CUBRID 2008 R2.1 User Manual

CUBRID 2008 R2.1 User Manual

Table of Contents

Introduction to Manual 15

Getting Started with CUBRID 17

Installing and Running 18

Installing and Running on Linux 18

Installing and Running on Windows 19

Before You Start CUBRID 22

Configuring the Environment Variable 22

Language Setting 23

CSQL Interpreter 25

Starting the CSQL Interpreter 25

Executing the SQL with CSQL 26

Getting Started with the CUBRID Manager 28

Running the CUBRID Manager 28

Using the Query Editor 32

Programming with JDBC 37

Setting up the JDBC Environment 37

JDBC Sample 39

Programming with PHP 41

Installing the PHP Module 41

PHP Sample 42

Programming with ODBC and ASP 44

Configuring the Environment of ODBC and ASP 44

ASP Sample 46

Programming with CCI 50

CCI Library 50

CCI Sample 52

Introduction to CUBRID 57

System Architecture 58

System Architecture 58

Database Volume Structure 59

Database Server 61

Broker 62

Interface Module 63

CUBRID Features 64

CSQL Interpreter 67

Introduction to the CSQL Interpreter 68

Executing CSQL 69

CSQL Execution Mode 69

How to Use CSQL (Syntax) 69

CSQL Startup Options 70

Session Commands 73

Administrator's Guide 79

CUBRID Utilities 80

CUBRID Controls 82

How to Use CUBRID Utilities (Syntax) 82

CUBRID Services 83

Database Server 85

Broker 87

CUBRID Manager Server 94

Database Administration 96

How to Use the CUBRID Administration Utilities (Syntax) 96

Database Users 96

databases.txt File 96

Creating a Database 97

Adding a Database Volume 101

Deleting a Database 103

Renaming a Database 104

Installing a Database 105

Checking and Compacting Used Space 106

Updating Statistics 107

Dumping Statistic 107

Checking the Lock Status 108

Checking Database Consistency 108

Killing Database Transactions 109

Checking the Query Plan Cache 110
Restoring Emergency Database Logs 110
Outputting Internal Information of a Database 111
Backup and Restore 112
Export and Import 112
Dumping Parameters 113
Database Migration 114
Migrating a Database 114
Unloading a Database 115
Loading a Database 117
How to Write a File to Load a Database 121
Database Backup and Restore 124
How to Use CUBRID Backup and Restore Utilities (Syntax) 124
Database Backup 129
Backup Strategy and Method 129
Managing a Backup File 131
Database Restore 132
Restore Strategy and Procedure 132
Restoring a Database to a Different Server 133
Database Replication 136
Concept of Database Replication 136
Configuration 141
Implementing Replication 156
Constraints on Replication 160
Direction for Replication 161
CUBRID HA 162
Overview 162
Glossary 162
HA Configuration 165
HA Utilities 166
HA-Related JDBC Configuration 167
HA-Related System Catalog 168

Performance Tuning 171

Database Server Configuration 172
Scope of the Dtabase Server Configuration 172
The cubrid_broker.conf Configuration File and Default Parameters 172
Connection-Related Parameters 176
Memory-Related Parameters 176
Disk-Related Parameters 178
Error Message-Related Parameters 179

Concurrency/Lock Parameters 180
Logging-Related Parameters 181
Transaction Processing-Related Parameters 182
Statement/Type-Related Parameters 183
Query Cache-Related Parameters 184
Utility-Related Parameters 185
Other Parameters 186
Changing Database Server Configuration 188
Broker Configuration 190
The cubrid_broker.conf Configuration File and Default Parameters 190
Common Parameters 191
Parameter by Broker 192
CUBRID Manager Server Configuration 196
The cm.conf Configuration File and Default Parameters 196
Parameters 197

CUBRID SQL Guide 199

CUBRID SQL Guide 200
Comment 201
Identifier 202
CUBRID Keywords 204
Data Types 209
Character Strings 209
Bit Strings 212
Numeric Type 214
Date/Time Type 218
Collection Type 220
Operators and Functions 223
Relational Operators 223
Arithmetic Operators 224
Set Operators 227
Containment Operators 228
CAST Operator 232
EXTRACT Operator 233
String Function and Operator 234
Numeric Functions 247
Date/Time Functions 253
Data Type Conversion 255
Aggregate Functions 263
Condition/Comparison Functions and Operators 267
User Name Function 271

Click Counter Functions 271
ROWNUM Function 272
Data Retrieval 275
SELECT Clause 275
FROM Clause 276
WHERE Clause 278
GROUP BY HAVING Clause 285
ORDER BY Clause 286
USING INDEX Clause 287
Outer Join 288
Subquery 290
Hierarchical Query 292
Query Optimization 301
Updating Statistics 301
Using SQL Hints 301
Viewing Query Plans 303
Table Manipulation 305
Creating Tables 305
Changing Tables 311
Dropping Tables 314
Class Inheritance 314
Resolving Class Conflict 318
Data Manipulation 325
Creating Data 325
UPDATE 326
DELETE 327
Virtual Tables (VIEW) 329
Creating Virtual Tables 329
Changing Virtual Tables 330
Dropping Virtual Tables 332
INDEX 333
Creating Indexes 333
Dropping Indexes 333
Altering Indexes 334
Transaction Management 335
Overview 335
Database Transaction 335
Database Concurrency 338
Lock Protocol 340
Transaction Isolation Level 345
Restore 361

Trigger 363
Creating Triggers 363
Changing Triggers 369
Dropping Triggers 370
Trigger Debugging 370
Deferred Actions/Conditions 372
Applications Using Triggers 373
Methods 375
Overview 375
Method Types 375
Calling Methods 375
Multimedia Data Management 379
glo Classes 379
Partitioning 385
What is Partitioning? 385
Range Partitions 385
Hash Partitions 388
List Partitions 389
Partition Management 391
Serials 397
Creating Serials 397
Accessing and Updating Serials 398
Altering Serials 398
Dropping Serials 399
Authorizing Access to Serial 399
Syntax for Granting Access Authorization 400
Database Users 400
Managing Users 400
Granting Authorization 401
Revoking Authorization 402
Java Stored Functions/Procedures 404
Overview 404
Environment Configuration for Java Stored Functions/Procedures 404
How to Write Java Stored Functions/Procedures 405
Using Internal JDBC Driver on Server-Side 406
Connecting to Other Databases 406
loadjava Utility 407
Registering Loaded Java Classes 408
Calling Java Stored Functions/Procedures 410
Note 413
CUBRID SQL Statements 416

List of SQL Statements 416
Search Conditions 429
Expressions 430
Table Specifications 432
Data Types 433
Identifier, Name, Variable 434
Literal 435
CUBRID System Catalog 437
Overview 437
System Catalog Classes 437
System Catalog Virtual Classes 449
Catalog Classes/Virtual Classes Authorization 467
Consistency of Catalog Information 467
Querying on Catalog 467

CUBRID Manager 469

Introduction to CUBRID Manager 470
CUBRID Manager Architecture 470
CUBRID Manager Client 470
Running the CUBRID Manager 475
Configuring the CUBRID Manager Server 475
Running the CUBRID Manager Server 476
Running the CUBRID Manager Client 477
Host Management 479
Default Host Information 479
Connect/Disconnect Host 480
Add/Delete Host 480
Managing Service 482
User Management 484
Setting User Authorization 484
Adding, Editing, and Deleting Users 485
Properties Management 487
Query Editor 490
Query Editor Structure 490
Toolbar Options 491
Query Edit Pane 493
Query Results Pane 495
Query Editor Options 498
Display Query Plan 499
Database 504
Database Functional Structure 504

Connecting to a Database 505
Starting a Database 507
Create Database 507
Database Configuration 512
Users 514
Table 516
Views 524
Triggers 527
Serials 528
Stored Procedures 529
Automation 531
Database Space 535
Load Database 538
Unload Database 539
Backup Database 541
Restore Database 543
Rename Database 545
Copy Database 546
Optimize Database 548
Compact Database 548
Check Database 549
Database Lock Information 549
Database Transaction Information 551
Delete Database 552
OID Navigator 553
Configuring Background Operations in a Multiple-Host Environment 554
Broker 556
Broker Structure 556
Broker Menu 556
Broker Function 558
SQL Logs 559
Status Monitor 563
Logs 566

API Reference 569

JDBC API 570
JDBC Programming 570
CUBRIDConnection 577
CUBRIDOID 578
CUBRIDPreparedStatement 586
CUBRIDResultSet 587

CUBRIDResultSetMetaData 588
CUBRIDStatement 589
ODBC API 590
ODBC Programming 590
OLE DB API 592
OLE DB Programming 592
PHP API 597
PHP Programming 597
cubrid_affected_rows 600
cubrid_bind 600
cubrid_close_request 601
cubrid_col_get 602
cubrid_col_size 602
cubrid_column_names 603
cubrid_column_types 603
cubrid_commit 604
cubrid_connect 604
cubrid_current_oid 605
cubrid_disconnect 606
cubrid_drop 606
cubrid_error_code 607
cubrid_error_code_facility 607
cubrid_error_msg 608
cubrid_execute 608
cubrid_fetch 610
cubrid_get 610
cubrid_get_class_name 611
cubrid_is_instance 612
cubrid_load_from_glo 612
cubrid_lock_read 613
cubrid_lock_write 613
cubrid_move_cursor 614
cubrid_new_glo 614
cubrid_num_cols 615
cubrid_num_rows 616
cubrid_prepare 616
cubrid_put 617
cubrid_save_to_glo 618
cubrid_schema 618
cubrid_send_glo 621
cubrid_seq_drop 621

cubrid_seq_insert 622
cubrid_seq_put 622
cubrid_set_drop 623
cubrid_rollback 624
cubrid_version 625
CCI API 626
CCI Overview 626
cci_connect 630
cci_disconnect 631
cci_end_tran 631
cci_prepare 631
cci_get_bind_num 632
cci_is_updatable 632
cci_get_result_info 632
CCI_GET_RESULT_INFO_TYPE 633
CCI_GET_RESULT_INFO_SCALE 633
CCI_GET_RESULT_INFO_PRECISION 633
CCI_GET_RESULT_INFO_NAME 634
CCI_GET_RESULT_INFO_ATTR_NAME 634
CCI_GET_RESULT_INFO_CLASS_NAME 634
CCI_GET_RESULT_INFO_IS_NON_NULL 635
cci_bind_param 635
cci_bind_param_array 636
cci_bind_param_array_size 637
cci_execute 637
cci_execute_result 638
cci_next_result 638
cci_execute_array 639
cci_execute_batch 639
CCI_QUERY_RESULT_RESULT 640
CCI_QUERY_RESULT_ERR_MSG 640
CCI_QUERY_RESULT_STMT_TYPE 641
cci_query_result_free 641
cci_get_db_parameter 641
cci_set_db_parameter 642
cci_close_req_handle 642
cci_cursor 643
cci_cursor_update 643
cci_fetch_size 644
cci_fetch_sensitive 644
cci_fetch 644

cci_fetch_buffer_clear 645
cci_get_data 645
cci_schema_info 646
cci_get_cur_oid 649
cci_oid_get 649
cci_oid_put 649
cci_oid_put2 650
cci_glo_new 651
cci_glo_save 651
cci_glo_load 652
cci_get_db_version 652
cci_get_class_num_objs 652
cci_oid 653
cci_oid_get_class_name 653
cci_col_get 654
cci_col_size 654
cci_col_set_drop 654
cci_col_set_add 655
cci_col_seq_drop 655
cci_col_seq_insert 656
cci_col_seq_put 656
CCI_IS_SET_TYPE, CCI_IS_MULTISET_TYPE, CCI_IS_SEQUENCE_TYPE, CCI_IS_COLLECTION_TYPE 656
CCI_GET_COLLECTION_DOMAIN 657
cci_set_isolation_level 657
cci_set_free 658
cci_set_size 658
cci_set_element_type 658
cci_set_get 658
cci_set_make 659
cci_set_max_row 660
cci_savepoint 660
cci_glo_read_data 661
cci_glo_write_data 661
cci_glo_insert_data 662
cci_glo_delete_data 662
cci_glo_truncate_data 663
cci_glo_append_data 663
cci_glo_data_size 663
cci_glo_compress_data 664
cci_glo_destroy_data 664

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, and shows the necessary sections of a syntax. Single quotes are also used to enclose strings.	{'{'const_list'}'}
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.	[ATTRIBUTE|COLUMN]
Braces around parameters({ })	Braces around parameters indicate that one of those parameters must be specified in a statement syntax.	CREATE {CLASS|TABLE}
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>

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 Programs using CCI

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 R2.0, CUBRID supports both 32-bit and 64-bit 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 binary shells; thus it can be installed automatically. Upload the setup file in binary mode to Linux server via the File Transfer Protocol (FTP).

Execute the setup program as the following example.

```
[cub_user@cubrid ~]$ sh CUBRID-8.2.0.1150-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 R2.0 servers are only compatible
with the CUBRID 2008 R2.0 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-8.2.0.1150-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
```

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

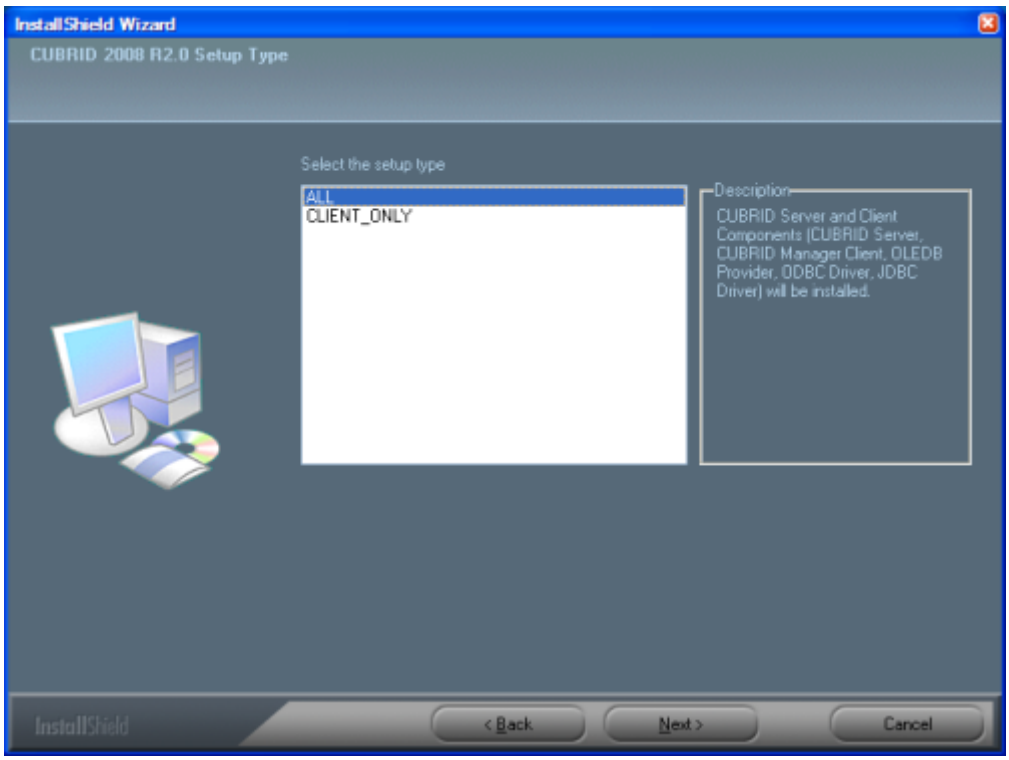
CUBRID 2008 R2.0 supports both 32-bit and 64-bit Windows. Make sure to install the CUBRID 32-bit version on 32-bit Windows and the CUBRID 64-bit version on 64-bit Windows.

The CUBRID Manager and java stored procedures require at least Java version 1.5. Windows XP or Windows Vista users must have the Microsoft Visual C++ 2008 Redistributable Package that is distributed separately from CUBRID.

Installing CUBRID

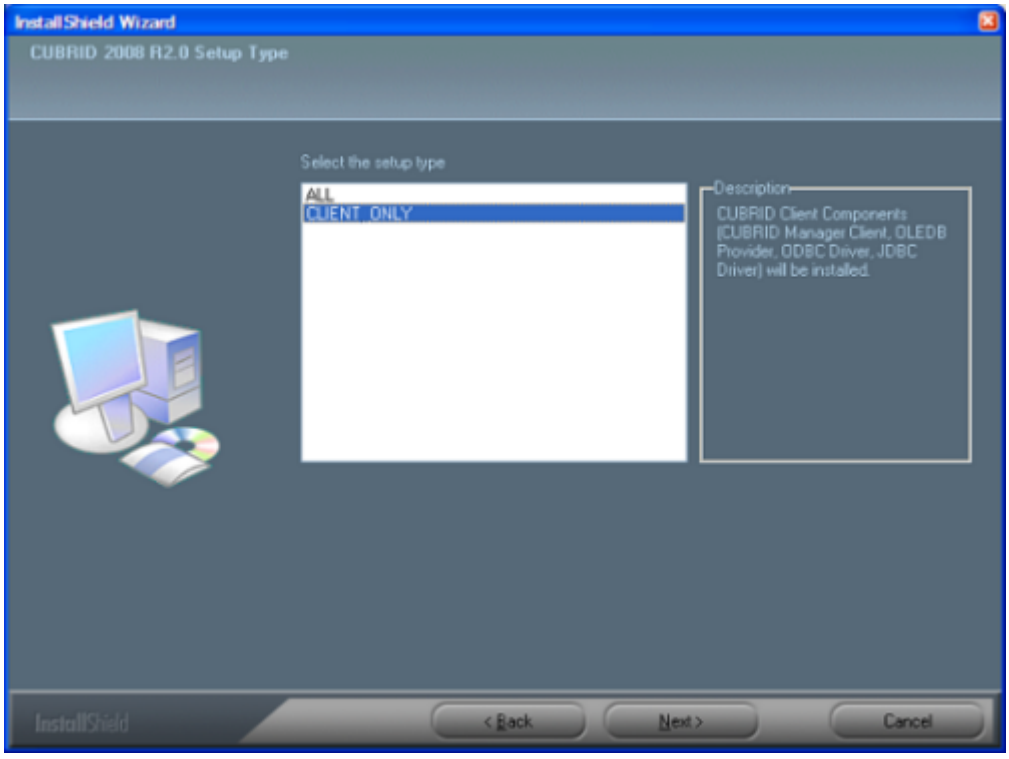
When Both Installing the Server and the Client

Select **ALL** if you need to install the database server for Windows with the client component. Note that Java needs to be installed in order to use the CUBRID Manager or the Java stored procedure.



When Installing the Client Module Only

To install the client module only, choose **`CLIENT_ONLY;`** selecting such will install only the CUBRID Manager and other connection environment for clients (OLEDB Provider, ODBC Driver, JDBC Driver, and CCI Library).



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 The 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 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 the **Databases.txt** file in the "Administrator's Guide").
- 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 the **Language Setting** section for details).

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
```

For Windows :

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

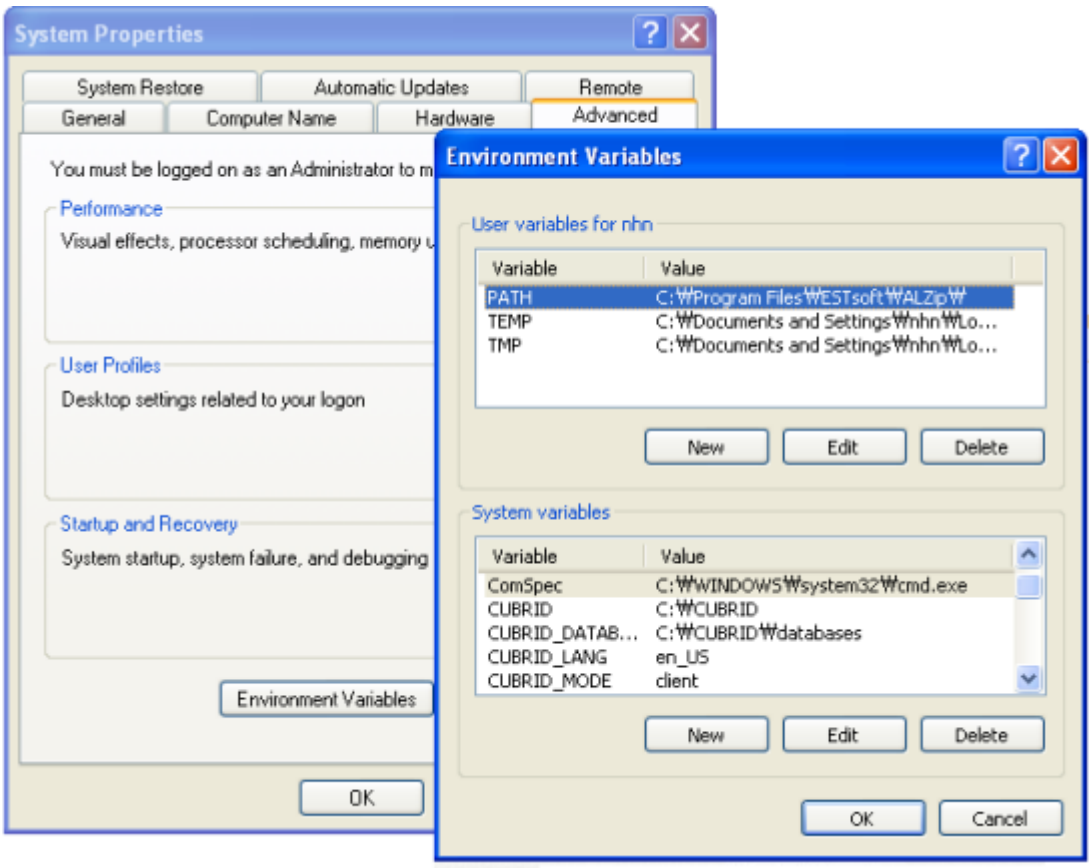
Related OS Environment Variables

- 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, 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) must be installed, and the **JAVA_HOME** environment variable must designate the concerned directory.

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.



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/cmclient:$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.

CSQL Interpreter

Starting the CSQL Interpreter

The CSQL Interpreter is a program used in CUBRID and is used to make SQL queries in command mode and view the results. The SQL statements, queried, and results can be saved in the file for later use. Besides the CSQL Interpreter, 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. 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 CUBRID

Enter the following commands, which start the CUBRID service, in the shell. Check whether the necessary environment variables have been correctly configured before executing the system.

```
% 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
This may take a long time depending on the amount of recovery works to do.
CUBRID 2008 R2.0
++ cubrid server start: sucess

% cubrid service status
@ cubrid master status

++ cubrid master is running.

@ cubrid server status

Server demodb (rel 8.2, pid 30549)
@ cubrid broker status
NAME             PID  PORT  AS  JQ     REQ  TPS  QPS  LONG-T  LONG-Q  ERR-Q
============================================================================
* query_editor  30522 40820   5   0      0    0     0    0/60     0/60   0
* broker1       30532 40821   5   0      0    0     0    0/60     0/60   0
@ cubrid manager server status
++ cubrid manager server is running.
@ cubrid replication status
```

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.

    ;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 at once. 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  clos
ing
_date  mascot                slogan                introduction
============================================================================
===
============================================================================
         2004  'Greece'              'Athens'              08/13/2004    08/2
9/2
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. For the
fir
```

```
st time ever a record 201 National Olympic

Committees (NOCs) participated in the Olympic Games. The overall tally for
event
s 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. Women''s wrestling was included in 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
beca
me the first athlete in any sport to win two

medals in each of 5 Olympics. Runner Hicham El Guerrouj 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
th
e men's football tournament without giving up a

goal, and the U.S. softball team won by outscoring their opponents 51-1.'
         2000  'Australia'           'Sydney'              09/15/2000    10/0
1/2
000    'Olly Syd Millie'     'Share the Spirit'    'The Sydney 2000 Games are
th
e biggest to date with a total of 199 nations and more than 10,500 athletes
part
...

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
```

Running the CUBRID Manager

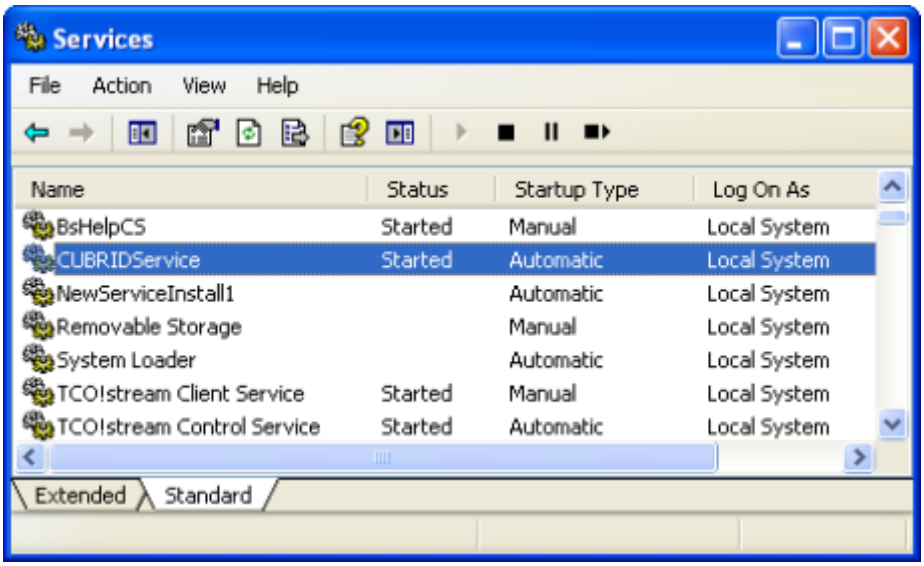
The "CUBRID Manager" is a GUI-based database management and query tool. It facilitates various management tasks and provides the "Query Editor," allowing users to execute SQL statements against the connected database.

The CUBRID Manager runs in the JAVA environment; you must have the Java Runtime Environment installed on your computer. To configure the Java Runtime Environment, see http://java.sun.com.

Starting the CUBRID Server

For Windows

- To start the CUBRID Server, go to [Control Panel] > [Performance and Maintenance] > [Administrative Tools] > [Services] and double-click CUBRIDService.



- You can also start the CUBRID Server by clicking the CUBRID Service Tray at the bottom right of the screen.



Note Make sure not to click [Exit] in the CUBRID Service Tray while the CUBRID is running. This causes the service tray to disappear from the screen and all of the running server processes to stop.

For Linux

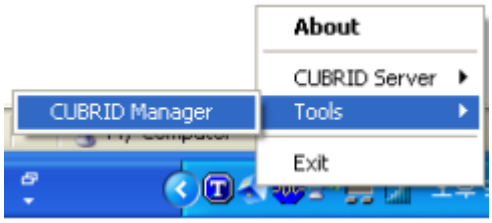
Enter the following command in the shell to start the CUBRID Server.

```
%cubrid service start
```

Starting the CUBRID Manager

For Windows

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



For Linux

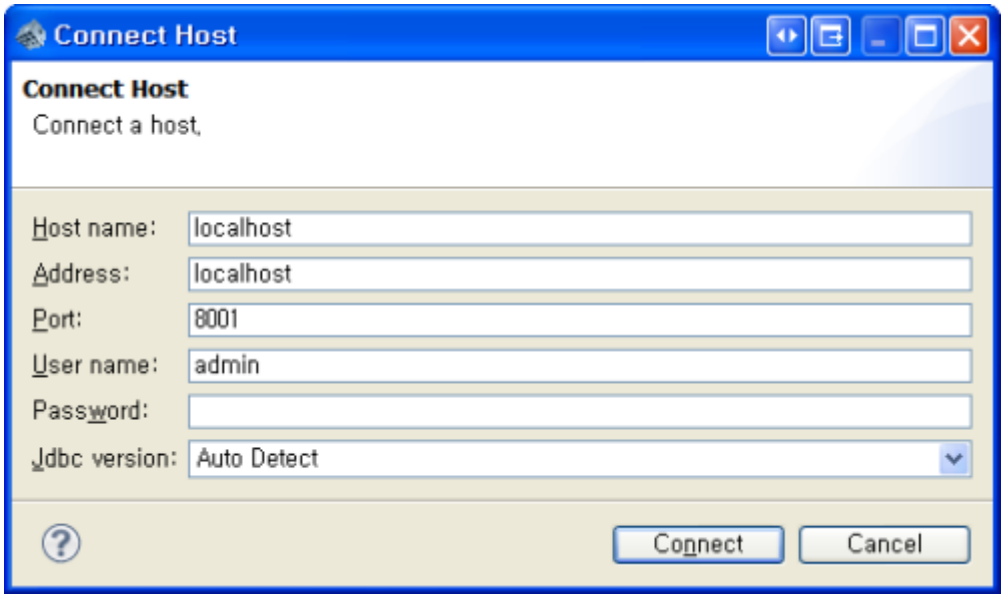
Enter the following command in the shell to start the CUBRID Manager.

- % cubridmanager
- % $CUBRID/cubridmanager/cmclient/cubridmanager

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.

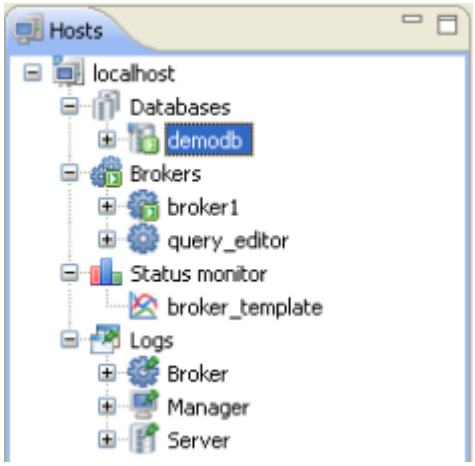
Right-click the mouse and then choose [Add host] to add a host in the host tab. The default value is localhost, the connection port is set to 8001, and the user ID and password are both set to "admin" by default. You must change the password of the admin ID to access the database; you cannot set it to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



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 initial host screen is as follows after connection.

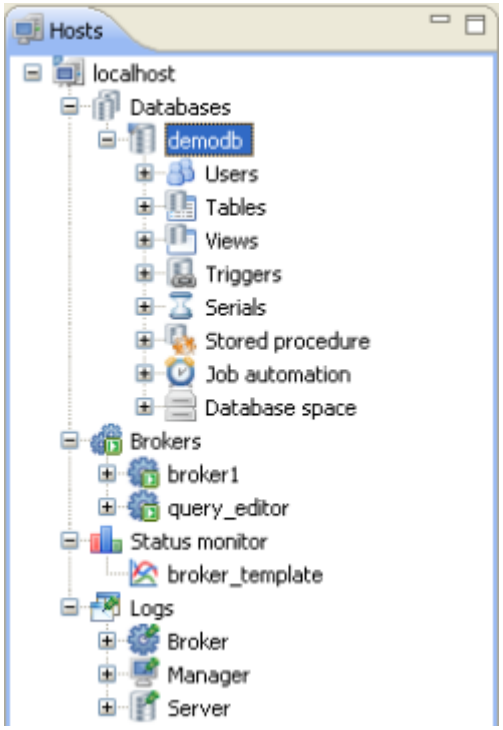


Connecting to a Database

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 databse, 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.

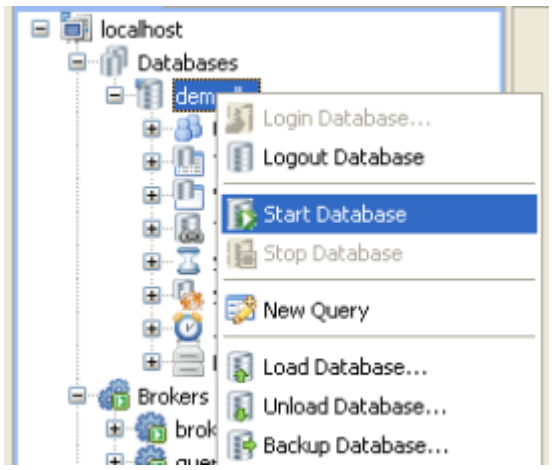


In case the CUBRID Manager is not running normally, an error message which means "Cannot connect to a server. Please check the configuration environment of the CUBRID Manager server and other connection." will be displayed. To resolve this problem, check the following list.

- 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 CUBRID Manager > Configuring 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.

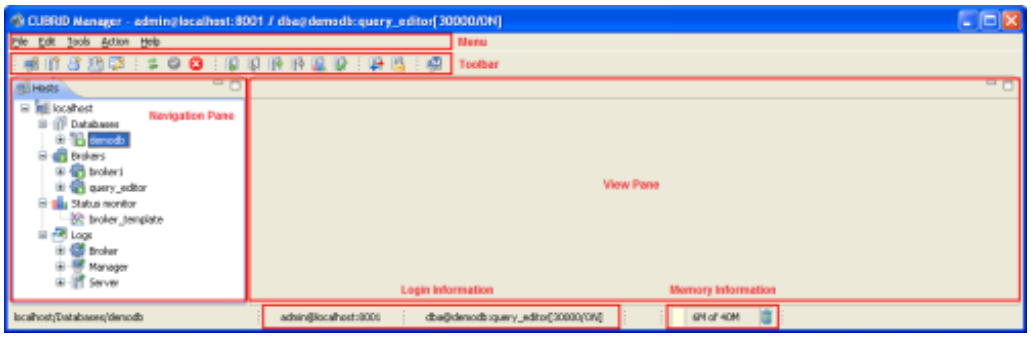
Starting a Database

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.



Menu

The menu bar includes File, Edit, Tools, Action, and Help at the top of the window. See CUBRID Manager Client.

Toolbar

A toolbar is displayed under the menu bar for fast access to frequently-used functions. The active icons indicate that the functions can be performed at the point; otherwise, displayed as inactive.

The following icons are provided in the toolbar:



Add host	Unload database
Create database	Load database
Add user	Back up database
Add table	Restore database
New query editor	Optimize database
Refresh	Check database
Start	Transaction info
Stop	Lock info
	Server version

Memory info

You can see the memory status in the status bar. You can check the current memory usage and click the recycle bin icon in the status bar to optimize the memory (garbage collection), if necessary.



Using the Query Editor

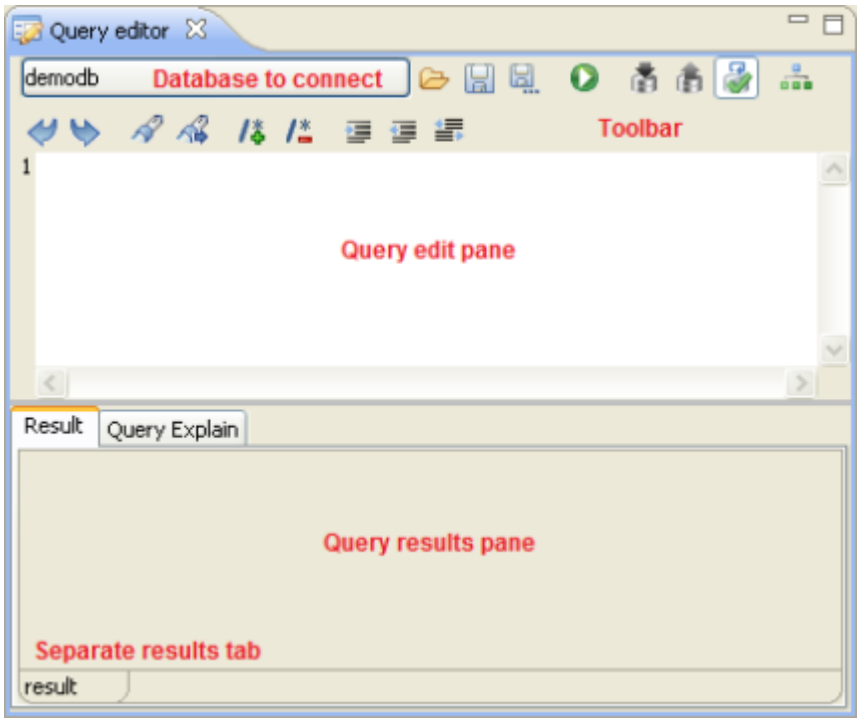
Starting the Query Editor

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 left, a query editor pane on the top-right, and a query result pane on the bottom-right. 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.



Toolbar

The following shows the toolbar provided in the query editor pane. The edit function is synchronized with the [Edit] menu. The shortcut key is provided for some functions.



Function	Icon	Shortcut Key	Description
Open			Opens text-based SQL statements and displays them in the query editor pane.
Save			Saves the content in the query editor pane.
Save as			Saves the content in the query editor pane with a different file name.
Execute		F5	Executes the entire queries or the block-selected queries in the query editor pane. An inactive icon ( ) will be shown in case of un-executable status.

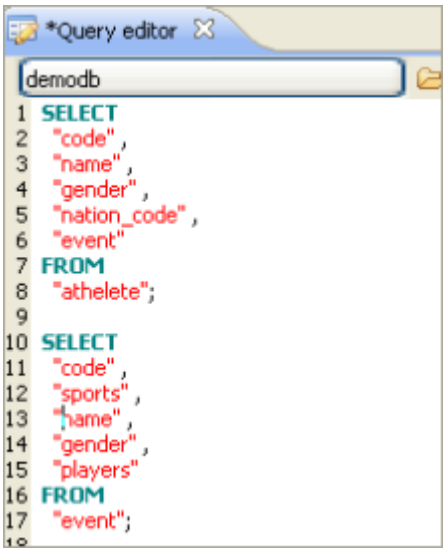
Commit			A user can decide whether to or not commit transactions by keeping an inactive mode () with the auto-commit selected; it changes the mode into the active () with the auto-commit un-selected.
Rollback			A user can decide whether to or not rollback transactions by keeping an inactive mode () with the auto-commit selected; it changes the mode into the active () with the auto-commit cleared.
Auto commit			Commits queries executed in the query editor pane, automatically. Toggles the icon; auto-commit selected :  , auto-commit cleared :  .
Display query plan		F6	Displays the query execution plan. For more info, see Display Query Plan.
Undo		Ctrl+Z	Cancels edited content.
Redo		Ctrl+Y	Re-performs canceled content.
Find/Replace		Ctrl+F	Finds and replaces content in the query editor pane.
Find next		F3	Finds the next occurrence of content.
Add comment		Ctrl+/	Adds a comment in the user selection area or the line where a cursor is located in the query editor pane. The comment is started with double dashes (--).
Delete comment		Ctrl+/	Deletes a comment in the user selection area or the line where a cursor is located in the query editor pane.
Indent		Tab	Indents the user selection area in the query editor pane.
Un-indent		Shift+Tab	Un-indents the user selection area in the query editor pane.
SQL formatting		Ctrl+Shift+F	Performs formatting SQL statements selected in the query editor 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. If 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 do not change.

Query Editor Pane

In the query editor pane, you can enter and edit queries to manipulate a database; additionally, you can use full functions supported in the toolbar. It provides functions such as Auto-Complete, Editing queries in the pop-up, and View schema information.

If you drag and drop the table you want to retrieve to the query editor pane from the host navigation tree, the **SELECT** statement is automatically generated. Note that you can check the database changes made by the DDL statement in the query editor after refreshing the changes in the host tab.

The following is an example where multiple queries are entered in the query edit pane.



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 as above, they are executed sequentially. Each query creates a corresponding tab in the query results pane. If you execute multiple queries without separating a semicolon, the execution is ignored after the first query.

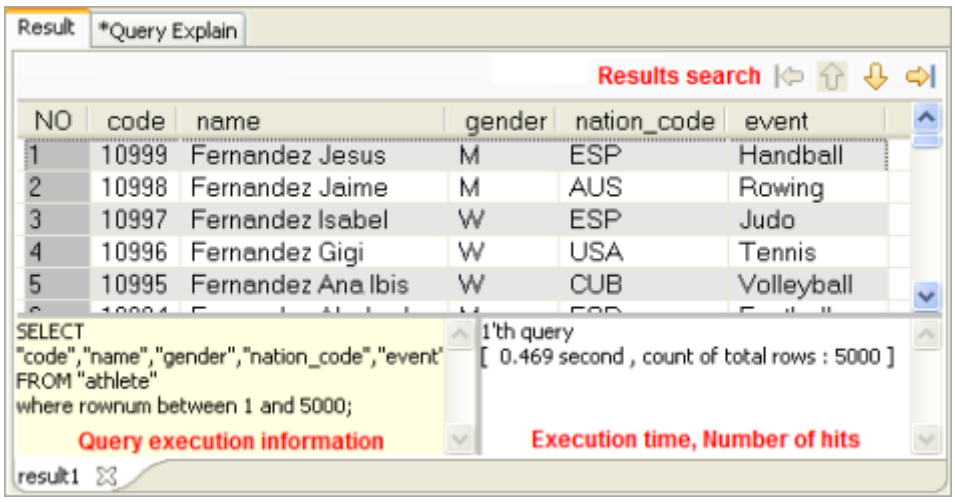
To execute a specific query statement only, select the area by dragging the query, and then click the Execute icon in the toolbar or enter the shortcut key.

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 execution pane displays the bottom-left of query result pane, and it shows executed query.

The query execution information pane displays the bottom-right of query result pane, and it shows query execution time and the number of query results. The time displayed indicates processing time only by a server database; it does not include time processed by the CUBRID Manager.


