of databases.
- **Delete** : Delete information registered in the list of databases.

Step 3: Select Broker

You can select the broker running in the host, and add it as a target to be monitored.



Add Host and Database

Select Broker

Please input the below broker information.

Select Broker

Broker name: query_editor

Alias name: query_editor

Broker port: 30000

Broker status: ON

Access mode: RW

Current broker list

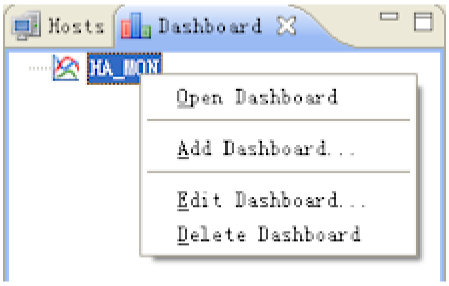
Alias Name	Broker Name	Broker Port	Broker Status	Access Mode	IP	Port

- **Alias name** : Enter an alias of the broker. If you skip this step, the broker name is used.
- **Add broker** : Add selected broker in the list of brokers.
- **Delete** : Delete the registered information in the list of brokers.

Note Dashboard is supported in CUBRID 2008 R3.1 or later.

Manage Dashboard

A popup menu to manage dashboard will appear when you right-click on each dashboard,



Open Dashboard

You can open dashboard window to monitor.

Edit Dashboard

You can edit information on host, database, and broker of the dashboard.

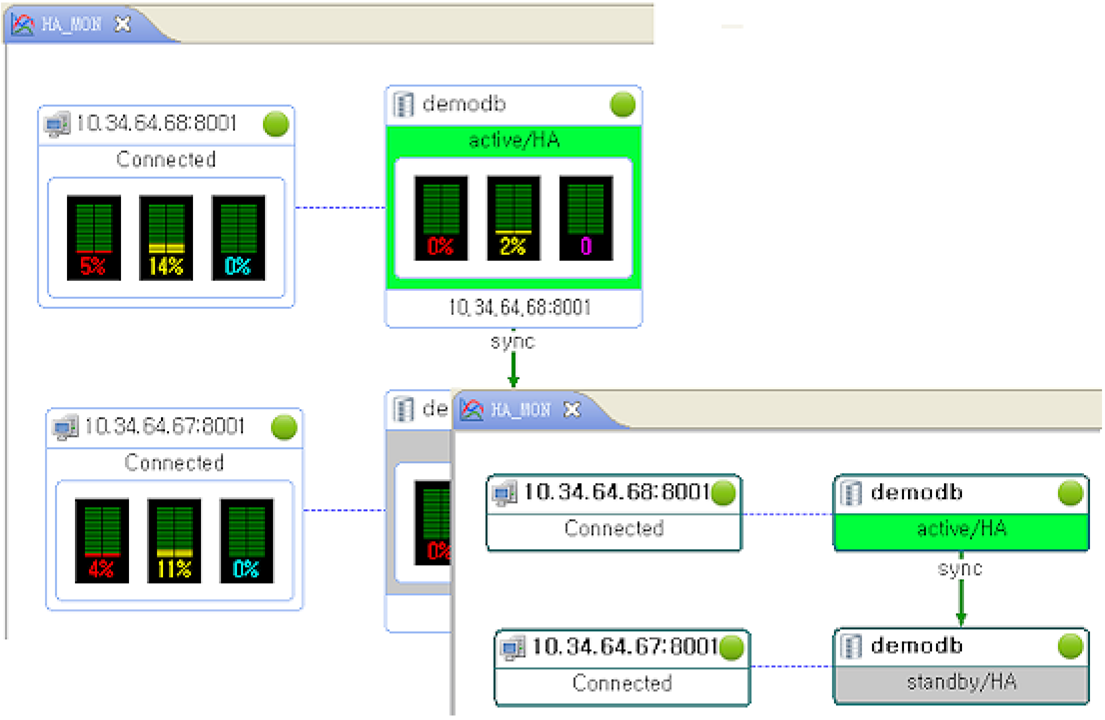
Delete Dashboard

You can delete the dashboard.

Dashboard Window

Manage Dashboard Window

You can use a mouse or keyboard direction key to adjust a target object displayed on the dashboard windows; the adjusted location information is stored in a local directory where CUBRID Manager is installed, and the final location information always will be used. You can also adjust screen ratio by using the <Ctrl> key with mouse scrolling or <+> or <-> keys in the keyboard.