Result
*Query Explain
Results search
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;
Query execution information
1'th query
[0.469 second , count of total rows : 5000]
Execution time, Number of hits
result1

Setting up the JDBC Environment

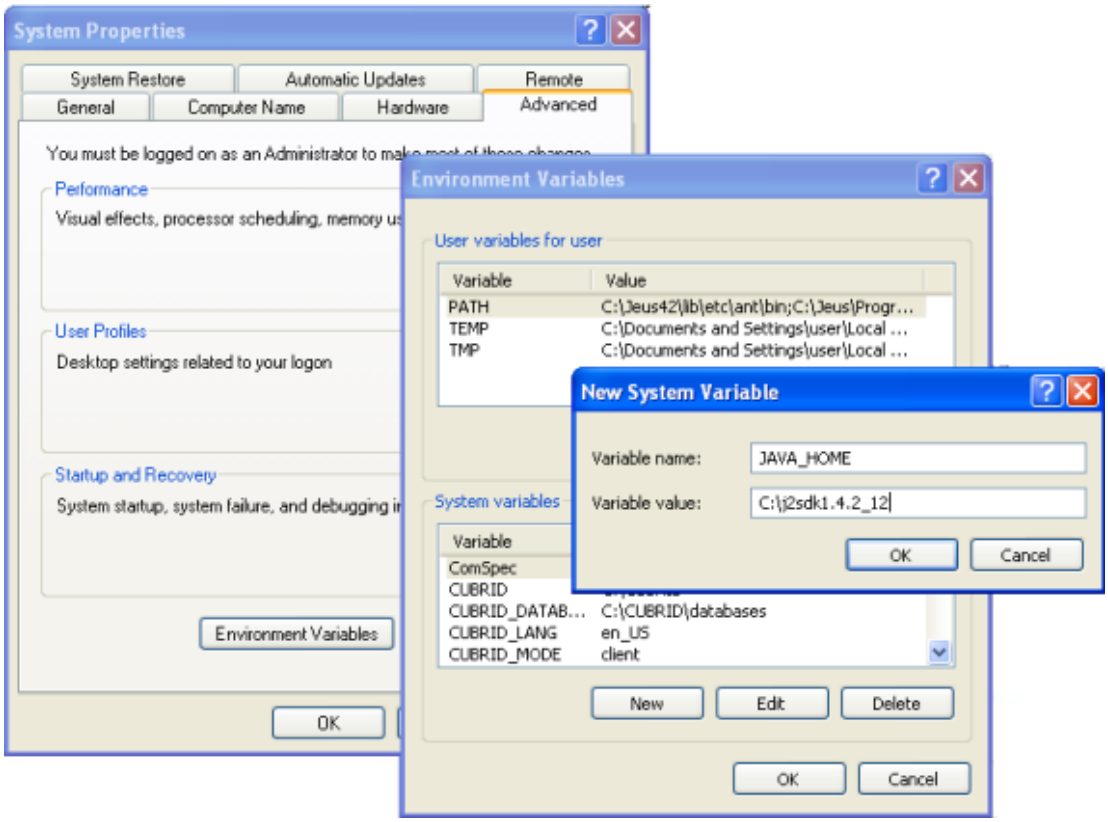
System Requirements

- JDK 1.5 or later
- CUBRID 2008 R1.0 or later
- CUBRID JDBC Driver 2008 R1.0 or later

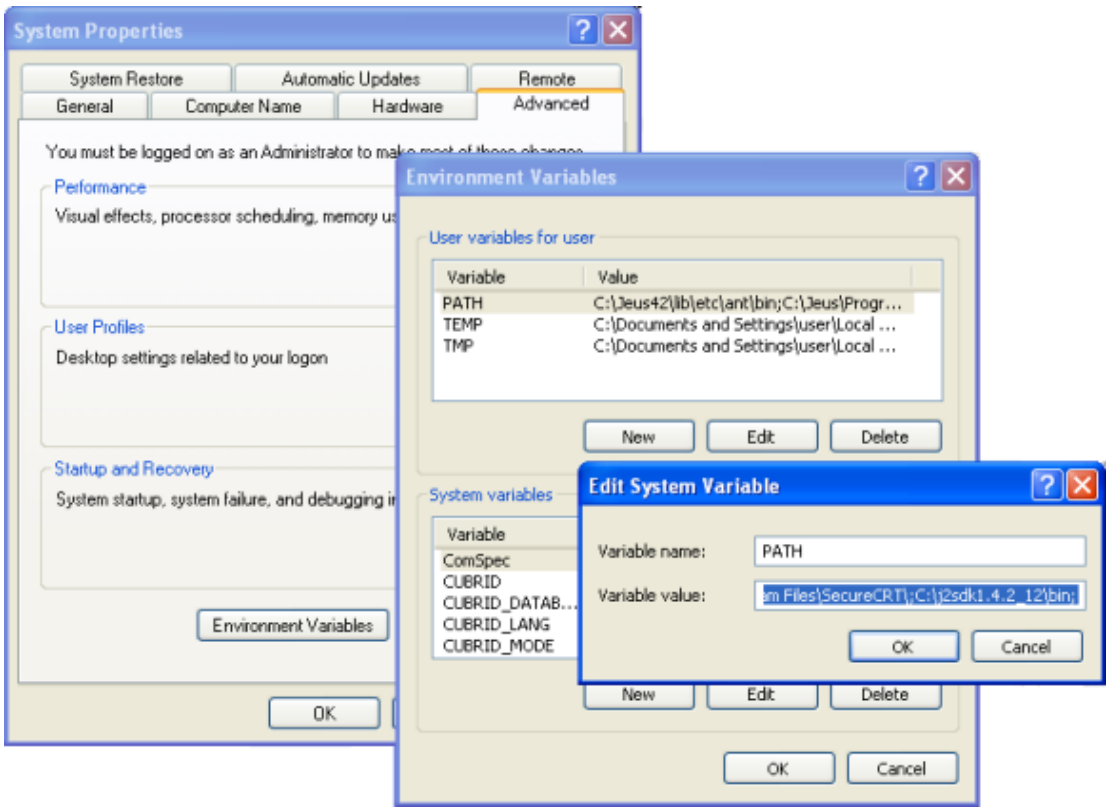
Installing Java

You must already have Java installed and the JAVA_HOME environment variable set on your system. To install Java, download it from Sun MicroSystems's Java homepage (http://java.sun.com). Once the installation is complete, add the installation path as the JAVA_HOME environment variable and append %JAVA_HOME%\bin to the PATH environment variable.

As shown below, go to [My Computer] > [Properties] > [Advanced] > [Environment Variables] > [New] > [New System Variable] and add the JAVA_HOME environment variable.



As shown below, change the PATH variable at [My Computer] > [Properties] > [Advanced] > [Environment Variables] > [System variables] (find a line starting with "Path" in this box) > [Edit] > [Edit System Variable].



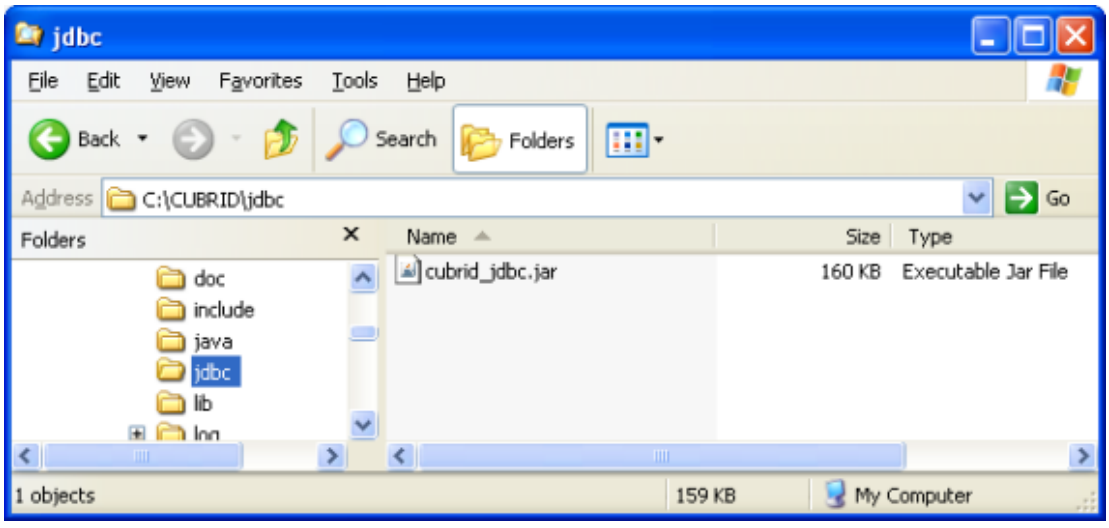
For Linux, set JAVA_HOME and PATH variables depending on the shell used (e.g.: sh | bash).

```
export JAVA_HOME=<Full Path of JAVA directory>
export PATH=$JAVA_HOME/bin:$PATH
```

Setting CLASSPATH

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

The CUBRID JDBC driver is located in $CUBRID/jdbc on Linux and UNIX, and in %CUBRID%\jdbc on Windows.



On Windows, set the following CLASSPATH the same way as you set the JAVA_HOME environment variable.

```
CLASSPATH=C:\CUBRID\jdbc\cubrid_jdbc.jar:.
```

On Linux, set the CLASSPATH as follows depending on the shell used (e.g. sh | bash).

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

If a CUBRID JDBC driver (cubrid_jdbc.jar) 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.

How to Make the Connection

```
Connection conn =
DriverManager.getConnection("jdbc:cubrid:IP:PORT:DB_NAME:::", "ID",
"PASSWORD");
```

- IP : An IP address of the server where the database is running
- PORT : A port number of the CUBRID Broker (default value: 33000)
- DB_NAME : A name of the database to connect
- ID : A user that connects to the database. There are two users in the database by default: DBA and PUBLIC. If you enter an empty string ("") in the ID parameter, you will connect to the database as a PUBLIC user.
- PASSWORD : A password of the user that connects to the database. If no password is specified for a user, enter an empty string ("").

selectData.java

The following is an example of data retrieval.

```
import java.sql.*;
public class selectData {
   public static void main(String[] args) throws Exception {
       Connection conn = null;
       Statement stmt = null;
       ResultSet rs = null;
       try {
       // Connecting to CUBRID
       Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
       conn
= DriverManager.getConnection("jdbc:CUBRID:localhost:33000:demodb:::","","");
       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();
       }
   }
}
```

- Loads the CUBRID JDBC driver class.
- Connects to the database.
- As in the sample code, you will be connected to the database as a PUBLIC user if no ID parameter value is specified. Enter the parameter value if you want to connect as a specific user or if no password is specified.
- Port 33000 is used for the connection broker.

insertData.java

The following is an example of data insertion.

```
import java.sql.*;
public class insertData {
   public static void main(String[] args) throws Exception {
       Connection conn = null;
       Statement stmt = null;
       try {
           // Connecting to CUBRID
           Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
           conn =
DriverManager.getConnection("jdbc:cubrid:localhost:33000:demodb:::","","");
           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 entered.");
           stmt.close();
       } catch ( SQLException e ) {
           System.err.println(e.getMessage());
       } catch ( Exception e ) {
           System.err.println(e.getMessage());
       } finally {
           if ( conn != null ) conn.close();
       }
   }
}
```

You can delete or modify data the same way as you insert data. This means that you can reuse the code above by simply changing the query statements.

Installing the PHP Module

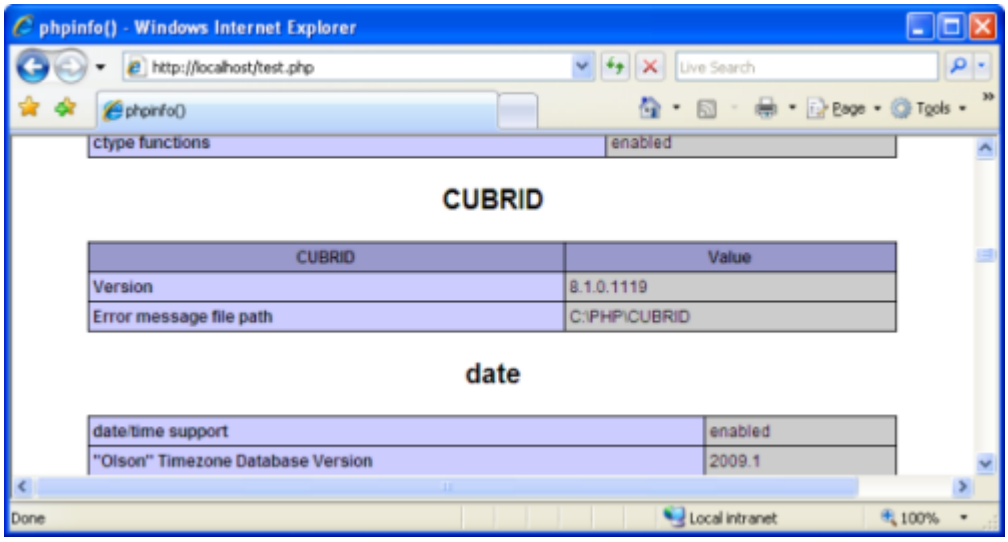
From the CUBRID homepage (http://www.cubrid.com), download the PHP module.

Installing PHP for Windows

Create a directory named CUBRID in the directory where PHP is installed, and then copy the cubrid_err.msg file and the .dll file of the cubrid_php version you just downloaded.
If you download CUBRID PHP version 5.1.4, 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
[CUBRID]
cubrid.err_path=C:\PHP\CUBRID
```

Once the configuration is complete, restart the web server. If you can see the CUBRID information as shown below when you check the PHP configuration using the phpinfo() function of PHP, it means that the installation was successful.



Installing PHP for Linux

Unlike Windows, Linux provides its source code. Download the source code from the CUBRID homepage (http://www.cubrid.com), and install it following the Setup Guide in the INSTALL file of the compressed file. Also in the Linux version, edit and save the php.ini file, and then restart the web server.

```
; Directory in which the loadable extensions (modules) reside.
extension_dir = /usr/lib/php5/lib/php/ext
extension=cubrid.so
;for Windows, extension=cubrid.dll
[CUBRID]
cubrid.err_path=/home/cubrid/CUBRID/msg
```

Check the configuration using phpinfo() function.

```
[root@localhost]#vi test.php
```

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 = 33000;
   $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.
```

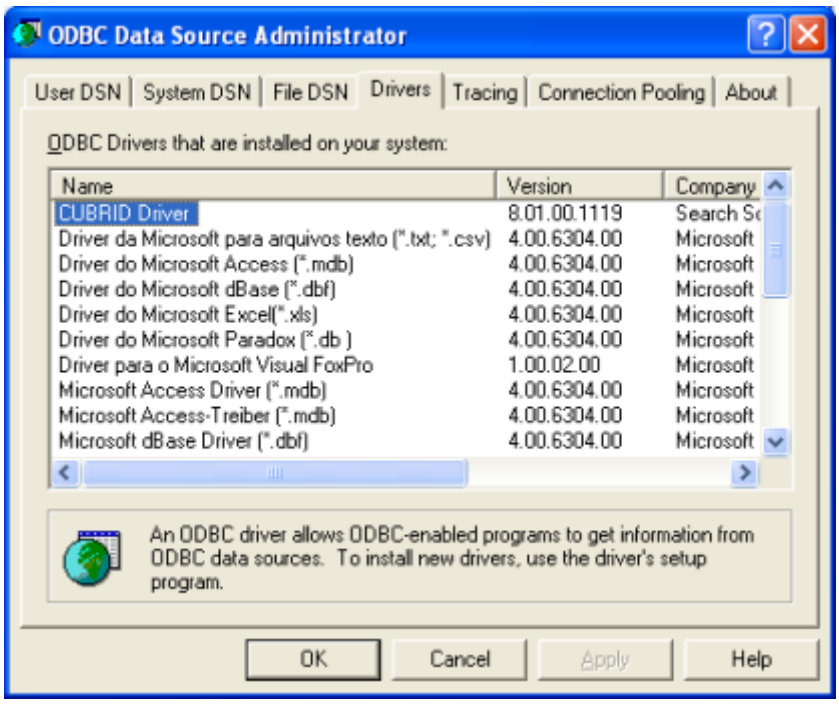
```
   cubrid_commit($cubrid_con);
   cubrid_disconnect($cubrid_con);
?>
</body></html>
```

Example of Data Insertion

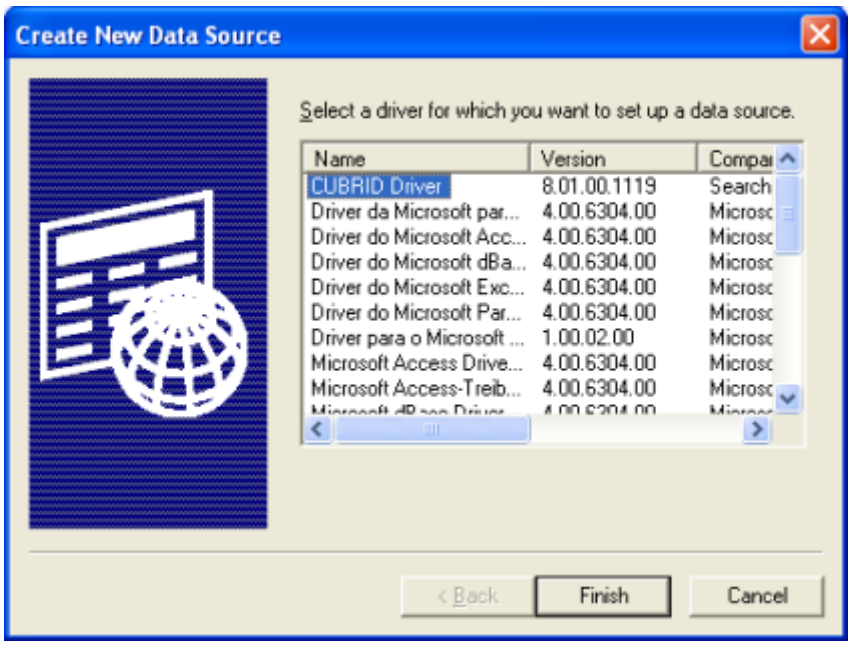
```
<html>
<head>
<meta http-equiv='content-type' content='text/html; charset=euc- kr'>
</head>
<body>
<center>
<table border=2>
<?
   $host_ip = "localhost";
   $host_port = 33000;
    $db_name = "demodb";
   $cubrid_con = @cubrid_connect($host_ip, $host_port, $db_name);
   if (!$cubrid_con) {
        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>
```

Configuring the Environment of ODBC and ASP

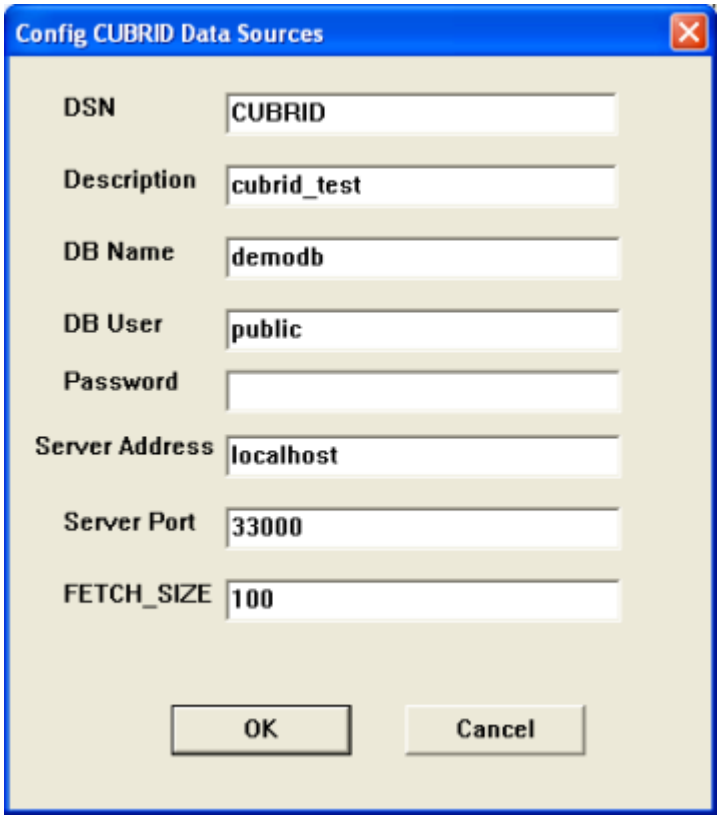
To configure the client environment for processing ODBC requests, you must have the CUBRID ODBC driver installed. The CUBRID ODBC driver is installed automatically during the CUBRID installation. You can install either the server or the client. To check the CUBRID ODBC driver installed, go to [Control Panel] > [Administrative Tools] and double-click the [Data Source (ODBC)].



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.

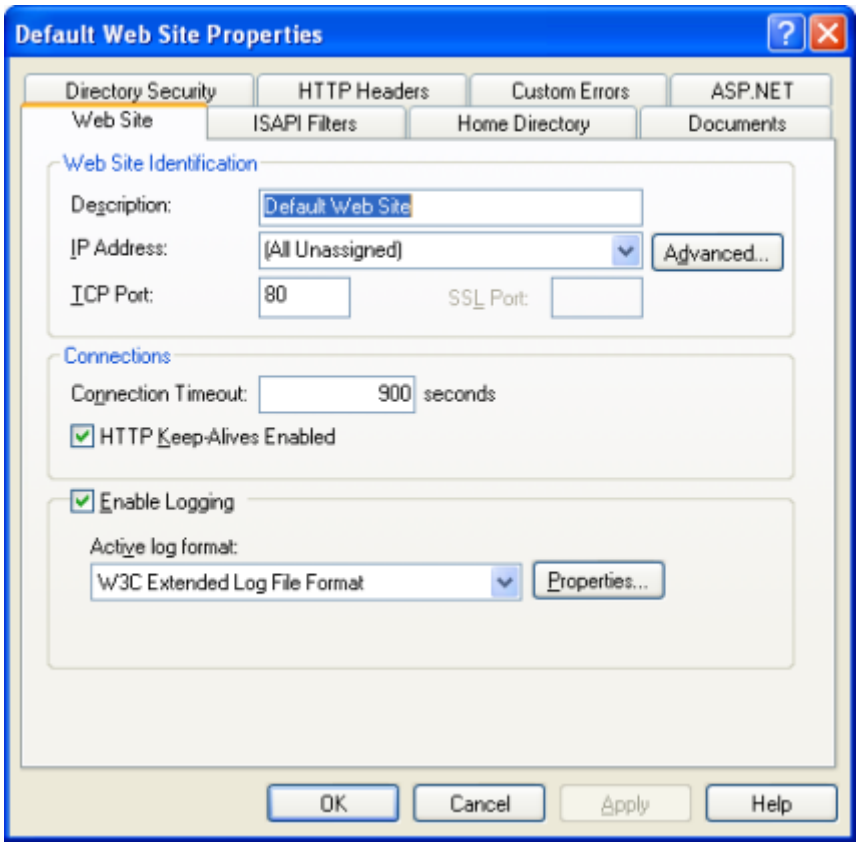


When the following [Config CUBRID Data Sources] text box appears, enter the database name that you try to connect to in the [DB Name] field, the port number of the CUBRID Broker in the [Server Port] field, and then click [OK] button.



ASP Sample

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



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
```

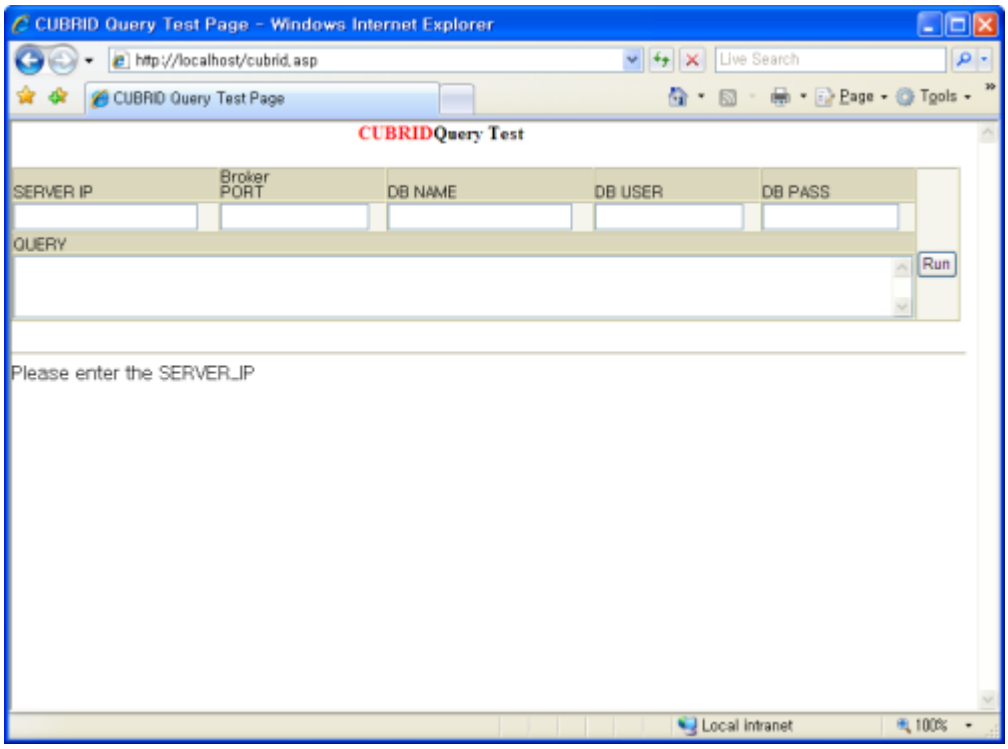
```
    end if
    if strName = "" then
       Response.Write "Please enter the DB_NAME "
        Response.End ' If no DB_NAME entered, end the page
    end if
    if strQuery = "" then
       Response.Write "Please enter the query you want to check"
        Response.End ' If no Query entered, end the 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>"

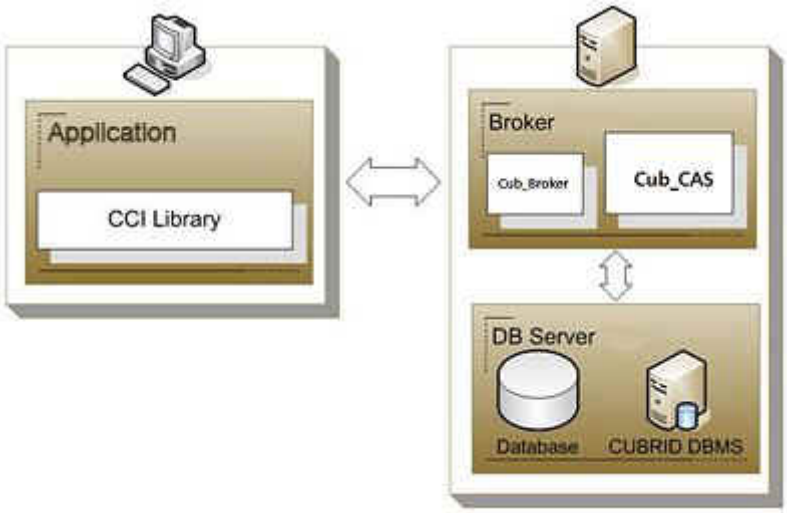
        rs.MoveNext
    Loop
%>
<%
    set  rs = nothing
%>
</table>
```

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.



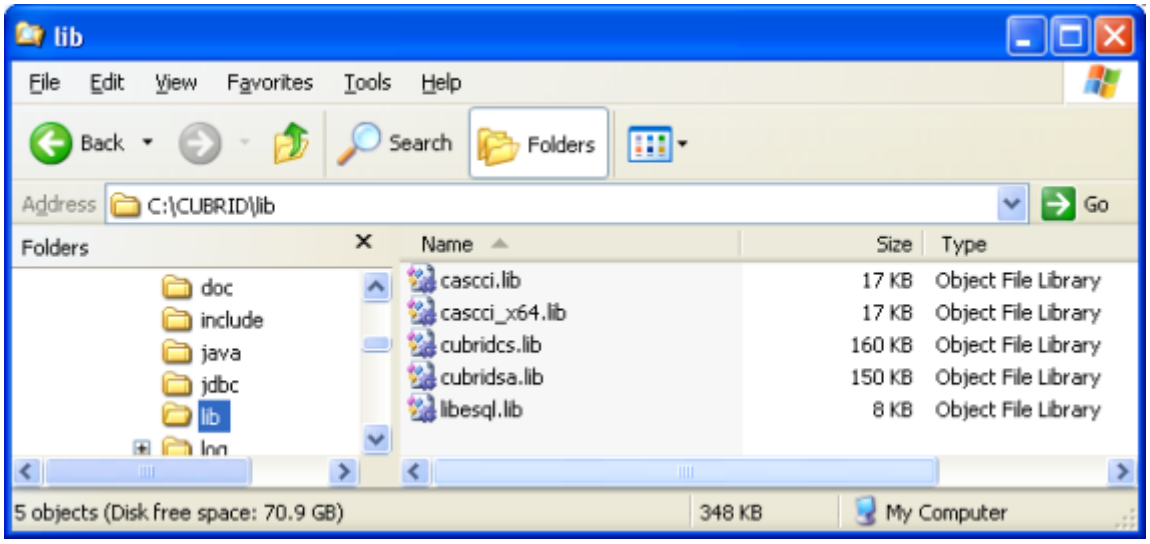
CCI Library

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

During installing CUBRID, CCI is deployed with the Broker. Therefore, you simply need to install CUBRID.



Operating System	Windows	UNIX/Linux
C header file	include/cas_cci.h	include/cas_cci.h
Static library (32-bit)	lib/cascci.lib	lib/libcascci.a
Static library (64-bit)	lib/cascci_x64.lib	lib64/libcascci.a

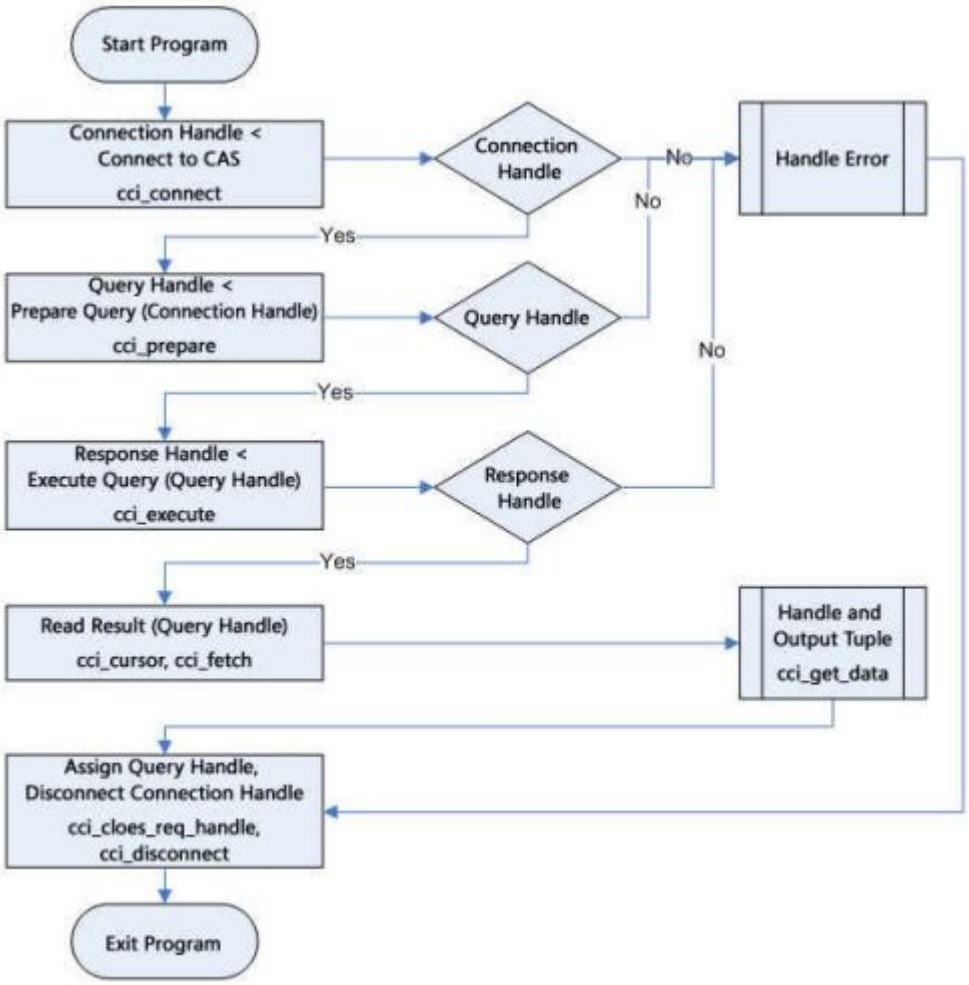
Dynamic library (32-bit)	lib/cascci.lib bin/cascci.dll	lib/libcascci.so
Dynamic library (64-bit)	llib/cascci_x64.lib bin/cascci_x64.dll	lib64/libcascci.so.8.1.0

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.



How to use

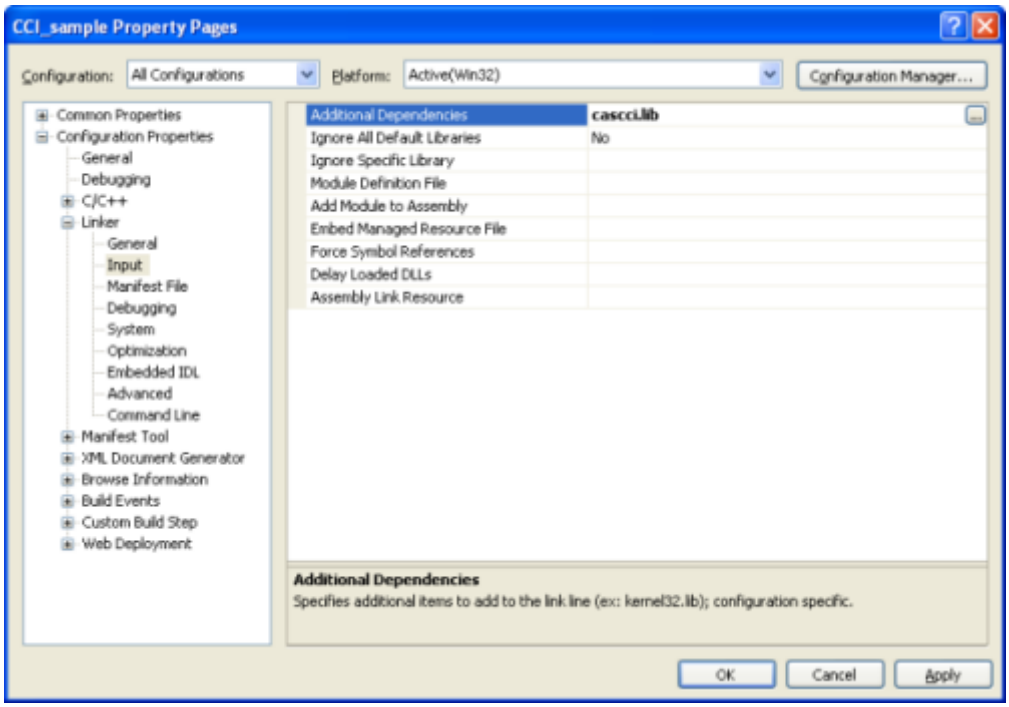
Once you have created the application using CCI, you should decide, according to its features, whether to execute CCI as a static link or dynamic link before you build it. Determine the library to use by referring to the table in the CCI Installation and Configuration section above.

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:

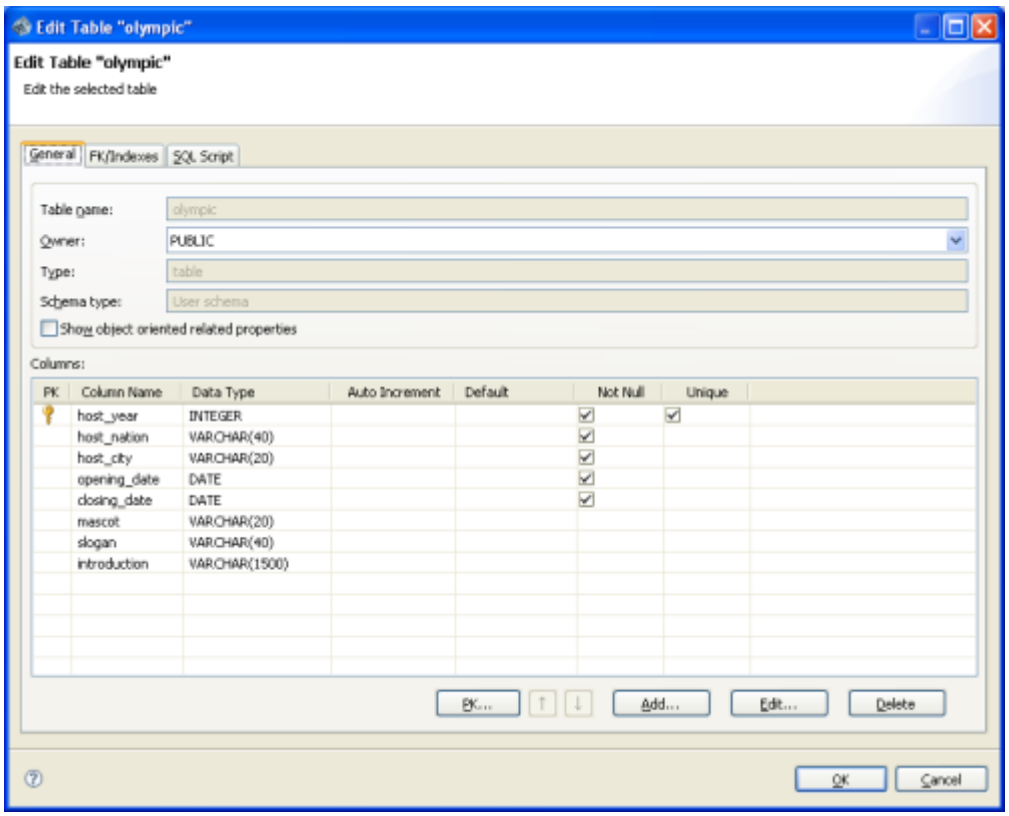


CCI Sample

Introduction

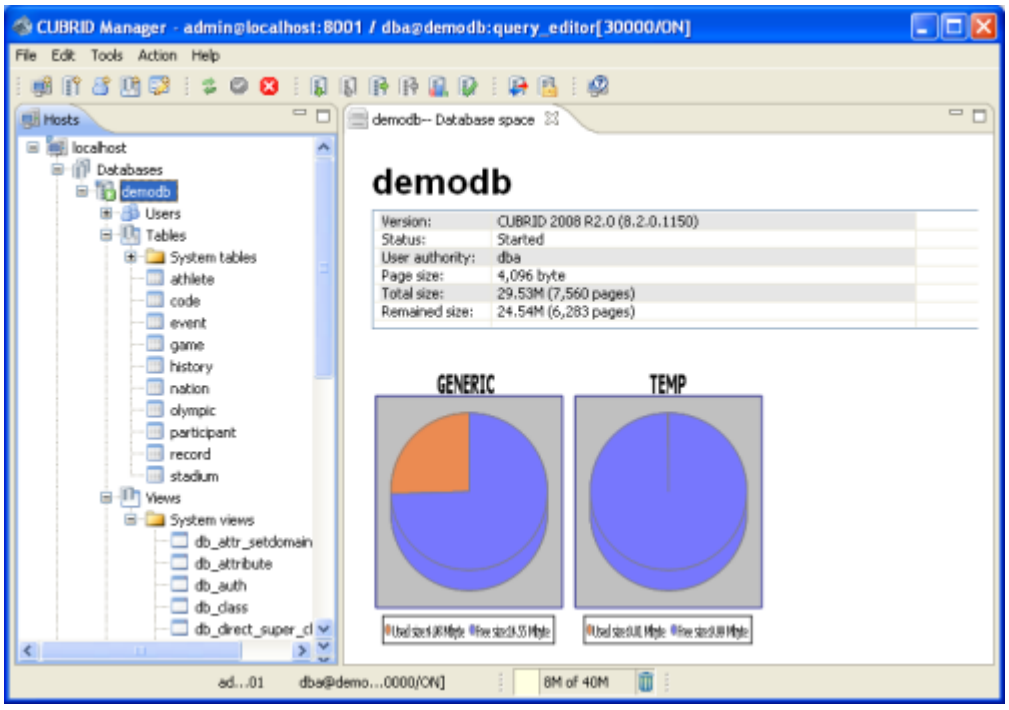
The sample program is to create a simple application using CCI through the connection to the **demodb** database deployed by default during the CUBRID installation. Follow the processes of connection to CAS, query preparation, query execution, response handling and disconnection in the sample. The sample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

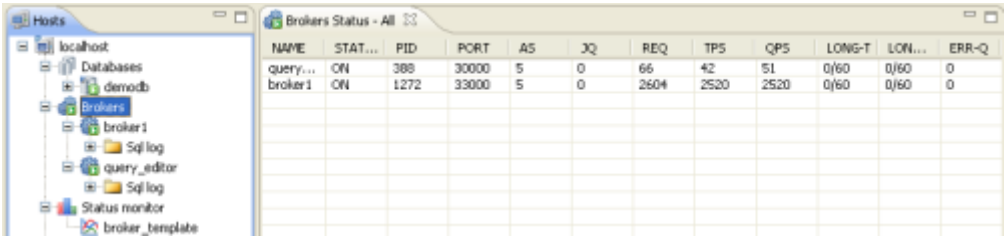


Preparation

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.



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



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, res, ind, i, col_count;
    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;
}
```

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.

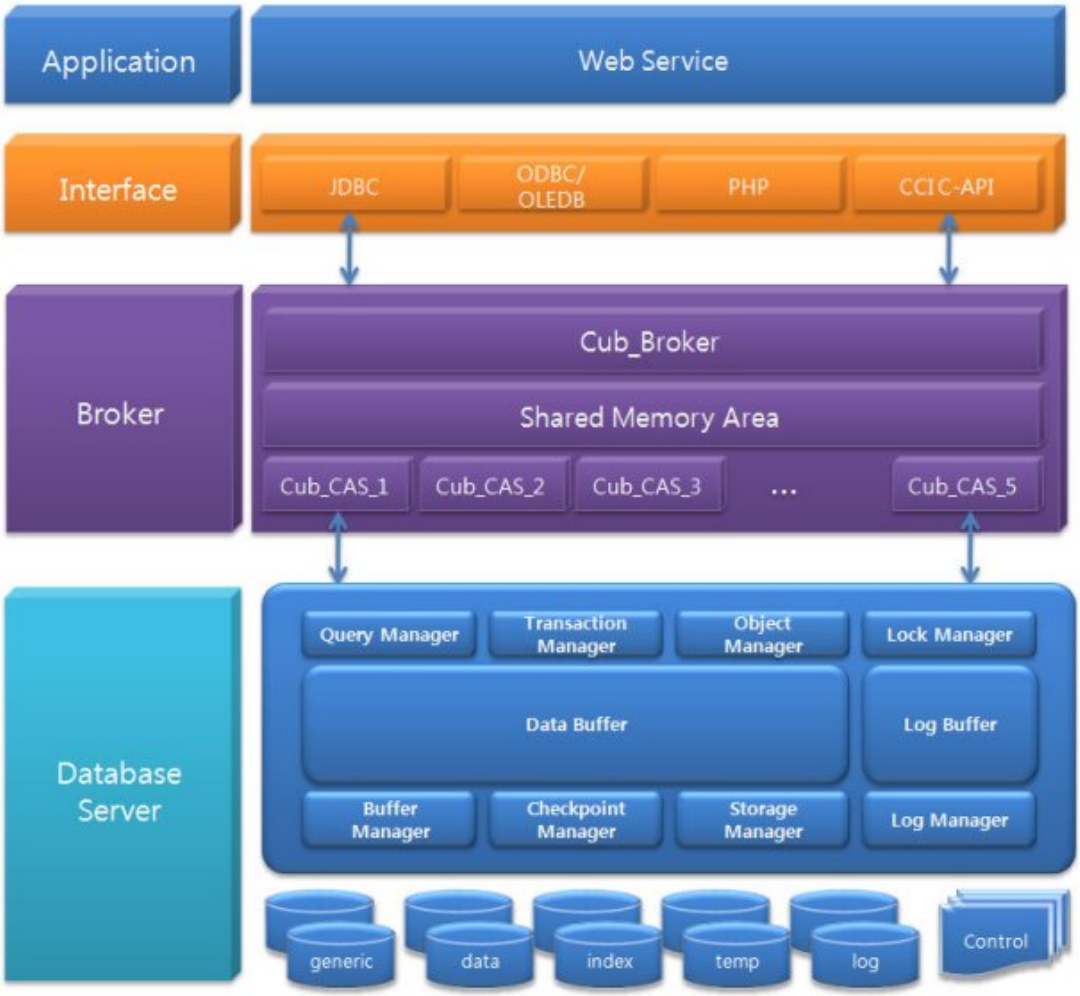
This chapter covers the following topics:

- System Architecture
- Features of CUBRID

System Architecture

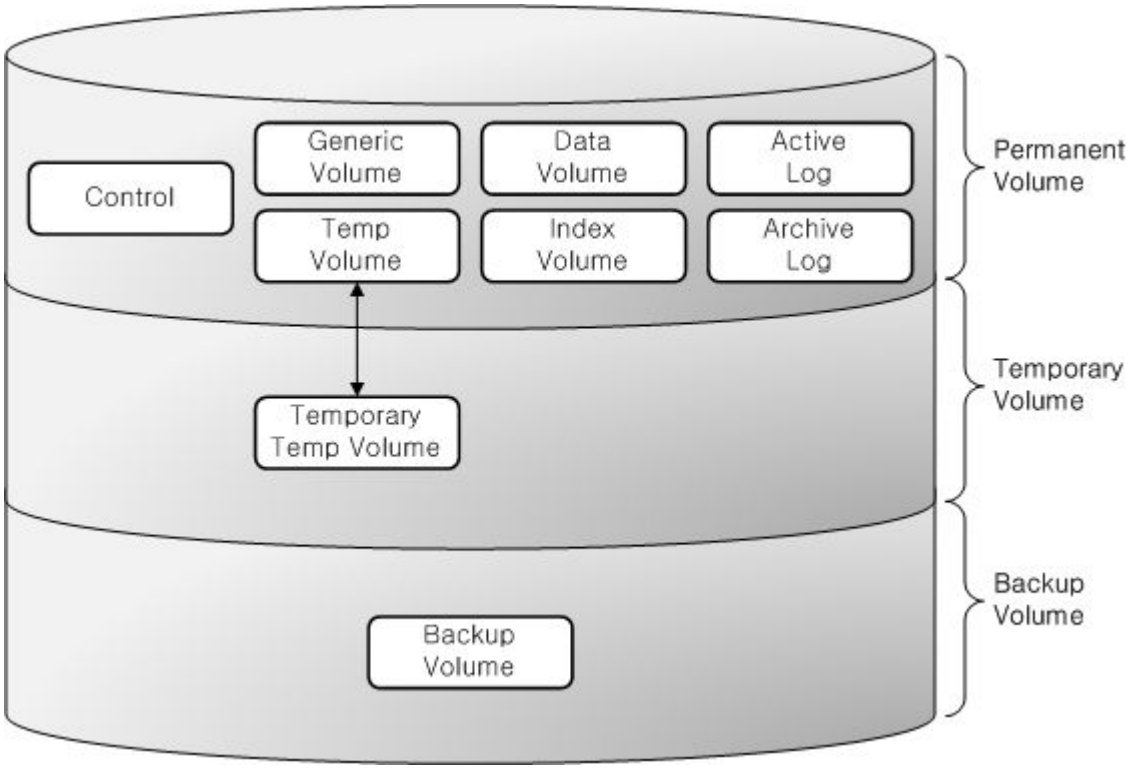
CUBRID is an object-relational database management system (DBMS) consisting of the Database Server, the Broker, and the CUBRID Manager.

- As the core component of the CUBRID Database Management System, the Database Server stores and manages data in a 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, you can specify 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 temporary volume 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 that is not necessary for a rollback or database restore. 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 temporarily to free up disk space needed for joining/sorting or index creation. Examples of such large-scale queries 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 transfers a socket to server process so that the server process can handle connection.

Execution Mode

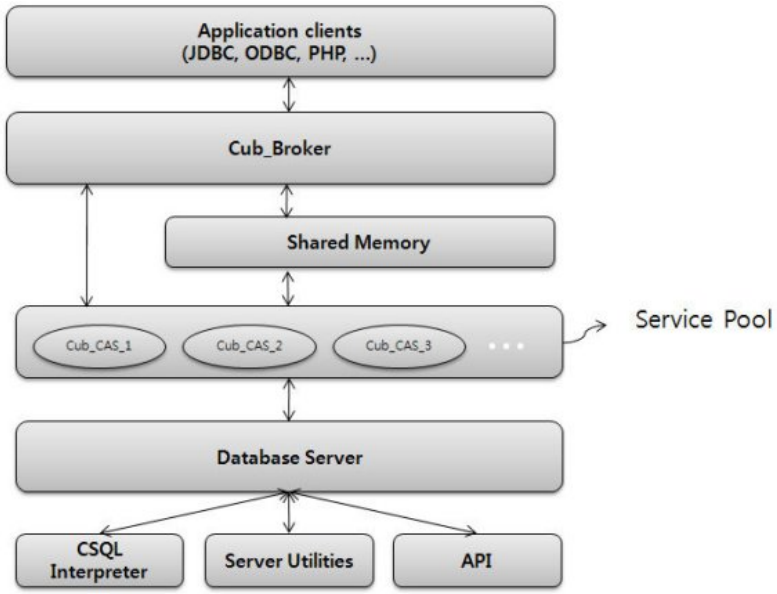
All CUBRID utilities except the server process have two execution modes: client/server mode and standalone mode.

- 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 a 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.

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

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. Collection data types are divided into **`SET`**, **`MULTISET`** and **`LIST`** depending on whether the duplication of elements is allowed or not.

- **`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 minimizes 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.

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.

How to Use CSQL (Syntax)

Running on a Local Host

Run 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
```

Running on a Remote Host

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 R2.0
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
```

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.
   ;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.
```

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> ;re 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 output file **outfile**.

```
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 SQL input from the user, **;REad** command, **;EDIT** command and so on.

```
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. If the CSQL Interpreter executes the command with auto-commit turned off, the changes made in the current transaction become permanent in the database. If the CSQL Interpreter is running in the auto-commit mode, the transaction is committed automatically whenever executing the **;Run** and **;Xrun** session commands.

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

- **Rolling back transaction (;ROllback)**

  This command rolls back the current transaction. As with **;COmmit**, it has no effect unless the Interpreter is running in the auto-commit mode.

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

- **Setting the auto-commit mode (;AUtocommit)**

  This command sets the auto-commit mode to **ON** or **OFF**. If you have not set the mode to **ON** or **OFF**, the current setting is displayed. Note that the CSQL Interpreter runs in an auto-commit mode by default.

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

- **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 information (;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 class. The information includes the class name, its properties 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> ===
     glo_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
......
```

- **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**. Use **;EDITOR_Cmd** command to specify a different editor.

```
csql> ;edit
```

- **Specifying the editor (;EDITOR_Cmd)**

  This command changes the editor to be used with **;EDIT** session command. As shown in the example below, you can specify **emacs** instead of the default editor **vi**, and then call the **emacs** editor.

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

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 "CUBRID Service" 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
- Database loading Utilities
  - cubrid loaddb

    - cubrid unloaddb
- Database backup/restore utility
    - cubrid backupdb
    - cubrid restoredb
- HA utilities
    - cubrid chagemode
    - 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 service utility, version 8.2.0
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

Available administrator's utilities:
     addvoldb
     alterdbhost
     backupdb
     checkdb
     compactdb
     copydb
     createdb
     deletedb
     diagdb
     emergency_patchlog
     installdb
     killtran
     loaddb
     lockdb
     optimizedb
     renamedb
     restoredb
     spacedb
     unloaddb
     paramdump
     changemode
     copylogdb
     applylogdb

cubrid is a tool for DBMS.
For additional information, see http://www.cubrid.com
```

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, rather than by the following commands.

```
% 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).

```
% 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 R2.1……

++ cubrid server start: success
@ cubrid server start: testdb

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

CUBRID 2008 R2.1……

++ 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 restore works to do.

CUBRID 2008 R2.1 ...

++ cubrid server start: success
@ cubrid server start: testdb

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

CUBRID 2008 R2.1 ...

++ cubrid server start: success
@ cubrid broker start
++ cubrid broker start: success
```

Checking the 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

NAME            PID    PORT    AS    JQ  REQ  TPS  OPS  AUTO    SES    SQLL
  CONN
================================================================================
========
* query_editor  336   30000     5      0  0   ---  ---   ON     OFF     ON:A
AUTO
* broker1       5100  33000     5      0  0   ---  ---   ON     OFF     ON:A
AUTO

@ 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 the 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 restore works to do.

CUBRID 2008 R2.1

++ 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 restore works to do.

CUBRID 2008 R2.1

++ 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 restore works to do.

CUBRID 2008 R2.1

++ cubrid server start: success
```

Checking the 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 the 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 already 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 already been stopped.

```
% cubrid broker stop
@ cubrid broker stop
++ cubrid broker is not running.
```

Checking the Broker Status

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 [args]
options : [ -b | -q | -s secs | -t | -f ]
args : broker_name
```

Options

The following table shows options that can be used together with cubrid broker status.

Options	Description
broker_name	Displays the status of the specified Broker. If this option 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.
-q	Displays standby jobs in the job queue.
-s secs	Displays the status of the Broker regularly according to the specified time period. Returns to the command prompt if q is entered.

-t	Displays the output in tty mode. The output can be redirected to a file.
-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.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
```

Managing a 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

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_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

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
```

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 **auto_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. 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 1111 -T 1112 -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 1111 -t 1112 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
% 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]
```

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 the 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 already 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

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 databse, 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

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 a Database

The **cubrid createdb** utility creates databases and initializes them with the built-in CUBRID system classes. 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] [--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] [ {-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 saved. Default value : The directory path specified with the **-F** option
-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
-s --page-size	Specifies the database page size in bytes. Default value : The page size optimized for I/O of the operating system
--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)**

```
cubrid createdb -p 10000 testdb
```

The above 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.

- **Database directory path (-F)**

```
cubrid createdb -F /dbtemp/new_db/ testdb
```

The above 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.

- **Log file directory path (-L)**

```
cubrid createdb -F /dbtemp/new_db/ -L /dbtemp/db_log/ testdb
```

The above 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.

- **Overwrite (-r)**

```
cubrid createdb -r testdb
```

The above 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.

- **Saving output messages to a file (-o)**

```
cubrid createdb -o db_output testdb
```

The above 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.

- **Verbose output (-v)**

```
cubrid createdb -v testdb
```

The above 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.

- **Log page (-l)**

```
cubrid createdb -l 1000 testdb
```

The above 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.

- **Page size (-s)**

```
cubrid createdb -s 8192 testdb
```

The above 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. 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.

- **Comment (--comment)**

```
cubrid createdb --comment "a new database for study" testdb
```

The above 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.

- **Server host name (--server-name)**

```
cubrid createdb --server-name aa_host testdb
```

The above 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.

- **Adding a database volume (--more-volume-file)**

```
cubrid createdb --more-volume-file vol_info.txt testdb
```

The above 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.

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.

```
#xxxx
```

by using the CSQL Interpreter. A schema can be created based on the the SQL statement contained in the file specified by the parameter.

- **User information file (--user-definition-file)**

```
cubrid createdb --user-definition-file user_info.txt testdb
```

The above 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 **`CREATE USER`** statement.

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 a Database Volume

For how to add new volumes to a database by using the CUBRID Manager, see "Storage."

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)**

```
cubrid addvoldb -S -n testdb_v1 testdb 1000
```

The above 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.

- **Path of the extended volume (-F)**

```
cubrid addvoldb -S -F /dbtemp/addvol/ testdb 1000
```

The above 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.

- **Purpose of the volume (-p)**

```
cubrid addvoldb -S -p index testdb 1000
```

The above 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."

- **Standalone mode (-S)**

```
cubrid addvoldb -S testdb 1000
```

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.

- **Client/server mode (-C)**

```
cubrid addvoldb -C -testdb 1000
```

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 defind, the system assumes to be in client/server mode.

- **Comment about the added volume (--comment)**

```
cubrid addvoldb -S --comment "data volume added_cheolsoo kim" testdb 1000
```

The above 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.

Deleting a Database

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)**

```
cubrid deletedb -o deleted_db.out testdb
```

By using the **-o** option, the above example shows deleting testdb and writes output messages to the file specified as an argument in the option. 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`**)**

```
cubrid deletedb -d testdb
```

By using the **-d** option, the above example shows deleting testdb and its backup volumes and backup information files simultaneously. If the **-d** option is not specified, backup volume and backup information files are not deleted.

Renaming a Database

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)**

```
cubrid renamedb -E /dbtemp/newaddvols/ testdb testdb_1
```

The above 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.

- **Specifying the input file where the directory information is stored (-i or --control-file)**

```
cubrid renamedb -i rename_path testdb testdb_1
```

The above 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.
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_x
001
```

- **Deleting and renaming backup files simultaneously (-d or --delete-backup)**

```
cubrid renamedb -d testdb testdb_1
```

By using the **-d** option, the above 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.

Installing a Database

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)**

```
cubrid installdb --server-name=cub_server1 testdb
```

The example above 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.

- **Registering the directory path of a database volume (-F or –file-path)**

```
cubrid installdb –F /home/cubrid/CUBRID/databases/testdb testdb
```

The example above 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.

- **Registering the directory path of a database log volume (-L or –log-path)**

```
cubrid installdb –L /home/cubrid/CUBRID/databases/logs/testdb testdb
```

The example above 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.

Checking and Compacting Used Space

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.

The **cubrid compactdb** utility is used to secure unused space of the database volume. It 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.

Checking Used Space

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)**

```
cubrid spacedb -o db_output testdb
```

The above example shows saving the result of checking the space information of testdb to a file named *db_output*.

- **Executing in stand-alone mode (-S or --SA-mode)**

```
cubrid spacedb --SA-mode testdb
```

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 recognize that a database is running in client/server mode.

- **Executing in client/server mode (-C or --CS-mode)**

```
cubrid spacedb --CS-mode testdb
```

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.

- **Outputing in megabytes (--size_unit=M)**

```
cubrid spacedb --size_unit=M testdb
```

- **Outputing in print-friendly version (--size_unit=H)**

```
cubrid spacedb --size_unit=H testdb
```

The unit is automatically determined as follows: M if 1MB = DB size < 1024MB, G if 1GB = DB size < 1024GB.

Compacting Space

Syntax

```
cubrid compactdb options database_name
options : [-v | --verbose]
```

- **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. It executes successfully only when the database is in a stopped state.
- *options* : The **-v** option is supported.
- *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.

Option

- **Displaying detailed messages during execution (-v)**

```
cubrid compactdb -v testdb
```

The above example shows displaying on the screen a message informing which class is currently being compacted and how many instances have been processed for the class by using the **-v** option.

Updating Statistics

Updates statistical information such as the number of objects, the number of pages to access, and the distribution of attribute values; such information exists in a tables used by the CUBRID query optimization.

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**

```
cubrid optimizedb testdb
```

The example above shows updating the query statistics information of all classes in the database.

- **Updating the query statistics of a specific class in the database (-n or --class-name)**

```
cubrid optimizedb -n event_table testdb
```

By using the **-n** option, the example above shows updating the query statistics information of the given class.

Dumping Statistic

The **cubrid statdump** utility allows you to view the statistics information which is generated by executing the CUBRID database server.

Syntax

```
cubrid statdump options database_name
options : [{-o|--output-file=} file_name]] [{-i|--interval=} secs]
```

- **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 and -i** options are available.
- *database_name* : The name of database which has the statistics data to be dumped.

Option

- **Storing the statistics information about database server execution to a file (-o or --output-file)**

  ```
  cubrid statdump -o statdump.log testdb
  ```

  The example above shows dumping the statistics information about the database server execution to a specified file by using an **-o** option.

- **Dumping the statistics information about database server execution periodically (-i or --interval)**

  ```
  cubrid statdump –F statdump.log testdb
  ```

  The example above shows dumping the statistics information about the database server execution at a specified interval (in seconds) by using an **-i** option.

Checking the Lock Status

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**

  ```
  cubrid lockdb testdb
  ```

  The above example shows displaying lock information of the testdb database on a screen without any option.

- **Displaying the lock information to the specified file (-o)**

  ```
  cubrid lockdb -o output.txt testdb
  ```

  The above example shows displaying lock information of the testdb databse as a output.txt by using the -o option.

Checking Database Consistency

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)**

```
cubrid checkdb -S testdb
```

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 recognize that a database is executing in client/server mode.

- **Checking the database consistency in client/server mode (-C or --CS-mode)**

```
cubrid checkdb -C testdb
```

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.

- **Repairing in case of a database consistency problem (-r or --repair)**

```
cubrid checkdb -r testdb
```

The **-r** option is used to repair an issue if a consistency error occurs in a database.

Killing Database Transactions

The **cubrid killtran** utility is used to kill transactions from a specific database based on the option specified.

Syntax

```
cubrid killtran options  database_name
options : [{-i|--kill-transaction-index=}index]  [--kill-user-name=id]  [--
kill-host-name=host]
            [--kill-program-name=name]  [--kill-program-name=name] [{-p|--
dba-password=}password]
            [-d|--display-information]  [-f|--force]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.
- **killtran** : Kills transactions from a specific database based on the option specified.
- *options* : Specifies the transactions to be killed. If no option is specified, all transactions in the given database are killed. You must enter the password of **DBA** after the **-p** option; if not, a prompt will enter it.
- *database_name* : The name of the database whose transactions are to be killed.

Options

- **Viewing all transactions in the testdb database**

```
cubrid killtran testdb
```

- **Killing transactions in the specified index (-i or --kill-transaction-index)**

```
cubrid killtran -i test_index testdb
```

- **Killing transactions for the specified user ID (--kill-user-name)**

```
cubrid killtran --kill-user-name myuser testdb
```

- **Killing transactions for the specified client host (--kill- host-name)**

```
cubrid killtran --kill-host-name myhost testdb
```

- **Killing transactions for the specified program (--kill-program-name)**

```
cubrid killtran --kill-program-name my_test_program testdb
```

- **Displaying information about active transactions (-d or --display)**

```
cubrid killtran -d testdb
```

- **Omitting a prompt to check transactions to be stopped (-f or --force)**

```
cubrid killtran -f -i 1 testdb
```

Checking the Query Plan Cache

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

The **cubrid emergency_patchlog** utility restores the log file of a corrupted database log file, or creates a new one afte compacting it. This utility can be used for the emergency restore when the server does not restart due to the corruption which has occurred to the database log file.

Syntax

```
cubrid emergency_patchlog options database_name
options : [ -r | --recreate-log ]
```

- **cubrid** : An integrated utility for the CUBRID service and database management.

- **emergency_patchlog** : A utility used to restore the corrupted database log file when it fails to restart such as system crash. It is used to restore the log file. It normally executes only when the database is in a stopped state.
- *options* : The **-r** option, which is used to compact the log file and create a new one, is supported. Thus, it is recommended that you executes the utility without the **-r** option for the first time. You should use this option in the worst-case situation where the log file is beyond restore.
- *database_name* : The name of the database to be installed.

Option

- **Restoring with the existing log**

```
cubrid emergency_patchlog testdb
```

- **Discarding the existing log and creating a new empty log (-r)**

```
cubrid emergency_patchlog -r testdb
```

Outputting Internal Information of a Database

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)

```
cubrid diagdb -d 1 myhost testdb
```

The above example displays the information of all files in the testdb database. If any option is not specified, the default value of 1 is used. The utility has 8 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.

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 "How to Use CUBRID Backup and Restore Utilities."

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 a Database."

Dumping Parameters

The **cubrid paramdump** utility displays current information of the parameters 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 command that displays information of the parameters 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 (**--**). The **-o**, **-b**, **-S** and **-C** options are available.
- *database_name* : The name of the database in which parameter information is to be displayed.

Option

- **Saving the output information to a file (-o)**

  ```
  cubrid paramdump -o db_output testdb
  ```

  The above command saves current information of the parameters used in the testdb database server into the db_output file instead of displaying on the console screen.

  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.

- **Displaying information of the server/client parameters (-b)**

  ```
  cubrid paramdump -b testdb
  ```

  The above command displays current information of the parameters used in the testdb database server on the console screen.

  The **-b** option is used to save information of the parameters used in the server/client process of the database into a specified file. If the **-b** option is not specified, only the parameter information of the server process is displayed.

- **Displaying the current parameter information of the server process in standalone mode (-S or --SA-mode)**

  ```
  cubrid paramdump -S testdb
  ```

- **Displaying the current parameter information of the server process in client/server mode (-C or --CS-mode)**

  ```
  cubrid paramdump -C testdb
  ```

Database Migration

Migrating a 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 a Database" and "Loading a Database."

- **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 "How to Use CUBRID Backup and Restore Utilities."

- **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 a Database."

- **Save configuration files of the existing version of CUBRID**

  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.

- **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."

- **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."

- **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 a Database" in "Administrator's Guide." For more information about database loading, see "Loading a Database."

- **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 "How to Use CUBRID Backup and Restore Utilities."

Unloading a Database

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 instances 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 instances 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)**

  -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; a line breaker (\n) must be included at the end of the input file.

  ```
  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 instances 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 instances (--estimated-size)**

  The **--estimated-size** option allows you to assign hash memory to save instances of the database to be unloaded. If the **--estimated-size** option is not specified, the number of instances 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 a Database

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 instances 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 instances.
-v --verbose	Displays detailed information about the data loading status on the screen.
-c --periodic-commit	Commits the transaction whenever a specified number of instances 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 instances.
--data-file-check-only	Performs checking only for statements and types included in the data file, but does not load instances.
--estimated-size	Specifies the number of instances expected.
--no-oid	Ignores the OID reference relationship included in the data file and loads instances.
--no-statistics	Loads instances 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 instances 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 instances. If the option is not specified, you will be prompted to enter the password.

```
cubrid loaddb -p admin -d demodb_objects newdb
```

- **Loading instances 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 instances 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 instances 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 instances are entered into the newdb by using the **-c** option. If the **-c** option is not specified, all instances 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 instance 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 instances 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 instances 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 instance 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 instances.

```
cubrid loaddb --data-file-check-only -d demodb_objects newdb
```

- **Number of expected instances (--estimated-size)**

  The **--estimated-size** option can be used to improve loading performance when the number of instances to be unloaded exceeds the default value of 5,000. That is, you can improve the load performance by assigning large hash memory for instance storage with this option.

```
cubrid loaddb --estimated-size 8000 -d demodb_objects newdb
```

- **Loading instances while ignoring the reference relationship (`--no-oid`)**

  The following is a command that loads instances into newdb ignoring the OIDs in demodb_objects.

```
cubrid loaddb --no-oid -d demodb_objects newdb
```

- **Loading instances 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 instances. All instances 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 a File to Load a 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 21
%id office 22
%id project 23
%id phone 24
```

Specifying the class and attribute

You can specify the classes and attributes 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 a "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
```

How to Use CUBRID Backup and Restore Utilities (Syntax)

Backup

The **cubrid backupdb** utility is used to backup a database to restore it in case of media failure or database file damage. To restore 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 --thread-count	Specifies the maximum number of threads allowed for a parallel 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 are 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 files of demodb are stored in the /home/cubrid/backup directory. 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.

- **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
  ```

Restore

The **cubrid restoredb** utility restores the database by using the active and archive logs created 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 utility for restore of the specified database. For a successful restore, you must prepare backup files, active log files and archive log files. This utility can be performed only in standalone mode.
- *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 restored.

Return Value

- 0 : Success
- Non-zero : Failure

Option

The following table shows options that can be used with **cubrid restoredb**. Note that options are case sensitive.

Option	Description
-d --up-to-date	Directly sets the time to back up the database or specifies the **backuptime** keyword.
-B --backup-file-path	Specifies the directory path name or device name where backup files are to be located.
-l --level	Configures the restore level to 0, 1 or 2. The default value is a full restore (0).
-p --partial-recovery	Performs a partial restore.
-o --output-file	Specifies the name of the file where restore information is to be displayed.
-u --use-database-location-path	Restores 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 restore by specifying a restore point (-d or --up-to-date)**

  The following utility restores **demodb**. If no option is specified, **demodb** is restored to the point of the last commit by default. If no active/archive log files are required to restore to the point of the last commit, the database is restored only to the point of the last backup.

  ```
  cubrid restoredb demodb
  ```

  demodb can be restored to the given point by using the **-d** option and the syntax which specifies the date and time of the restore. The user can specify the restore 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 restore to the point specified, the database is restored only to the point of the last backup.

```
cubrid restoredb -d 14-10-2008:14:10:00 demodb
```

The following utility specifies the restore point by using the **-d** option and the **backuptime** keyword and restores **demodb** to the point of the last backup.

```
cubrid restoredb -d backuptime demodb
```

- **Performing a restore 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 restore by specifying the backup level (-l or --level)**

  You can perform a restore by specifying the backup level of the database to 0, 1, or 2. For more information about backup levels, see "Incremental Backup."

  ```
  cubrid restoredb -l 1 demodb
  ```

- **Performing a partial restore (-p or --partial-recovery)**

  The following example performs a partial restore 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 restore can be performed directly without such a request message by using the **-p** option. Therefore, if the **-p** option is used when performing a restore, data is always restored 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 restore.
- Option 1 : An administrator enters 1 to perform a partial restore without log files.
- Option 2 : An administrator enters 2 to perform a restore after moving archive logs to the current device.
- Option 3 : An administrator enters 3 after changing a log location to resume a restore.

- **Saving restore progress information in the specified file (-o or --output-file)**

  The following example writes the restore progress of the database to the info_restore file by using the **-o** option.

```
cubrid restoredb -o info_restore demodb
```

- **Restoring data to the directory specified in the database location file (-u or --use-database-location-path)**

  The following example restores 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 restore 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 a restore.

```
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 restore the database in the example, you must prepare backup files for backup levels 0 and 1.

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.

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.

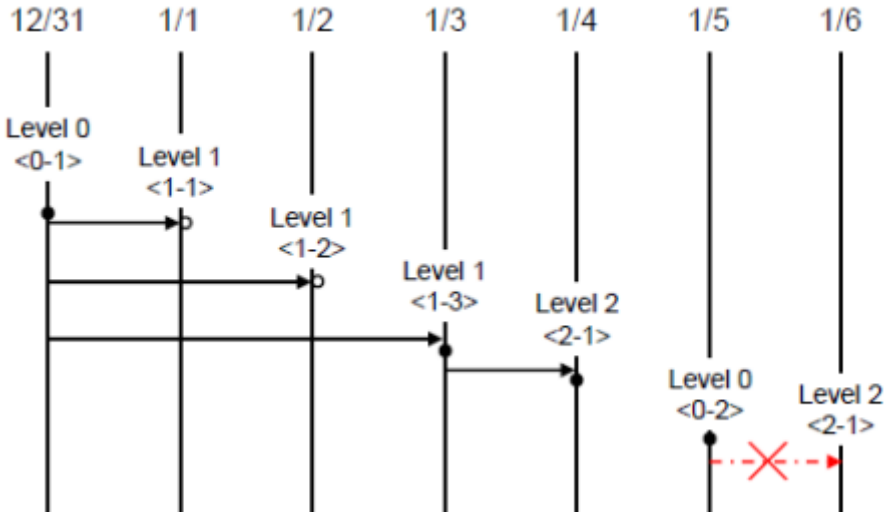
To perform an offline backup, use the **`cubrid backupdb -S`** command.

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 a Backup File

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.

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.

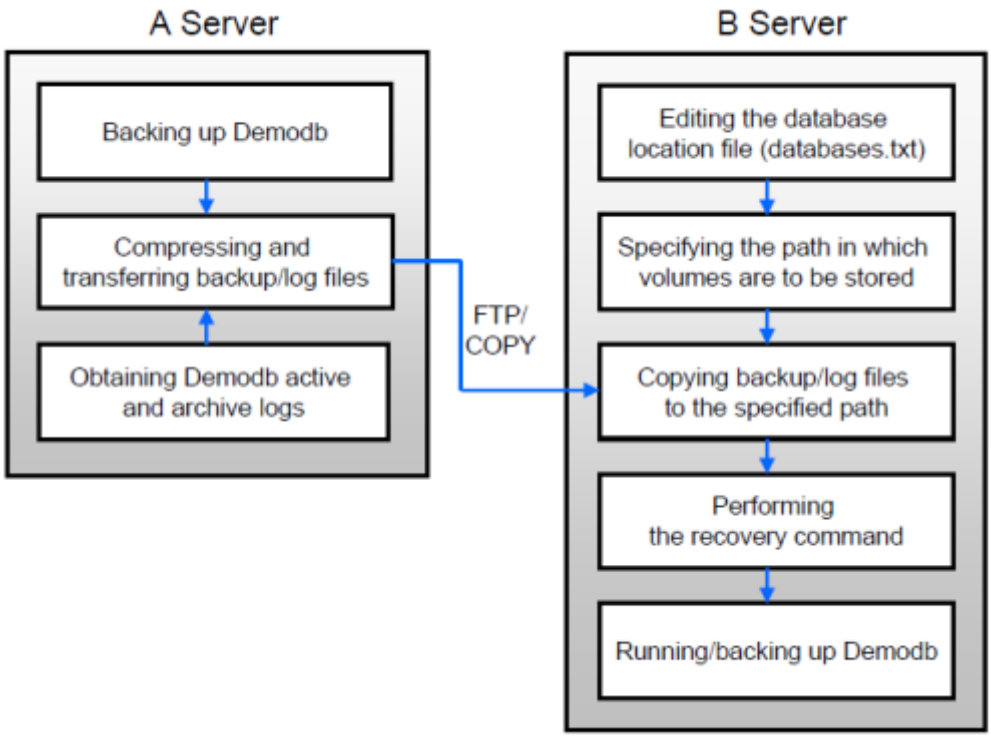
Time	Command	Description
2008/8/14 04:25	cubrid server stop demodb	Shuts down demodb.
2008/8/14 04:30	cubrid backupdb -S -D /home/backup -l 0 demodb	Performs a full backup of demodb in offline mode and creates backup files in the specified directory.
2008/8/14 05:00	cubrid server start demodb	Starts 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 15:30		A system failure occurs.
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 a Database to a 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.



- **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 "How to Use CUBRID Backup and Restore Utilities."

  ```
  cubrid backupdb -z -t demodb
  ```

- **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
  ```

- **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.

- **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
```

- **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:

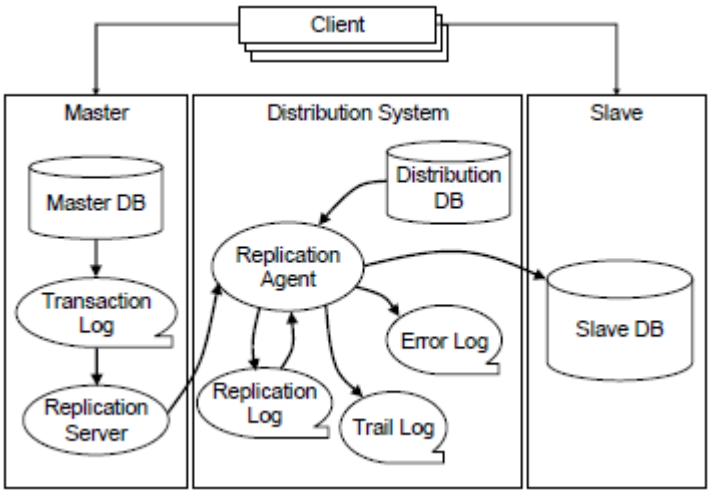
- To improve performance by distributing access of applications using the same object to multiple database servers.
- To satisfy different operational requirements by using the replicated database server for other purposes.
- To urgently cope with failure by switching to the replicated database server when a failure occurs in the currently running database server.

Note Currently, replication feature is supported only on Linux series; it is not supported on Windows.

Architecture and Terms of the 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.

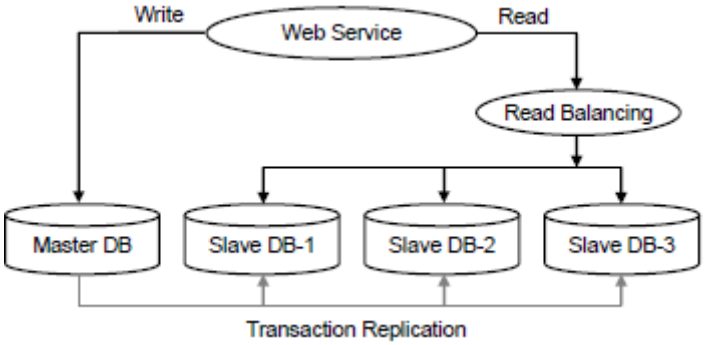
Slave Database

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

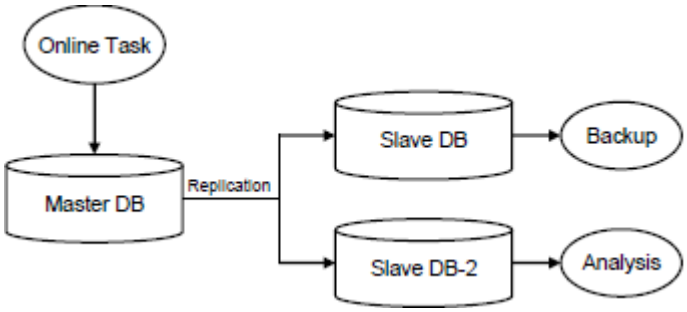
If you set a slave using database replication, you can respond effectively by replacing the master with the slave when a failure occurs in the master. Recovery functions using the master database backup might be enough depending on the type of the failure. However, if a failure occurs on the master host, 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

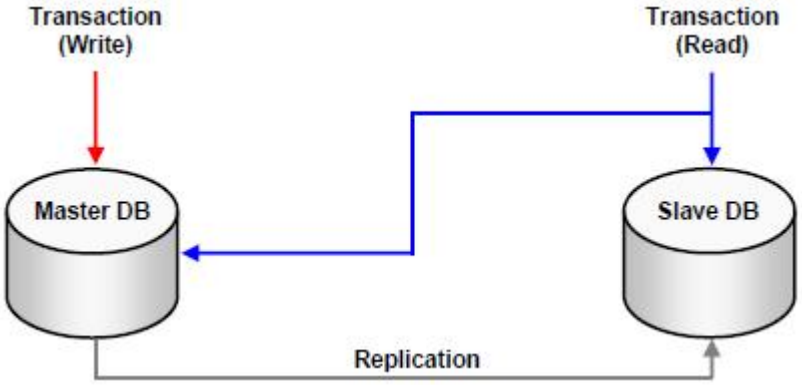
operations that might affect the online tasks (e.g. backup, analysis, etc.) in any time period, you can perform batch operations not affecting the master by creating a slave as shown below.



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 systme 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

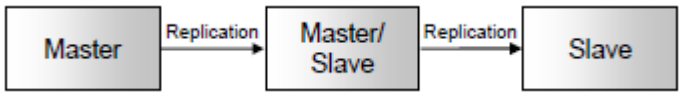
Using the Primary Key

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

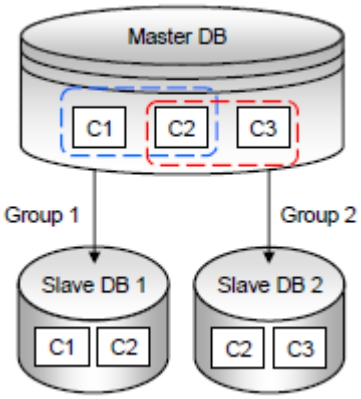
Replication performed in a hierarchical manner is called chained replication. It means that a slave database can become a master database at the same time, so it can transfer changed replication data to the other slave.



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.

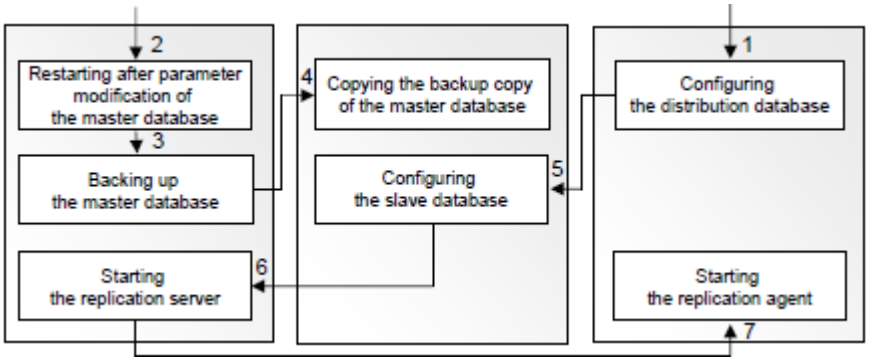


Such method is useful when you want to configure the slave server for a specific purpose. For example, if you want to build up the slave server by replicating only statistics data, you can do such a work by specifying a replication group.

Configuration

Replication Setup Procedure

You can make the replication as follows:



- **Configure the distribution database**

  Run the **repl_make_distdb** utility to create the distribution database and to configure parameters for the replication.

- **Change the replication parameters of the master database**

  Modify the value of replication in the **cubrid.conf** file to **yes**. If anything does not occur, it has already been set to **yes**.

  You must restart the master database if **cubrid.conf** file is modified.

- **Back up the master database**

  Back up the master database with full level.

- **Transfer backup volumes and an information file**

  Copy backup volumes and an information file to the slave host.

- **Configure the slave database**

  Run the **repl_make_slavedb** utility to restore a database with backup files of the master and to configure parameters in the slave host.

- **Start the** `repl_server` **utility**

  Start the **repl_server** utility in the master host.

- **Start the** `repl_agent` **utility**

  Start the repl_agent utility in the slave host.

This procedure is completed successfully. All changes of the master after backup will be synchronized to slave.

Configuring the Distribution Database

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 a 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 a 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 a Replication Group.

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 the Replication Server and Agent

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

- **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 a 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 and 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 a Replication Group

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

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.

- **perf_poll_interval**
  - Allowable range : 10 - 60
  - A parameter that measures the replication delay time (in seconds). The default value is **10**.
- **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**.
- **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**.
- **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**.
- **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**.
- **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**.