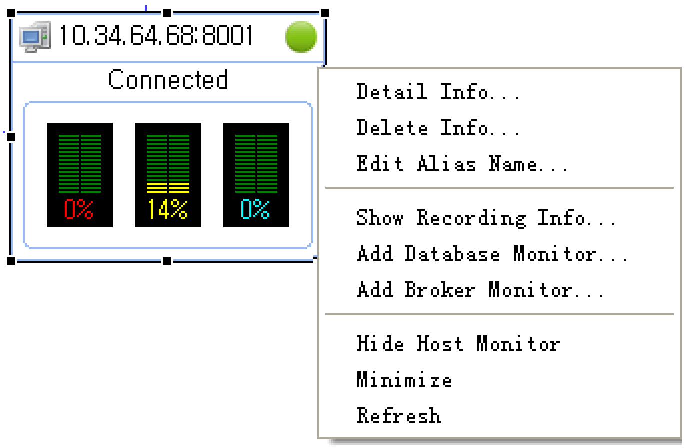


A popup menu will appear when you click right-click on the dashboard window.

- **Select All** : You can select all the objects indicated in the monitoring window to move their location.
- **Add Host Monitor** : You can add new host information that you want to monitor in the monitoring dashboard.
- **Refresh** : You can manually refresh the monitoring dashboard.

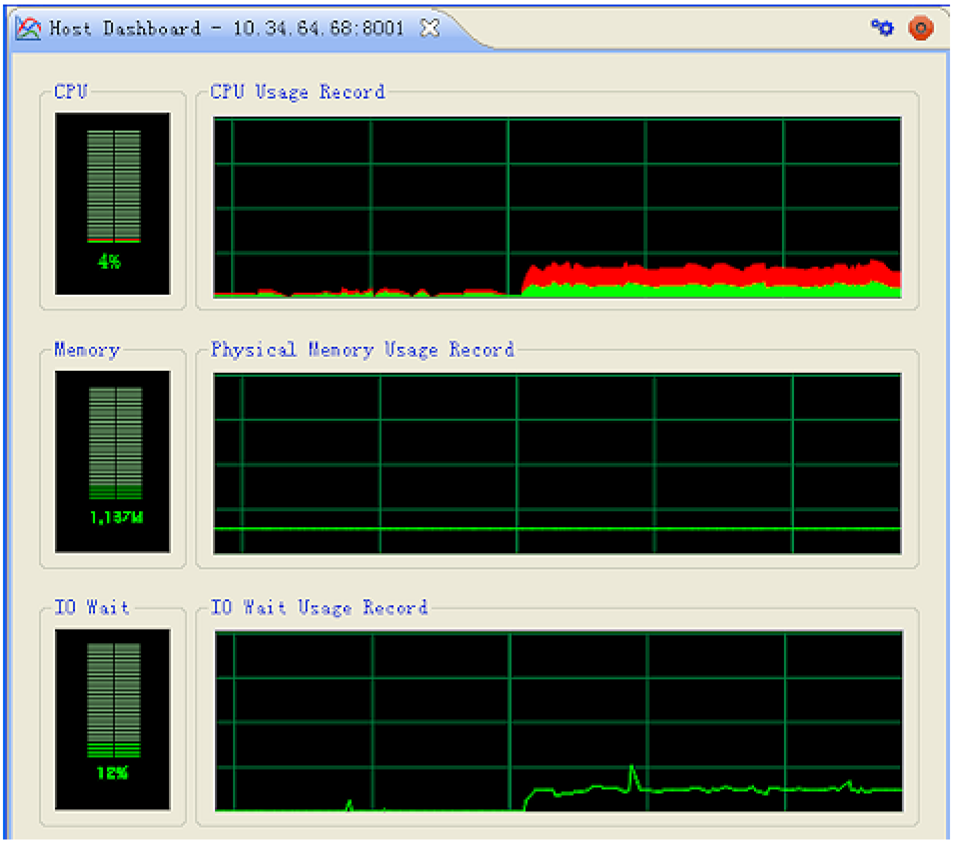
Host Monitor

It monitors the host in which database or broker is running and shows the current value of the CPU, MEMORY, and IO WAIT as percentage. A popup menu where you can configure functions below will appear when you right-click on the host monitor.



View Detailed Information

[View Detailed Information] shows changes with time of CPU, MEMORY, and IO WAIT.



You can configure the chart properties by clicking the icon (⚙) on the upper right in the [View Detailed Information] screen.

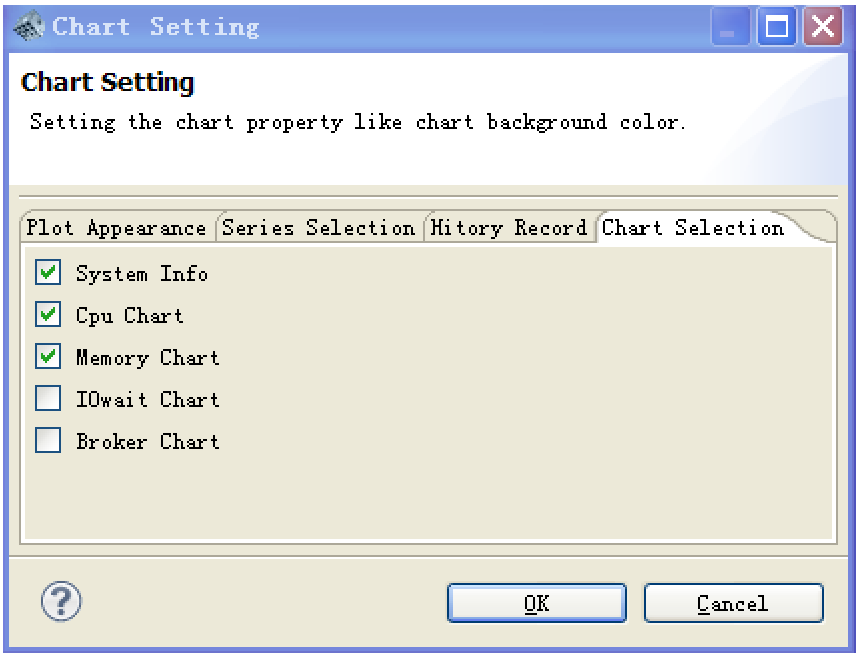


Chart Setting

Setting the chart property like chart background color.

☑ System Info

☑ Cpu Chart

☑ Memory Chart

□ IOwait Chart

□ Broker Chart

- **Plot Appearance** : Configure the colors of cross gridlines and the background of a monitor host chart.
- **Series Selection** : Select items to display in a monitor host chart, and configure the colors and line thickness of the graph.
- **History Record** : Specify a directory where history will be saved when recording is on.
- **Chart Selection** : Select a chart to be monitored.

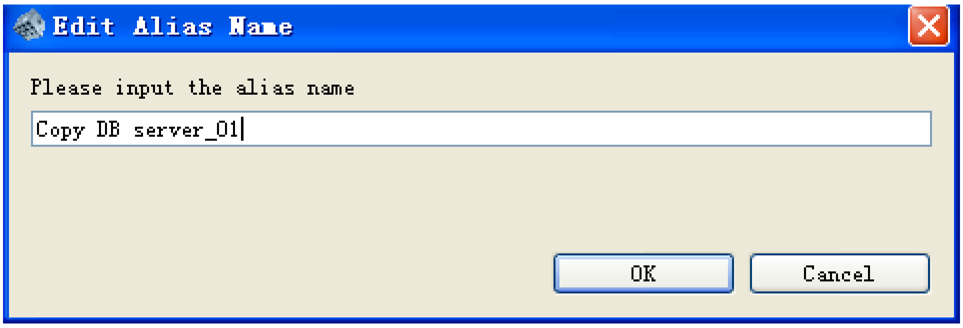
Clicking the start recording icon (●) in the [View Detailed Information] screen will save all statistical data being recorded into a local folder. To stop recording, click stop recording icon (●).

Delete

You can delete information on the host to be monitored, including database and broker information.

Edit Alias Name

You can edit an alias shown in the host monitor window. By default, an IP address of the host is displayed. You can edit this name with a service name that can easily be identified by system operators.