- **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**.
- **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 a 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 a 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

```
92.168.2.100 5627
92.168.2.200 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	1234/123	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   EQ   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        delay
```

```
------------------------------------------------------------------------------
---
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 synchorized 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 .pert (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

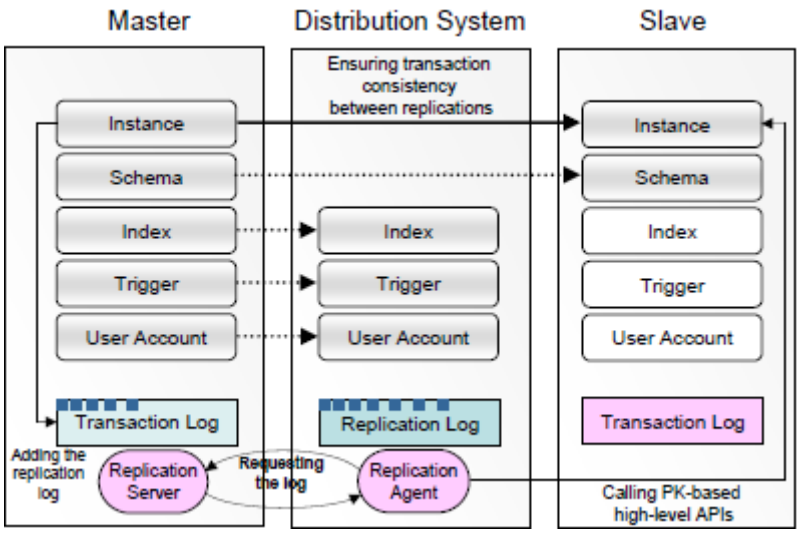
Implementing Replication

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

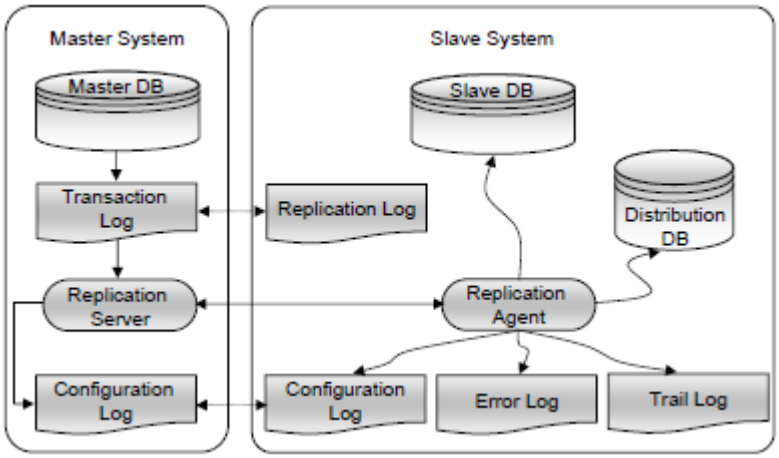
synchronized if the master database is replaced by the slave database due to a system failure in the master; however, triggers and user accounts are not synchronized automatically.

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 superclass 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 Constraints.
- 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 a 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. After the failback, the synchronization process is performed automatically to maintain the same data.

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. Heartbeat is installed and runs in every system where a CUBRID server operates.

Note As with the replication feature, the log of a table with the primary key can be multiplexed.

Caution The CUBRID HA feature is supported only in Linux-based systems.

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 automatically 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 automatically-restored active server to resume services after the failover when it is restored to the original state.

Role Change

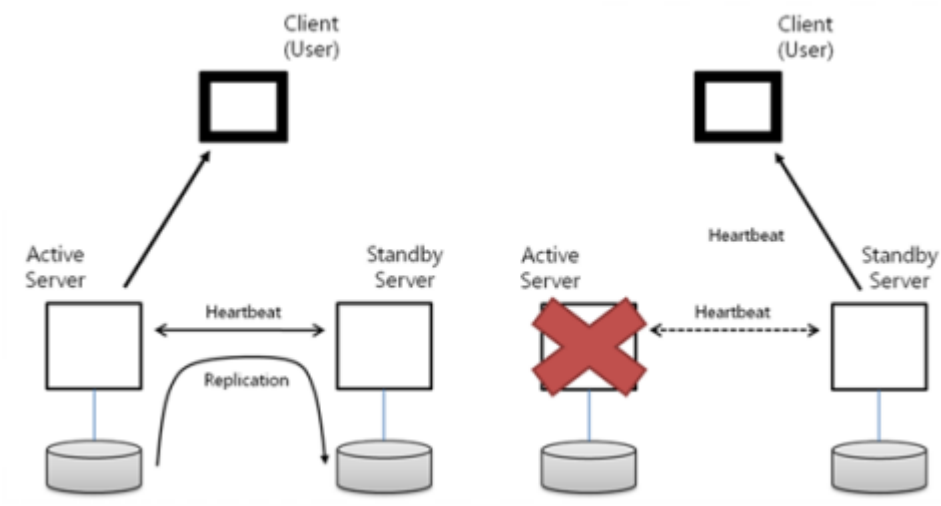
A feature that allows services to be provided continuously without failback even when the failure in the previous active server is restored.

Heartbeat

An essential element for providing HA features. This feature checks the availability of important resources comprising the system such as data, application programs and services, and provides failover, failback and monitoring functionalities.

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).



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.

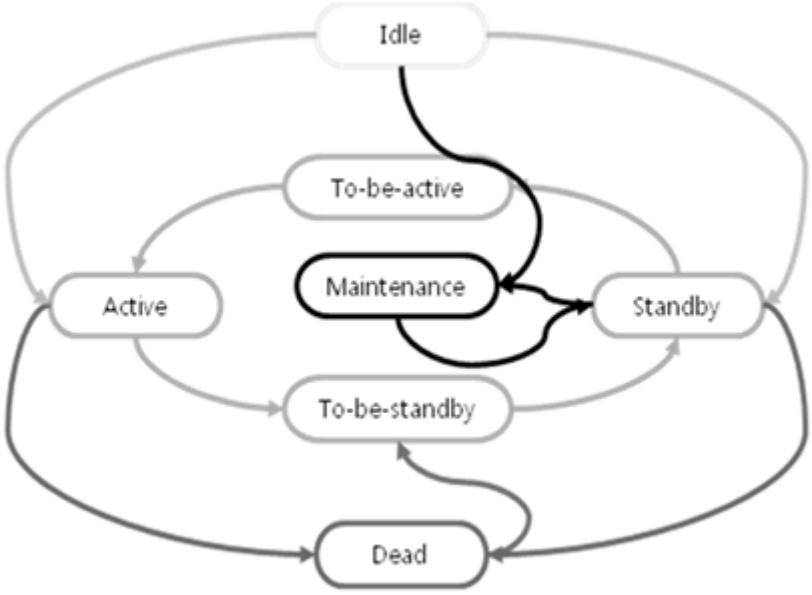
- **`Read-only broker`** : A broker that performs read operations only. It provides services through the connection with a standby server. If a standby server does not exist, it can send a read request to an active server. That is, the order of attempting to connect to a database server is as follows: 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.

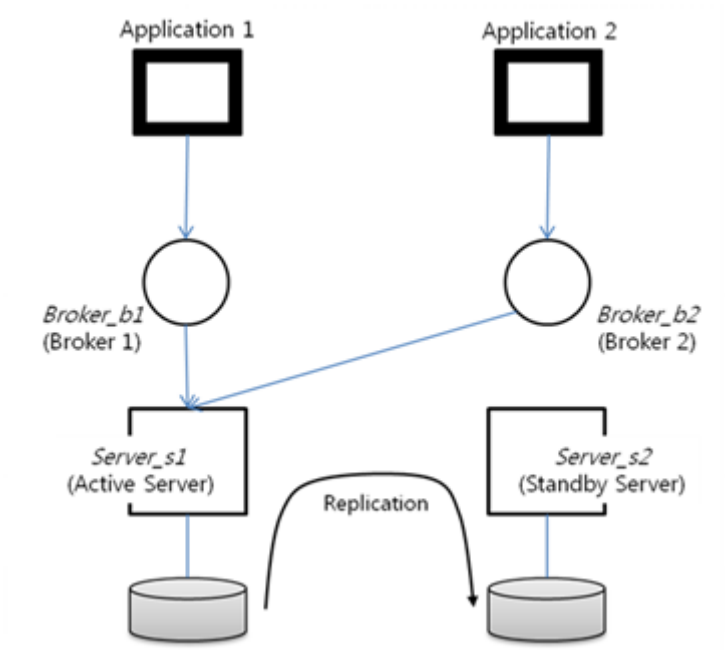


- State change of the database server during failover
  - active server: active -> dead
  - standby server: standby -> to-be-active -> active
- State change of the database server during failback
  - active server: active -> to-be-standby -> standby
  - failure restore server: to-be-standby -> standby -> to-be-active -> active

HA Configuration

To use the HA feature, the following configuration is required.

System Architecture



To perform failover and failback, the Heartbeat solution of Linux-HA project should be installed in each system where the database server is running. For Heartbeat configuration, see a separate Heartbeat installation document.

You can also configure the system in a way it can more flexibly address failures that may cause the system to stop by configuring the broker and the database in separate systems.

Configuring the Database Server

To use the HA feature, you should set the **ha_mode** parameter to **on** in the **cubrid.conf** file. If the **ha_mode** parameter is not set, the default value is **off**. Then, it runs as a general server without the HA feature.

```
#cubrid.conf
ha_mode=on
```

To see if the database server is using the HA feature and which HA mode is running, the **cubrid changemode** utility can be used. You can also use the CSQL command (;database) to check the HA mode of the currently runing database server. The HA mode is one of the followings: active, standby, or maintenance.

```
% cubrid changemode demodb
The server 'demodb''s current HA running mode is active.
```

```
csql> ;database
```

```
      demodb@localhost (active)
```

Configuring the Broker

The default operation mode of the broker is one that requires read and write operations. If necessary, 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 the 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 (**database.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 (:).

```
#databases.txt
#db-name         vol-path         db-host                    log-path
nbd2            /home/db/db2      server_s1:server_s2        /home/db/db2
nbd1            /home/db/db1      server_s1:server_s2        /home/db/db1
```

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

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.

```
cubrid changemode [option] <database_name>@<hostname>
option:
-m, --mode=<MODE> : Specifies the mode to change. Available values are active, standby
and maintenance.
```

Saving the Transaction Log of the Database Server

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:

```
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.
```

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 copylog –m** value.
- Semi-Synchronous : A database server sends all transaction logs to the copylog 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 copylog –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 copylog –m** value.

Reflecting the Stored Transaction Log to the Database

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:

```
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 See the **APPL_SERVER_MAX_SIZE** description to get information about the requirements of CAS process restart; For **cubrid applylogdb**, you can specify the maximum memory size directly by using the **--max-mem-size** option.

HA-Related JDBC Configuration

System Requirements

- JDK 1.5 or higher
- CUBRID 2008 R2.0 or higher
- CUBRID JDBC Driver 2008 R2.0 or higher

Connection

Syntax

```
<url> := jdbc:CUBRID:<host>:<dbname>:<username>:<password> [?<properties>]
```

```
<properties> := <property> [&<property>]
<property> := ... | althosts=<alternative_hosts> | rctime=<time>
<alternative_hosts> := <host>:<port> [,<host>:<port>]
<host> := HOSTNAME | IP_ADDR
<time> := SECOND
```

Description

To use the HA feature in JDBC, add the following keywords to the existing JDBC connection method.

- **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 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

In the following example, if a failure occurs in Broker_b1 (default broker), the system attempts to make a connection to Broker_b2 and then to Broker_b3. If the failover is performed to the broker, the system attempts failback at every 600 seconds.

```
URL=jdbc:CUBRID:Broker_b1:31000:nbd:nbd:nbd:?althosts=Broker_b2:31000,Broker_
b3:31000&rctime=600
```

HA-Related System Catalog

Class Name : 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. 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

Scope of the Dtabase 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 configured in the **cubrid.conf** file, are divided into client and server parameters depending on their application scope. (Some are client as well as server parameters.) Client parameters affect a client process such as the Broker; server parameters affect a server process.

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.

The cubrid_broker.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 the 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
block_ddl_statement	client parameter	bool	no
block_nowhere_statement	client parameter	bool	no
intl_mbs_support	client parameter	bool	no
oracle_style_empty_string	client parameter	bool	no
data_buffer_pages	server parameter	int	25000
dont_reuse_heap_file	server parameter	bool	no
index_scan_oid_buffer_pages	server parameter	int	4
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
auto_restart_server	server parameter	bool	yes
deadlock_detection_interval_in_secs	server parameter	int	1
file_lock	server parameter	bool	yes
isolation_level	server parameter	int	3
lock_escalation	server parameter	int	100000
lock_timeout_in_secs	server 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
log_flush_interval_in_msecs	server parameter	int	1000
pthread_scope_process	server parameter	bool	yes
backup_volume_max_size_bytes	server parameter	int	-1
checkpoint_interval_in_mins	server parameter	int	720

log_buffer_pages	server parameter	int	50
media_failure_support	server 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
compat_numeric_division_scale	client/server parameter	bool	no
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_client	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 default value is **50**. However, it is necessary to limit the number appropriately considering the number of concurrent users of the server.

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	int	4	1	16
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** pages. The minimum value is 1 and the maximum value is 16.

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	0.1	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

dont_reuse_heap_file is a parameter that configures not to reallocate the empty space created by deleting tables. If this parameter is configured to 0, the heap file is reallocated; if it is configured to 1, the heap file is not reallocated. 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

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 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.

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 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`** : The highest level of consistency can be ensured because concurrency-related problems such as dirty, non-repeatable and phantom reads do not occur.
- **`TRAN_REP_CLASS_REP_INSTANCE`** : It does not read records modified by another transaction, and not perform non-repeatable reads, either. This isolation level does not allow tables or records accessed by the current transaction to be updated by another transaction. However, phantom reads may occur for some records.
- **`TRAN_REP_CLASS_COMMIT_INSTANCE`** : It does not read records modified by other transactions, and not allow tables or records accessed by the current transaction to be updated by another transaction. However, non-repeatable reads may occur for some records.
- **`TRAN_REP_CLASS_UNCOMMIT_INSTANCE`** : It does not read records modified by other transactions, and not allow tables or records accessed by the current transaction to be updated by another transaction. However, dirty reads may occur for uncommitted records.
- **`TRAN_COMMIT_CLASS_COMMIT_INSTANCE`** : It does not read records modified by another transaction. However, this isolation level allows tables or records accessed by the current transaction to be updated by another transaction. Non-repeatable reads may occur.
- **`TRAN_COMMIT_CLASS_UNCOMMIT_INSTANCE`** : It does not read tables modified by another transaction, but reads modified records. Non-repeatable reads for both tables and records may occur.

lock_escalation

`lock_escalation` is a parameter that configures the maximum number of locks permitted before record locks are extended to table locks. 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 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, some information about lock ownership is included. If it is configured to 2, all information about lock ownership is included.

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
checkpoint_interval_in_mins	int	720	1	
log_buffer_pages	int	50	3	
media_failure_support	bool	yes		
log_max_archives	int	INT_MAX	0	

background_archiving	bool	yes	
log_flush_interval_in_msecs	int	1000	0

checkpoint_interval_in_mins

checkpoint_interval_in_mins is a parameter that configures the interval (in minutes) in which checkpoints are performed. The greater the value of the **checkpoint_interval_in_mins** parameter, the more time is needed for database restore. Therefore, it is recommended to configure the interval to a value from 20 to 30 minutes. The default value is **30**.

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.

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.

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
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

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**.

compat_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 multibyte character set. If the parameter is configured to no, a multibyte character set is not allowed; if it is configured to yes, a multibyte 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 multibyte 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**. Therefore, if the **oracle_style_empty_string** parameter is configured to yes, the empty string is processed as **NULL** as in Oracle DBMS.

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
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**.

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, data can be retrieved without errors when Unicode is used.

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. The following are three types of **INSERT** statements for execution modes.

- **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) VALUES('Mixed');
```

Each execution mode has the following value:

- **INSERT_SELECT** = 1
- **INSERT_VALUES** = 2
- **INSERT_DEFAULT** = 4

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** and **INSERT_DEFAULT** on the server, the **insert_execution_mode** is 5. (1 + 4 = 5)

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.

Parameter Syntax Rules

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

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}...]'
```

Description

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 'r;csql_history_num=70; index_scan_in_oid_order=1'
```

Using Session Commands of the CSQL Interpreter

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
```

The 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
LONG_QUERY_TIME	int	60000

LONG_TRANSACTION_TIME	int	60000
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
APPL_SERVER_MAX_SIZE    =40
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
APPL_SERVER_MAX_SIZE    =40
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 log file where the time and information messages related with execution of the CUBRID Broker are stored. The default file name is the **cubrid_broker.log** in the **$CUBRID/log** directory.

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 **30,000**, and the port for broker1 is configured to **33,000**.

MIN_NUM_APPL_SERVER

MIN_NUM_APPL_SERVER is a parameter that configures the minimum number of application servers (CAS) that can be created by the CUBRID Broker. 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) that can be created by the CUBRID Broker. The default value is **40**.

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.sql*.**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.

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**.

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.

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**.

CCI_PCONNECT

CCI_PCONNECT is a parameter that specifies whether or not to use the CCI connection pooling. The default value is **OFF**.

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.

The 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	8,001
monitor_interval	int	5
allow_user_multi_connection	string	YES
auto_start_broker	string	YES
execute_diag	string	OFF
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 is used to configure a parameter value; an equal sign (=) is not allowed.
- 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

#
# Auto start CUBRID Broker.
#
auto_start_broker YES

################################
# diagnostics parameter
################################
#
# turn on/off diag
#

execute_diag OFF

#
# 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.

execute_diag

execute_diag is a parameter that activates the database diagnosis functionality provided by the CUBRID Manager. If the parameter is set to **OFF**, which is the default value, the database diagnosis functionality is deactivated; if it is set to **ON**, the functionality can be used through the CUBRID Manager. However, the broker diagnosis functionality will be still active even if this parameter is set to OFF; this affects only the database diagnosis functionality.

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.

CUBRID SQL Guide

This chapter describes SQL syntax such as datatypes, 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

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> } & ldots; ]
```

<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. Except that, special characters are allowed.

The following are examples of legal identifiers.

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

Enclosing in Double Quotes or Square Brackets

If an identifier begins with a number or a symbol, the identifier must always be enclosed in double quotes or square brackets.

The following are examples of legal identifiers.

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

Illegal Identifiers

Beginning with a Letter

If an identifier contains special characters such as an underline, it is not allowed.

```
_a
#ack
%nums
```

Beginning with a Number

An identifier which begins with a number is not allowed.

- Example

```
2fer
88abs
```

CUBRID Keywords

The following keywords are previously reserved as a command, a function name or a type name in CUBRID; thus you are restricted to use these words for a class name, an attribute name, a variable name. However, if the keywords are enclosed in double quotes or square brackets, they can be used.

A

ABSOLUTE	ALLOCATE	ASSERTION
ACTION	ALTER	ASYNC
ADD	AND	AT
ADD_MONTH	ANY	ATTACH
AFTER	ARE	ATTRIBUTE
ALIAS	AS	AVG
ALL	ASC	-

B

BEFORE	BIT	BREADTH
BEGIN	BIT_LENGTH	BY
BETWEEN	BOOLEAN	
BIGINT	BOTH	

C

CALL	CLOSE	CORRESPONDING
CASCADE	CLUSTER	COUNT
CASCADED	COALESCE	CREATE
CASE	COLLATE	CROSS
CAST	COLLATION	CURRENT
CATALOG	COLUMN	CURRENT_DATE
CHANGE	COMMIT	CURRENT_DATETIME
CHAR	COMPLETION	CURRENT_TIME
CHARACTER	CONNECT	CURRENT_TIMESTAMP
CHARACTER_LENGTH	CONNECTION	CURRENT_USER
CHAR_LENGTH	CONSTRAINT	CURSOR
CHECK	CONSTRAINTS	CYCLE
CLASS	CONTINUE	
CLASSES	CONVERT	

D

DATA	DEFERRABLE	DICTIONARY
DATA_TYPE	DEFERRED	DIFFERENCE
DATE	DELETE	DISCONNECT
DATETIME	DEPTH	DISTINCT
DAY	DESC	DOMAIN
DEALLOCATE	DESCRIBE	DOUBLE
DECLARE	DESCRIPTOR	DROP
DEFAULT	DIAGNOSTICS	

E

EACH	ESCAPE	EXECUTE
ELSE	EXCEPT	EXISTS
ELSEIF	EXCEPTION	EXTERNAL
END	EXCLUDE	EXTRACT
EQUALS	EXEC	-

F-H

FALSE	GENERAL	HAVING
FETCH	GET	HOUR
FILE	GLOBAL	-
FIRST	GO	-
FLOAT	GOTO	-
FOR	GRANT	-
FOREIGN	GROUP	-
FOUND	-	-
FROM	-	-
FULL	-	-
FUNCTION	-	-

I

IDENTITY	INHERIT	INTERSECT
IF	INITIALLY	INTERSECTION
IGNORE	INNER	INTERVAL
IMMEDIATE	INOUT	INTO

IN	INSERT	IS
INDEX	INT	ISOLATION
INDICATOR	INTEGER	-

J-K

JOIN	KEY	-

L-N

LANGUAGE	MATCH	NA
LAST	MAX	NAMES
LDB	METHOD	NATIONAL
LEADING	MILLISECOND	NATURAL
LEAVE	MIN	NCHAR
LEFT	MINUTE	NEXT
LESS	MODIFY	NO
LEVEL	MODULE	NOCYCLE
LIKE	MONETARY	NONE
LIMIT	MONTH	NOT
LIST	MULTISET	NULL
LOCAL	MULTISET_OF	NULLIF
LOCAL_TRANSACTION_ID	-	NUMERIC
LOOP	-	-
LOWER	-	-

O

OBJECT	ON_	ORDER
OCTET_LENGTH	OPEN	OTHERS
OF	OPERATION	OUT
OFF	OPERATORS	OUTER
OID	OPTIMIZATION	OUTPUT
OLD	OPTION	OVERLAPS
ONLY	OR	-

P

PARAMETERS	PREORDER	PRIVATE

PARTIAL	PREPARE	PRIVILEGES
PENDANT	PRESERVE	PROCEDURE
POSITION	PRIMARY	PROTECTED
PRECISION	PRIOR	PROXY

Q

QUERY	-	-

R

READ	RENAME	ROLE
REAL	REPLACE	ROLLBACK
RECURSIVE	RESIGNAL	ROUTINE
REF	RESTRIC	ROW
REFERENCES	RETURN	ROWNUM
REFERENCING	RETURNS	ROWS
REGISTER	REVOKE	-
RELATIVE	RIGHT	-

S

SAVEPOINT	SHARED	SUBSET
SCHEMA	SIBLINGS	SUBSETEQ
SCOPE	SIGNAL	SUBSTRING
SCROLL	SIMILAR	SUM
SEARCH	SIZE	SUPERCLASS
SECOND	SMALLINT	SUPERSET
SECTION	SOME	SUPERSETEQ
SELECT	SQL	SYSDATE
SENSITIVE	SQLCODE	SYSDATETIME
SEQUENCE	SQLERROR	SYSTEM_USER
SEQUENCE_OF	SQLEXCEPTION	SYSTIME
SERIALIZABLE	SQLSTATE	SYS_DATE
SESSION	SQLWARNING	SYS_DATETIME
SESSION_USER	STATISTICS	SYS_TIME
SET	STRING	SYS_TIMESTAMP
SETEQ	STRUCTURE	SYS_USER

SET_OF	SUBCLASS	

T

TABLE	TIMESTAMP	TRANSLATE
TEMPORARY	TIMEZONE_HOUR	TRANSLATION
TEST	TIMEZONE_MINUTE	TRIGGER
THEN	TO	TRIM
THERE	TRAILING	TRUE
TIME	TRANSACTION	TYPE

U-W

UNDER	VALUE	WAIT
UNION	VALUES	WHEN
UNIQUE	VARCHAR	WHENEVER
UNKNOWN	VARIABLE	WHERE
UPDATE	VARYING	WHILE
UPPER	VCLASS	WITH
USAGE	VIEW	WITHOUT
USE	VIRTUAL	WORK
USER	VISIBLE	WRITE
USING	-	-
UTIME	-	-

Y-Z

YEAR	ZONE	-

Data Types

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*)

All types of character strings are enclosed within single quotes. If 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 two character strings are identical.

```
'abcdef'

'abc'
'def'
```

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 character strings that are not part of the English alphabet. National character strings differ from non-national character strings in that they are prefixed by the character **N** (must be in uppercase). For example:

```
 'Härder'
```

Characteristics

Length

The length of a character string is represented by the number of characters in it. Whether it has a fixed- or variable- length, the size of the character string is given when an attribute is defined.
When the length of a character string exceeds the maximum length defined, the exceeding characters are truncated if they are space characters (ASCII 32), or processed as an error otherwise. When a character string shorter than the defined length is stored in a fixed-length character string, the remainder of the character string is filled with space characters. For a variable-length character string, however, only the entered character string is stored with no added trailing space.
The maximum length of a character string is 1,073,741,823 bytes or 1GB. A string longer than this is truncated. The maximum length of a string that can be entered or processed by a single **CSQL** statement is between 16 and 8192KB (i.e. 8,388,608 bytes). The maximum length of a national character string is 536,870,911 characters, equal to half of the non-national character string limit because more than one byte may be needed for a character in a national character string. For the same reason, the maximum length of a national character string that can be input or output in a **CSQL** statement is also 536,870,911 characters.

Character Code Set

CUBRID supports the following character code sets:

- 8-bit ASCII
- 8-bit ISO 8859-1 Latin

- KSC 5601-1992 (Korean character standard)

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

Collating Character Code Sets

Character codes are sorted based on certain rules in the character code set. Such rules are called collation. The rules determine whether to compare character codes from the left to the right or vice-versa, and whether the trailing spaces will be used in the comparison. Each character code set includes a pre-defined basic collation.
For a national character set, the collation is determined by its encoding algorithm.

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 code 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

Fixed-length character strings are represented as **CHAR**(*n*), where *n* is the number of ASCII character strings. Each character takes up one byte. If *n* is not specified, the length is set to the default value 1. When the length of a character string exceeds *n*, the exceeding characters are truncated if they are space characters, or processed as an error otherwise. When a character string shorter than *n* is stored, the remainder of the character string is filled with space characters.

CHAR(*n*) and **CHARACTER**(*n*) can be used interchangeably.

Note

- The **CHAR** data type is always based on the ISO 8859-1 (Latin-1) character set.
- *n* must be a number greater than 0.
- Space characters used as filling characters are considered to be smaller than any other characters, including special characters.

Example

```
If you specify 'pacesetter' as CHAR(12), 'pacesetter ' is stored (a 10-
character string plus two space characters).
If you specify 'pacesetter ' as CHAR(10), 'pacesetter' is stored (a 10-
character string; two space characters are truncated).
If you specify 'pacesetter' as CHAR(4), an error occurs (because 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).
```

VARCHAR(n) or CHAR VARYING(n)

Description

Variable-length character strings are represented as **VARCHAR**(*n*), where *n* is the maximum number of ASCII character strings. Each character takes up one byte. 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*, the exceeding characters are truncated if they are space characters, or processed as an error otherwise. When a character

string shorter than *n* is stored, the trailing space is not filled with space characters. That is, only necessary parts are stored.

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

Note

- **STRING** is the same as the **VARCHAR** (maximum length).
- *n* must be a number greater than 0.

Example

```
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 space characters).
If you specify 'pacesetter ' as VARCHAR(10), 'pacesetter' is stored (a 10-
character string; two space characters are truncated).
If you specify 'pacesetter' as VARCHAR(4), an error occurs (because the
length of the character string is greater than 4).
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 space characters).
```

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

Fixed-length character strings are represented as **NCHAR**(*n*), where where *n* is the number of characters. Character strings that can be stored in this data type must belong to the character sets explained earlier. If *n* is not specified, the length is set to the default value 1. When the length of a character string exceeds *n*, the exceeding characters are truncated if they are space characters, or processed as an error otherwise. When a character string shorter than *n* is stored, the trailing space is filled with space characters. In some language character sets, one character is stored as more than one byte. Therefore, *n* must not be considered as the byte size.

The locale used by **NCHAR** is the locale of the operating system, or is specified by the **CUBRID_LANG** environment variable.

NCHAR(*n*), **NATIONAL CHAR**(*n*) and **NATIONAL CHARACTER**(*n*) can be used interchangeably.

Note

- *n* must be a number greater than 0.
- 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 N'Härder' as NCHAR(8), 'Härder ' is stored (a 6-character
string plus two space characters).
If you specify N'Härder ' as NCHAR(6), 'Härder' is stored (a 6-character
```

```
string; two space characters are truncated).
If you specify N'Härder' as NCHAR, an error occurs (because the length of the
character string is greater than the default value 1).
If you specify 'pacesetter' as NCHAR(12), an error occurs (that is, if a non-
national string is specified for an attribute declared as a national
character string).
```

NCHAR VARYING(n)

Description

Variable-length national character strings are represented as **NCHAR VARYING**(*n*), where *n* is the number of characters. As with **NCHAR**, *n* must not be considered as the byte size because one character is stored as more than one byte in some language character sets. If *n* is not specified, the maximum length is set to 536,870,911. When the length of a character string exceeds *n*, the exceeding characters are truncated if they are space characters, or processed as an error otherwise. When a character string shorter than *n* is stored, the trailing space is not filled with space characters. That is, only necessary parts are stored.

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

Note

- *n* must be a number greater than 0.
- 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 JIS X 208: 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 N'Härder' as NCHAR VARYING(8), 'Härder' is stored (a 6-
character string).
If you specify N'Härder' as NCHAR VARYING(6), 'Härder' is stored (a 6-
character string).
If you specify 'pacesetter' as NCHAR(12), an error occurs (that is, if a non-
national string is specified for an attribute declared as a national
character string).
```

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** as shown in the example.

```
B'0100110010100011'
```

Hexadecimal format uses less digits than the binary format to represent the same number. For hexadecimal format, append the string consisting of the numbers 0 - 9 and the letters A - F to the uppercase letter **X**. The following is hexadecimal representation of the same number that was represented above in binary format.

```
X'4ca3'
```

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.

Note

- *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

```
If you specify B'0001' is as BIT(8), B'00010000' (or X'10') is stored (the
remainder is filled with 0s).
If you specify X'12c34A' as BIT(4), an error occurs(because the length of the
string is greater than 4).
If you specify X'12c34A' as BIT, an error occurs (because the length of the
string is greater than the default value 1).
```

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.

Note

- 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

```
If you specify B'0001' as BIT VARYING(8), B'0001' (or X'1') is stored (the
remainder is filled with 0s).
If you specify X'12c34A' as BIT VARYING(4), an error occurs (because the
length of the string is greater than 4).
If you specify X'12c34A' as BIT VARYING, X'12c34A' is stored (if n is not
```

```
specified, the maximum length is set to the maximum value of 1,073,741,823
and the remainder is not filled when saving it).
```

Numeric Type

Definition and Characteristics

Definition

Numeric data types are divided into exact and approximate numeric data types. Exact numeric data types are used for numbers whose values must be precise and consistent. Generally, exact numeric data types are used in most financial calculations and analyses for accuracy to prevent the system from adjusting significant figures automatically.

CUBRID supports the following exact numeric data types:

- **NUMERIC** (or **DECIMAL**)
- **INTEGER**
- **SMALLINT**
- **BIGINT**

CUBRID supports the following approximate numeric data types:

- **FLOAT** (or **REAL**)
- **DOUBLE PRECISION**
- **MONETARY**

These data types have the same literal value, but are considered as approximate values since the system may interpret them differently. For example, suppose that you define an attribute as **FLOAT** and another as **DOUBLE**, and then store the same literal values in these two attributes. In this case, the system may interpret them differently.

To store **FLOAT**, **DOUBLE PRECISION** and **MONETARY** values, 4, 8 and 12 bytes are needed respectively.

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 than CUBRID 2008 R2.0, the input constant value exceeds **INTEGER**, it is handled as **NUMERIC**. However, in 2008 R2.0, it is handled as **BIGINT**.

INTEGER

Description

INTEGER is an exact numeric data type. The precision is 10, and the scale is 0.

INTEGER value is represented by the word size supported by the computer hardware, which is usually 4 bytes (32-bit). Therefore, the range of values that can be expressed is from -2147483648 to +2147483647.

```
INTEGER
```

Note

- If you assign a value with digits after the decimal point to an **INTEGER**, all digits after the decimal point are rounded.
- **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).
```

SMALLINT

Description

SMALLINT data type is an exact numeric data type. **SMALLINT** data type has the precision of 5 and the scale of 0. **SMALLINT** value is usually 2 bytes (16-bit). Therefore, the range of values that can be expressed is from -32768 to +32767.

```
SMALLINT
```

Note

If you assign a number with digits after the decimal point to **SMALLINT**, all digits after the decimal point are rounded.

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 an exact numeric data type. The precision is 19, and the scale is 0. The **BIGINT** type stores an 8 byte (64 bit) integer value, and supports the range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

```
BIGINT
```

Note

- If you assign a value with digits after the decimal point to a **BIGINT**, all digits after the decimal point are rounded.
- 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 7823467 as BIGINT, 7823467 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 or DECIMAL

Description

NUMERIC or **DECIMAL** is an exact numeric data type. You can specify the precision and scale as an option as shown below.

If the precision *p* is not specified, the default value is 15; the scale *s* is not specified, the default value is 0. The minimum value for precision is 1; the maximum value is 38.

NUMERIC, **DECIMAL** and **DEC** are used interchangeably.

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

Note

- Precision must be equal to or greater than scale.
- A scale greater than precision will cause an error.

Example

```
If you specify 55555.3333 as NUMERIC, 55555 is stored (all digits after the
decimal point are truncated because the default scale value is 0).
If you specify 555.33 as NUMERIC(5), 555 is stored (all digits after the
decimal point are truncated because the default scale value is 0).
If you specify 55555.3333 as NUMERIC(9,4), 55555.3333 is stored.
If you declare NUMERIC(3,4), an error occurs (because precision is less than
scale).
If you specify .533333 as NUMERIC(4,4), .5333 is stored (numbers after the
fourth decimal point are truncated because the precision is 4).
```

FLOAT or REAL

Description

FLOAT (or **REAL**) data type is an approximate numeric data type. The precision can be specified as an option.

If the precision *p* is specified as 7 or less, the data is represented as a single-precision number. If the precision is greater than 7 and less than or equal to 19, it is considered a **DOUBLE or DOUBLE PRECISION**

data type. The allowable value range varies depending on the system being used, but usually conforms to the ANSI/IEEE 754-1985 standard. According to this standard, a value between -10e+38 and +10e+38 can be stored and can be represented as a normalized number in the range from 1.175494e-38 to 3.402823e+38.

FLOAT is **REAL** without precision and scale specified.

```
FLOAT(p)
```

Note

The range between the minimum and maximum that can be represented by the **FLOAT** data type depends on the system where CUBRID is running. Extra caution is required when using a **FLOAT** for comparison because the **FLOAT** data type stores approximate values unlike the **INTEGER** data type. For example, you can convert a value to a **FLOAT**, but it is not guaranteed that the original and converted value will be identical. Therefore, when you execute a **SELECT** operation with a condition using the **FLOAT** data type, it is better to have a condition with a value range rather than with a specific **FLOAT** value (e.g. equation condition). This is also the case with the **DOUBLE** data type. This characteristic of **FLOAT** data types is not CUBRID-specific, but common to all computers handling approximate numeric data types.

Example

```
If you specify 8934 as FLOAT, 8.934000e+03 is stored.
If you specify 34 as FLOAT(0), 3.400000e+01 is stored.
If you specify -234.67 as FLOAT(5), -2.346700e+02 is stored.
```

DOUBLE or DOUBLE PRECISION

Description

DOUBLE PRECISION data type is an approximate numeric data type. **DOUBLE PRECISION** data type represents a floating point number with double precision.

Therefore, its precision is higher than that of a **FLOAT or REAL**. As with the **FLOAT or REAL**, the range of allowable numbers depends on the system being used, but usually conforms to the ANSI/IEEE 754-1985 standard. According to this standatd, a value between -10e+308 and +10e+308 can be stored and can be represented as a normalized number in the range from 2.2250738585072014e-308 to 1.7976931348623157e+308.

DOUBLE PRECISION and **DOUBLE** are used interchangeably.

```
DOUBLE PRECISION
```

Note

- Extra caution is required when using a DOUBLE for comparison because the **DOUBLE** data type stores approximate values unlike the **INTEGER** data type. For example, you can convert a value to a **DOUBLE**, but it is not guaranteed that the original and converted value will be identical. Therefore, when you execute a **SELECT** operation with a condition using a **DOUBLE** data type, it is better to have a condition with a value range rather than with a specific **DOUBLE** value (e.g. equation condition). This is also the case with the **FLOAT or REAL** data type explained earlier. This characteristic of FLOAT data types is not CUBRID-specific, but common to all computers handling approximate numeric data types.
- The number of significant digits of **DOUBLE** is 15.

Example

```
If you specify 89.998765431132 as DOUBLE, 8.999876543113200e+01 is stored.
If you specify -18.256743237573 as DOUBLE, -1.825674323757300e+01 is stored.
```

MONETARY

Description

MONETARY data type is an approximate numeric data type. The actual currency to be used is determined by the locale of the system. The precision and the scale can vary depending on the system used, but the precision up to 15 and the scale between -308 and +308 are usually allowed. The range of allowed values is the same as that of **DOUBLE or DOUBLE PRECISION**.

```
MONETARY
```

Note

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

Example

```
If you specify 8934 as MONETARY when the locale is English, $8,934.00 is
stored.
```

Date/Time Type

Definition and Characteristics

Definition

Date/time data types are used to represent the date, time or both. CUBRID supports the following data types:

- **DATE**
- **TIME**
- **TIMESTAMP**
- **DATETIME**

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 0 - 9999 AD.

  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**

  Date/time values can be compared only when they have the same items. That is, automatic coercion between different date/time types is not supported; only explicit coercion is allowed between different date/time types using the **CAST** operator.

  DATETIME can be coerced to **TIMESTAMP**, **DATE**, or **TIME**. However, the data or the time parts may be truncated after coercions.

  TIMESTAMP can be coerced to **DATE**, **DATETIME**, or **TIME**. However, the date or the time parts may be truncated after coercions.

  DATE can be coerced to **DATETIME** or **TIMESTAMP**.

  TIME can not be coerced to **DATETIME** or **TIMESTAMP.** However, reverse coercion is available.

DATE

Description

DATE data type consists of day, month and year. A **DATE** literal is represented as a character string. The character string follows the **DATE** keyword as shown below.

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

mm is month, *dd* is day, and *yyyy* is year. If there is no year entered, the current year will be specified automatically.

Note

All fields must be entered in integral form.

Example

```
DATE '11/11/1994' is stored as '11/11/1994'.
DATE '11/11' is stored as '11/11/2009'. (if a value for year is omitted, the
current year is automatically specified.)
```

TIME

Description

TIME data type consists of hour, minute and second. A **TIME** literal is represented as a character string. The character string follows the **TIME** keyword as shown below.

```
TIME 'hh:mm [:ss] [am | pm]'
```

hh is the hour, *mm* is the minute, and *ss* is the second. If there is no second entered, 0 will be specified automatically.
Time is automatically converted to 24-hour clock format.

Note

- All items must be entered in integral form.
- You can enter time using either 12-hour clock(AM/PM) format or 24-hour clock format, but the CSQL always uses AM/PM format to display time. You can also use AM/PM format even when entering time in 24-hour clock format; this is allowed only when the input value and the AM/PM value converted from the input value match.
- All time values are stored in 24-hour clock format; The C-API function **db_time_decode** returns time values in 24-hour clock format.

Example

```
TIME '1:15:45 pm' is stored as '01:15:45 PM'.
TIME '16:08:33 am' is an error (because the input value does not match the
AM/PM value).
TIME '16:08:33 pm' is stored as '04:08:33 PM'.
TIME '1:15' is stored as '01:15:00 AM'.
```

TIMESTAMP

Description

TIMESTAMP data type represents the data values that combine date and time. As shown below, the **TIMESTAMP** syntax consists of 'Date Time' or 'Time Date' character strings, following the **TIMESTAMP** keyword.

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

Note

The meaning of each item is the same as explained in the **DATE** and **TIME** sections. One difference is that the range allowed for a **TIMESTAMP** value 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.

Example

```
TIMESTAMP '01/31/1994 8:15:00 pm' is stored as '8:15:00 PM 01/31/1994.'
TIMESTAMP '16:08:33 am 1/1/1946' is an error (because the input time value
does not match the AM/PM value and the allowable limit has been exceeded).
```

DATETIME

Description

The **DATETIME** type consists of year, month, date, hour, minute, second and millisecond. A **DATETIME** literal is represented as a character string. The character string follows the **DATETIME** keyword as shown below.

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

mm indicates month, *dd* indicates date, *yyyy* indicates the year, *hh* indicates the time, *mm* indicates the minute, *ss* indicates the second and *ff* indicates the millisecond. If there is no year entered, the current year is automatically specified. If there are no time, minute and second entered, 12:00:00.000 is automatically specified; If there is no millisecond entered, 000 millisecond is automatically specified. The DATETIME type is represented from 0001-01-01 00:00:00.000 to 9999-12-31 23:59:59:999 in GMT.

Note

All fields must be entered in integral form.

Example

```
DATETIME '1999-10-10' is stored as '1999/10/10 12:00:00.000'.
DATETIME '10-10' is stored as '2009/10/10 12:00:00.000' (if there is no year
entered, the current year is automatically specified).
```

Collection 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.

CUBRID provides the following data types as collections.

- **SET** *domain*
- **SET** (*domain_list*)
- **LIST** or **SEQUENCE** *domain*
- **LIST** or **SEQUENCE** (*domain_list*)
- **MULTISET** *domain*
- **MULTISET** (*domain_list*)

In the syntax above, you can add OF after the collection type keyword. For example, instead of **SET** *domain* you can use **SET OF** *domain*. Parentheses are not needed if there is only one *domain*, but multiple domains (e.g. *domain_list*) must be enclosed in parentheses.

To list values of **SET**, **MULTISET**, **LIST** or **SEQUENCE**, they must be enclosed in braces and separated by commas ('{', '}') (e.g. {value_1, value_2, value_n}). If there is a {*value_list*} without any collection type specified, it is considered as a MULTISET. In some cases, type rules can be applied. For example, if you add a literal element to a SEQUENCE attribute, the attribute type remains as SEQUENCE.

domain_list can be the list of system default data types or user-defined classes. For example, SET (INTEGER, resort) is an example in which the set of row values of the type INTEGER or user-defined class "resort" are specified as the domain. If there is no *domain_list* such as SET (), it means all data types and classes are allowed as the elements of the set. If there are more than one domain lists, you can retrieve the elements using the **csql** utility and C-API. You cannot retrieve the elements of such collection using the CUBRID Manager and the interfaces via the CUBRID Broker (JDBC, ODBC, OLEDB, PHP, CCI).

Characteristics

- **Coercions**

  SET can be coerced to **MULTISET** automatically. Explicit coercions can be performed using the **CAST** operator, with the following limitations:

  - **LIST** or **SEQUENCE** can be **CAST**ed to **LIST**, **SEQUENCE**, **SET** or **MULTISET**.
  - **SET** can be **CAST**ed to **SET** or **MULTISET**.
  - **MULTISET** can be **CAST**ed to **MULTISET** or **SET**.