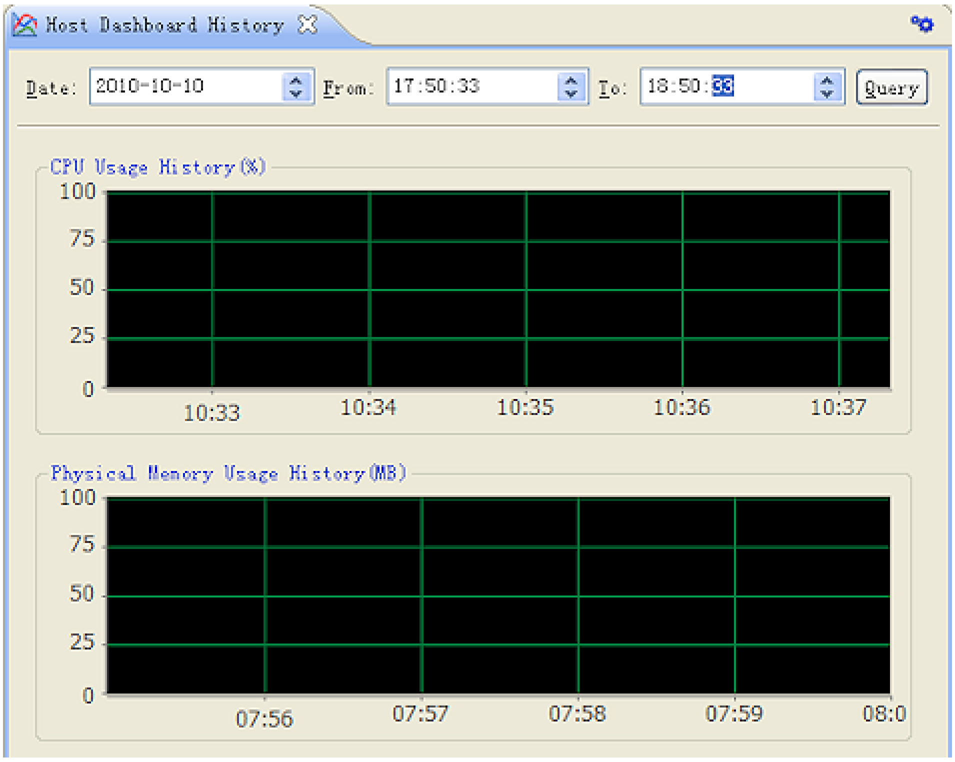


Please input the alias name

Copy DB server_01

View History Record

The function that provides the history record information. You can track trend of a specific time by selecting date and time.



Add Database

You can directly add a database running in the host.

Add Broker

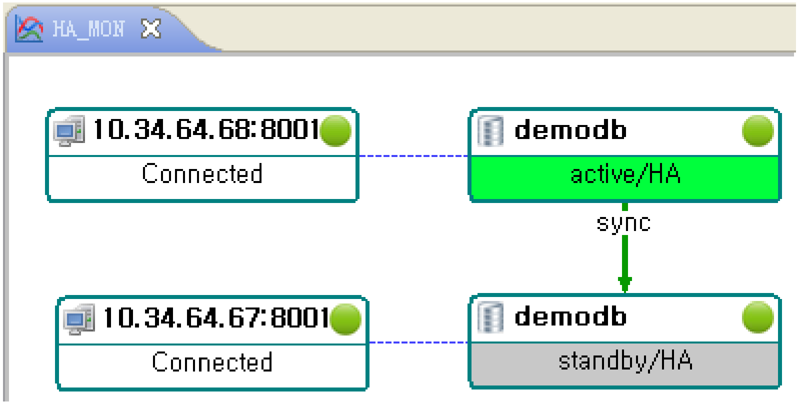
You can directly add a broker running in the host.

Hide Host

You can hide the host monitor in the monitoring dashboard.

Minimize

You can minimize the host monitor window. A host alias and connection status are only shown.

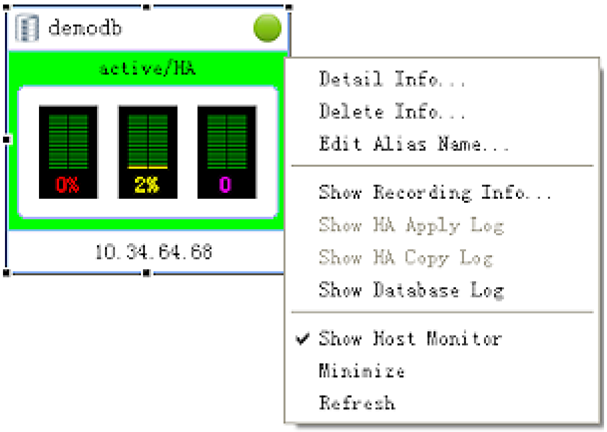


Refresh

You can manually refresh the host monitor.

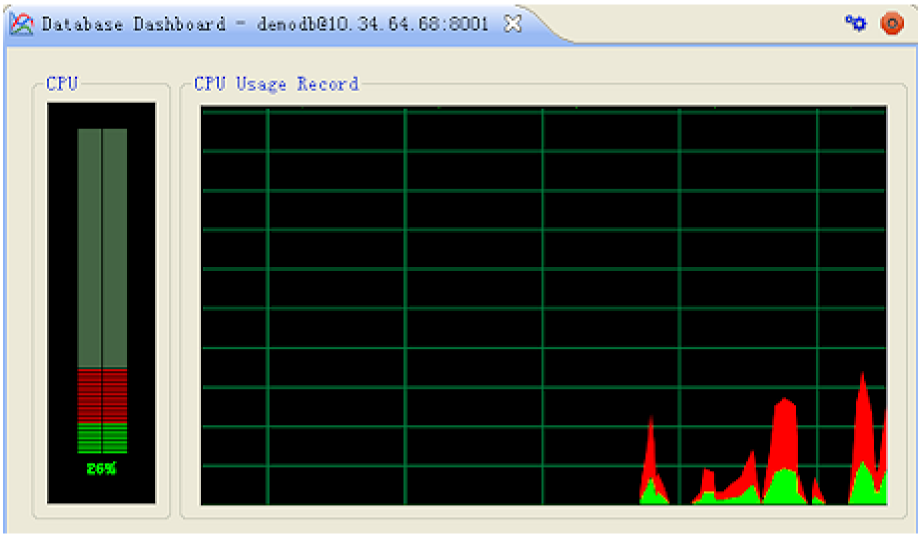
Database Monitor

It monitors the database server which is currently running and shows the current value of CPU, MEMORY, and HA delay (msec) of the database server. A popup menu where you can configure functions below will appear when you right-click on the database monitor.



View Detailed Information

[View Detailed Information] shows changes with time of CPU, MEMORY, and HA delay.



You can configure the chart properties by clicking the icon (⚙) on the upper right in the [View Detailed Information] screen.

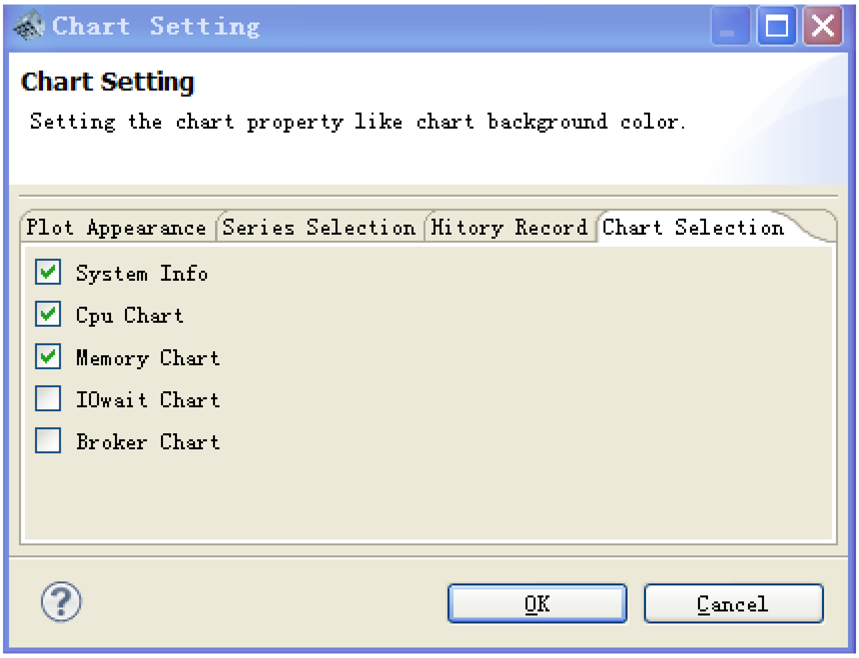


Chart Setting

Setting the chart property like chart background color.

☑ System Info

☑ Cpu Chart

☑ Memory Chart

□ IOwait Chart

□ Broker Chart

- **Plot Appearance** : Configure the colors of cross gridlines and the background of a monitor host chart.
- **Series Selection** : Select items to display in a monitor host chart, and configure the colors and line thickness of the graph.
- **History Record** : Specify a directory where history will be saved when recording is on.
- **Chart Selection** : Select a chart to be monitored.

Clicking the start recording icon (◉) in the [View Detailed Information] screen will save all statistical data being recorded into a local folder. To stop recording, click stop recording icon (●).

Delete

You can delete the database information from host information.

Edit Alias Name

You can edit an alias shown in the database monitor window. By default, a name of the database is displayed. You can edit this name with another name that can easily be identified by system operators.

View History Record

The function that provides the history record information. You can track trend of a specific time by selecting date and time.

View HA Applied Log

This is enabled in the HA standby database server where HA is configured; you can check the log generated upon execution of **cubrid applylogdb** command.

View HA Copied Log

This is enabled in the HA active database server where HA is configured; you can check the log generated upon execution of **cubrid copylogdb** command.

View Database Log

You can check an error log file of database.

View Host

You can check the host information in which the database server is running.

Minimize

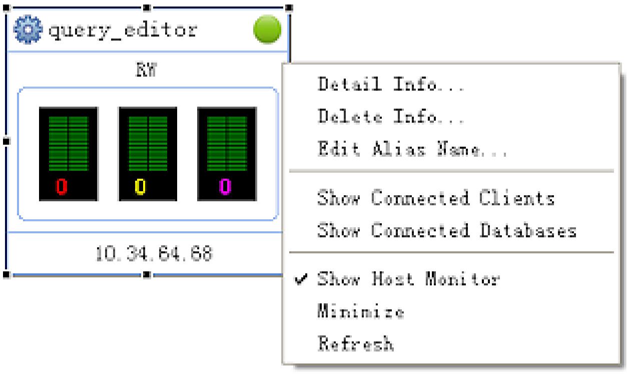
You can minimize the host monitor window. A host alias and connection status are only shown.

Refresh

You can manually refresh the host monitor.