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

Definition	Example of collection values
SET(VARCHAR(20), INTEGER)	{'golf', 'handicap', 10}
SET CHAR(5)	{'aaa', 'bbbb', 'ccccc'}

The following is an example in which duplications are removed from the SET.

Definition	Input data	Stored data
SET VARCHAR(10)	{'golf', 'scuba', 'golf'}	{'golf', 'scuba'}

Caution

- If there are duplicate values, the duplicates are removed automatically.

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

Definition	Example of collection values
MULTISET INTEGER	{10, 20, 10, 80}
MULTISET CHAR(5)	{'aaa', 'bbbb', 'ccccc'}

LIST or 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

Definition	Example of collection values
SEQUENCE INTEGER	{20, 40, 60, 80}
LIST CHAR(5)	{'aaa', 'bbbb', 'ccccc'}

Relational Operators

Description

The relational operators compare the values of two expressions and returns true or false. The comparison is performed based on the order of two arguments. Invariably, the first expression is compared to the second. The arguments compared by the relational operators must be of the same data type or the type that can be implicitly coerced to a common type by the system. Otherwise, an explicit type cast is required.

Syntax

```
value_expression  relational_operator  value_expression

value_expression :
bit string
character string
numeric value
date-time value
collection value
NULL

relational_operator :
=    (equal to)
>    (greater than)
<    (less than)
<>   (not equal to)
<=   (less than or equal to)
>=   (greater than or equal to)
```

- *value_expression* : Declares an expression to be compared.
  - bit string : A Boolean operation can be performed on bit strings, and all relational 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 relational operator, two character strings must have the same character code 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 : All numeric values can be compared by a relational operator, and all numeric types can be used for Boolean comparison. 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 relational operator.
  - 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. Relational operation on the **NULL** value does not return a result.

- *relational_operator* : Compares values of two expressions by using a relational operator.

Arithmetic Operators

Arithmetic Operators

Description

You can use an arithmetic operator with two expressions to add, subtract, multiply, or divide numeric values.

Syntax

```
value_expression  mathematical_operator  value_expression
value_expression :
bit string
character string
numeric value
date-time value
collection value
NULL

mathematical_operator :
set_arithmetic_operator
arithmetic_operator

arithmetic_operator :
+       (Addition)
-       (Subtraction)
*       (Multiplication)
/       (Division)

set_arithmetic_operator :
UNION                 (Union)
DIFFERENCE        (Difference)
INTERSECTION    (Intersection)
```

- *value_expression* : Declares the value to be calculated.
- *mathematical_operator* : Specifies the mathematical operation to perform. You can specify the arithmetic or set operator.
- *set_arithmetic_operator* : Specifies the set operator to be used. A union, difference, or intersection can be specified.
- *arithmetic_operator* : Specifies the arithmetic operation to be used. Addition, subtraction, multiplication, or division can be specified.

Character String Operation

Description

You can use the double pipe symbol (||) operator to concatenate characters or bit strings. For more information about concatenation operators, see "Concatenation Operators."

Example

The following is an example of returning a single character string after concatenating two.

```
SELECT 'new generation' || ' database system'
FROM db_root;
```

Result : 'new generation database system'

Numeric Operation

All numeric data types can be used in numeric operations. An expression divided by 0 is represented as **INF** to indicate infinity.

The result type of an arithmetic operation is determined by the data type of the operand and the operation type. Arithmetic operations on **NUMERIC** types and their results are determined as follows:

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.	

Type Casting of Numeric Data Types

If a numeric data type needs to be cast to another type when an arithmetic expression is calculated, type casting is performed applying the rules in the following order:

- If one of the two arguments is of **DOUBLE** type, the other argument is also cast to **DOUBLE**, and **DOUBLE** is returned as the result of the operation.
- If two arguments are of the same data type and the operation is not division, type casting is not required. The data type of the operation result is the same as that of the arguments.
- If one of the two arguments is of **FLOAT** type and the other is not **DOUBLE**, the latter is cast to **FLOAT**, and the result will be of **FLOAT** type.
- If neither of the two arguments is **DOUBLE** in a division operation, both arguments are cast to **FLOAT**, and the result will be of **FLOAT** type.
- If one argument is a **SMALL INTEGER** and the other is an **INTEGER**, the **SMALL INTEGER** argument is cast to **INTEGER**, and an **INTEGER** is returned as the result of the operation.

If none of the above rules apply, type casting is performed in the following order:

- **SMALL INTEGER**
- **INTEGER**
- **BIGINT**
- **NUMERIC**
- **FLOAT**
- **MONETARY**
- **DOUBLE**

Data types in the upper part of the list are cast to the ones in the lower part. For example, in a comparison between a **SMALL INTEGER** and a **FLOAT**, the **SMALL INTEGER** is cast to **FLOAT**, and a **FLOAT** is returned as the result.

Monetary Data Value

The **MONETARY** type represents a currency value. The default value follows the locale specified in the **CUBRID_LANG** environment variable. In case **CUBRID_LANG** is specified as 'en_US', the default value is 'US dollar' ($). You can request your system administrator to change default currency to other types.

Arithmetic operators + and - can be used to add or subtract two monetary data. In both cases the result will be of **MONETARY** type. The **MONETARY** type can be used in multiplication and division operations together with any data type. Generally, the **MONETARY** type adopts the same operation system as the **DOUBLE** type. The following are valid expressions that return the results of **MONETARY** type.

```
MONETARY + MONETARY = MONETARY
MONETARY + NUMERIC = MONETARY
MONETARY - MONETARY = MONETARY
MONETARY - NUMERIC = MONETARY
MONETARY * NUMERIC = MONETARY
MUMERIC * MONETARY = MONETARY
MONETARY / NUMERIC = MONETARY
```

Date/Time Operation

Description

If you subtract two date/time values of the same type, the difference of the two operands is returned. If the data type of both operands is **DATE**, the difference in days between the two **DATE** values is returned as the result. If the data type of both operands is **TIME** or **TIMESTAMP**, the difference in seconds between the two values is returned as the result. If the data type of both operands is **DATETIME**, the difference in milliseconds between the two values is returned as the result.

You cannot add two **TIMESTAMP** values. However, you can add or subtract an **INTEGER** value (in seconds) to or from the **TIMESTAMP**. A **TIMESTAMP** is returned as the result of the operation.

```
TIMESTAMP - TIMESTAMP = INTEGER (in seconds)
TIMESTAMP + INTEGER (in seconds) = TIMESTAMP
TIMESTAMP - INTEGER (in seconds) = TIMESTAMP
```

As with **TIMESTAMP**, you cannot add two **TIME** values. However, if you add or subtract an **INTEGER** value to or from a **TIME** value, **TIME** value is returned.

```
TIME - TIME = INTEGER (in seconds)
TIME + INTEGER (in seconds) = TIME
TIME - INTEGER (in seconds) = TIME
```

If you add or subtract **INTEGER** value to or from a **DATE** value, a **DATE** value is returned. The **INTEGER** operand represents the number of days. You can subtract a **DATE** value from another **DATE** value; and **INTEGER** value is returned as the result. For example, the expression DATE'11/1/1994' - DATE'12/1/1994' returns an **INTEGER** value of 30. However, note that you cannot add two **DATE** values.

```
DATE - DATE = INTEGER (days)
DATE + INTEGER (days) = DATE
DATE - INTEGER (days) = DATE
```

If you subtract **DATETIME** value from the **DATETIME** value, a **BIGINT** value is returned The return value means the difference in millisecond. If you add or subtract a **DATETIME** value to or from a **DATETIME** value, a **BIGINT** (millisecond) value is returned. If you add or subtract a **BIGINT** (millisecond) to or from a **DATETIME** value, a **DATETIME** value is returned.

```
DATETIME - DATETIME = BIGINT (in millisecond)
DATETIME + BIGINT(Millisecond) = DATETIME
DATETIME - BIGINT(Millisecond) = DATETIME
```

Example

The following is an example that calculates the duration in days of Olympic Games.

```
SELECT host_year, host_city, closing_date-opening_date+1 FROM olympic;
```

Caution

- You cannot add two date-time values.
- If any of the date-time arguments contains **NULL**, **NULL** is returned.

Set 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.

Collection Operand	Set Arithmetic Operator	Collection Operand	Result Collection Type
SET	+ - *	**SET**	**SET**
SET	+ - *	**MULTISET** **LIST (SEQUENCE)**	**MULTISET**
MULTISET	+ - *	**SET** **MULTISET** **LIST (SEQUENCE)**	**MULTISET**
LIST (SEQUENCE)	+	**LIST (SEQUENCE)**	**LIST (SEQUENCE)**
LIST (SEQUENCE)	+	**SET** **MULTISET**	**MULTISET**
LIST (SEQUENCE)	- *	**SET** **MULTISET** **LIST (SEQUENCE)**	**MULTISET**

- Union

  The + operator joins all elements in two collections to generate a single collection. If you combine two **LIST**s with the + operator, a concatenated **LIST** is returned as the result. The individual order of the input **LIST**s does not change. If the operands of the + operator are all of the same collection type, the result becomes the input collection type. For example, if you add two **SET**s, a **SET** is returned as a result, and duplicate elements are removed as can be expected of **SET**s. When computing the union of two **MULTISET**s, a **MULTISET** is returned with all the elements sorted in order. For example, the expression {1, 3, 4, 3} + {3, 5, 4} returns a **MULTISET** {1, 3, 3, 3, 4, 4, 5}. If two different collection types are given, both operands are converted to **MULTISET**s before calculation, which returns a **MULTISET** as the result.

- Difference

  The - operator is used to identify only elements of a collection that do not belong to another collection. Calculating the set difference between two collections returns the elements in the collection on the left of the - operator that do not belong to the collection on the right. For example, the expression {1, 3, 4, 3} - {3, 5, 4} returns a **MULTISET** {1, 3}. A **SET** is returned only when both operands are of the **SET** type. Otherwise, a **MULTISET** is returned.

- Intersection

  The * operator is used to identify elements that belong to both collections. If both collection operands are of the **SET** type, a **SET** type collection is returned as the result with all the duplicates removed. Except for this case, all other combinations of operands return a **MULTISET** as the result. In such cases, both operands are converted to **MULTISET**s before calculation, which returns a **MULTISET** as the result. For example, the expression {1, 3, 4, 3} * {3, 5, 4} returns a **MULTISET** {3, 4}.

Type Casting

Arithmetic operations on collection data types comply with the following data type casting rules:

- If two arguments are of the same type, type casting is not necessary; the same type as the arguments is returned as the result.

- If two arguments are of different collection types, both arguments are cast to **MULTISET**s if necessary. The result is a **MULTISET**.

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;
```

Containment Operators

Containment Operators

Description

The containment operator is used to compare set values. For example, you can use a containment operator to compare a set-value attribute that has a homogeneous or heterogeneous domain with a certain set-value.

You can also use the result of a subquery as a comparison set. To create a set-value from an attribute which is not a set-value, you must add an appropriate keyword for the set type to the subquery (e.g. **SET**(*subquery*)). If the attribute in the subquery is a set-value attribute, you can use the subquery statement without the keyword for the set type. However, the subquery must return only one set as the result.

Syntax

```
collection_operand  containment_operator  collection_operand

collection_operand:
• set
• multiset
• sequence (or list)
• NULL

containment_operator:
• SETEQ
• SETNEQ
• SUPERSET
• SUBSET
• SUPERSETEQ
• SUBSETEQ
```

If the operands in the expression containing a set have different collection types, both collections are converted to a **MULTISET** before the operation is performed.

The Containment operator is used to compare two **SET**s or **MULTISET**s generated from two expressions or from one expression and one subquery. This expression can be an arithmetic expression that includes a single SET-valued attribute or SET operator, or a set that is represented as a list of constant values enclosed in braces. **SETEQ** and **SETNEQ** can be used for **SET**, **MULTISET** and **SEQUENCE** (or **LIST**), but **SUPERSET**, **SUPERSETEQ**, **SUBSET** and **SUBSETEQ** can be used only for **SET** and **MULTISET**. If a subquery is included in the comparison, it must retrieve only one set-value attribute.

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.

If any of the operands in the expression is **NULL**, **NULL** is returned as the result.

Suppose that the following table and data were created to explain containment operators.

```
create table city (city_name varchar(30) primary key, sports set varchar(20));
insert into city values ('Seoul', {'baseball','basketball','golf', 'bowling',
'soccer'});
insert into city values ('Busan', {'baseball','basketball','soccer','scuba'});
insert into city values ('Daejeon', {'baseball','soccer'});
insert into city values ('Gwangju', {'baseball','soccer','basketball'});
insert into city values ('Incheon', {'baseball','basketball','golf'});
insert into city values ('Suwon', {'soccer','basketball'});
insert into city values ('Anyang', {'soccer'});
insert into city values ('Pyongchang', {'ski','snowboard'});
insert into city values ('Jeju', {'tennis','golf','scuba'});
insert into city values ('Cheonan', {});
```

SETEQ Operator

Description

The **SETEQ** operator determines whether two collections are identical. If every element of each collection is the same, it returns TRUE.

Syntax

```
collection_operand SETEQ collection_operand
```

The *collection_operand* must always be a single collection value. It is checked if the collection created by the first *collection_operand* is the same as the other collection (in this case, the second *collection_operand*). It should be possible to form the same set as the collection on the left of the **SETEQ** operator using all the elements of the collection on the right.

Example

The following query retrieves the city_name and sports that meets WHERE condition. The condition meets when every element is the same as that of the compared set, regardless of its order.

```
csql> select city_name, sports from city where sports seteq {'tennis', 'golf',
'scuba'};
csql> ;x

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

  city_name             sports
=========================================
  'Jeju'                {'golf', 'scuba', 'tennis'}
```

SETNEQ Operator

Description

The **SETNEQ** operator determines whether one collection has a different set from another.

Syntax

```
collection_operand SETNEQ collection_operand
```

The *collection_operand* must always be a single collection value. It is checked if the collection created by the first *collection_operand* is the same as the one created by the second *collection_operand*. Each element in one collection is compared to every element in the other collection to determine whether they are the same as each other.

Example

The following query retrieves all the city_names and sports when elements of sports are not {'baseball', 'soccer'}. The result also includes a empty set.

```
csql> select city_name,sports from city where sports setneq
{'baseball','soccer'};
csql> ;x

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

  city_name             sports
=========================================
  'Seoul'               {'baseball', 'basketball', 'bowling', 'golf',
'soccer'}
  'Busan'               {'baseball', 'basketball', 'scuba', 'soccer'}
  'Gwangju'             {'baseball', 'basketball', 'soccer'}
  'Incheon'             {'baseball', 'basketball', 'golf'}
  'Suwon'               {'basketball', 'soccer'}
  'Pyongchang'          {'ski', 'snowboard'}
  'Jeju'                {'golf', 'scuba', 'tennis'}
  'Cheonan'             {}
  'Anyang'              {'soccer'}
```

SUPERSET Operator

Description

The **SUPERSET** operator compares two collections to determine whether the first collection operand includes at least all the values of the collection operand on the right of the operator. Equal sets are excluded. The first collection is considered to be greater than the compared collection.

Syntax

```
collection_operand SUPERSET collection_operand
```

The *collection_operand* must always be a single collection value. It is checked whether the collection created by the first *collection_operand* is the same as the one created by the second *collection_operand*. All elements of the collection on the right of the **SUPERSET** operator must be included in the collection on the left.

Example

The following query retrieves city_names and sports of all cities that have 'baseball', 'basketball', 'soccer' plus one or more facilities.

```
csql> select city_name,sports from city where sports superset
{'baseball','soccer', 'basketball'};
csql> ;x

=== <Result of SELECT Command in Line 5> ===

  city_name             sports
=========================================
  'Seoul'               {'baseball', 'basketball', 'bowling', 'golf',
'soccer'}
  'Busan'               {'baseball', 'basketball', 'scuba', 'soccer'}
```

SUPERSETEQ Operator

Description

The **SUPERSETEQ** operator compares two collections to determine whether the first collection is equal to or at least have all the values of the second collection. For this reason, the first collection is considered to be greater than or equal to the second collection.

Syntax

```
collection_operand SUPERSETEQ collection_operand
```

The *collection_operand* must always be a single collection value. It is checked whether the collection created by the first *collection_operand* is the same as the one created by the second *collection_operand*. All elements of the collection on the right of the **SUPERSETEQ** operator must be included in the collection on the left or identical to those of the second collection.

Example

The following query retrieves city_names and sports of all cities that have 'soccer', 'baseball' and 'basketball' facilities. The cities that have only such two sports facilities are also included in the query result.

```
csql> select city_name, sports from city where sports superseteq
{'soccer','baseball', 'basketball'};
csql> ;x

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

  city_name             sports
=========================================
  'Seoul'               {'baseball', 'basketball', 'bowling', 'golf',
'soccer'}
  'Busan'               {'baseball', 'basketball', 'scuba', 'soccer'}
  'Gwangju'             {'baseball', 'basketball', 'soccer'}
```

SUBSET Operator

Description

The **SUBSET** operator determines whether the first collection is a proper subset of the second collection. That is, every element of the first collection must be also an emlement of the second collection. If the first and the second collection have the same elements, FALSE is returned; if the first collection has the empty set, it is a proper subset of the second one, which means it returns TRUE.

Syntax

```
collection_operand SUBSET collection_operand
```

The *collection_operand* must always be a single collection value. It is checked whether the collection created by the first *collection_operand* is the same as the one created by the second *collection_operand*.

Example

The following query retrieves city_name and sports that is a proper subset of the compared set. The result tuples have smaller sets than the compared set. Empty sets are also included.

```
csql> select city_name, sports from city where sports subseteq
{'soccer','basketball'};
csql> ;x

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

  city_name             sports
=========================================
  'Cheonan'             {}
  'Anyang'              {'soccer'}
```

SUBSETEQ Operator

Description

The **SUBSETEQ** operator determines whether the first collection is a subset of or identical to another. Every element of the first collection is also element of the second collection including itself.

Syntax

```
collection_operand SUBSETEQ collection_operand
```

The *collection_operand* must always be a single collection value. It is checked if the collection created by the first *collection_operand* is the same as the one created by the second *collection_operand*. All the elements of the first collection must belong to the collection on the right of the **SUBSETEQ** operator or must be identical to those of the second collection.

Example

The following query retrieves the city_name and sports that is a subset of the second collection. In this case, the first collection becomes a subset of the second when every element of two collections is identical. The following query results show instances including the compared set itself, the empty set, and the proper subsets.

```
csql> select city_name, sports from city where sports subseteq
{'soccer','basketball'};
csql> ;x

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

  city_name             sports
=========================================
  'Suwon'               {'basketball', 'soccer'}
  'Cheonan'             {}
  'Anyang'              {'soccer'}
```

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**	**D**	**T**	**UT**	**DT**	**S**	**MS**	**SQ**
EN	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No	No	No
AN	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No	No	No
VC	Yes	Yes	Yes[1]	Yes*	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
FC	Yes	Yes	Yes*	Yes*	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
VB	No	No	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No
FB	No	No	Yes	Yes	Yes	Yes	No	No	No	No	No	No	No
D	No	No	Yes	Yes	No	No	Yes	No	Yes	Yes	No	No	No
T	No	No	Yes	Yes	No	No	No	Yes	No	No	No	No	No
UT	No	No	Yes	Yes	No	No	Yes	Yes	Yes	Yes	No	No	No
DT	No	No	Yes	Yes	No	No	Yes	Yes	Yes	Yes	No	No	No
S	No	No	No	No	No	No	No	No	No	No	Yes	Yes	Yes
MS	No	No	No	No	No	No	No	No	No	No	Yes	Yes	Yes

SQ	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*))
- **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**.

```
SELECT CAST(score AS FLOAT), unit
FROM history
WHERE unit = 'kg';
```

Caution

Any value expression can be explicitly cast to an appropriate data type. The target type must be a valid CUBRID data type. The **CAST** conversion is allowed only between data types having the same character code set. Furthermore, if you want to cast an approximate numeric data type (e.g. **FLOAT**, **DOUBLE**) to a character string, you must input a character string long enough to express all significant digits.

EXTRACT Operator

Description

The **EXTRACT** operator extracts the value of a specified part of a date-time expression.

Syntax

```
EXTRACT ( field FROM date-time_argument )

field :
YEAR
MONTH
DAY
HOUR
MINUTE
SECOND
MILLISECOND

date-time_argument :
expression
```

- *field* : Specifies the value to be extracted from a date-time expression.
- *date-time argument* : Declares a date-time expression that contains a part to be extracted. The type of the date-time argument must be one value of **TIME**, **DATE**, **TIMESTAMP** or **NULL**. The extracted value is an **INTEGER**. If the date-time argument is **NULL**, **NULL** is returned.

Example

The following is an example that extracts the month from a date-time expression.

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '07/31/1994 8:15:00 pm') FROM db_root;
```

Result : 7

The following is an example that extracts the millisecond from **DATETIME**.

```
SELECT EXTRACT(MILLISECOND FROM DATETIME '07/31/1994 8:15:00.031 pm') FROM
db_root;
```

Result : 31

String Function and Operator

Concatenation Operators

Description

Concatenation operations are performed between two character or bit strings. The second string is concatenated to the first string. The plus sign (+) or double pipe symbol (||) can be used as a concatenation operator. You must use the || operator to concatenate character strings.

Syntax

```
concat_operand1   +  concat_operand1
concat_operand2   ||  concat_operand2

concat_operand1 :
bit string
NULL

concat_operand2 :
bit string
character string
NULL
```

- *concat_operand1* : Specifies the first string for the concatenation.
- *concat_operand2* : Specifies the second string that will be concatenated to the first string.

Example

- **Example 1**

The following is an example that returns a single string by concatenating the capital city and the country name.

```
SELECT capital ||', '|| name
FROM nation
WHERE name = 'Korea'
```

Result : Seoul, Korea

- **Example 2**

  The following is an example that returns a single string by concatenating the first bit string and the second bit string.

```
SELECT B'1011111' ||', '|| B'0100'
FROM db_root
```

  Result : 'be, 4'

LENGTH Function

Description

The **LENGTH** function returns the length of a given character string. **CHAR_LENGTH** and **LENGTHB** work the same as **LENGTH**.

Syntax

```
LENGTH( length_operand )

length_operand :
character string
NULL
```

- *length_operand* : Specifies the character string whose length is to be calculated.

Example

- **Example 1**

  The following is an example that returns the length of the string 'Cubrid'.

```
SELECT LENGTH('Cubrid') FROM db_root;
```

  Result : 6

- **Example 2**

  The following is an example that returns the length of the string ' Cubrid ' which includes space characters.

```
SELECT LENGTH(' Cubrid ') FROM db_root;
```

  Result : 8

- **Example 3**

  The following is an example that returns the length when there is no input string.

```
SELECT LENGTH('') FROM db_root;
```

  Result : 0

- **Example 4**

  The following is an example that returns the length when the input string is **NULL**.

```
SELECT LENGTH(NULL) FROM db_root;
```

  Result : NULL

CHAR_LENGTH Function

Description

The **CHAR_LENGTH** function returns the number of characters in a character string. It is used to calculate the length of a **CHAR**, **NCHAR**, **VARCHAR** or **VARNCHAR** (and the synonym of such data type) string.

Syntax

```
CHAR_LENGTH ( length_operand )

length_operand :
character string
NULL
```

- *length_operand* : Specifies the character string whose number of characters is to be calculated. If the character string is **NULL**, **NULL** is returned. The length of the character string is calculated in characters and returned as an integer value. Space characters in the character string are also reflected in the return value. The single quote used to represent a character string is not considered for the calculation.

Example

The following is an example that returns countries whose names consist of 15 - 16 characters from the nation table.

```
SELECT CHAR_LENGTH(name), name
FROM nation
WHERE CHAR_LENGTH(name) BETWEEN 15 AND 16
ORDER BY 1 ASC;
```

Result :

15 'Solomon Islands'
16 'Papua-New Guinea'

BIT_LENGTH Function

Description

The **BIT_LENGTH** function returns the bit length for the bit string of character string as an integer.

Syntax

```
BIT_LENGTH ( bit_length_operand )

bit_length_operand :
bit string
character string
NULL
```

- *bit_length_operand* : Specifies the character string or bit string whose number of bits is to be calculated. If this value is **NULL**, **NULL** is returned.

Example

- **Example 1**

  The following is an example that returns the number of bits for the character string 'Cubrid'.

  ```
  SELECT BIT_LENGTH ('Cubrid')
  FROM db_root;
  ```

  Result : 48

- **Example 2**

  The following is an example that returns the number of bits for the bit string B'010101010'.

```
SELECT BIT_LENGTH (B'010101010')
FROM db_root;
```

Result : 9

OCTET_LENGTH Function

Description

The **OCTET_LENGTH** returns the length of a string in bytes.

Syntax

```
OCTET_LENGTH ( octet_length_operand )

octet_length_operand :bit string
character string
NULL
```

- *octet_length_operand* : Specifies the character or bit string whose length is to be returned in bytes. If the value is **NULL**, **NULL** is returned. If the length of the bit string in bits is not a multiple of 8, the value ceiled to the multiple of 8 is returned. For example, a 8-bit bit string returns 1, and a 9-bit bit string returns 2.

Example

- **Example 1**

  The following is an example that returns the length of a character string 'Cubrid' in bytes.

```
SELECT OCTET_LENGTH ('Cubrid')
FROM db_root;
```

  Result : 6

- **Example 2**

  The following is an example that returns the length of a bit string B'010101010' in bytes.

```
SELECT OCTET_LENGTH (B'010101010')
FROM db_root;
```

  Result : 2

UPPER Function

Description

The **UPPER** function returns a string in which all lowercase ASCII or ISO 8859-1 (Latin-1) characters have been converted to uppercase characters.

Syntax

```
UPPER ( upper_operand )

upper_operand :
character string
NULL
```

- *upper_operand* : Specifies the string in which lowercase characters are to be converted to uppercase. If the value is **NULL**, **NULL** is returned.

Example

The following is an example that converts the characters in the string 'Cubrid' to uppercase.

```
SELECT UPPER('Cubrid')
FROM db_root;
```

Result : 'CUBRID'

LOWER Function

Description

The **LOWER** function returns a string in which all uppercase ASCII or ISO 8859-1 (Latin-1) characters have been converted to lowercase characters.

Syntax

```
LOWER (lower_operand )

lower_operand :
character string
NULL
```

- *lower_operand* : Specifies the string in which uppercase characters are to be converted to lowercase. If the value is **NULL**, **NULL** is returned.

Example

The following is an example that converts the characters in the string 'Cubrid' to lowercase and returns the result.

```
SELECT LOWER('Cubrid')
FROM db_root;
```

Result : 'cubrid'

TRIM Function

Description

The **TRIM** function trims leading and/or trailing characters from a character string.

Syntax

```
TRIM ( [ [ LEADING | TRAILING | BOTH ]
  [ trim_character_set ] FROM ] trim_operand )trim_character_set :

trim_character_set :
character string
NULL

trim_operand :
character string literal
string valued attribute
NULL
```

- *trim_character_set* : The optional *trim_character_set* represents the leading and/or trailing characters to trim from the string specified by *trim_operand*. If the *trim_character_set* value is not set, a single space character (' ') is specified automatically.
- *trim_operand* : If the value is **NULL**, **NULL** is returned.
- To specify the position to perform trimming, you can use the **LEADING**, **TRAILING** or **BOTH** keyword. If there is no keyword specified, **BOTH** is used.
- The strings in *trim_character* and *trim_operand* must have the same character code set.

Example

- **Example 1**

  The following is an example that trims the leading and trailing characters equal to 'i' from the character string.

```
SELECT TRIM ('i' FROM 'iiiiiOlympiciiiii')
FROM db_root;
```

  Result : Olympic

- **Example 2**

  The following is an example that trims only the leading characters equal to 'i' from the character string.

```
SELECT TRIM (LEADING 'i' FROM 'iiiiiOlympiciiiii')
FROM db_root;
```

  Result : Olympiciiiii

LTRIM Function

Description

The **LTRIM** function removes all specified characters from the left-hand side of a string until reaching a different character.

Syntax

```
LTRIM( char1 [, set])

char1 :
character string
NULL

set :
character string
NULL
```

The **LTRIM** function removes specified characters from the left of *char1* until it reaches a character that is not in the *set*.

- *char1* : Specifies the string. If the value is **NULL**, **NULL** is returned.
- *set* : Specifies the character to remove from *char1*. The default value is a single space character (' '). If the value is **NULL**, **NULL** is returned.

Example

The following is an example of removing X from the left-hand side of the string.

```
SELECT LTRIM('XXX123', 'X') FROM db_root;
```

Result : '123'

RTRIM Function

Description

The **RTRIM** function removes all specified characters from the right-hand side of a string until reaching a different character.

Syntax

```
RTRIM( char1 [, set ] )

char1 :
character string
NULL
```

```
set :
character string
NULL
```

The **RTRIM** function removes specified characters from the right of *char1* until it reaches a character that is not in the *set*.

- *char1* : Specifies the string. If the value is **NULL**, **NULL** is returned.
- *set* : Specifies the character to remove from *char1*. The default value is a single space character (' '). If the value is **NULL**, **NULL** is returned.

Example

The following is an example that removes X from the right-hand side of the string.

```
SELECT RTRIM('123XXX', 'X') FROM db_root;
```

Result : '123'

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
NULL

n :
integer
NULL

char2 :
character string
NULL
```

The **LPAD** function pads the left side of *char1* with *char2* until its length reaches *n*.

- *char1* : Specifies the string to pad characters to. If the length of *char1* is greater than *n*, *char1* is returned after being truncated to the length of *n*. If *char1* is truncated at the first byte of a double-byte character set (e.g. Korean), the first byte is removed and then a space character is added. If the value is **NULL**, **NULL** is returned.
- *n* : Specifies the length. If the value is **NULL**, **NULL** is returned.
- *char2* : Specifies the string to pad. The default value for *char2* is a single space character (' '). If the value is **NULL**, **NULL** is returned.

Example

- **Example 1**

  The following is an example of padding the character X on the left of a string until the length of the string reaches 6.

  ```
  SELECT LPAD('123', 6, 'X') FROM db_root;
  ```

  Result : 'XXX123'

- **Example 2**

  The following is an example of truncating a string to the length of 4 and returns the result.

  ```
  SELECT LPAD('123456', 4, 'X') FROM db_root;
  ```

Result : '1234'

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
NULL

n :
integer
NULL

char2 :
character string
NULL
```

The **RPAD** function pads the right side of *char1* with *char2* until its length reaches *n*.

- *char1* : Specifies the string to pad characters to. If the length of *char1* is greater than *n*, *char1* is returned truncated to the length of *n*. If *char1* is truncated at the first byte of a double-byte character set (e.g. Korean), the first byte is removed and then a space character is added. If the value is **NULL**, **NULL** is returned.
- *n* : Specifies the length. If the value is **NULL**, **NULL** is returned.
- *char2* : Specifies the string to pad. The default value for *char2* is a single space character (' '). If the value is **NULL**, **NULL** is returned.

Example

- **Example 1**

  The following is an example that pads the character X on the right of a string until the length of the string reaches 6.

  ```
  SELECT RPAD('123', 6, 'X') FROM db_root;
  ```

  Result : '123XXX'

- **Example 2**

  The following is an example that truncates a string to the length of 4 and returns the result.

  ```
  SELECT RPAD('123456', 4, 'X') FROM db_root;
  ```

  Result : '1234'

LIKE Function

Description

The **LIKE** function searches for instances matching the specific text. Use the **LIKE** function to compare only attributes of fixed-length or variable-length character string type. For the **LIKE** operation, the compared strings must have the same character sets. The **LIKE** function cannot be used to compare bit strings.

Syntax

```
like_operand [ NOT ] LIKE like_operand [ ESCAPE like_operand ]

like_operand :
```

```
character string
NULL
```

Beginning with first character of the first *like_operand*, the system searches for the string pattern given by the second *like_operand*.

The **LIKE** function can contain the following characters:

- The "_" (underscore) sign in the string pattern represents a single character.
- The "%" sign in the string pattern represents combination of characters. It may represent no character.
- The other characters represent themselves.

  If you want to search for the actual _ or % sign, you must specify the *like_operand* consisting of a single character using the **ESCAPE** option. The **ESCAPE** character in a string must always precede the _ or % sign. In addition, the _ or % sign cannot be used as an escape character. For example, to search for '10%', you can use an escape character such as backslash (\). The actual string to be entered must be '10\%.' That is, enter the following statement to search for '10%.'

  ```
  SELECT name FROM nation WHERE name LIKE '10\%' ESCAPE '\';
  ```

  The **LIKE** function does not have any default values defined for escape characters.
  If the **LIKE** function have a **NULL** value entered for any of its *like_operand*, **NULL** is returned as the result.

Example

The following is an example that searches for country names matching the string pattern '_in%' in the **nation** table. In this example, the first character of the string must be an actual character. The second and third characters must be 'in'. The rest can be any combination of characters.

```
SELECT name FROM nation WHERE name LIKE '_in%';
```

Result :
Finland
Singapore

CHR Function

Description

The **CHR** function returns a character corresponding to the given ASCII code value.

Syntax

```
CHR( number_operand )
```

number_operand : Specifies the number whose ASCII character is to be returned.

Example

The following is an example that returns the character corresponding to the ASCII code 90.

```
SELECT CHR(90) FROM db_root;
```

Result : 'Z'

POSITION Function

Description

The **POSITION** function returns the position of one string within another. You can have two bit strings or two character strings as the input of the function, but attempting to get the position of a bit string within a character string will cause an error.

The **POSITION** function returns an integer index value starting from 1. This means the search begins from the first position of the string. If the input is a character string, it is measured in characters; if it is a bit string, it is measured in bits.

Syntax

```
POSITION ( position_operand   IN  position_operand )

position_operand :
bit string
character string
NULL
```

This function determines the position of one string within another. If the *position_operand* is not found, 0 is returned.

- *position operand* : Specifies the character string whose position is to be returned. If the value is a blank string, 1 is returned. If the value is **NULL**, **NULL** is returned.

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. See the **SUBSTRING** Function section for such examples.

Example

The following is an example that returns the position of 'of' within the string 'United States of America'.

```
SELECT POSITION ('of' IN 'United States of America')
FROM db_root;
```

Result : 15

INSTR Function

Description

The **INSTR** function, similarly to the **POSITION** function, returns the position of a substring within another string; the position where the search begins can be specified in the **INSTR** function.

Syntax

```
INSTR( string , substring [, position] )
```

Searches the position of a *substring* in a *string*.

- *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* and then searches towards the beginning of the *string*.

Example

- **Example 1**

  The following is an example that returns the position of 'ra' in the character string 'United Arab Emirates'. It returns the position at which the first occurrence of 'ra' begins.

  ```
  SELECT INSTR('United Arab Emirates', 'ra', 1) FROM db_root;
  ```

  Result : 9

- **Example 2**

The following is an example that searches the string 'United Arab Emirates', beginning with the 13th character, for the string 'ra'. It returns the position at which the second occurrence of 'ra' begins.

```
SELECT INSTR('United Arab Emirates', 'ra', 13) FROM db_root;
```

Result : 16

- **Example 3**

  The following is an example that searches the string 'United Arab Emirates' backward from the last character. It returns the position at which the second occurrence of 'ra' begins.

```
SELECT INSTR('United Arab Emirates', 'ra', -1) FROM db_root;
```

  Result : 16

- **Example 4**

  The following is an example that searches the string 'United Arab Emirates' backward from the last character to the fifth character from the end, looking for 'ra'. It returns the position at which the first occurrence of 'ra' begins.

```
SELECT INSTR('United Arab Emirates', 'ra', -5) FROM db_root;
```

  Result : 9

REPLACE Function

Description

The **REPLACE** function replaces a string in an expression with another string and returns the result.

Syntax

```
REPLACE( char1, search_string [, replacement_string ] )

char1 :
character string
NULL

search_string :
character string
NULL

replacement_string :
character string
NULL
```

The **REPLACE** function replaces *search_string* with *replacement_string* and returns *char1*.

- *char1* : Specifies the original string. If the value is **NULL**, **NULL** is returned.
- *search_string* : Specifies the string whose value is to be replaced. If the value is **NULL**, **NULL** is returned.
- *replacement_string* : Specifies the string to replace the *search_string*. If this value is omitted, *char1* is returned with the *search_string* removed. If the value is **NULL**, **NULL** is returned.

Example

The following is an example that replaces the value of the **gender** column with Men's if it is M.

```
SELECT REPLACE(gender, 'M', 'Men''s' ), sports, name FROM event
WHERE name = '100m Freestyle'
AND gender = 'M';
```

Result : Men's Swimming 100m Freestyle

TRANSLATE Function

Description

The **TRANSLATE** replaces a sequence of characters in a string with another set of characters. For example, TRANSLATE(char1, 'abc', '123') replaces all occurrences of a, b and c in char1 with 1, 2 and 3 respectively.

Syntax

```
TRANSLATE( char1, from [, to] )

char1 :
character string
NULL

from :
character string
NULL

to :
character string
NULL
```

The **TRANSLATE** function returns *char1* with all occurrences of each character in the *from* argument replaced by its corresponding character in the *to* argument.

- *char1* : Specifies the original string. If the value is **NULL**, **NULL** is returned.
- *from* : Specifies the string to be replaced. If the number of characters in *from* is greater than that of *to*, the characters in *from* which do not have corresponding characters in *to* are removed from *char1*. If the value is **NULL**, **NULL** is returned.
- *to* : Specifies the characters that correspond to ones in the *from* argument. If the value is **NULL**, **NULL** is returned.

Example

The following is an example that returns the result after replacing all occurrences of 1, 2, 3, 4, 5 in the string with A, B, C, D, E respectively.

```
SELECT TRANSLATE('1234567890123', '12345', 'ABCDE' ) FROM db_root;
```

Result : ABCDE67890ABC

SUBSTR Function

Description

The **SUBSTR** function extracts a portion of a character string.

Syntax

```
SUBSTR( string, position [, substring_length])

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. The starting position of the string is always 1. If the value is negative, the system counts backward from the end of the string. If the value is **NULL**, **NULL** is returned.
- *substring_length* : Specifies the length of the string to be extracted. If this argument is omitted, the whole string is returned.

Example

- **Example 1**

  The following is an example that returns the result when the input value is **NULL**.

  ```
  SELECT SUBSTR(NULL, 2) FROM db_root;
  ```

  Result : NULL

- **Example 2**

  The following is an example that extracts characters starting at the 13th position.

  ```
  SELECT SUBSTR('United Arab Emirates', 13) FROM db_root;
  ```

  Result : 'Emirates'

- **Example 3**

  The following is an example that extracts 4 characters starting at the 8th position.

  ```
  SELECT SUBSTR('United Arab Emirates', 8, 4) FROM db_root;
  ```

  Result : 'Arab'

- **Example 4**

  The following is an example that extracts 2 characters starting at the 8th position from the end of the string.

  ```
  SELECT SUBSTR('United Arab Emirates', -8, 2) FROM db_root;
  ```

  Result : 'Em'

SUBSTRING Function

Description

The **SUBSTRING** function extracts a specified portion from a given character string starting at the position specified. If the number of characters to be extracted is omitted, the **SUBSTRING** function returns the characters starting at the position specified to the end of the string.

Syntax

```
SUBSTRING(substring_operand FROM integer [FOR integer] )

substring_operand :
bit string
character string
NULL
```

- *substring_operand* : Specifies the input character string. A bit or character string can be specified. If the input value is **NULL**, **NULL** is returned.
- *integer* : The mandatory **FROM** *integer* value represents the starting position of the substring to be extracted. If the value is smaller than 1, the starting position is set to the beginning of the string (the default 1). The optional **FOR** *integer* value represents the number of bits or characters to be extracted depending on the string type. If this value is smaller than 0, **FOR** *integer* value is ignored. An integer value must be specified after the **FOR** keyword.
- If the sum of the **FROM** *integer* value is larger than the length of the string whose substring is to be extracted, an empty string is returned as the result.

- If the sum of the **FROM** *integer* value and the **FOR** *integer* value is smaller than the length of the input string, the substring starting at the position **FROM** *integer* is extracted for the length specified by the **FOR** *integer* value. If any of the input value is **NULL**, **NULL** is returned.

Example

- **Example 1**

  The following is an example that returns the result if **NULL** exists in an input value.

  ```
  SELECT SUBSTRING(NULL FROM 2) FROM db_root;
  ```

  Result : NULL

- **Example 2**

  The following is an example that extracts characters starting at the 13th position.

  ```
  SELECT SUBSTRING('United Arab Emirates' FROM 13) FROM db_root;
  ```

  Result : 'Emirates'

- **Example 3**

  The following is an example that returns 4 characters starting at the 8th position.

  ```
  SELECT SUBSTRING('United Arab Emirates' FROM 8 FOR 4) FROM db_root;
  ```

  Result : 'Arab'

- **Example 4**

  The following is an example that extracts 2 characters from the first position (default value) if the **FROM** value is smaller than 1 in the **SUBSTRING** function.

  ```
  SELECT SUBSTRING('United Arab Emirates' FROM -9 FOR 2) FROM db_root;
  ```

  Result : 'Un'

Numeric Functions

RANDOM/RAND Function

Description

The **RANDOM/RAND** functions return a uniform distributed value as an integer from the range [0, 2^{31}]. **RAND** generate a single value regardless of the number of output rows displayed; **RANDOM** generates different values for each output row.

Syntax

```
RANDOM( )
RAND( )
```

Example

The following is an example that returns a random integer between 0 and 2^{31}. The example below shows difference between **RAND** and **RANDOM**.