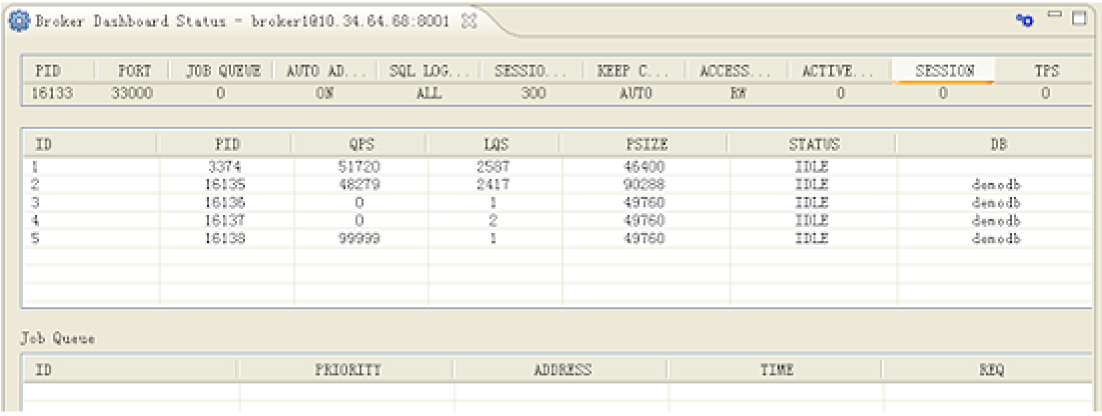
Broker Monitor

It monitors the broker which is currently running and shows the current value of the number of session and active session, including TPS. A popup menu where you can configure functions below will appear when you right-click on the broker monitor.



View Detailed Information

[View Detailed Information] shows values for the broker.



Broker Dashboard Status - broker1@10.34.64.68:8001

PID	PORT	JOB QUEUE	AUTO AD...	SQL LOG...	SESSIO...	KEEP C...	ACCESS...	ACTIVE...	SESSION	TPS
16133	33000	0	ON	ALL	300	AUTO	RW	0	0	0

ID	PID	QPS	LQS	PSIZE	STATUS	DB
1	3374	51720	2587	46400	IDLE	
2	16135	48279	2417	90288	IDLE	demodb
3	16136	0	1	49760	IDLE	demodb
4	16137	0	2	49760	IDLE	demodb
5	16138	99999	1	49760	IDLE	demodb

Job Queue

ID	PRIORITY	ADDRESS	TIME	REQ

Delete

You can delete the broker information from host information.

Edit Alias Name

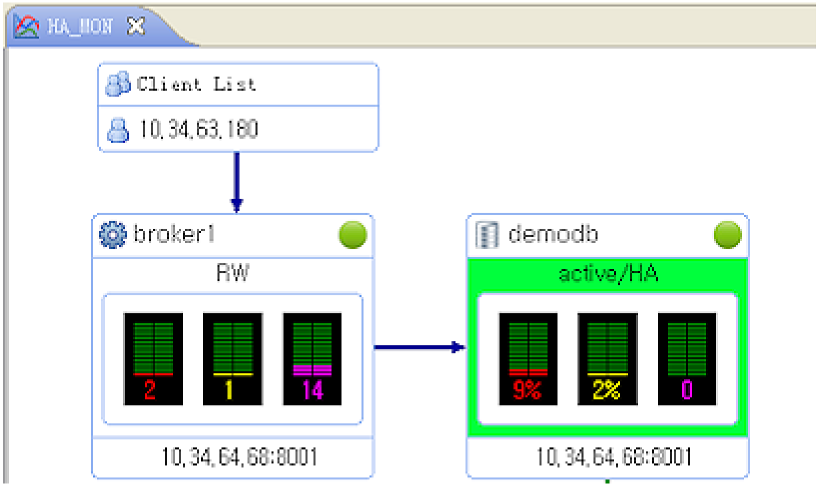
You can edit an alias shown in the broker monitor window. By default, a name of the broker is displayed. You can edit this name with another name that can easily be identified by system operators.

View History Record

The function that provides the history record information. You can track trend of a specific time by selecting date and time.

View Connected Client

This enables the window that represents an application client connected to a broker. An IP address of the connected application client is displayed.



View Connected Database

You can check database information connected by using the broker. The information based on the values of DB items are shown among broker monitoring items.

View Host

You can check the host information in which the database server is running.

Minimize

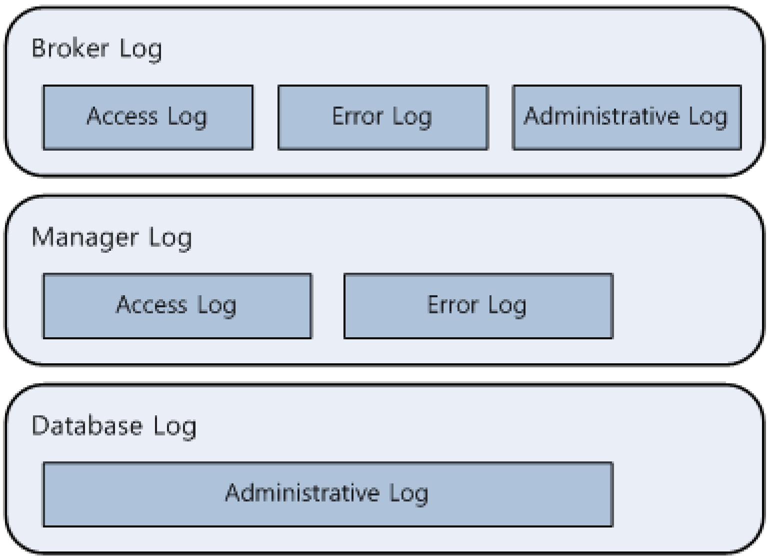
You can minimize the host monitor window. A host alias and connection status are only shown.