```
SELECT RAND(), RANDOM() FROM db_class;
        rand()     random()
=========================
        11766         2104
        11766        10548
        11766         5125
...
        11766        10900
        11766        19111
```

DRANDOM/DRAND Function

Description

The **DRANDOM/DRAND** functions return a uniform distributed value as a double precision float from the range [0.0, 1.0]. **DRAND** generate a single value regardless of the number of output rows displayed; **DRANDOM** generates different values for each output row.

Syntax

```
DRANDOM( )
DRAND( )
```

Example

The following is an example that returns a random floating point value between 0.0 and 1.0. The example below shows difference between **DRAND** and **DRANDOM**.

```
SELECT DRAND(), DRANDOM() FROM db_class;
                  drand()                  drandom()
=================================================
    1.661732840968047e-001    3.210547196874904e-001
    1.661732840968047e-001    8.232062746055483e-001
    1.661732840968047e-001    7.202368236335337e-001
...
    1.661732840968047e-001    1.604052858058412e-001
    1.661732840968047e-001    9.024323252052370e-001
```

ROUND Function

Description

The **ROUND** function returns a number rounded to the specified number of places. The data type of the return value is the same as that of the argument.

Syntax

```
ROUND( number_operand, integer )
```

- *number_operand* : Specifies the number to be rounded.
- *integer* : Specifies the number of places to be rounded to the right of the decimal point. If the value is negative, the number is rounded off to the left of the decimal point.
- The data type of the return value is the same as that of the *number_operand*.

Example

- **Example 1**

  The following example rounds 15.456 to two decimal places.

  ```
  SELECT ROUND(15.456, 2) FROM db_root;
  ```

  Result : 15.460

- **Example 2**

  The following example rounds 15.456 to one decimal place.

  ```
  SELECT ROUND(15.456, 1) FROM db_root;
  ```

  Result : 15.500

- **Example 3**

  The following example rounds a number to zero decimal place to represent the integer part only.

  ```
  SELECT ROUND(15.456, 0) FROM db_root;
  ```

  Result : 15.000

- **Example 4**

  The following example rounds 15.456 one digit to the left of the decimal point.

  ```
  SELECT ROUND(15.456, -1) FROM db_root;
  ```

  Result : 20.000

CEIL Function

Description

The **CEIL** function returns the smallest integer that is not less than its argument. The data type of the return value is the same as that of the argument.

Syntax

```
CEIL( number_operand )
```

- *number_operand* : Specifies the number to calculate the ceiling of.

Example

- **Example 1**

  The following is an example that outputs the ceil of 10.145.

  ```
  SELECT CEIL(10.145) FROM db_root;
  ```

  Result : 11.000

- **Example 2**

  The following is an example that outputs the ceil of 10.

  ```
  SELECT CEIL(10) FROM db_root;
  ```

  Result : 10

- **Example 3**

  The following is an example that outputs the ceil of -10.145.

  ```
  SELECT CEIL(-10.145) FROM db_root;
  ```

  Result : -10.000

- **Example 4**

  The following is an example that outputs the ceil of -10.

  ```
  SELECT CEIL(-10) FROM db_root;
  ```

  Result : -10

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* : Specifies the number to calculate the floor of.

Example

- **Example 1**

  The following is an example that outputs the floor of 10.145.

```
SELECT FLOOR(10.145) FROM db_root;
```

Result : 10.000

- **Example 2**

  The following is an example that outputs the floor of 10.

```
SELECT FLOOR(10) FROM db_root;
```

  Result : 10

- **Example 3**

  The following is an example that outputs the floor of -10.145.

```
SELECT FLOOR(-10.145) FROM db_root;
```

  Result : -11.000

- **Example 4**

  The following is an example that outputs the floor of -10.

```
SELECT FLOOR(-10) FROM db_root;
```

  Result : -10

MOD Function

Description

The **MOD** function returns the remainder of the first argument divided by the second argument.

The **MOD** function is different from the typical operation 'm-n*FLOOR(m/n)'.

The following table shows the results from the **MOD** function.

m	n	MOD(m. n)	Classical Modulus m-n*FLOOR(m/n)
9	5	4	4
-9	5	-4	1
9	-5	4	-1
-9	-5	-4	-4
9	0	9	Divided by 0 error

Syntax

```
MOD(number_operand_m, number_operand_n)
```

Returns the remainder of the first argument *number_operand_m* divided by the second argument *number_operand_n*.

- *number_operand_m* : Represents the dividend, the number that is divided.
- *number_operand_n* : Represents the divisor, the number that divides *number_operand_m*. If the number is 0, *number_operand_m* itself is returned.

Example

The following is an example that returns the remainder of 10.945 divided by 0.4.

```
SELECT MOD(10.945, 0.4) FROM db_root;
```

Result : .145

POWER Function

Description

The **POWER** function returns the value of the given number to the specified power. The data type of the return value is the same as that of the argument. If the result value has too many or too few digits or decimal places, an overflow or truncation can occur.

Syntax

```
POWER( base_operand, exponent_operand )
```

Returns the *base_operand* to the *exponent_operand* power. If the *base_operand* is negative, the exponent *exponent_operand* must be an integer.

- *base_operand* : Represents the number to be raised.
- *exponent_operand* : Represents the exponent.

Example

- **Example 1**

  The following is an example that returns 5 raised to the power of 3.1.

  ```
  SELECT POWER(5, 3.1) FROM db_root;
  ```

  Result : 146.8

- **Example 2**

  The following is an example that returns 9.05 raised to the power of 2.1.

  ```
  SELECT POWER(9.05, 2.1) FROM db_root;
  ```

  Result : 102.085

- **Example 3**

  The following is an example that returns -9.05 raised to the power of 3.

  ```
  SELECT POWER(-9.05, 3) FROM db_root;
  ```

  Result : -741.22

- **Example 4**

  The following is an example that returns -5 raised to the power of 3.2.

  ```
  SELECT POWER(-5, 3.2) FROM db_root;
  ```

  Result : Error

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* : Specifies the number to calculate the absolute number.

Example

The following is an example that returns the absolute value of -0.5.

```
SELECT ABS(-0.5) FROM db_root;
```

Result : .5

SIGN Function

Description

The **SIGN** function returns the sign of a number. It returns 1 for a positive value, -1 for a negative value, and 0 for zero.

Syntax

```
SIGN(number_operand)
```

- *number_operand* : Specifies the number to return the sign of.

Example

The following is an example that returns the sign of -15.03.

```
SELECT SIGN(-15.03) FROM db_root;
```

Result : -1

TRUNC Function

Description

The **TRUNC** function returns a number truncated to a specified number of digits. The data type of the return value is the same as that of the argument.

Syntax

```
TRUNC( number_operand, integer )
```

- *number_operand* : Specifies the number to be truncated.
- *integer* : Specifies the number of decimal places to truncate to. If this value is negative, the function truncates digits left of the decimal point.

Example

- **Example 1**

  The following is an example that returns 235.1538 truncated to two decimal places.

  ```
  SELECT TRUNC(235.1538, 2) FROM db_root;
  ```

  Result : 235.1500

- **Example 2**

  The following is an example that returns 235.1538 truncated to one decimal place.

  ```
  SELECT TRUNC(235.1538, 1) FROM db_root;
  ```

  Result : 235.1000

- **Example 3**

  The following is an example that returns only the integer part of 235.1538.

  ```
  SELECT TRUNC(235.1538, 0) FROM db_root;
  ```

  Result : 235.0000

- **Example 4**

  The following is an example that returns 235.1538 truncated to one digit left of the decimal point.

  ```
  SELECT TRUNC(235.1538, -1) FROM db_root;
  ```

  Result : 230.0000

- **Example 5**

The following is an example that returns 235.1538 truncated to two digits left of the decimal point.

```
SELECT TRUNC(235.1538, -2) FROM db_root;
```

Result : 200.0000

Date/Time Functions

ADD_MONTH Function

Description

The **ADD_MONTHS** function returns a new date with the specified number of months added to the input date. If the calculated date is greater than the last day of the month specified, the latter is returned. Except for this, the former is returned.

Syntax

```
ADD_MONTHS ( date_argument , month )

date_argument :
date
NULL

month :
integer
NULL
```

- Adding a specified date (*date_argument*) with the specified number of months (*month*) returns a date. If the calculated value exceeds the **DATE** limit, an error occurs.
- *date_argument* : Specifies the date. If the value is **NULL**, **NULL** is returned. To apply the **TIMESTAMP** value, an explicit cast to the **DATE** type must be performed.
- *month* : Specifies the number of the months to be added to the *date_argument*. If the value is **NULL**, **NULL** is returned. If the given value is not an integer type, an implicit type cast to an integer takes place. The value can be specified as a negative number.

Example

The following is an example that returns the date after adding 2/29/1980 with 12 months.

```
SELECT ADD_MONTHS(DATE '2/29/1980',12) FROM db_root;
```

Result : '1981-02-28'

MONTHS_BETWEEN Function

Description

The **MONTHS_BETWEEN** function returns the number of months between two **DATE** values.

Syntax

```
MONTHS_BETWEEN(date_argument, date_argument)

date_argument :
date
NULL
```

If the first argument is earlier than the second, a negative value is returned. When the two dates have the same day component or are both the last day of the month, the return value is an integer. Otherwise, the return value includes a fraction that considers the difference in the days based on a 31-day month. Since it is represented in **DOUBLE** format, scientific notations are used.

- *date_argument* : If any of the two arguments is **NULL**, **NULL** is returned as the result. To apply the **TIMESTAMP** value to the *date_argument*, an explicit cast to the **DATE** type must be performed.

Example

- **Example 1**

  The following is an example that returns the difference between the dates 1/13/1999 and 2/3/1999 divided by 31.

  ```
  SELECT MONTHS_BETWEEN(DATE '1/13/1999',DATE '2/3/1999') FROM db_root;
  ```

  Result : -6.774193548387097e-01

- **Example 2**

  The following is an example that returns the difference in months between the dates 8/15/2008 and 8/15/2006.

  ```
  SELECT MONTHS_BETWEEN(DATE '8/15/2008',DATE '8/15/2006') FROM db_root;
  ```

  Result : 2.400000000000000e+01

LAST_DAY Function

Description

The **LAST_DAY** function returns the last day of the month for a given date.

Syntax

```
LAST_DAY ( date_argument )

date_argument :
date
NULL
```

- *date_argument* : Specifies the date used to calculate the last day of the month. If the value is **NULL**, **NULL** is returned. To apply the **TIMESTAMP** value, an explicit cast to the **DATE** type must be performed.

Example

The following is an example that returns the last day of January 2000.

```
SELECT LAST_DAY(DATE '1/01/2000') FROM db_root;
```

Result : 2000-01-31

SYS_DATE (SYSDATE, CURRENT_DATE) Function

Description

SYS_DATE (or **SYSDATE**, **CURRENT_DATE**) function returns the current date in **DATE** format.

Syntax

```
SYS_DATE
SYSDATE
CURRENT_DATE
```

Example

The following are examples that return the current date in the **DATE** format.

```
SELECT SYS_DATE FROM db_root;
SELECT SYSDATE FROM db_root;
SELECT CURRENT_DATE FROM db_root;
```

SYS_TIME (SYSTIME, CURRENT_TIME) Function

Description

SYS_TIME (or **SYSTIME**, **CURRENT_TIME**) function returns the current time in **TIME** format.

Syntax

```
SYS_TIME
SYSTIME
CURRENT_TIME
```

Example

The following are examples that return the current time in the **TIME** format.

```
SELECT SYS_TIME FROM db_root;
SELECT SYSTIME FROM db_root;
SELECT CURRENT_TIME FROM db_root;
```

SYS_TIMESTAMP (SYSTIMESTAMP, CURRENT_TIMESTAMP) Function

Description

The **SYS_TIMESTAMP** (or **SYSTIMESTAMP**, **CURRENT_TIMESTAMP**) function returns the current date and time in **TIMESTAMP** format.

Syntax

```
SYS_TIMESTAMP
SYSTIMESTAMP
CURRENT_TIMESTAMP
```

Example

The following are examples that return the current date and time in **TIMESTAMP** format.

```
SELECT SYS_TIMESTAMP FROM db_root;
SELECT SYSTIMESTAMP FROM db_root;
SELECT CURRENT_TIMESTAMP FROM db_root;
```

SYS_DATETIME (SYSDATETIME, CURRENT_DATETIME) Function

Description

The **SYS_DATETIME** (or **SYSDATETIME**, **CURRENT_DATETIME**) function returns the current date as a **DATETIME** type.

Syntax

```
SYS_DATETIME
SYSDATETIME
CURRENT_DATETIME
```

Example

All of the following are examples that return the current date as a **DATETIME** type.

```
SELECT SYS_DATETIME FROM db_root;
SELECT SYSDATETIME FROM db_root;
SELECT CURRENT_DATETIME FROM db_root;
```

Data Type Conversion

TO_CHAR Function

- Date Type (DATE, TIME, TIMESTAMP, DATETIME)
- Number Type (NUMERIC)

Date Type (DATE, TIME, TIMESTAMP, DATETIME)

Description

The **TO_CHAR** function converts date-time values (**DATE**, **TIME**, **TIMESTAMP, DATETIME**) to **VARCHAR** type according to the specified "date-time" format. If the date-time format is omitted, 'MM/DD/YYYY' format is used for the **DATE** type, 'HH:MI:SS' format is used for the **TIME** type, 'HH:MI:SS AM MM/DD/YYYY' format is used for the **TIMESTAMP** type, and 'HH:MI:SS.FF AM MM/DD/YYYY' format is used for the **DATETIME** type.

Syntax

```
TO_CHAR( date_time_argument [, format_argument[, date_lang_string_literal ]] )

date_time_argument :
date
time
timestamp
datetime
NULL

format_argument :
character strings (see the Date-Time Format table)
NULL

date_lang_string_literal : (see the date_lang_string_literal table)
'en_US'
'ko_KR'
```

- *date_time_argument* : Specifies the date-time value to be converted to **VARCHAR** type. If the value is **NULL**, **NULL** is returned.
- *format_argument* : Specifies the format of the return value. If the value is **NULL**, **NULL** is returned.
- *date_lang_string_literal* : Specifies the language for the return value (see the date_lang_string_literal table). The default is 'en_US'. You can modify the value by using the **CUBRID_DATE_LANG** environment variable.

Date-Time Format

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)
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 (0 - 11)
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 taken by each element in the result format is as follows:

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

- **Example 1**

  The following is an example that returns only the date value in a **DATE** type.

  ```
  SELECT TO_CHAR(DATE '12/25/1999', 'dd') FROM db_root;
  ```

  Result : 25

- **Example 2**

  The following is an example that returns only the hour value in a **TIME** type.

  ```
  SELECT TO_CHAR(TIME '10:20:30 AM', 'HH24') FROM db_root;
  ```

  Result : 10

- **Example 3**

  The following is an example that returns the month in characters, abbreviated month name and year values in a **TIMESTAMP** type.

  ```
  SELECT TO_CHAR(TIMESTAMP '10:20:30 AM 12/25/1999', 'MONTH MON yyyy') FROM
  db_root;
  ```

  Result : DECEMBER DEC 1999

- **Example 4**

  The following is an example that returns the day of the week in number value in a **TIMESTAMP** type.

  ```
  SELECT TO_CHAR(TIMESTAMP '10:20:30 AM 12/25/1999', 'D') FROM db_root;
  ```

  Result : 7

- **Example 5**

  The following is an example that returns the second and millisecond from **DATETIME**.

  ```
  SELECT TO_CHAR(DATETIME'10:12:13 6/5/2009','ss ff') FROM db_root;
  ```

  Result : 13 000

Number Type (NUMERIC)

Description

The **TO_CHAR** function converts a value of **NUMERIC** type to **VARCHAR** type according to the number format.

Syntax

```
TO_CHAR(numeric_argument[, format_argument ])

numeric_argument :
numeric(decimal)
integer
smallint
float(real)
double
NULL

format_argument :
character strings (see the Number Format table)
NULL
```

- *numeric_argument* : Specifies the numeric value to be converted to **VARCHAR**. If the value is **NULL**, **NULL** is returned.

format_argument : Specifies the format of the return value. If this argument is omitted, the default format is used. 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. A space is returned for a leading 0.
0	0999	0, not a space is returned for a leading 0.
S	S9999	Returns "+" for a positive and "-" for a negative number at the specified position.
C	C9999	Returns the ISO currency code at the specified position.
,	9,999	Returns a comma (",") at the specified position.
.	9.999	Returns a punctuation mark (".") that separates the integer part of a number from the fractional part at the specified position.
EEEE	9.99EEEE	Returns a scientific notation number.

Example

- **Example 1**

  The following is an example that returns a 4-digit **VARCHAR** type with a sign.

  ```
  SELECT TO_CHAR(1234,'S9999') FROM db_root;
  ```

  Result : +1234

- **Example 2**

  The following is an example that converts a scientific notation number to a **VARCHAR** type.

  ```
  SELECT TO_CHAR(1.234E-4) FROM db_root;
  ```

  Result : 0.0001234

TO_NUMBER Function

Description

The **TO_NUMBER** function converts a string to **NUMERIC** type according to the number format.

Syntax

```
TO_NUMBER(string_argument[, format_argument ])

string_argument :
character strings
NULL

format_argument :
character strings
NULL
```

The precision and scale of the return value are determined by those of the *string_argument* value. For example, the precision of '123.45' is 5, and its scale is 2. For '1.23e+1', the precision is 3, and the sclae is 1.

- *string_argument* : Specifies the string to be converted to **NUMERIC** type. If the value is **NULL**, **NULL** is returned.
- *format_argument* : Specifies the format of the return value. If this argument is omitted, a string is returned in the **NUMERIC** format. If the value is **NULL**, **NULL** is returned.

Example

- **Example 1**

  The following is an example that converts the string '-1234' to **NUMERIC** type.

  ```
  SELECT TO_NUMBER('-1234') FROM db_root;
  ```

  Result : -1234

- **Example 2**

  The following is an example that converts the string '0.123' to **NUMERIC** type in the format 0.0000.

  ```
  SELECT TO_NUMBER('0.123','0.0000') FROM db_root;
  ```

  Result : .1230

DescriptionTO_DATE Function

The **TO_DATE** function converts a string to **DATE** type according to the date-time format.

Syntax

```
TO_DATE(string_argument[,format_argument[,date_lang_string_literal]])

string_argument :
character strings
NULL

format_argument :
character strings (see the Date-Time Format table)
NULL

date_lang_string_literal : (see the date_lang_string_literal table)
'en_US'
'ko_KR'
```

- *string_argument* : Specifies the string to be converted to DATE type. **NULL**, **NULL** is returned.

- *format_argument* : Specifies the format by which an input string is to be interpreted. If omitted, the default format 'MM/DD/YYYY' is used to interpret character strings. **NULL**, **NULL** is returned.
- *date_lang_string_literal* : Specifies the language for the return value by using the **CUBRID_DATE_LANG** environment variable.

Example

- **Example 1**

  The following is an example that converts a string to **DATE** type according to the default format 'MM/DD/YYYY'.

  ```
  SELECT TO_DATE('9/6/1976') FROM db_root;
  ```

  Result : 09/06/1976

- **Example 2**

  The following is an example that converts a string to **DATE** type according to the 'YYYY MM DD' format.

  ```
  SELECT TO_DATE('1999 3 4', 'YYYY MM DD') FROM db_root;
  ```

  Result : 03/04/1999

TO_TIME Function

Description

The **TO_TIME** function converts a string to **TIME** type according to the date-time format.

Syntax

```
TO_TIME(string_argument[,format_argument [,date_lang_string_literal]]):

string_argument :
character strings
NULL

format_argument :
character strings (see the Date-Time Format table)
NULL

date_lang_string_literal : (See the date_lang_string_literal table)
'en_US'
'ko_KR'
```

- *string_argument* : Specifies the string to be converted to **TIME** type. If the value is **NULL**, **NULL** is returned.
- *format_argument* : Specifies the format by which an input string is to be interpreted. If the value is **NULL**, **NULL** is returned. If this argument is omitted, the default format 'HH:MI[:SS] [am|pm]' is used to interpret character strings.
- *date_lang_string_literal* : Specifies the language for the return value by using the **CUBRID_DATE_LANG** environment variable.

Example

- **Example 1**

  The following is an example that converts the string '10:30:20 AM' to **TIME** type according to the default format.

  ```
  SELECT TO_TIME('10:30:20 AM') FROM db_root;
  ```

  Result : 10:30:20 AM

- **Example 2**

The following is an example that converts a string to **TIME** type according to the specified format.

```
SELECT TO_TIME('HOUR: 10 MINUTE: 30 SECOND: 20', '"HOUR:" HH24 "MINUTE:" MI
"SECOND:" SS')
FROM db_root;
```

Result : 10:30:20 AM

TO_TIMESTAMP Function

Description

The **TO_TIMESTAMP** function converts a string to **TIMESTAMP** type according to the date-time format.

Syntax

```
TO_TIMESTAMP(string_argument[, format_argument[,date_lang_string_literal]])

string_argument :
character strings
NULL

format_argument :
character strings (see the Date-Time Format table)
NULL

date_lang_string_literal : (see the date_lang_string_literal table)
'en_US'
'ko_KR'
```

- *string_argument* : Specifies the string to be converted to **TIMESTAMP** type. If the value is **NULL**, **NULL** is returned.
- *format_argument* : Specifies the format by which an input string is to be interpreted. If the value is **NULL**, **NULL** is returned. If this argument is omitted, the default format 'HH:MI[:SS] [am|pm] MM/DD/YYYY' is used to interpret character strings.
- *date_lang_string_literal* : Specifies the language for the return value. You can modify the value using the **CUBRID_DATE_LANG** environment variable.

Example

- **Example 1**

  The following is an example that converts a string to **TIMESTAMP** type according to the default format.

  ```
  SELECT TO_TIMESTAMP('10:30:20 AM 12/25/1999') FROM db_root;
  ```

  Result : 1999-12-25 10:30:20

- **Example 2**

  The following is an example that converts a string to **TIMESTAMP** type according to the specified format.

  ```
  SELECT TO_TIMESTAMP('YEAR: 1999 MONTH: 12 DAY: 25 HOUR: 10 MINUTE: 30 SECOND:
  20','"YEAR:" YYYY "MONTH:" MM "DAY:" DD "HOUR:" HH24 "MINUTE:" MI "SECOND:"
  SS') FROM db_root;
  ```

  Result : 1999-12-25 10:30:20

TO_DATETIME Function

Description

The **TO_DATETIME** function converts a string to **DATETIME** type according to the date-time format.

Syntax

```
TO_DATETIME (string_argument[,format_argument[,date_lang_string_literal]])

string_argument :
```

```
character strings
NULL

format_argument :
character strings (see the Date-Tme Format table)
NULL

date_lang_string_literal : (see the date_lang_string_literal table)
'en_US'
'ko_KR'
```

- *string_argument* : Specifies the string to be converted to a **DATETIME** type. If the value is **NULL**, **NULL** is returned.
- *format_argument* : Specifies the format by which an input string is to be interpreted. If this argument is omitted, the default format 'HH:MI:SS.FF [am|pm] MM/DD/YYYY' is used. If the value is **NULL**, **NULL** is returned.
- *date_lang_string_literal* : Specifies the language for the return value. You can modify the value by specifying the **CUBRID_DATE_LANG** environment variable.

Example

- **Example 1**

  The following is an example of converting a string to a **DATETIME** type by using the basic format.

  ```
  SELECT TO_DATETIME('6/5/2009 10:12:13') FROM db_root;
  SELECT TO_DATETIME('10:12:13 AM 6/5/2009') FROM db_root;
  ```

  Result:
  10:12:13.000 AM 06/05/2009

  10:12:13.000 AM 06/05/2009

- **Example 2**

  The following is an example of converting a string to a **DATETIME** type by using the 'HH:MI:SS YYYY/MM/DD' format.

  ```
  SELECT TO_DATETIME('10:12:13 2009/6/5', 'HH:MI:SS YYYY/MM/DD') FROM db_root;
  ```

  Result: 10:12:13.000 AM 06/05/2009

Aggregate Functions

MIN Function

Description

The **MIN** function retrieves the minimum value of the specified expression or path expression. The character strings are compared in dictionary order. The minimum value is the one closest to the first entry in the dictionary. For other data types, the minimum value is the smallest value. The **MIN** function cannot be used for attributes that include collections or object domains (user-defined classes or multimedia classes).

Syntax

```
MIN ( [ DISTINCT | UNIQUE | ALL ] expression )
```

With *expression* for the **MIN** function, you can specify a single-value expression.

- *expression* : Specifies the expression to calculate the minimum value.
- **ALL** : Gets the minimum value of all instances (default).
- **DISTINCT** or **UNIQUE** : Gets the minimum value of unique values without duplicates.

- However, whether you use the keyword **DISTINCT**, **UNIQUE** or **ALL**, you will get the same minimum value as the result.

Example

The following is an example of returning the minimum number of gold medals that Korea won in the Olympic Games from 1988 to 2004.

```
SELECT MIN(gold)
FROM participant
WHERE nation_code = 'KOR';
```

Result : 7

MAX Function

Description

The **MAX** function retrieves the maximum value of the specified expression. The character strings are compared in dictionary order. The maximum value is the one closest to the last entry in the dictionary. For other data types, the maximum value is the largest value. The **MAX** function cannot be used for attributes that include collections or object domains (user-defined classes or multimedia classes).

Syntax

```
MAX ( [ DISTINCT | UNIQUE | ALL ] expression )
```

With *expression* for the **MAX** function, you can specify a single-value expression.

- *expression* : Specifies the expression to calculate the maximum value.
- **ALL** : Gets the maximum value of all instances (default).
- **DISTINCT** or **UNIQUE** : Gets the maximum value of unique values without duplicates.
- However, whether you use the keyword **DISTINCT**, **UNIQUE** or **ALL**, you will get the same maximum value as the result.

Example

The following is an example of returning the maximum number of gold medals that Korea won in the past Olympic Games.

```
SELECT MAX(gold)
FROM participant
WHERE nation_code = 'KOR';
```

Result : 12

SUM Function

Description

The **SUM** function returns the sum of the values in expression. Numeric data types with the addition operator can also be calculated by the **SUM** function. String data types as well as **SET**, **MULTISET** and **LIST (SEQUENCE)** types cannot be calculated by the **SUM** function; they can be concatenated with the + operator. In addition, the **SUM** function cannot be used for expressions that contain attributes with object domains (user-defined classes or multimedia classes).

Syntax

```
SUM ( [ DISTINCT | UNIQUE | ALL ] expression )
```

With the input for the **SUM** function, you can specify a single-value expression.

- *expression* : Specifies the expression to calculate the sum.
- **ALL** : Gets the sum of all instances of the *expression* (default).

- **DISTINCT** or **UNIQUE** : Gets the sum of unique values without duplicates.

Example

The following is an example that returns the total number of gold medals in the 1998 Olympics.

```
SELECT SUM(gold)
FROM participant
WHERE host_year = 1988;
```

Result : 241

AVG Function

Description

The **AVG** function calculates the arithmetic average of an expression with a single value. You can use any data type that can be added or divided by a number. The data type of the attribute to be averaged is maintained. For example, if the attribute is a floating point value, the **AVG** function also returns a **FLOAT**. You cannot get the average of a collection.

syntax

```
AVG ( [ DISTINCT | UNIQUE | ALL ] expression )
```

When calculating the average of a value specified in an *expression*, you can determine whether to include all instances or only unique values.

- *expression* : Specifies the expression to be averaged.
- **ALL** : Includes all values to calculate the average (default).
- **DISTINCT** or **UNIQUE** : Includes only unique values to calculate the average.

Example

The following is an example that returns the average number of gold medals that Korea won in the Olympic Games from 1988 to 2004.

```
SELECT AVG(gold)
FROM participant
WHERE nation_code = 'KOR';
```

Result : 9

COUNT Function

Description

The **COUNT** function returns the number of instances in the query results with an entered expression. It also calculates the number of attributes that have a collection attribute and object domain (user-defined class or multimedia class). If the expression is specified in the brace, only instances that are not **NULL** can be counted. If wildcard (*) is specified, however, all instances are counted even though some instances are duplicated or having **NULL**. In addition, **COUNT** function never returns **NULL**, only returns numbers.

syntax

```
COUNT ( * | [ DISTINCT | UNIQUE | ALL ] expression )
```

The **COUNT** function returns the number of instances for an expression in the parentheses. All instances that meet the conditions of the **WHERE** clause can be counted by specifying a wildcard (*) symbol. A unique value can be counted by specifying a **DISTINCT** or **UNIQUE** keyword before the expression.

- *expression* : Specifies the expression to calculate the number.
- **ALL** : Used to calculate the number of all the instances of a given expression (default).

- **DISTINCT** or **UNIQUE** : Gets the number of unique values without duplicates.

Example

The following is an example of returning the number of Olympic Games that had a mascot.

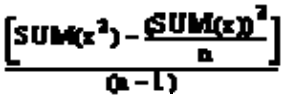
```
SELECT COUNT
FROM olympic
WHERE mascot IN NOT NULL;
```

Result : 9

VARIANCE Function

Description

The **VARIANCE** function calculates the variance of a given expression using the following formula:



Syntax

```
VARIANCE( [DISTINCT | UNIQUE | ALL] expression )
```

- *expression* : Specifies the expression to calculate the variance.
- **ALL** : Gets the variance of 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 that Korea won.

```
SELECT VARIANCE(gold)
FROM participant
WHERE nation_code = 'KOR';
```

Result : 5

STDDEV Function

Description

The **STDDEV** function calculates the standard deviation of a given expression.

Syntax

```
STDDEV( [DISTINCT | UNIQUE | ALL] expression )
```

- *expression* : Specifies the expression to calculate the standard deviation.
- **ALL** : Gets the standard deviation of all values (default).
- **DISTINCT** or **UNIQUE** : Gets the standard deviation of unique values without duplicates.

Example

The following is an example that returns the standard deviation of the number of gold medals that Korea won in the Olympic Games from 1988 to 2004.

```
SELECT STDDEV(gold)
FROM participant
WHERE nation_code = 'KOR';
```

Result : 2

Condition/Comparison Functions and Operators

CASE

Description

The **CASE** expression is used to get a value that changes depending on the conditions in the CUBRID statement. The **CASE** expression compares one or more comparison expressions with a value expression or a query to check if the condition is true. If it is true, a specified expression or a **NULL** value is returned.

Syntax

The **CASE** expression supports the following four syntax forms:

```
CASE control_expression simple_when_list
[ else_clause ]
END
CASE searched_when_list
[ else_clause ]
END
NULLIF ( expression_comma_list )
COALESCE ( expression_comma_list )

simple_when :
WHEN expression THEN result

searched_when :
WHEN search_condition THEN result

else_clause :
ELSE result

result :
expression | NULL
```

The **CASE** expression includes one or more *simple_when* or *searched_when* clauses. The **NULLIF and COALESCE** statement can be used as a conditional expression of specific types. In each **CASE** statement, all **WHEN** clauses must be the *simple_when* or *searched_when* form without mixing the two types. A **CASE** statement must end with the **END** keyword.

The expression value of the **THEN** clause that satisfies the conditions of the **WHEN** clause first becomes the value of the **CASE** expression. The *simple_when* clause is satisfied when the value of the **WHEN** expression equals to that of the *control_expression* in the **CASE** statement. *searched_when* clause is satisfied when the corresponding *search_condition* is true. If no **WHEN** clause is satisfied, the value of the **ELSE** clause becomes the value of the **CASE** expression. If the **ELSE** clause is omitted, the **ELSE NULL** clause is assumed.

To use a **CASE** expression using a *simple_when* clause, the data types of all **WHEN** clauses must be comparable with that of *control_expression*. Likewise, the data types of all **THEN** clauses must also be comparable. However, the data types of the **WHEN** and **THEN** clauses do not have to be comparable with each other.

According to the following rules, the data type of the value returned by the **CASE** expression is determined depending on the data type of the **THEN** or **ELSE** (optional) clause.

If the data types of all the **THEN** clauses are identical, they will be the data type of the **CASE** statement.

If the data types of the **THEN** clauses are not identical but comparable, the most generic type of the **THEN** clauses will be the data type of the **CASE** statement.

If one of the data types of the **THEN** clauses is a variable-length character or bit string, the result will be a variable-length type. If there is no variable-length character strings used, the longest character or bit string is returned as the result.

If none of the data types of the **THEN** clauses are approximate numeric, an exact numeric is returned as the result. The scale is determined based on the number having the largest number of decimal places.

The precision of the result is determined so that it can include all the numbers from the smallest to the largest specified by any of the **THEN** clauses in the **CASE** statement.

Example

The following is an example that outputs 'MAN' if the athlete's gender is M; it outputs 'WOMAN' if that is W.

```
SELECT name,
CASE WHEN gender = 'M' THEN 'MAN'
         WHEN gender = 'W' THEN 'WOMAN'
         END as GENDER
FROM athlete;
```

NULLIF and COALESCE

Description

The **NULLIF** and **COALESCE** functions are simple ways to process special **CASE** expressions, used as special forms of conditional expressions.
The **NULLIF** compares two expressions, and returns **NULL** if they are identical. Otherwise, it returns the first argument value.

NULLIF(*a, b*) works the same as the following **CASE** statement.

```
CASE
WHEN a = b THEN NULL
ELSE a
END
```

The **COALESCE** statement takes two expressions with the same data type as arguments, and returns the first value that is not **NULL**. If all the values are **NULL**, it returns **NULL**. **COALESCE**(*a, b*) works the same as the following **CASE** statement.

```
CASE WHEN a IS NOT NULL
THEN a
ELSE b
END
```

Specifying more than two expressions in the **COALESCE** statement works the same as adding a **THEN** clause to a **CASE** statement. The statement proceeds, sequentially checking the expression list for **NULL**, until it meets the non-**NULL** value. If there is no expression left, **COALESCE** returns **NULL**.

Example

- **Example 1**

  The following is an example that returns **NULL** if the mascot attribute value is 'HODORI', and the value itself otherwise.

  ```
  SELECT host_year, NULLIF(mascot, 'HODORI') FROM olympic;
  ```

- **Example 2**

  The following is an example that returns the value itself if the mascot attribute value is not **NULL**, and 'Not Exist' if it is **NULL**.

  ```
  SELECT host_year, COALESCE(mascot, 'Not Exist') FROM olympic;
  ```

NVL and NVL2 Functions

Description

The **NVL** and **NVL2** functions replace **NULL** with a specified substitute value.

Syntax

```
NVL( expr, expr2 )
NVL2( expr, expr1, expr2 )
```

The **NVL** returns the value of the *expr2* if the value of the *expr* is **NULL**, and the value of the *expr* itself if it is not **NULL**.

The **NVL2** function returns the value of the *expr1* if the value of the *expr* is not **NULL**, and the value of the *expr2* if it is **NULL**.

- *expr* : Specifies a substitute value when it is **NULL**.
- *expr1* : Specifies a return value when the value of the *expr* is not **NULL**.
- *expr2* : Specifies a return value when the value of the *expr* is **NULL**.

Example

- **Example 1**

  The following is an example that returns the value itself if the mascot attribute value is not **NULL**, and 'Not Exist' if it is **NULL**.

  ```
  SELECT host_year, NVL( mascot, 'Not Exist' ) FROM olympic;
  ```

- **Example 2**

  The following is an example that returns 'Exit' if the mascot attribute value is not **NULL**, and 'Not Exist' if it is **NULL**.

  ```
  SELECT host_year, NVL2( mascot, 'Exist', 'Not Exist' ) FROM olympic;
  ```

DECODE Function

Description

The **DECODE** function works the same as the **IF-THEN-ELSE** statement.

Syntax

```
DECODE( expression, search, result [, search, result]* [, default] )
```

- *expression* : Specifies the value to be compared.
- *search* : Specifies the value to be compared with the *expression*.
- *result* : Specifies the value to be returned when the *expression* and the *search* are identical.
- default : Specifies the value to be returned when there is no match. If there is no match without the default value specified, **NULL** is returned.
- Note that the **NULL** value of the *search* is treated the same as the **NULL** value of the *expression*.

Example

The following is an example where 'length: 4' is returned if the length of the character string is 4, 'length: 5' if it is 5, or 'length: 6' if it is 6.

```
SELECT DECODE( LENGTH('Korea'), 4, 'length: 4',
                                5, 'length: 5',
                                6, 'length: 6',
                                   'length: unknown')
FROM db_root;
```

Result : length: 5

GREATEST Function

Description

The **GREATEST** function returns the greatest value in the argument list.

Syntax

```
GREATEST( expression [, expression]* )
```

- *expression* : Lists one or more values. The data types of all arguments must be comparable. If the data types of all arguments are the same, such type is returned as the result. Otherwise, the return type is determined according to the following rules: If the argument type is a **VARCHAR(n) or CHAR VARYING(n)** or **CHAR(n)**, the return value is a **VARCHAR(n) or CHAR VARYING(n)** whose expression is the longest. If there is no **VARCHAR(n) or CHAR VARYING(n)** type, a **CHAR(n)** type is returned as the result. If there is an argument represented as an approximate numeric data type, the return type is also approximate numeric.

Example

The following is an example that returns the number of the gold, silver and bronze medals that Korea won in the 1988 Olympic Games and the greatest number among the returned ones.

```
SELECT gold, silver , bronze, GREATEST (gold, silver, bronze) FROM
participant
WHERE host_year=1988
AND nation_code = 'KOR';
Result)
       gold       silver       bronze  greatest(gold, silver, bronze)
=============================================================================
        12            10         11                                  12
```

LEAST Function

Description

The **LEAST** function returns the smallest value in the argument list.

Syntax

```
LEAST( expression [, expression]* )
```

- *expression* : Lists one or more values. The data types of all arguments must be comparable. If the data types of all arguments are the same, such type is returned as the result. Otherwise, the return type is determined according to the following rules: If the argument type is a **VARCHAR(n) or CHAR VARYING(n)** or **CHAR(n)**, the return value is the longest one of **VARCHAR(n) or CHAR VARYING(n)** in expression If there is no **VARCHAR(n) or CHAR VARYING(n)** type, a **CHAR(n)** type is returned as the result. If there is an argument represented as an approximate numeric data type, the return type is also approximate numeric.

Example

The following is an example that returns the number of the gold, silver and bronze medals Korea won in the 1988 Olympic Games and the smallest number.

```
SELECT gold, silver , bronze, LEAST(gold, silver, bronze) FROM participant
WHERE host_year=1988
AND nation_code = 'KOR';
Result)
         gold       silver       bronze  least(gold, silver, bronze)
=============================================================================
           12           10           11                           10
```

User Name Function

CURRENT_USER or USER Function

Description

The **CURRENT_USER** or **USER** function returns a string representing the name of the user who is currently logged in to the database.

Syntax

```
CURRENT_USER
USER
```

Example

The following is an example that returns the name of the user currently logged in to the database.

```
SELECT name FROM db_user WHERE name = CURRENT_USER;
SELECT name FROM db_user WHERE name = USER;
```

Click Counter Functions

INCR and DECR Functions

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.

Note

- 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 :
'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 :
'text...' 1

ROWNUM Function

ROWNUM/INST_NUM() Function

Description

The **ROWNUM** function returns the number representing the order of the tuples that will be generated by the query result. The first result tuple is assigned 1, and the second result tuple is assigned 2.

Syntax

```
INST_NUM()
ROWNUM
```

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 tuples 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 tuples.

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.

Note

- 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 tuple is written to the query result set. At this moment, the counter value that generates the serial number of the result set tuples 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 tuples.
- 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 instance from one table and then insert it into another by using the **ROWNUM** function in the **WHERE** clause.

Example

The following is an example that returns country names ranked first to fourth in overall medal standings in the 1988 Olympics.

```
SELECT  *
FROM (SELECT nation_code FROM participant
     WHERE host_year = 1988
     ORDER BY gold DESC, silver DESC, bronze DESC)
AS T
WHERE ROWNUM <5;
```

Result :
'URS'
'GDR'
'USA'
'KOR'

GROUPBY_NUM() Function

Description

The **GROUPBY_NUM()** function works the same as the **ROWNUM** or **INST_NUM()** function, but it is used in the **GROUP BY**...**HAVING** clause. Whereas **INST_NUM()** is a scalar function, **GROUPBY_NUM()** is kind of an aggregate function. If the **GROUP BY** clause is included in the **SELECT** statement, the results of the **GROUPBY_NUM()** function specified in the **HAVING** clause is calculated after the query results are grouped.

Syntax

```
GROUPBY_NUM()
```

ORDERBY_NUM() Function

Description

The **ORDERBY_NUM()** function works the same as the **ROWNUM** or **INST_NUM()** function, but it is used in the **ORDER BY** clause. If the **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. 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 serial number of the result tuples.

Syntax

```
ORDERBY_NUM()
```

Example

The following is an example of searching athlete names ranked 3rd to 5th and their records in the history table.

```
SELECT athlete, score FROM history
ORDER BY score FOR ORDERBY_NUM() BETWEEN AND 5;
```

Result :
Luo Xuejuan 01:07.0
Rodal Vebjorn 01:43.0
Thorpe lan 01:45.0

SELECT Clause

Description

The **SELECT** statement specifies columns that you want to retrieve from a table.

Syntax

```
SELECT [ qualifier ] select_expression [ { TO | INTO }
variable [ {, variable } ] ]

qualifier :
ALL
DISTINCT
UNIQUE

select_expression :
*
table_name. *
expression [ {, expression}...]

variable :
[:] identifier
```

- *qualifier* : A qualifier. It can be omitted. When omitted, it is set to **ALL**.
  - **ALL** : Retrieves all instances of the table.
  - **DISTINCT** : Retrieves only instances with unique values without allowing duplicates.
  - **UNIQUE** : Like **DISTINCT**, retrieves only instances 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.
  - *table_name*. * : Specifying the table name with * works the same as specifying all the columns from the given table.
  - *expression* [{, *expression*}...] : *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.
    As **AVG**, **COUNT**, **MAX**, **MIN**, or **SUM**, an aggregate function that manipulates the retrieved data can also be used in the *expression*. When an aggregate function is used as the *expression*, but not used together with the **GROUP BY** clause, all elements of the retrieval list must be the aggregate function.
- *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

- **Example 1**

  The following is an example of retrieving host countries of the Olympic Games without any duplicates. This example generates, as the result, a list with no duplicated **host_nation** values from the **olympic** table.
  The **DISTINCT** or **UNIQUE** keyword allows only unique values in the query result set. For example, when there are multiple **olympic** instances 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 of retrieving all information on the Olympic Games that have been held. To retrieve all the columns, you can use the asterisk (*) instead of the list of column names. In the query result, the column values are displayed in the order defined in the table.