Refresh

You can manually refresh the host monitor.

Logs

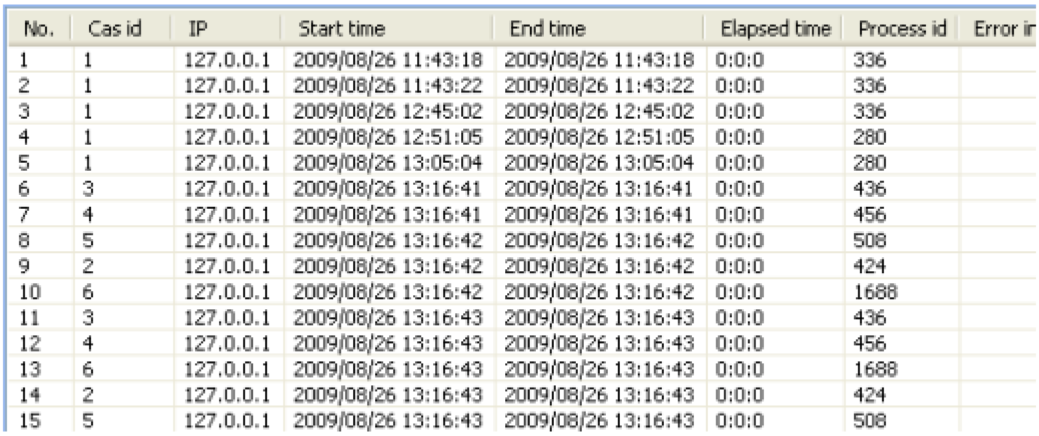
Logs consist of Broker, Manager and Database logs. There can be sub-nodes such as Access, Error and Admin logs. Each log consists as follows:



Broker Log

Access Log

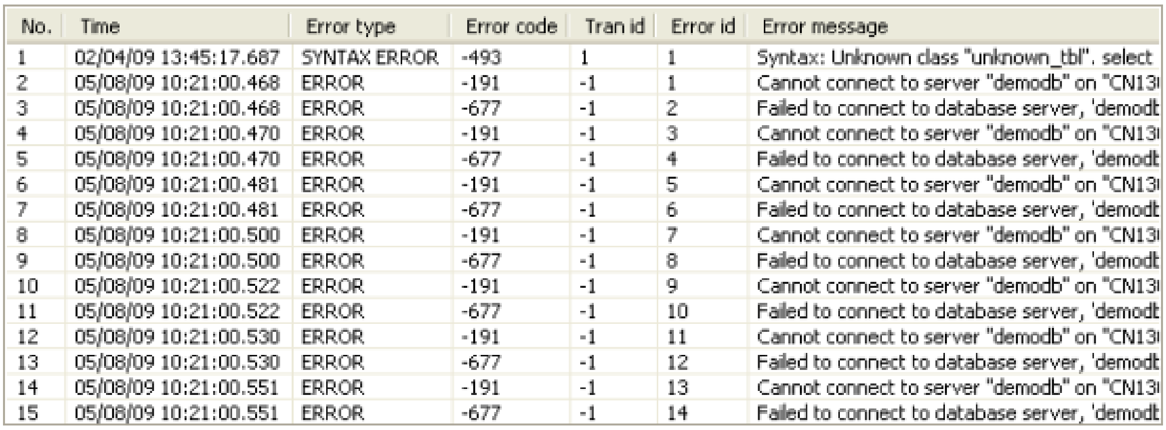
The access log file records information about application client access, and it analyzes and outputs what is saved with the name of " broker_name.access." In addition, if the **LOG_BACKUP** parameter is configured to " ON" in the broker configuration file, the information about the termination date and time is stored additionally to the log file upon successful termination of the broker operation.