```
SELECT * FROM olympic;

=== <Result of SELECT Command in Line 1> ===
    host_year  host_nation              host_city     opening_date  closing_d
ate  mascot      slogan                            introduction
============================================================================
================================================
        1988  'Korea'                    'Seoul'         09/17/1988    10/02/
1988  'HODORI'    'Harmony and progress'             'The 1988 Seoul Games
we're the first Olympics to allow professional athletes to compete in certain
events…
         1992 'Spain'                    'Barcelona
'    07/25/1992    08/09/1992   'Cobi'     'Friends Forever
'                 'For the first time in decades, no nations boycotted the
1992 Barcelona Games…
         1996 'United States of America' 'Atlanta
'      07/19/1996    08/09/1996   'Izzy'     'The Celebration of the Century
'  'The 1996 Atlanta Games celebrated 100 years of the Modern Olympic Games.
...
25 rows selected.
```

FROM Clause

- General
- Derived Table
- Subquery Derived Table

General

Description

The **FROM** clause specifies the table in which data is to be retrieved in the query. Retrieval paths are as follows:

• Single table

• Subquery

• Derived table

Syntax

```
FROM from_specification

from_specification :
table_spec [ {, table_spec | outerjoin_table_spec }...]

table_spec :
single_table_spec [ correlation ] [ WITH (lock_hint [ {, lock_hint } ] ) ]
( single _table_spec [ {, single_table_spec}...] )
CLASS table_name [ correlation ]
subquery correlation
```

```
TABLE (expression) correlation

outerjoin_table_spec :
{ LEFT | RIGHT } [OUTER ] JOIN table_spec ON search_condition

single_table_spec :
[ ONLY ] table_name
ALL table_name [ ( EXCEPT table_spec ) ]

correlation :
[ AS ] identifier

lock_hint :
READ UNCOMMITTED
```

- *table_spec* : To retrieve class attributes from a table, you must use the keyword **CLASS** followed by the table name in 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.

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.

Syntax

```
FROM (subquery) [ AS ] derived_table_name [( column_name [ {,
column_name }_ ] )]
```

A derived table created form a subquery can have any number of columns and instances.

However, the number of *column_name* and the number of columns created by the *subquery* must be identical.

Example

- **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

- General
- BETWEEN Predicate
- Comparison Predicate
- EXISTS Predicate
- IN Predicate
- LIKE Predicate
- NULL Predicate
- Quantifier

General

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
in_predicate
like_predicate
null_predicate
quantified predicate
exists_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.)

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.

- *search_condition* : It is described in detail in the following sections.

BETWEEN Predicate

Description

The **BETWEEN** predicate allows you to select instances of a table that fall within a range of values or outside a range of values. The column evaluated by **BETWEEN** must be default data types which are single values.

Syntax

```
expression [ NOT ] BETWEEN expression AND expression
```

- *expression* : *expression* can be a column name, path expression, constant value, an arithmetic expression or aggregate function. Character string expressions are evaluated alphabetically. For example, a predicate **i BETWEEN g AND m** is equivalent to the compound predicates **i >= g AND i <= m**.

Example

The following is an example of selecting the name, gender and number of players of sports whose number of players is between 5 and 10. In this query, 20 rows are selected in the **event** table. The **players** values of selected rows fall between the specified range.

```
SELECT name, gender, players FROM event WHERE players BETWEEN 5 AND 10;
=== <Result of SELECT Command in Line 2> ===
  name                  gender                players
============================================================
  'Synchronized Team'   'W'                          8
  'Indoor'              'M'                          6
  'Indoor'              'W'                          6
  'Water Polo'          'M'                          7
  'Water Polo'          'W'                          7
...
20 rows selected.
```

You can use **NOT** in the **BETWEEN** clause. If you use **NOT BETWEEN**, data outside the specified range are selected. The **players** value in the **event** table is exactly 5 or 10 is not included in the query results of the **NOT BETWEEN** clause.

```
SELECT name, gender, players FROM event WHERE players NOT BETWEEN 5 AND 10;
=== <Result of SELECT Command in Line 2> ===
  name                  gender                players
============================================================
  'Freestyle 48kg'      'W'                          1
  'Freestyle -48kg'     'M'                          1
```

```
  'Freestyle 52kg'      'M'                             1
  'Freestyle -52kg'     'M'                             1
  'Freestyle 54kg'      'M'                             1
 ...
402 rows selected.
```

Comparison Predicate

Description

A *comparison_predicate* evaluates an expression by comparing it against another or subquery. The first expression is often a column name or path expression that is compared against a second expression. The following table shows comparison operators that can be used in *comparison_predicate*.

Comparison Operator	Meaning
=	Equal to
<>	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Comparison against floating point and double precision data types should be regarded as approximate. Because of the conversion that occurs internal to your system, the value you enter may be altered during query manipulation and argument passing. This loss of precision affects the actual value stored of float point or double precision number, even though the output value may look like the desired value.

Example

- **Example 1**

  The following is an example of selecting the years and host cities of Olympic Games held in the United States of America. The following query selects **host_year**, **host_nation** and **host_city** from the **olympic** table. The predicate in the **WHERE** clause selects data for which **host_nation** is 'United States of America.' Comparisons on string values are case-sensitive. Strings must be enclosed in single quotes (' '). For more information on character strings, see the "Data Types" section.

```
SELECT host_year, host_nation, host_city FROM olympic WHERE
host_nation='United States of America';
=== <Result of SELECT Command in Line 2> ===
    host_year  host_nation           host_city
============================================================
         1996  'United States of America'  'Atlanta '
         1904  'United States of America'  'St. Louis'
         1984  'United States of America'  'Los Angeles'
         1932  'United States of America'  'Los Angeles'
4 rows selected.
```

- **Example 2**

  The following is an example of selecting the name, gender and number of players of sports for which the number of players is greater than 4 and less than 10. The following query selects the **name**, **gender** and **players** columns from the **event** table. The comparison predicates and logical operators in the **WHERE** clause specify the range of the **players** value to retrieve.

```
SELECT sports, name, gender, players FROM event WHERE players > 4 AND players
< 10;
=== <Result of SELECT Command in Line 1> ===
  sports               name                 gender                  playe
rs
```

```
========================================================================
==
'Swimming'            'Synchronized
Team'          'W'                        8
  'Volleyball'        'Indoor'                    'M'
 6
  'Volleyball'        'Indoor'                    'W'
 6
  'Water Polo'        'Water
Polo'                 'M'                       7
  'Water Polo'        'Water
Polo'                 'W'                       7
...
20 rows selected.
```

EXISTS Predicate

Description

The **EXISTS** predicate is used to determine whether the result of a subquery is an empty set.

Syntax

```
EXISTS expression
```

- *expression* : If a subquery is specified in *expression*, the result of the **SELECT** statement in the subquery is evaluated. If the subquery does not produce any results, then **EXISTS** is false.

Example

The following is an example of selecting the names of the countries that participated in the 1988 Olympic Games, and their continents. The **EXISTS** predicate is used to determine whether a certain value exists in the results of a subquery. If any countries participated in the Olympic Games, the results of the subquery exist; data for 156 countries and their continents is outputted.

```
SELECT n.name, n.continent FROM nation n
WHERE EXISTS (SELECT * FROM participant p WHERE p.host_year=1988 AND
p.nation_code=n.code);
=== <Result of SELECT Command in Line 2> ===
  name                  continent
========================================
  'Somalia'             'Africa'
  'Sri Lanka'           'Asia'
  'Sudan'               'Africa'
  'Switzerland'         'Europe'
  'Surinam'             'Americas'
...
156 rows selected.
```

IN Predicate

Description

The **IN** predicate compares a single value against a set, multiset or sequence set of values.

Syntax

```
expression [ NOT ] IN expression
```

- *expression* (left) : A single-value column, path expression, constant value, or an arithmetic function that produces a single value.
- *expression* (right) : A list of values to compare. The expression can be a subquery. The list can be an explicit list of constant values entered within braces or parentheses. In the **IN** predicate, only constant values enclosed in parentheses (()) are purpose of ANSI compliance.

Example

- **Example 1**

  The following is an example of selecting the name, gender and number of players of sports for which the number of players is 5 or 6. The following query selects the data whose column value is 5 or 6 in the **event** table. The data from the **event** table that have one of the values specified in the query condition are retrieved as the result.

```
SELECT name, gender, players FROM event WHERE players IN (5, 6);
=== <Result of SELECT Command in Line 2> ===
  name                  gender                      players
============================================================
  'Indoor'              'M'                             6
  'Indoor'              'W'                             6
  'Hockey'              'M'                             6
  'Hockey'              'W'                             6
  'Rhythmic Group Competition'  'W'                     6
  'Rhythmic Team'       'M'                             6
  'Rhythmic Team'       'W'                             6
  'Basketball'          'M'                             5
  'Basketball'          'W'                             5
9 rows selected.
```

- **Example 2**

  The following is an example of selecting the names of the countries that won one or more gold medals, and their continents. In the following query, a subquery is specified in the **IN** predicate. The subquery selects the **nation_code** from the **participant** table and then selects the data that have the **code** value included in the result of the subquery from the **nation** table.

```
SELECT continent, name FROM nation WHERE code IN (SELECT nation_code FROM
participant WHERE gold>0);
=== <Result of SELECT Command in Line 2> ===
  continent             name
============================================
  'Africa'              'Algeria'
  'Americas'            'Argentina'
  'Europe'              'Armenia'
  'Oceania'             'Australia'
  'Europe'              'Austria'...
81 rows selected.
```

LIKE Predicate

Description

The **LIKE** predicate retrieves the data that matches a specified text pattern from a table. Comparison using the **LIKE** predicate is supported only with columns that have **STRING** or **CHAR** domains. The **LIKE** predicate does not support **NCHAR** or **BIT** domains.

Syntax

```
expression [ NOT ] LIKE expression [ ESCAPE char]
```

- *expression* (left) : The first *expression* represents a column that has a domain of **STRING** or **CHAR**(n) type. Pattern matching starts from the first character of the column value.
- *expression* (right) : The second *expression* to be retrieved can be a string consisting of the following characters:
  - The symbol "_" (underscore) in the string pattern represents any single character.
  - The symbol "%" (percent) in the string pattern represents any sequence of characters, which may be an empty string.
  - All other characters represent themselves.
  - In the **LIKE** predicate, the default values of escape characters do not exist.

- **ESCAPE** *char* : If the string pattern to be retrieved includes an actual "_" or "%" character, **ESCAPE** must be specified. That is, an **ESCAPE** character must precede the "_" or "%" character. For example, if you retrieve '10%', then an **ESCAPE** character such as "\" (backlash) can be used, the string would be actually represented as '10\%'.

Example

The following is an example of selecting the nation codes and names of the countries that include 'K' in their nation codes. This query selects **code**s whose string pattern is '%K_' from the **nation** table. In this case, the first character can be any string, the second character must be 'K', and the last character must be any single characters. Here, the **code** consists of three characters. Therefore, as shown in the result, only **code**s that include 'K' as the second character are retrieved.

```
SELECT code, name FROM nation WHERE code LIKE '%K_';
=== <Result of SELECT Command in Line 1> ===
  code                  name
==========================================
  'TKM'                 'Turmenistan'
  'UKR'                 'Ukraine'
  'SKN'                 'Saint Kitts & Nevis'
  'MKD'                 'Former Yugoslav Republic of Macedonia'
  'HKG'                 'Hong Kong'
5 rows selected.
```

NULL Predicate

Description

The **NULL** predicate determines whether columns in a table have a value specified as **NULL**. You can also select those columns that have values by using the **NOT NULL** keyword in the predicate.

Syntax

```
expression IS [ NOT ] NULL
```

- *expression* : It refers to the columns in the table that you want to check for **NULL** or **NOT NULL**.

Example

The following is an example of selecting the year, host country, host city, mascot and slogan of all Olympic Games that had a mascot. The following query selects specified columns from the **olympic** table for instances in which the **mascot** column values are not **NULL**.

```
SELECT host_year, host_nation, host_city, mascot, slogan FROM olympic WHERE
mascot is NOT NULL;
=== <Result of SELECT Command in Line 1> ===
    host_year  host_nation               host_city             mascot
        slogan
================================================================================
========================
         1988  'Korea'                   'Seoul'               'HODORI'
         'Harmony and progress'
         1992  'Spain'                   'Barcelona
'          'Cobi'               'Friends Forever '
         1996  'United States of America' 'Atlanta
'            'Izzy'               'The Celebration of the Century '
         2000  'Australia'               'Sydney '             'Olly Syd
Millie '    'Share the Spirit'
         1976  'Canada'                  'Montreal'            'Amik'
         NULL
         1972  'Germany'                 'Munich'              'Waldi'
         NULL
         1980  'U.S.S.R.'                'Moscow'              'Misha'
         NULL
         1984  'United States of America' 'Los
Angeles'         'Sam'                'Play part in History'
```

```
         2004  'Greece'                  'Athens
'             'Athena  Phevos'      'Welcome Home'
...
25 rows selected.
```

Quantifier

Description

The **ALL** quantifier compares a single value against every value returned by another expression. The **SOME** quantifier compares a single value against each value returned by another expression. The single value must compare at least one of the values returned by the expression. The keyword **ANY** can be used instead of keyoword **SOME** which returns the same query results.

Syntax

```
expression comp_op ALL expression
expression comp_op SOME expression
```

- *comp_op* : A comparison operator >, <, =, >= or <= can be used.
- *expression* (right) : The second *expression* can be a column name, path expression, a list (set) of constant values, a subquery or an arithmetic operator. If a subquery is used as the second *expression* in a query, each result produced by the subquery is compared against a single value associated with the first *expression*.

Example

- **Example 1**

  The following is an example of an **ALL** quantifier. In this example, the **gold** column in the **participant** table is compared against each value of the **silver** column in the **participant** table. The query result shows that the **gold** column value of one instance of the **participant** table is greater than the **silver** column values of all instances of the **participant** table.

```
SELECT nation_code, gold, silver FROM participant
WHERE gold > ALL (SELECT silver FROM participant) AND host_year=1992;
=== <Result of SELECT Command in Line 1> ===
  nation_code                  gold       silver
============================================
  'EUN'                          45           38
1 rows selected.
```

- **Example 2**

  The following is an example of a **SOME** quantifier. The **gold** column in the **participant** table is compared against each value of the **silver** column in the **participant** table. The query result shows that the **gold** column values for 37 instances of the **participant** table are greater than at least one of the **silver** column values in the **participant** table.

```
SELECT nation_code, gold, silver FROM participant
WHERE gold > SOME(SELECT silver FROM participant) AND host_year=1992;
=== <Result of SELECT Command in Line 1> ===
  nation_code                  gold       silver
============================================
  'ITA'                          6            5
  'GBR'                          5            3
  'ROU'                          4            6
  'TCH'                          4            2
  'PRK'                          4            0
...
37 rows selected.
```

GROUP BY HAVING Clause

Description

The **GROUP BY** clause is used when more than one aggregate functions are given in the **SELECT** statement.

Query results are formed into a group. By using the **HAVING** clause, you can set the condition so that only groups satisfying the specified condition will be retrieved instead of all groups created by the **GROUP BY** clause.

Syntax

```
GROUP BY expression_list [ HAVING search_condition]
```

- *expression_list* : The *expression_list* consists of one or more column names or expressions. Each field must be separated by a comma (,).
- *search_condition* : The query result groups can be limited by *search_condition* specified in the **HAVING** option. Even when multiple groups are created as the result of the query, you can set the condition so that only groups satisfying a specific logical expression will be displayed.

Example

- **Example 1**

  The following is an example of retrieving the number of gold medals in each Olympic Games since 1988. In the **participant** table, the sum of **gold** is calculated in each **host_year**. In this example, the **participant** table's instance has 5 different **host_year** values. Therefore, 5 groups are created, and for each **host_year** value group, **sum(gold)** value is calculated.

```
SELECT host_year, SUM(gold) FROM participant WHERE host_year >= 1988 GROUP BY
host_year;
=== <Result of SELECT Command in Line 2> ===
    host_year    sum(gold)
=========================
         1988          241
         1992          260
         1996          271
         2000          298
         2004          301
5 rows selected.
```

- **Example 2**

  The following is an example of grouping the athletes that participated in the Olympic Games by the nation and retrieving the nation codes and the number of athletes of countries with more than 200 participants. You can retrieve the instances satisfying the specified conditions by using the **GROUP BY** clause in combination with the **HAVING** clause. In the example above, only instances whose **count(code)** is greater than 200 are retrieved, grouping by **nation_code**. Whereas **WHERE** is the conditional clause of **SELECT**, **HAVING** is the conditional clause of **GROUP BY**.

  In a **GROUP BY** clause, expressions as well as columns can be listed. In this case, the expressions can be composed of columns, functions and constant values.

```
SELECT nation_code, COUNT(code) FROM athlete GROUP BY nation_code HAVING
COUNT(code) > 200;
=== <Result of SELECT Command in Line 2> ==">
  nation_code           count(code)
===================================
  'AUS'                         355
  'CHN'                         296
  'CUB'                         203
  'GER'                         417
  'ITA'                         226
  'JPN'                         205
  'KOR'                         263
  'NED'                         214
  'RUS'                         369
```

```
  'USA'                          810
10 rows selected.
```

ORDER BY Clause

Description

The **ORDER BY** clause sorts the query result set in ascending or descending order. The **ORDER BY** clause must be placed at the end of the query statement. If the **ORDER BY** clause is not specified, the order of instances to be queried may vary depending on query.

Syntax

```
ORDER BY sort_spec [ {, sort_spec}  ]
sort_spec :
integer_literal [ ASC | DESC ]
expression [ ASC | DESC ]
alias [ ASC | DESC ]
```

- *sort_spec* : The *sort_spec* can be one or more column names, expressions, aliases or integer values.
  - *integer_literal* [**ASC**| **DESC**] : An integer value *integer_literal* is used as a reference to the position of the column or expression in the **SELECT** clause.
  - *expression* [**ASC**| **DESC**]
  - *alias* [**ASC**| **DESC**]

Each field in the **ORDER BY** clause must be separated by a comma (,). If the keyword **DESC** is not specified after *expression* or *integer_literal* in the **ORDER BY** clause, the query result is returned in ascending order.

Example

- **Example 1**

  The following is an example of retrieving the years and host countries of the Olympic Games and sorting the results by host country in ascending order. The results are sorted by the column after the **ORDER BY** clause in ascending order. To sort in descending order, place **DESC** after the column.

```
SELECT host_year, host_nation FROM olympic ORDER BY host_nation;
=== <Result of SELECT Command in Line 2> ===
    host_year  host_nation
==================================
         2000  'Australia'
         1956  'Australia'
         1920  'Belgium'
         1976  'Canada'
         1952  'Finland'
...
 25 rows selected.
```

- **Example 2**

  The following example of retrieving the countries which won between 10 and 15 gold medals. If the numbers of gold medals are the same, the results are sorted by nation code in ascending order. The query statement below shows how integer values are used in the **ORDER BY** clause. The integer value corresponding to the position of the column or expression used in the **SELECT** statement is used to determine the sorting order of the query results. Multiple columns are displayed below to indicate the sorting order.

```
SELECT host_year, nation_code, gold FROM participant WHERE gold BETWEEN 10
AND 15
ORDER BY 3 DESC, 2;
 === <Result of SELECT Command in Line 1> ===
    host_year  nation_code                gold
============================================
         1996  'FRA'                        15
         1992  'CUB'                        14
```

```
        1992  'ESP'                          13
        2000  'FRA'                          13
        2000  'GER'                          13
        2004  'GER'                          13
...
20 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 query outputs a result based on the values in the common columns of two tables to be joined. If the joined result of the two tables meets all join conditions, it is called an **inner join** or simply a **join**.

The query result of an **outer join** can include all instances from the left table (a left outer join), all instances of the right table (a right outer join), or both (a full outer join). If the table has no instance that meets the join conditions, the corresponding column in the result will have a **NULL** value.

Syntax

```
FROM table_specification [{, table_specification |
qualified_join_table_specification}...]

table_specification :
table_specification [ correlation ]
CLASS table_name [ correlation ]
subquery correlation
TABLE (expression) correlation

qualified_join_table_specification :
[ INNER | {LEFT | RIGHT} [ OUTER ] ] JOIN table_specification join_condition

join_condition :
ON search_condition
```

- *qualified_join_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.

Example

- **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'
21 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 instances. 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 instance that has a single column. If no instance is returned by the subquery, the subquery expression has a **NULL** value. If the subquery is supposed to return more than one instances, 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 instances 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 instances 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** values for **nation** instances, as well as a set that contains **host_city** values of the **olympic** instances with **host_nation** values. If the **name** value is an empty set in the query result, it is excluded. If there is no **olympic** instance 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 **SELECT ... 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
1	NULL	KIM
2	NULL	Moy
3	1	Jonas
4	1	Simth
5	2	Verma
6	2	Foster
7	6	Brown

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));

INSERT INTO tree VALUES (1,NULL,'Kim');
INSERT INTO tree VALUES (2,NULL,'Moy');
INSERT INTO tree VALUES (3,1,'Jonas');
INSERT INTO tree VALUES (4,1,'Smith');
INSERT INTO tree VALUES (5,2,'Verma');
INSERT INTO tree VALUES (6,2,'Foster');
INSERT INTO tree VALUES (7,6,'Brown');

-- 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 *
    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 *
    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 Joins

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

Description

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; 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
```

Operator 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

To better understand this, let us consider the following scenario:
We have one parent node (named "**R**") which has two children, **C1** and **C2**. Each of these children has three children of it's own, **C1_1**, **C1_2**, and **C1_3** and **C2_1** and **C2_2**, **C2_3**. If we don't specify **ORDER BY**, the rows are obtained in depth-first order but without further ordering. Let's say we want them sorted. Assuming that we sort the rows with a simple **ORDER BY** we obtain the following order:

ROWS	COLUMN VALUES	LEVEL
R(parent node)	A	Level 1
C1	B	Level 2
C1_1	G	Level 3
C1_2	F	Level 3

C1_3	H	Level 3
C2	D	Level 2
C2_1	C	Level 3
C2_2	E	Level 3
C2_3	I	Level 3

We have correctly ordered them but the hierarchical ordering is broken. The **ORDER SIBLINGS BY** clause will cause the order of the row to be as displayed below.

ROWS	COLUMN VALUES	LEVEL
R	A	Level 1
C1	B	Level 2
C2_1	C	Level 3
C2	D	Level 2
C2_2	E	Level 3
C1_2	F	Level 3
C1_1	G	Level 3
C1_3	H	Level 3
C2_3	I	Level 3

Basically, the tree is displayed vertically with the child nodes of a parent sorted according to the column list found in the clause. All siblings are in successive rows and sorted according to the desired criteria.

Example 2

The following is an example of sorting rows which share a parent.

```
--Displaying siblings in successive rows
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
```

Using Scenarios - Hierarchical Structures and Efficiency Details

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.

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_spec ) ]
```

- **ALL CLASSES** : If the **ALL CLASSES** keyword is specified, the statistics on all the tables existing in the database are updated.

Using SQL Hints

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 1

```
SELECT /*+ hint [ { hint } ... ] */
or
SELECT --+ hint [ { hint } ... ]
or
SELECT //+ hint [ { hint } ... ]

hint :
 USE_NL[(spec-name[{, spec-name}...])]
 USE_IDX[(spec-name[{, spec-name}...])]
 USE_MERGE[(spec-name[{, spec-name}...])]
 ORDERED
```

Syntax 2

```
CREATE /*+ NO_STATS */ [TABLE | CLASS] ...;
ALTER /*+ NO_STATS */ [TABLE | CLASS] ...;

CREATE /*+ NO_STATS */ INDEX ...;
ALTER /*+ NO_STATS */ INDEX ...;
DROP /*+ NO_STATS */ INDEX ...;
```

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.

Hint	Description

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 Plans

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 "Viewing Query Execution Plans."

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.
```

Table Manipulation

Creating Tables

CREATE TABLE (Syntax)

Description

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

Syntax

```
CREATE { CLASS | TABLE } table_name
[ {UNDER | AS SUBCLASS OF } super_class_name [ {, super_class_name }_ ] ]
[ TABLE ATTRIBUTE
( table_attr_definition_list ) ]
[ ( attr_definition | table_constraint
[ { , attr_definition | table_constraint_definition }_ ] ) ]
[ METHOD method_definition_list ] ]
[ FILE path_name_list ] ]
[ INHERIT resolution_list [ {, resolution_list }_ ] ] [ ; ]
super_class_name :
[ user_name.]classname
table_attr_definition :
attribute_name datatype { [ DEFAULT value ] | [ NOT NULL ] }
attr_definition :
attribute_name datatype [ { SHARED [ value ] | DEFAULT value } ]
[AUTO_INCREMENT ...] [ constraints ]
constraints :
NOT NULL
UNIQUE
PRIMARY KEY
FOREIGN KEY ...
table_constraint_definition :
[ CONSTRAINT constraint_name ] {[ UNIQUE | PRIMARY KEY | FOREIGN KEY ]}
(attribute_name [{, attribute_name} ...])
method_definition :
[ CLASS ] method_name [ ( [ arg_type_list ] ) ]
[ result_type ] [ FUNCTION method_implementation_name ]
arg_type :
datatype
result_type :
datatype
resolution_list :
attr_mthd_name OF super_class_name [ AS alias ]
```

- *table_name* : Specifies the name of the table to be created.
- *attr_definition* :
  - *attribute_name* : Specifies the name of the column to be created.
  - *datatype* : Specifies the data type of the column.
  - [**SHARED** [*value*] | **DEFAULT** *value*] : Specifies the initial value of the column (see "Columns" for more information).
  - *constraints* : Specifies the constraint of the column. Available constraints are **NOT NULL**, **UNIQUE**, **PRIMARY KEY** and **FOREIGN KEY** (see "Constraints" for more information).

Example

The following is an example of creating the olympic table in the olympic database provided as a demo.

```
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

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

- Column Name
- Setting the Column Initial Value (SHARED, DEFAULT)
- Auto Increment

Column Name

Description

A column name is a word consisting of alphabets including '_', '#' and '%'. It is recommended that a column name include a word representing the domain.

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

The initial value of a column can be set when it is created by using the **SHARED** or **DEFAULT** clause.

- If the **SHARED** clause is used, all instances in the table will have the same value.
- If the **DEFAULT** clause is used, the value specified in the **DEFAULT** clause is saved when a new instance is created even without specifying the value.

Example

- **Example 1**

  The following is an example of creating the man table that contains the information about male athletes. The gender column is specified as **SHARED**.

  ```
  CREATE TABLE man ( name VARCHAR(40), gender CHAR(1) SHARED );
  ```

- **Example 2**

  The following is an example of inserting athletes AAA and BBB. Because the gender column is defined as **SHARED**, all instances share the value. You can see that the gender of the athlete BBB is 'M' even though you did not enter the value.

  ```
  INSERT INTO man VALUES ( 'AAA','M');
  INSERT INTO man (name) VALUES ('BBB');
  csql> SELECT * FROM man;
  csql> ;x
  === <Result of SELECT Command in Line 1> ===
    name                  gender
  ============================================
  ```

```
  'AAA'                 'M'
  'BBB'                 'M'
2 rows selected.
```

- **Example 3**

  The following is an example of creating the woman table that contains the information about female athletes. The default value of gender is specified as W. Even when the value of the gender is not inserted, the default value 'W' is entered.

```
CREATE TABLE woman ( name VARCHAR(40), gender CHAR(1) DEFAULT 'W' );
INSERT INTO woman (name) VALUES ('AAA');
csql> SELECT * FROM woman;
csql> ;x
=== <Result of SELECT Command in Line 6> ===
  name                  gender
=========================================
  'AAA'                 'W'
1 rows selected.
```

Caution

SHARED and **DEFAULT** values can be modified by using the **ALTER CLASS** statement.

Auto Increment

Description

The auto increment is created by increasing values of a numeric column. It is defined by specifying the initial and increment values.

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

- **Example 1**

  The following is the schema of the athlete table where the auto increment is used.

```
CREATE TABLE athlete (
   code             INTEGER AUTO_INCREMENT(16693, 1) PRIMARY KEY,
   name             VARCHAR(40) NOT NULL,
   gender           CHAR(1)     ,
   nation_code   CHAR(3)     ,
   event            VARCHAR(30)
);
```

- **Example 2**

  The following is an example of inserting the information about the athletes into the athlete table. When you insert values using auto increment, you must specify accurately the names of columns (except for the auto increment columns) after the table name.

```
INSERT INTO athlete (name, gender, nation_code, event) VALUES ('Jang Mi-Ran',
'W', 'KOR', 'Weightlifting');
```

Caution

- Auto increment can be defined only for **SMALLINT**, **INTEGER**, **DECIMAL**(p,0) and **NUMERIC**(p,0) domains.
- **DEFAULT**, **SHARED** clause.

- Even if a column has auto increment, the **UNIQUE** constraint is not satisfied.
- You can insert, modify or delete values even when auto increment is enabled. That is, the value specified by users has a higher priority than auto increment. It is recommended to use auto increment only when users know the previous value deleted and specify data with the value.
- Auto increment is not supported in some statements such as **INSERT INTO** <table name> **VALUES**(...) where columns are not defined. In such cases, you must define all columns except for the auto increment columns in the Insert statement to use auto increment. 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 tb1(A smallint auto_increment, B char(5));
```

Constraints

- NOT NULL Constraint
- UNIQUE Constraint
- PRIMARY KEY Constraint
- FOREIGN KEY Constraint

NOT NULL Constraint

Description

The **NOT NULL** constraint enforces a column to always contain a value other than **NULL**. Any column can have a **NOT NULL** constraint on the data. An error occurs if a **NULL** value is inserted or updated by using the **INSERT** or **UPDATE** statement.

Example

The following is an example of specifying the name column of the manager2 table as **NOT NULL**.

```
CREATE TABLE manager2
(name VARCHAR(40) NOT NULL,
event VARCHAR(50));
```

UNIQUE Constraint

Description

The **UNIQUE** constraint enforces a column to have a unique value. You can place a **UNIQUE** constraint on either a column or a set of 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.

```
CREATE TABLE "test"(
a INTEGER,
b INTEGER,
UNIQUE(a,b))
INSERT INTO "test" VALUES(1, 2);
INSERT INTO "test" VALUES(1, 3);
```

```
-- Succeeds because the value of column b is unique
INSERT INTO "test" VALUES(1, 2);
-- error because the values of columns a and b are the same as those in the
first statement
```

PRIMARY KEY Constraint

Description

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

Example

- **Example 1**

  As an example, take a look at the schema of the nation table provided in the olympic database as a demo. The code column is set as a single primary key.

```
CREATE TABLE nation (
   code             CHAR(3)     NOT NULL PRIMARY KEY,
   name             VARCHAR(40) NOT NULL,
   continent        VARCHAR(10) ,
   capital          VARCHAR(30)
);
```

- **Example 2**

  The record table is an example that has a 4-column composite key.

```
CREATE TABLE record (
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

```
CREATE { TABLE | CLASS } table_name
[ {UNDER | AS SUBCLASS OF } super_class_name [ {, super_class_name }_ ] ] ...
[( column_definition [ { , column_definition }...; ]
| unique_constraint
| referential_constraint
)]

referential_constraint:
[FOREIGN KEY [constraint-name] (attribute_name[{, attribute_name }...; ])
REFERENCES referenced_class_name (attribute_name [{ , attribute_name }...; ])
[referential_triggered_action] ]

referential_triggered_action:
update_rule [ delete_rule [ cache_object_rule ] ]
```

```
update_rule:
ON UPDATE referential_action

delete_rule:
ON DELETE referential_action

cache_object_rule:
ON CACHE OBJECT cache_object_column_name

referential_action:
CASCADE
| RESTRICT
| NO ACTION
```

- *table_name* : Specifies the name of the table to be created.
- *referential_constraint* :
  - *constraint-name* : Specifies the name of the constraint. If omitted, it is automatically specified by CUBRID.
  - *attribute_name* : Specifies the name of the referencing foreign key. There is no limit to the number of foreign keys defined (the number of attributes identified), but the number must be identical to that of primary keys.
  - *referenced_table_name* : Specifies the name of the table to be referenced.
  - *attribute_name* : Specifies the name of the primary key to be referenced.
- *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** or **RESTRICT** option. The default is **RESTRICT**.
  - **RESTRICT** : Prevents the primary key from being changed.
  - **NO ACTION** : Does not change the foreign key even when the primary key is changed.
- **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** or **CASCADE** option. The default is **RESTRICT**.
  - **CASCADE** : Specifies that all instances containing the referencing foreign key will be deleted when the primary key is deleted.
  - **RESTRICT** :Prevents the primary key from being deleted.
  - **NO ACTION** : Does not change the foreign key even when the primary key is deleted.
- **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.

Example

The following is an example of showing the foreign key reference relationship in the olympic table provided as a demo. When creating the nation and olympic tables, specify that each table's primary key will be referenced by the foreign key in the participant table.

```
CREATE TABLE nation (
   code             CHAR(3)     NOT NULL PRIMARY KEY,
  ....
);
```

```
CREATE TABLE olympic (
   host_year        INT    NOT NULL PRIMARY KEY,
   ....
);

CREATE TABLE participant (
   host_year        INT    NOT NULL ,
   nation_code      CHAR(3)     NOT NULL ,
   ....
   PRIMARY KEY(host_year, nation_code)   ,
   FOREIGN KEY(host_year) REFERENCES olympic(host_year),
   FOREIGN KEY(nation_code) REFERENCES nation(code)
);
```

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.

Changing Tables

Changing Table Names

Description

You can change the table name by using the **RENAME** statement.

Syntax

```
RENAME  [ TABLE | CLASS | VIEW | VCLASS] old_talbe_name AS new_table_name
[ ; ]
```

- *old_table_name* : Specifies the old table name to be changed.
- *new_table_name* : Specifies a new table name.

Example

The following is an example of renaming the stadium table to stadium_info.

```
RENAME TABLE stadium AS stadium_info;
```

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).

Altering Columns and Constraints

You can add, alter, or drop columns or constraints by using the **ALTER** statement.

Adding Columns and Constraints

Description

You can add a column or constraint by using the **ADD** [**ATTRIBUTE** | **COLUMN**] in the **ALTER** statement.

Syntax

```
ALTER [ table_type ] table_name
ADD [ ATTRIBUTE | COLUMN ] attr_definition | table_constraint_definition
[ { , attr_definition | class_constraint }_ ]
```

```
[ INHERIT resolution [ { , resolution }_ ] ] [ ; ]

ALTER [ table_type ] table_name
ADD CLASS ATTRIBUTE table_attr_definition [ {, class_attr_definition }_ ]
[ INHERIT resolution [ { , resolution }_ ] ] [ ; ]

table_type:
[TABLE | CLASS]
[VIEW | VCLASS]

resolution:
[ TABLE ] attr_mthd_name OF superclass_name [ AS alias ]
table_constraint_definition :
[ CONSTRAINT constraint_name ] [ UNIQUE | PRIMARY KEY | FOREIGN KEY]
(attribute_name [{, attribute_name } ...])
```

- *table_type* : Specifies the type of the table (normal or virtual) to be modified.
- *table_name* : Specifies the name of the table to be added.
- *attr_definition* : Specifies the name and data type of the column to be added.
- *constraint_name* : Specifies the name of the constraint to be added.

Example

- **Example 1**

  The following example shows various ways to add a column.

```
ALTER TABLE athlete ADD COLUMN age INT;
ALTER TABLE athlete ADD COLUMN age INT DEFAULT 0 NOT NULL;
ALTER TABLE athlete ADD COLUMN retire CHAR(1) SHARED 'N';
ALTER TABLE athlete ADD COLUMN phone VARCHAR(13) DEFAULT '000-0000-0000';
```

- **Example 2**

  The following example shows various ways to add a constraint.

```
ALTER TABLE company ADD CONSTRAINT pk_id PRIMARY KEY (comp_id);
ALTER TABLE nation ADD CONSTRAINT u_name UNIQUE (name);
```

Changing Column Names

Description

You can change the column name by using the **RENAME** in the **ALTER** statement after a table is created.

Syntax

```
ALTER [ table_type ] table_name
RENAME rename_clause
[ INHERIT resolution [ {, resolution }_ ] ] [ ; ]

table_type:
[TABLE | CLASS]
[VIEW | VCLASS]

rename_clause:
[ ATTRIBUTE | COLUMN ]
[ CLASS ] old_attr_mthd_name AS new_attr_mthd_name
FUNCTION OF [ CLASS ] method_name AS method_implementation_name

FILE file_path_name AS file_path_name

resolution:
[ CLASS ] attr_mthd_name OF superclass_name [ AS alias ]
```

- *table_type* : Specifies the type of the table (normal or virtual) to be modified.
- *table_name* : Specifies the name of the table whose column is to be modified.

- *rename_clause* :
  - *old_attr_mthd_name* : Specifies the old column name to be changed.
  - *new_attr_mthd_name* : Specifies a new column name.

Example

The following is an example of altering the name of gender to gender_type in the athlete table.

```
ALTER TABLE athlete RENAME COLUMN gender AS gender_type;
```

Changing the Column Default

Description

You can specify or change the default of a column by using the **CHANGE** in the **ALTER** statement.

Syntax

```
ALTER [ table_type ] table_name
CHANGE [ CLASS ] attribute_name DEFAULT value
 [ { , [ CLASS ] attribute_name DEFAULT value }_ ] [ ; ]

table_type:
[TABLE | CLASS]
[VIEW | VCLASS]
```

- *table_type* : Specifies the type of the table (normal or virtual) to be changed.
- *table_name* : Specifies the name of the table whose column is to be changed.
- *attribute_name* : Specifies the name of the column to which apply the default.
- *value* : Specifies the default to be changed.

Example

The following is an example of changing the default of the gender column in the athlete table. In the beginning, there is no default value exists on the gender column. The first query sets the default of the gender column to 'M' and the second query changes its default to 'W'.

```
ALTER TABLE athlete CHANGE gender DEFAULT 'M';
ALTER TABLE athlete CHANGE gender DEFAULT 'W';
```

Dropping Columns and Constraints

Description

You can drop a column or constraint by using the **DROP** in the **ALTER** statement. Multiple columns or constraints could be dropped by a single **ALTER** statement.

Syntax

```
ALTER [ table_type ] table_name
DROP [ ATTRIBUTE | COLUMN ] [ CLASS ] attr_mthd_name
[ {, [ CLASS ] attr_mthd_name }_ ]
[ INHERIT resolution [ {, resolution }_ ] ] [ ; ]

ALTER [ table_type ] table_name
DROP CONSTRAINT constraint_name

table_type:
[TABLE | CLASS]
[VIEW | VCLASS]

resolution:
[ CLASS ] attr_mthd_name OF superclass_name [ AS alias ]
```

- *table_type* : Specifies the type of the table (normal or virtual) to be altered.

- *table_name* : Specifies the name of the table whose column is to be dropped.
- *attr_mthd_name* : Specifies the name of the column to be dropped.
- *constraint_name* : Specifies the name of the constraint to be dropped.

Example

- **Example 1**

  The following is an example of dropping the address and seats columns from the stadium table.

```
ALTER TABLE stadium DROP ATTRIBUTE address, seats;
```

- **Example 2**

  The following is an example of dropping the **PRIMARY KEY** constraint set on the code column of the stadium table.

```
ALTER TABLE stadium DROP CONSTRAINT k_stadium_code;
```

Dropping Tables

Description

You can drop an existing table by the **DROP** statement. Multiple tables can be dropped by a single **DROP** statement. All instances of table are also dropped.

Syntax

```
DROP [ table_type ] table_spec [ { ,table_spec }_ ] [ ; ]

table_type:
[TABLE | CLASS]
[VIEW | VCLASS]

table_spec:
single_table_spec
( single_table_spec [ { , single_class_spec }_ ] )

single_table_spec:
[ ONLY ] table_name
ALL table_name [ ( EXCEPT table_spec ) ]
```

- *table_type* : Specifies the type of the table (normal or virtual) to be dropped.
- *table_name* : Specifies the name of the table to be dropped.

Example

The following is an example of dropping the history table.

```
DROP TABLE history ;
```

Class Inheritance

Overview

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 **superclass** of the Manager class. Inheritance can simplify class creation by reusing the existing class hierarchy.

Inheritance

Description

CUBRID allows multiple inheritance, which means that a class can inherit attributes and methods from more than one superclass. However, inheritance can cause conflicts when an attribute or method of the superclass is added or deleted.

Such conflict occurs in multiple inheritance if there are attributes or methods with the same name in different superclasses. For example, if it is likely that a class inherits attributes of the same name and type from more than one superclass, you must specify the attributes to be inherited. In such a case, if the inherited superclass is deleted, a new attribute of the same name and type must be inherited from another superclass. 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 superclass, it is possible that their names are to be the same, while their domains are different. For example, two superclasses 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.