No.	Cas id	IP	Start time	End time	Elapsed time	Process id	Error in
1	1	127.0.0.1	2009/08/26 11:43:18	2009/08/26 11:43:18	0:0:0	336	
2	1	127.0.0.1	2009/08/26 11:43:22	2009/08/26 11:43:22	0:0:0	336	
3	1	127.0.0.1	2009/08/26 12:45:02	2009/08/26 12:45:02	0:0:0	336	
4	1	127.0.0.1	2009/08/26 12:51:05	2009/08/26 12:51:05	0:0:0	280	
5	1	127.0.0.1	2009/08/26 13:05:04	2009/08/26 13:05:04	0:0:0	280	
6	3	127.0.0.1	2009/08/26 13:16:41	2009/08/26 13:16:41	0:0:0	436	
7	4	127.0.0.1	2009/08/26 13:16:41	2009/08/26 13:16:41	0:0:0	456	
8	5	127.0.0.1	2009/08/26 13:16:42	2009/08/26 13:16:42	0:0:0	508	
9	2	127.0.0.1	2009/08/26 13:16:42	2009/08/26 13:16:42	0:0:0	424	
10	6	127.0.0.1	2009/08/26 13:16:42	2009/08/26 13:16:42	0:0:0	1688	
11	3	127.0.0.1	2009/08/26 13:16:43	2009/08/26 13:16:43	0:0:0	436	
12	4	127.0.0.1	2009/08/26 13:16:43	2009/08/26 13:16:43	0:0:0	456	
13	6	127.0.0.1	2009/08/26 13:16:43	2009/08/26 13:16:43	0:0:0	1688	
14	2	127.0.0.1	2009/08/26 13:16:43	2009/08/26 13:16:43	0:0:0	424	
15	5	127.0.0.1	2009/08/26 13:16:43	2009/08/26 13:16:43	0:0:0	508	

Error Log

The error log file records information about errors that occurred during the client's request processing and is saved with the name of " broker_name_app_server_num.err."



No.	Time	Error type	Error code	Tran id	Error id	Error message
1	02/04/09 13:45:17.687	SYNTAX ERROR	-493	1	1	Syntax: Unknown class "unknown_tbl". select
2	05/08/09 10:21:00.468	ERROR	-191	-1	1	Cannot connect to server "demodb" on "CN13
3	05/08/09 10:21:00.468	ERROR	-677	-1	2	Failed to connect to database server, 'demod
4	05/08/09 10:21:00.470	ERROR	-191	-1	3	Cannot connect to server "demodb" on "CN13
5	05/08/09 10:21:00.470	ERROR	-677	-1	4	Failed to connect to database server, 'demod
6	05/08/09 10:21:00.481	ERROR	-191	-1	5	Cannot connect to server "demodb" on "CN13
7	05/08/09 10:21:00.481	ERROR	-677	-1	6	Failed to connect to database server, 'demod
8	05/08/09 10:21:00.500	ERROR	-191	-1	7	Cannot connect to server "demodb" on "CN13
9	05/08/09 10:21:00.500	ERROR	-677	-1	8	Failed to connect to database server, 'demod
10	05/08/09 10:21:00.522	ERROR	-191	-1	9	Cannot connect to server "demodb" on "CN13
11	05/08/09 10:21:00.522	ERROR	-677	-1	10	Failed to connect to database server, 'demod
12	05/08/09 10:21:00.530	ERROR	-191	-1	11	Cannot connect to server "demodb" on "CN13
13	05/08/09 10:21:00.530	ERROR	-677	-1	12	Failed to connect to database server, 'demod
14	05/08/09 10:21:00.551	ERROR	-191	-1	13	Cannot connect to server "demodb" on "CN13
15	05/08/09 10:21:00.551	ERROR	-677	-1	14	Failed to connect to database server, 'demod

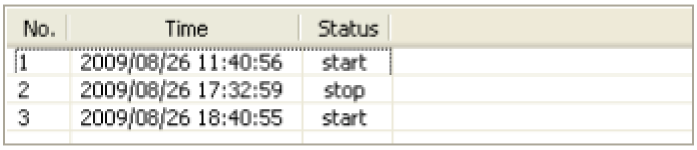
The following is an example and description of an error log:

```
Time: 02/04/09 13:45:17.687 - SYNTAX ERROR *** ERROR CODE = -493, Tran = 1, EID = 38
Syntax: Unknown class " unknown_tbl" . select * from unknown_tbl
```

- **Time : 02/04/09 13:45:17.687** : Time when the error occurred
- **SYNTAX ERROR** : Type of the error (SYNTAX ERROR, ERROR, etc.)
- ***** ERROR CODE = -493** : Error code
- **Tran = 1** : Transaction ID. -1 if no transaction ID is assigned.
- **EID = 38** : Error ID. This ID is used to find the SQL log related to the server or client logs when an error occurs during SQL statement processing.

Admin Log

The admin log records the history about the service operation and termination.



No.	Time	Status
1	2009/08/26 11:40:56	start
2	2009/08/26 17:32:59	stop
3	2009/08/26 18:40:55	start

Manager Log

Access Log

The access log files records information about the CUBRID Manager access. You can see user accounts, operation history, and time when operations are performed.

Error Log

The error log file records information about errors that occurred while connecting the CUBRID Manager.

Database Log

Admin Log

This admin log records information about errors that occurred while server is running. The format of output file is as follows: < *database_name*> _< *date*> _< *time*> .err.