Syntax

```
CREATE { CLASS | TABLE } class_name
[ {UNDER | AS SUBCLASS OF } super_class_name [ {, super_class_name }_ ] ]
[ CLASS ATTRIBUTE
( class_attr_definition_list ) ]
[ ( attr_definition | class_constraint
[ { , attr_definition | class_constraint_definition }_ ] ) ]
[ METHOD method_definition_list ] ]
[ FILE path_name_list ] ]
[ INHERIT resolution_list [ {, resolution_list }_ ] ] [ ; ]

super_class_name:
[ user_name.]classname

class_attr_definition:
attribute_name datatype [ DEFAULT value ] [ NOT NULL ]

attr_definition:
attribute_name datatype [ { SHARED [ value ] | DEFAULT value } ]
[ constraints ]

constraints:
NOT NULL
UNIQUE

class_constraint_definition:
[ CONSTRAINT constraint_name ] UNIQUE
(attribute_name [{, attribute_name} ...])

method_definition:
[ CLASS ] method_name [ ( [ arg_type_list ] ) ]
[ result_type ] [ FUNCTION method_implementation_name ]

arg_type:
datatype

result_type:
datatype

resolution_list:
attr_mthd_name OF super_class_name [ AS alias ]
```

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 superclass 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.

A superclass name can begin with the user name so that the owner of the class can be easily identified.

For compatibility with relational DBMS, a **TABLE** keyword may be used instead of **CLASS** in the **CREATE** statement. The attribute has the same meaning as the column of a table in the relational DBMS.

Inheriting Attributes and Methods

Description

When a class is created as a subclass, the class inherits all attributes and methods of the superclass. 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:
attr_mthd_name OF super_class_name [ AS alias ]
```

For the *attr_mthd_name* in the **INHERIT** clause, specify the name of the attribute or method of the superclass to inherit. With the **ALIAS** clause, you can resolve a name conflict that occurs in multiple inheritance statements by inheriting a new name.

Class Attributes and Methods

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 superclass, 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 superclass 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 Rules 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 superclass, attributes are defined in the same order as in the **CREATE** statement (an ANSI standard).

- If there is one superclass, locally created attributes are placed after the superclass attributes. The order of the attributes inherited from the superclass follows the one defined during the superclass definition. For multiple inheritance, the order of the superclass attributes is determined by the order of the superclasses specified during the class definition.
- If more than one superclass inherits the same class, the attribute that exists in both superclasses is inherited to the subclass only once. At this time, if a conflict occurs, the attribute of the first superclass is inherited.
- If a name conflict occurs in more than one superclass, you can inherit only the ones you want from the superclass attributes by using the **INHERIT** clause in order to resolve the conflict.
- If the name of the superclass attribute is changed by the alias option of the **INHERIT** clause, its position is maintained.

Adding a Superclass

Description

To extend class inheritance, add a superclass to a class. A relationship between two classes is created when a superclass is added to an existing class. Adding a superclass does not mean adding a new class.

Syntax

```
ALTER [ class_type ] class_name
ADD SUPERCLASS [ user_name.]class_name [ { , [ user_name.]class_name }_ ]
[ INHERIT resolution [ {, resolution }_ ] ] [ ; ]

resolution:
[ CLASS ] attr_mthd_name OF superclass_name [ AS alias ]
```

For the first *class_name*, specify the name of the class where a superclass is to be added. Attributes and methods of the superclass can be inherited by using the syntax above.

Name conflicts can occur when adding a new superclass. If a name conflict cannot be resolved by the database system, attributes or methods to inherit from the superclass 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 superclass 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;
```

Dropping a Superclass

Description

Deleting a superclass from a class means removing the relationship between two classes. If a superclass is deleted from a class, it changes inheritance relationship of the classes as well as of all their subclasses.

Syntax

```
ALTER [ class_type ] class_name
DROP SUPERCLASS class_name [ { , class_name }_ ]
[ INHERIT resolution [ {, resolution }_ ] ] [ ; ]

resolution:
[ CLASS ] attr_mthd_name OF superclass_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 superclass to be deleted. If a name conflict occurs after deleting a superclass, see the Resolving Class Conflict section for the resolution.

Example

- **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 superclass 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 SUPERCLASS event;
```

Resolving Class Conflict

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 superclass when the superclass is modified. The following are operations that may cause conflicts between classes.

- Adding an attribute
- Deleting an attribute
- Adding a superclass
- Deleting a superclass
- 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:
[ CLASS ] attr_mthd_name OF superclass_name [ AS alias ]
```

Superclass Conflict

Adding a Superclass

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.

superclass_name specifies the name of the superclass 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 superclasses (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 Superclass

A name conflict may occur again if a superclass 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 superclass.

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 superclass. Because nation_code and host_year are explicitly inherited from the participant class, you must resolve their name conflicts before deleting it from the superclass. 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 SUPERCLASS participant
INHERIT nation_code OF stadium;
```

Compatible Domains

When an attribute conflict occurs among two or more superclasses, the statement resolving the conflict is not possible only if all attributes have compatible domains.

For example, the class that inherits a superclass with the phone attribute of integer type cannot have another superclass with the phone attribute of string type. If the types of the phone attributes of the two superclasses are both String or Integer, you can add a new superclass 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 superclass, 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 superclass has the same name as one already inherited by another superclass. 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 superclasses 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 superclass, the cost attribute of the stadium class will be inherited automatically.

Dropping Attributes and Methods

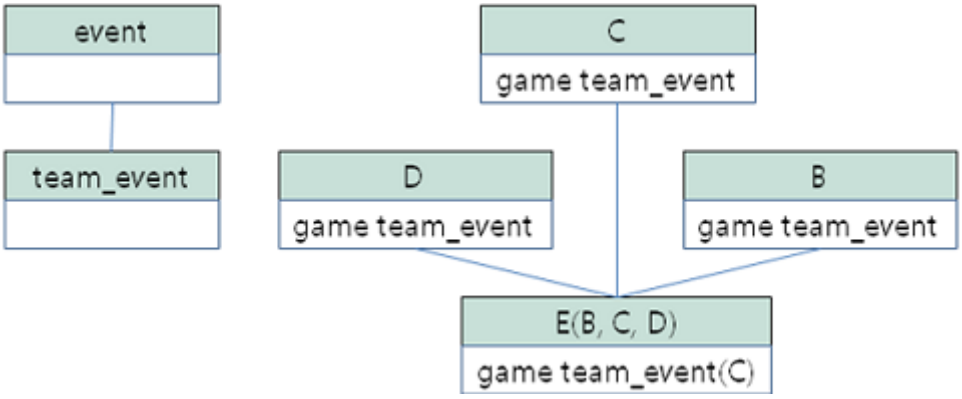
When an attribute is dropped from a class, any resolution specifiers which refer to the attribute by using the **INHERIT** clause are also removed. If a conflict occurs due to the deletion of an attribute, the system will determine a new inheritance hierarchy. If you don't like the inheritance hierarchy 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 superclasses. If a name conflict occurrs in all superclasses 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 superclasses 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 superclass. 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 superclass is event and that of another superclass 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 Invariants

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 superclasses, and cannot become their own superclasses. 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 superclasses. 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.

Rules 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 superclass, the relationship with the superclass is to be removed. If the relationship with the superclass 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 superclass. 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 superclass 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 superclasses, three aspects of the attribute (method) of each superclass 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 superclasses. In Case 1 (two different superclasses 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 superclasses 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 superclasses 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.

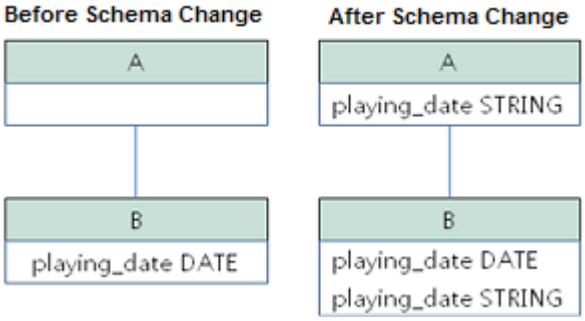
- **Rule 3** : If two or more superclasses have attributes with different sources and domains but the same name, attributes (methods) with more detailed (lower in the inheritance hierarchy) domains are inherited. If there is no inheritance relationship between domains, schema change is not allowed.

  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 superclass.

  The philosophy of Rule 4 is that "an inheritance is a privilege a subclass obtained from a superclass, so changes in a subclass cannot affect the superclass." Rule 4 means that the name of the attribute (method) included in the superclass cannot be changed to resolve conflicts between class C and superclasses. 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 superclass 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 superclass, 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 superclass is deleted when the attribute (method) is added to the superclass or the one inherited from the superclass 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 superclass can cause a conflict between the attribute (method) of the superclass 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 superclass. The addition/deletion of a superclass 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.

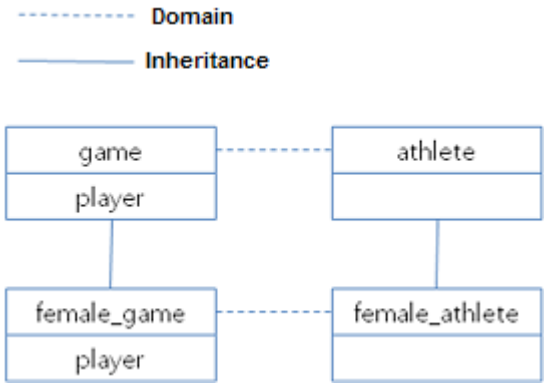
  This rule limits the application of Rule 5 and the invariant of inheritance. Conflicts can be detected and resolved by applying Rule 2 and 3.

- **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 superclass of D, the new domain is less generic than the corresponding domain in the superclass 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 superclass of female_athlete.



Class-Hierarchy Rule

- **`Rule 9`** : A class without a superclass becomes a direct subclass of object. The class-hierarchy rule defines characteristics of classes without superclasses. If you create a class without a superclass, object becomes the superclass. If you delete the superclass S, which is a unique superclass of class C, class C becomes a direct subclass of object.

Data Manipulation

Creating Data

INSERT INTO (Syntax)

Description

The **INSERT INTO** statement is used to insert a record of data into a table.

Syntax

```
INSERT INTO table_name [ ( attribute_list ) ] VALUES ( value_list ) [ ; ]
INSERT INTO table_name DEFAULT [ VALUES ] [ ; ]
```

- *table_name* : Specifies the name of the table where the data is to be inserted.
- *attribute_list* : Specifies column names for the values to be inserted. If *attribute_list* is not specified, all the specified columns in the table must be filled with values. If only some of the columns in the *attribute_list* are specified, a default value is assigned to the rest of the columns. If there is no default value, **NULL** is assigned.
- *value_list* : Specifies values corresponding to the columns of the *attribute_list* . The *value_list* field can be an expression, method or a call. Its attribute location and domain type must be the same as those of *attribute_list*. Each name and value is separated by a comma (,).
- **DEFAULT** : The **INSERT** statement in the second example above creates data by assigning a default value to each attribute. If the default value is not set for the column in the table definition, **NULL** is assigned as the column's value.

Example

- **Example 1**

  The following is an example of inserting the information about Nam Hyun-Hee, a 2008 Beijing Olympics silver medalist in men's fencing into the athlete table. code is an auto_increment column; a value should not be entered. For more information, see the Auto Increment section in Column.

```
INSERT INTO athlete (name, gender, nation_code, event) VALUES ('Nam Hyun-Hee',
'W', 'KOR', 'Fencing');
```

- **Example 2**

  The following is an example of inserting the general information about 2008 Beijing Olympics into the olympic table.

```
INSERT INTO olympic (host_year, host_nation, host_city, opening_date,
closing_date) VALUES (2008, 'China', 'beijing', '2008-08-08','2008-08-24');
```

- **Example 3**

  The following is an example of inserting data using **DEFAULT**. For this example, create the de_test table and insert data. As shown below, the specified default value is inserted automatically.

```
CREATE TABLE de_test(
a INT DEFAULT 10,
b INT DEFAULT 20,
c CHAR(1) DEFAULT 'M' );

INSERT INTO de_test DEFAULT;
csql> SELECT * FROM de_test;
csql> ;x
=== <Result of SELECT Command in Line 1> ===
            a            b  c
=============================================
           10           20  'M'
1 rows selected.
```

INSERT Using Queries

Description

Using a query in an **INSERT** statement, you can create multiple data with a single **INSERT** statement.

If you want to extract data from another table based on a specific search condition, use a query to include all the data that satisfies the condition in the table specified in the **INSERT** statement.

Syntax

```
INSERT INTO table_name [ (attribute_list ) ] query_statement [ ; ]
```

- *table_name* : Specifies the name of the table where the data is to be inserted.
- *attribute_list* : Specifies the column names for the data to be inserted. The data type defined here must be the same as that of the result of the *query_statement*.
- *query_statement* : Defines a query to extract the data to be inserted.

Example

The following is an example of creating a table named man and then inserting the result of the query that retrieves the names of male athletes from the athlete table. You can see that 4087 rows are inserted by a single **INSERT** statement.

```
CREATE TABLE man (
name VARCHAR(40)
);
csql> INSERT INTO man (name) SELECT name FROM athlete WHERE gender ='M';
csql> ;x
4087 rows inserted.
Current transaction has been committed.
1 command(s) successfully processed.
```

INSERT Using Subqueries

Description

You can insert data by including a query as one of the **VALUES** items in the **INSERT** statement.

Syntax

```
INSERT INTO table_name [ ( attribute_list ) ]
VALUES ( value_list, query_statement ) [ ; ]
```

- *table_name* : Specifies the name of the table where the data is to be inserted.
- *attribute_list* : Specifies the column names for the data to be inserted.
- *value_list* : Specifies the values corresponding to the columns of the *attribute_list*.
- *query_statement* : Defines a subquery. The result of the subquery must be a value corresponding to one of the columns.

Example

The following is an example of inserting the information about Wang Ki-chun, 2008 Beijing Olympics silver medalist in men's 73kg Judo into the athlete table.

```
INSERT INTO athlete VALUES (16800, 'Wang Ki-chun','M',SELECT code FROM nation
WHERE name ='Korea', 'Judo');
```

UPDATE

Description

Use the **UPDATE** statement to update the existing data inserted.

Syntax

```
UPDATE table_name
SET assignment [ {, assignment }... ]
[ WHERE search_condition ] [ ; ]

assignment:
(attribute_name_comma_list) = { (expression_comma_list) | (select_stmt)}
```

- *table_name* : Specifies the name of the table whose columns are to be updated.
- *attribute_name* : Specifies the columns to be updated.
- *expression* : Specifies a new value to be updated or **NULL** in the column.
- *select_stmt* : A query result can be used for the update operation. However, the query must return only one instance as a result. Also, the number of columns in the query result and the number of the columns specified in *attribute_name* must be same.
- *search_condition* : By using the **WHERE Clause**, you can limit the operation so that only instances that meet the *search_condition* will be updated.

Note

One column can be updated only once in the same **UPDATE** statement.

Example

- **Example 1**

  The following is an example of changing the capital city of Korea to Busan in the nation table.

  ```
  UPDATE nation SET capital ='Busan' WHERE name = 'Korea';
  ```

- **Example 2**

  The following is an example of inserting the information about Lim Su-jeong, a 2008 Beijing Olympics gold medalist in women's Taekwondo into the athlete table and updating the athlete's nationality with the query result.

  ```
  INSERT INTO athlete(name, gender, event) VALUES ('Lim Su-
  jeong','W','Taekwondo');

  UPDATE athlete SET nation_code = (SELECT code FROM nation WHERE name =
  'Korea') WHERE name = 'Lim Su-jeong';
  ```

DELETE

Description

The **DELETE** statement is used to delete unnecessary data from the table.

Syntax

```
DELETE
FROM table_spec [ correlation ]
[ WHERE search_condition ] [ ; ]

correlation :
[ AS ] identifier [ {, identifier }... ]

table_spec :
single_table_spec
( single_table_spec [ {, single_table_spec }... ] )

single_table_spec :
[ ONLY ] table_name
ALL table_name [ EXCEPT table_spec ]
```

- *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* using the **WHERE Clause**. If it is not specified, all the data in the table will be deleted.

Example

- **Example 1**

  The following is an example of deleting all the data from the stadium table keeping the table schema.

  ```
  DELETE FROM stadium;
  ```

- **Example 2**

  The following is an example of deleting the information about the 1980 Moscow Olympics using the **WHERE Clause**.

  ```
  DELETE FROM olympic WHERE host_year = 1980;
  ```

Virtual Tables (VIEW)

Creating Virtual Tables

CREATE VIEW (Syntax)

A virtual table is a table that does not physically exist and is defined as a query on an existing table or another virtual table. In CUBRID, **VIEW** or **VCLASS** can be used to represent a virtual table.

Description

You can create a virtual table by using the **CREATE VIEW** statement.

Syntax

```
CREATE [ VIEW | VCLASS ] view_name
[ { UNDER | AS SUBCLASS OF } super_vclass_name [ {, super_vclass_name }... ] ]
[ CLASS ATTRIBUTE ( class_attr_definition [ {,
class_attr_definition }... ] ) ] [ ( view_attr_definition [ { ,
view_attr_definition }... ] ) ]
[ METHOD method_definition [ { , method_definition }... ] ]
[ FILE path_name [ { , path_name }... ] ]
[ INHERIT resolution_list ]
[ AS query_specification
[ { UNION ALL query_specification }... ] ]
[ WITH CHECK OPTION] [ ; ]

super_vclass_name :
[ user_name.]classname

class_attr_definition :
attribute_name datatype [ DEFAULT value ]
[ NOT NULL ]

view_attr_definition :
column_name [ datatype ] [ { SHARED [ value ]
[ NOT NULL ] | DEFAULT value } ]
```

- *view_name* : Specifies the name of the virtual table to be created.
- *query_specification* : Defines the query on the virtual table. A query specification is assigned an integer number based on the specified order.
- *view_attr_definition*
  - *column_name* : Defines a column of the virtual table.
  - *datatype* : Specifies the data type of the column.

Example

The following is an example of creating the virtual table 'game_2004' that shows the names of athletes who participated in the 2004 Olympic Games and medals won by them.

```
CREATE VIEW game_2004 (name varchar(40), medal char(6) default 'none')
AS SELECT a.name, g.medal
FROM game g, athlete a
WHERE g.host_year = 2004 AND g.athlete_code = a.code ;
```

Conditions for Creating Modifiable VIEW

Description

To modify data in a virtual table, it must be modifiable because an option is needed to define data.

A virtual table is modifiable if it satisfies the following conditions:

- The **FROM** clause must include only one table or modifiable virtual table. However, two tables included in parentheses as in **FROM** (class_x, class_y) can be modified 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 modified by **UNION ALL**, not by **UNION**. However, the table must exist only in one of the queries that constitute **UNION ALL**.
- If an instance is inserted into a virtual table created by using the **UNION ALL** statement, the system determines which table the instance will be inserted into. This cannot be done by the user. To control this, the user must manually insert the instance or create a separate virtual table for insertion.

Even when all rules above are satisfied, each column of the modifiable virtual table may not be modifiable. For a column to be modifiable, the following rules must be observed:

- Path expressions must not be modifiable.
- Columns of number type with an arithmetic operator must not be modifiable.

Even though the column defined in the virtual table is modifiable, the virtual table can be modified only when there is an appropriate modification 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.

Example

The following is an example of creating the modifiable virtual table 'swimming' to retrieve an athlete's name, gender and event from the athlete table.

```
CREATE VIEW swimming(
name VARCHAR(50),
gender CHAR(1),
event VARCHAR(50)
)
AS SELECT name, gender, event FROM athlete WHERE event = 'Swimming';
```

Changing Virtual Tables

Renaming Virtual Tables

Description

You can change the name of a virtual table by using the **RENAME** 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 the name of the game_2004 virtual table to 2004_info.

```
RENAME VIEW game_2004 AS info_2004;
```

Adding New Queries

Description

You can add a new query to a query specification of a virtual table by using the **ADD QUERY** reserved word of the **ALTER** statement. Each query is assigned a sequential integer value.

Syntax

```
ALTER [ VIEW | VCLASS ] view_name
ADD QUERY select_statement
[ INHERIT resolution [ {, resolution }... ] ] [ ; ]

resolution :
[ CLASS ] attr_mthd_name OF superclass_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

The following is an example of creating the virtual table 'swimming' from the athlete table to retrieve the swimmers' name, gender and event, and adding a query in the swimming table to retrieve the athletes' information as well. Because there is one existing query in the swimming table, the added query is assigned number 2.

```
CREATE VIEW swimming(
name VARCHAR(50),
gender CHAR(1),
event VARCHAR(50)
)
AS SELECT name, gender, event FROM athlete WHERE event = 'Swimming';

ALTER VIEW swimming
ADD QUERY SELECT name, gender, event FROM athlete WHERE event = 'Athlete';
```

Changing Queries

Description

You can change the query defined in the query specification of a virtual table by using the **CHANGE QUERY** reserved word of the **ALTER** 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

The following is an example of changing the second query of the virtual table 'swimming' so that the information on Taekwondo players, not the athletes, is displayed.

```
ALTER VIEW swimming
CHANGE QUERY SELECT name, gender,event FROM athlete WHERE event = 'Taekwondo';
```

Dropping Queries

Description

You can drop a query defined in the query specification of a virtual table by using the **DROP QUERY** reserved word of the **ALTER** statement.

Example

The following is an example of dropping the second query of the swimming table.

```
ALTER VIEW swimming
DROP QUERY;
```

Dropping Virtual Tables

Description

You can drop a virtual table by using the **DROP** statement. The way to drop a virtual table is the same as to drop a regular table.

Syntax

```
DROP [ VIEW | VCLASS ] view_spec [ { ,class_spec }... ] [ ; ]

view_spec :
single_view_spec
( single_view_spec [ { , single_view_spec }... ] )

single_view_spec :
[ ONLY ] view_name
ALL view_name [ ( EXCEPT view_spec ) ]
```

- *view_name* : Specifies the name of the virtual table to be dropped.

Example

The following is an example of dropping the virtual table 'swimming.'

```
DROP VIEW swimming;
```

INDEX

Creating Indexes

Description

Use the **CREATE INDEX** statement to create an index in the specified table.

Syntax

```
CREATE [ REVERSE ] [ UNIQUE ] INDEX [ index_name ]
ON table_name ( attr_name [ASC | DESC] [ {, attr_name  [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.
- *table_name* : Specifies the name of the table where the index is to be created.
- *attr_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

- **Example 1**

  The following is an example of creating a single column index.

```
CREATE INDEX ON game(medal);
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);
```

Dropping Indexes

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 ( attr_name [ {, attr_name } ...) ] [ ; ]

DROP [ REVERSE ] [ UNIQUE ] INDEX
 ON table_name ( attr_name [ {, attr_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.
- *attr_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);
```

Altering Indexes

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 ( attr_name [ {, attr_name } ...) ] REBUILD [ ; ]

ALTER [ REVERSE ] [ UNIQUE ] INDEX
ON { ONLY } table_name ( attr_name [ {, attr_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.
- *attr_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.

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;
```

Transaction Management

Overview

This chapter covers issues relating to concurrency and restore, as well as how to commit or roll back transactions.

In a 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 table 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 a 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 instance is inserted into a table, only the user who inserted the instance 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 table. 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.
```

The three **UPDATE** statements must have the current values of seats in each stadium. After the command is executed, you can retrieve related columns of the seats table to find out whether the data is inserted correctly.

```
update stadium
set seats = seats + 1000
where code in (30138, 30139, 30140);
;xrun
3 rows updated.
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.
```

If updates are made correctly, you can make them permanent by using the following **COMMIT WORK** statement.

```
;commit work
```

Note If the application exits the database without committing the transaction when the autocommit mode is off, all updates made after the last **COMMIT WORK** are automatically rolled back. Therefore, CSQL Interpreter performs a procedure to check whether the transaction should be committed or aborted. For the AUTOCOMMIT session command, 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 instance of the same table.

```
alter table 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 by **ALTER TABLE** statement. 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 table 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 rolback 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

```
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

```
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 a 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 is required every time multiple transactions are executed at the same time. The simultaneous execution of multiple transactions must work in the same manner as executing each transaction sequentially. This principle is based on the assumption that database concurrency will be protected if transactions are executed atomically (i.e., the entire transaction either commits or rolls back). Serializability in CUBRID is managed by the well-known two-phase locking technique. This will be covered in the Lock Protocol section.

The transaction to commit must ensure database concurrency, and each transaction must guarantee appropriate results. When other transactions are being executed, events in one transaction must be hidden from other transactions. This is called isolation. Transaction isolation level is the degree to which a transaction is separated from all other concurrent transactions. The higher the isolation level, the lower the influence from other transactions. The lower the isolation level, the higher the concurrency. Consistency and concurrency are controlled by the isolation level. Therefore, you need to determine the isolation level based on the system to be applied.

For isolation levels supported by CUBRID, see Concurrency/Lock Parameters in Database Server Configuration.

CUBRID controls concurrent accesses by using a sophisticated locking technique. For any type of data, the lock required for access is determined automatically, without the need for any hint from the application; that is, the user's intervention is not required. CUBRID acquires read and write locks to execute retrieval and update operations, respectively. You can lock a specific table, instance or index, as well as the entire database.

The isolation levels can be explained by the following three concepts.

- **`Dirty read`**: This occurs when a transaction reads/updates an object that has been updated by another transaction, but has not yet been committed, and it will never be committed (i.e. the transaction can be rolled back). In this case, the second transaction may access an object that does not exist in fact. The following describes several dirty read scenarios:
- Transaction T1 updates an object. Transaction T2 reads the object before transaction T1 commits or aborts. If T1 aborts, T2 will see the contents of an object that has not been committed.
- Transaction T1 deletes instance O1 of table C, and inserts O2. Then transaction T2 executes a query on table C. If T1 is aborted, T2 will see the instance O2 that has not been committed, or will not see O1, which has not been deleted.
- Two transactions concurrently update the same object. The two transactions may have a new value derived from the object, by an uncommitted value. If one transaction aborts or both transactions commit, it is not certain which value will be stored in the database and which will be lost. This is often called a lost update.
- **`Non-repeatable read`**: If a transaction reads an object that it has previously read, it will see a different value of the object. The following describes several non-repeatable read scenarios:
- Transaction T1 reads an object. Then transaction T2 updates (deletes) the object and commits. If T1 attempts to read the object again, it will see new contents of the object. (It may find that the object no longer exists.)
- Transaction T1 executes a query on table C. Then transaction T2 deletes and inserts some instances of C, and commits. If T1 executes the same query again, it may get different results due to the inserted/deleted objects.
- Transaction T1 executes a query on table C. Then transaction T2 deletes an attribute of table C and commits. If T1 executes the same query again, it may not see the deleted column.

- **`Phantom read`**: If a transaction retrieves an object multiple times with a search condition, it will see newly inserted objects, which are called phantoms. The following describes several phantom read scenarios:
- Transaction T1 retrieves an object with a single search condition. Then transaction T2 inserts a new instance satisfying the search condition and commits. If T1 attempts to read the object again using the same search condition, it may see the new object inserted by T2.
- Transaction T1 retrieves an object with a single search condition. Then transaction T2 updates an object that no longer satisfies the search condition, and commits. If T1 attempts to read the object again with the same search condition, it may see the object updated by T2.
- Transaction T1 retrieves an object after transaction T2 deletes one of the objects satisfying the search condition of T1. Then T2 rolls back. If T1 retrieves the object again with the same search condition, it may see the object T2 tried to delete.

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 transactions does 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.

Determining Lock Permissions

CUBRID determines the lock to obtain based on the operation to be performed. The user can never direct CUBRID to obtain a certain type of lock. A lock can be granted depending on whether another transaction has a lock on the required data or, if so, what kind of lock mode the transaction has. For example, if another transaction has an exclusive lock on the data desired, the required lock cannot be granted for the transaction requiring it.

The following lock compatibility table shows which granted lock is compatible with which required lock. This table indicates whether or not a lock in the M2 mode can be permitted for transaction T2 when transaction T1 currently has a lock in the M1 mode. For example, if transaction T1 has a **X_LOCK**, no lock in any mode is granted for another transaction. This transaction must stay in a standby mode until the **X_LOCK** is released. However, if transaction T1 has a **S_LOCK**, **IS_LOCK** or **S_LOCK** is granted for another transaction T2. N/A means 'Not Applicable.'

Lock Compatibility

		Lock Holder						
		NULL_LOCK	**IS_LOCK**	**S_LOCK**	**IX_LOCK**	**SIX_LOCK**	**U_LOCK**	**X_LOCK**
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.
- **IS_LOCK** : Intention shared lock.

  The system obtains **IS_LOCK** on a table to permit reading the table instance definition (schema definition). This lock permits reading some instances by obtaining **S_LOCK** on the instances of the table. Whenever **S_LOCK** is permitted on an instance of the table, **IS_LOCK** is also permitted on the table. **IS_LOCK** on the table prevents other transactions from updating all instances of the table or the table definition (schema definition) of the instance.

  IS_LOCK on the instance can never be obtained because it is the smallest unit in the CUBRID protocol.

- **IX_LOCK** : Intention exclusive lock.

  The system obtains **IX_LOCK** on the table to permit reading the table instance definition (schema definition). This lock permits updating (reading) some instances by obtaining an exclusive (shared) lock on the instances of the table. Whenever **X_LOCK** is permitted on the instances of the table, **IX_LOCK** is also permitted on the table. **IX_LOCK** on the table prevents other transactions from updating the table definition (schema definition) of the instance or reading/updating all instances of the table.

  IX_LOCK on the instance can never be obtained because it is the smallest unit in the CUBRID protocol.

- **S_LOCK** : Shared lock.

  The system obtains **S_LOCK** on a table to permit reading the table definition (schema definition) of the table instance. This lock also permits reading all instances without obtaining **S_LOCK** on all instances. **S_LOCK** on the table prevents other transactions from updating the schema definition of the instance or any value of the table instance.

  The system obtains **S_LOCK** on an instance to read it. In this case, **IS_LOCK** must be obtained on the table of the instance.

- **X_LOCK** : Exclusive lock.

  The system obtains **X_LOCK** on a table to permit updating the table definition (schema definition) of the table instance. This lock also permits updating (reading) all instances of the table without obtaining **X_LOCK** (**S_LOCK**) on the instance. **X_LOCK** on the table prevents other transactions from reading or updating the table definition or any instance value of the table.

  The system obtains **X_LOCK** on an instance to update it. In this case, **IX_LOCK** must be obtained on the table of the instance.

- **SIX_LOCK** : Shared intention exclusive lock.

  This lock is the combination of **S_LOCK** and **IX_LOCK** and obtained on the table. This lock permits reading the table definition (schema definition) of the table instance and reading all instances of the table without obtaining **S_LOCK** on the instance. It also permits updating some instances of the table by obtaining **X_LOCK**.

  SIX_LOCK on the instance can never be obtained because it is the smallest unit in the CUBRID protocol.

- **U_LOCK** : Update lock.

  The **DELETE** or **UPDATE** statement retrieves and reads the instances to be deleted or updated and then deletes them or updates some of their values. It obtains **S_LOCK** to read the instances and **X_LOCK** to delete or update them. In this process, when there are two transactions that are trying to delete or update the same instance, if both of them require **X_LOCK** already holding **S_LOCK**, these two transactions fall into a deadlock.

  Such deadlock can be avoided by using the **U_LOCK** mode. If a transaction retrieves the index to reads the instances for delete or update purpose, it obtains **U_LOCK** instead of **S_LOCK**. If it deletes or updates the instances, it obtains **X_LOCK**. Two or more transactions cannot obtain **U_LOCK** simultaneously on the same instance because **U_LOCK**s are incompatible each other. Therefore, transactions can be executed serially without falling into a deadlock. If a transaction retrieves the index to reads the instances for delete or update purpose, it obtains **U_LOCK** by getting each OID satisfying the key range and filter through the index search. In the second step, it obtains a **X_LOCK** on each OID that also satisfies the delete or update condition. If an OID obtains an **U_LOCK** in the first step but fails to obtain an **X_LOCK** in the second step by not satisfying the data filter condition, its **U_LOCK** is demoted to **S_LOCK** based on the transaction isolation level or is released.

Example

In the following example, the isolation level is set to **REPEATABLE READ CLASS** with **READ COMMITTED INSTANCES** and **AUTOCOMMIT** is set to OFF (;autocommit off).

Suppose that two users try to access the same table simultaneously. T1 needs to query data from the table, and T2 is already updating one of the attributes selected by T1 in the query statement (execute commands to the "------ 1st" in the example T2).

- User 1 (T1):

```
;autocommit off
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ CLASS, READ COMMITTED
INSTANCES;
select host_year, nation_code, gold from participant where gold > 10 and
host_year >= 2000;
;xrun
```

- User 2 (T2):

```
;autocommit off
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ CLASS, READ COMMITTED
INSTANCES;
update participant
set gold = 11
where nation_code = 'KOR' and host_year=2000;
;xrun
------ 1st
update participant
set gold = 29
where nation_code = 'RUS' and host_year=2004;
```

```
update participant
set gold = 37
where nation_code = 'USA' and host_year=2004;
commit work;
;xrun
```

In this case, to maintain consistency, the system places transaction T1 in a standby mode until the data is accessible. The instance updated by the transaction T2 can be read by transaction T1 after T2 commits. As soon as transaction T2 is committed or rolled back, CUBRID releases the lock acquired by transaction T2 and allows T1 to resume. Once the changes made in transaction T2 become permanent, transaction T1 can access the updated data.

Transaction Deadlocks

There is a situation where one transaction is waiting for another one to finish (commit or rollback). Such case is called a deadlock because transactions prevent each other from proceeding. If a deadlock occurs, CUBRID resolves the problem by rolling back one of the transactions. The transaction to be rolled back is usually the transaction which made least updates, i.e. the one that started latest.

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. Deadlock situations are irregular and impossible to predict. However, it is likely that a deadlock occurs if there are increases in the number of users who want to update the same data.

Example

In the following example, both **`isolation_level`** = 6 (**`SERIALIZABLE`**) and **`AUTOCOMMIT`** are OFF.

Suppose that two users try to update the same data simultaneously. In this example, User1 (T1) creates the participant2 table and inserts some instances. User1 (T1) and User2 (T2) query the participant2 table and get the same result.

- User1 (T1):

```
;autocommit off
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
create class participant2 (host_year integer, nation_code char(3), gold
integer, silver integer, bronze integer)
insert into participant2 (host_year, nation_code, gold, silver, bronze)
values (2008, 'KOR', 20,10,10);
insert into participant2 (host_year, nation_code, gold, silver, bronze)
values (2008, 'NED', 12,22,18);
insert into participant2 (host_year, nation_code, gold, silver, bronze)
values (2008, 'GER', 13,17,19);
commit work;
;xrun
select * from participant2;
;xrun

1 command(s) successfully processed.
```

- User2 (T2):

```
;autocommit off
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
select host_year, nation_code from participant2;
=== <Result of SELECT Command in Line 2> ===
    host_year  nation_code
===================================
         2008  'KOR'
         2008  'NED'
         2008  'GER'
3 rows selected.
```

Then User1 tries to delete the 'GER' instance from the participant2 table but fails because User2 did not either commit or abort. User2 tries to insert instances into the participant2 table, but fails because User1 did not commit or abort. The system resolves this deadlock by aborting the transaction of User1.

- User1 (T1):

```
delete from participant2 where host_year=2008 and nation_code = 'GER';
;xrun
------- <T1 aborts after T2 executes ;xrun>
In line 1, column 1,
ERROR: Your transaction (index 2, brightes@cdbs006.cub|27967) has been
unilaterally aborted by the system.
0 command(s) successfully processed.
```

- User2 (T2)

```
insert into participant2 (host_year, nation_code, gold, silver, bronze)
values (2008, 'AUS', 13,17,19);
;xrun
```

Transaction Timeout

CUBRID provides a lock timeout feature. Applications can inform CUBRID of the minimum waiting time for a lock by using the **lock_timeout_in_secs** system parameter or the **SET TRANSACTION** statement. If the timeout value is not specified, by default, the waiting continues until the lock is allowed, or until the transaction is rolled back due to a deadlock.

If the lock is not allowed within the specified timeout period, it is rejected and the execution of the operation is canceled. CUBRID returns an error informing that the operation has been canceled due to lock timeout.

If a transaction with a specified timeout value goes into a deadlock, CUBRID selects the transaction based on the timeout value instead of rolling it back. If multiple transactions in a deadlock have expired timeout values, CUBRID attempts to select the one closest to the specified timeout value. Therefore, timeout can be used to avoid indefinite waiting for a lock, or to prevent a transaction from being rolled back by the system.

Setting Timeout Values

The following is the syntax of the **SET TRANSACTION** statement.

Syntax

```
SET TRANSACTION LOCK TIMEOUT timeout_spec [ ; ]
timeout_spec:
_ INFINITE
_ OFF
_ unsigned_integer
_ variable
```

If *timeout_spec* is set to **INFINITE** (default), the transaction is allowed to hold a lock, or waits indefinitely until it is rolled back due to a deadlock. If it is set to **OFF**, the transaction does not wait for the lock when another transaction is holding an incompatible lock. In this case, the lock is rejected, and the operation canceled. If the lock is not allowed within the timeout period (in seconds) specified by the *unsigned_integer*, it is rejected and the operation canceled. If a *variable* is used in this statement, waiting for the lock to be allowed continues for the period specified by the variable. If the operation is canceled, CUBRID returns an error informing that the operation has been canceled due to the lock timeout.

Getting Timeout Values

You can also get a specified timeout value by using the **GET TRANSACTION** statement.

Syntax

```
GET TRANSACTION LOCK TIMEOUT [ { INTO | TO } variable ] [ ; ]
```

This command retrieves the value of a timeout parameter and assigns it to a given variable.

Lock Timeout Error Message

If timeout occurs after blocking a lock request (timeout occurs if you press the <Ctrl+C> key when the lock is blocked in CSQL, or when a transaction whose **lock_timeout_in_secs** value is not -1 reaches **lock_timeout_in_secs**), information about the other transactions that blocked the lock request is outputted as an error. To output the information about transactions, modify the **lock_timeout_message_type** system parameter. The default value is **0**. If the value is 1, one of the transactions that blocked the lock request is outputted. If it is 2, all transactions that blocked the lock request are outputted.

The following are the results of the tests in which the isolation level is set to **SERIALIZABLE** and the **lock_timeout_message_type** values are changed each time. After changing, you must restart the database server in oder to apply the **lock_timeout_message_type** value.

If the value is 0, the following error message is outputted:

```
ERROR: Your transaction (index 3, brightes@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 the value is 1, the following error message is outputted:

```
ERROR: Your transaction (index 3, brightes@cdbs006.cub|15668) timed out
waiting on    X_LOCK lock on instance 0|636|34 of class participant. You are
waiting for user(s) brightes@cdbs006.cub|15615 to finish.
```

If the value is 2, the following error message is outputted:

```
ERROR: Your transaction (index 3, brightes@cdbs006.cub|15668) timed out
waiting on    X_LOCK lock on instance 0|636|34 of class participant. You are
waiting for user(s) brightes@cdbs006.cub|15615, brightes@cdbs006.cub|15596 to
finish.
```

Transaction Isolation Level

Overview

CUBRID specifies a transaction isolation level so that the transaction is executed with less restrictive consistency. The transaction isolation level is the degree to which a transaction is separated from all other concurrent transactions. The higher the isolation level, the lower the influences from other transactions. The lower the isolation level, the higher concurrency. A transaction is valid if the transaction acquires a lock in an exclusive (shared/exclusive) mode before it updates (reads) an object and does not acquire any more locks after they are released.

Note A transaction can be restored in all supported isolation levels because updates are not committed before the end of the transaction.

Setting Isolation Levels

Description

You can set an isolation level by using the **isolation_level** system parameter and the **SET TRANSACTION** statement. On the initial installation of CUBRID, the isolation level set in **cubrid.conf** is **REPEATABLE READ CLASS, READ UNCOMMITTED INSTANCES**. If an isolation level is not specified, the default transaction isolation level is **REPEATABLE READ CLASS, READ COMMITTED INSTANCES**. **isolation_level** applies to the combination of tables (schema) and instances.

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

The following command sets a repeatable read isolation level on tables and instances.

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ CLASS, REPEATABLE READ
INSTANCES
```

Some combinations of isolation levels are not allowed in CUBRID. Invalid isolation levels are discussed in the Unsupported Combinations of Isolation Levels section. The following table summarizes some isolation level combinations that can be set by the **SET TRANSACTION** statement.

Isolation levels supported by CUBRID

Isolation Level	Description
SERIALIZABLE	Problems concerning concurrency (e.g. dirty read, non-repeatable read, phantom read, etc.) do not occur.
REPEATABLE READ CLASS with **REPEATABLE READ INSTANCES**	A transaction does not read an object being modified by another transaction, and non-repeatable reads on a given instance do not occur. Another transaction cannot execute an update operation on the object (table or instance) being read by the current transaction. Retrieving instances that satisfy a specific condition may cause a phantom read.
REPEATABLE READ CLASS with **READ COMMITTED INSTANCES** (or **CURSOR STABILITY**)	A transaction does not read an object being modified by another transaction. This isolation level does not allow another transaction to update the table (part of schema) being accessed by the transaction. The transaction may experience a non-repeatable read on an instance. This means it may read two different (committed) values if it reads the same object twice.
REPEATABLE READ CLASS with **READ UNCOMMITTED INSTANCES**	A transaction does not read a table being modified by another transaction. This isolation level does not allow another transaction to update the table (part of schema) being accessed by the transaction. However, it allows the transaction to read uncommitted dirty instances, which can be committed or rolled back later, as well as committed instances.
READ COMMITTED CLASS with **READ COMMITTED INSTANCES**	A transaction does not read an object (table or instance) being modified by another transaction. This isolation level does not prevent another transaction from updating the table or instance read by the transaction. The transaction might not be able to read the table (schema) or instance twice (non-repeatable read).
READ COMMITTED CLASS with **READ UNCOMMITTED INSTANCES**	A transaction does not read a table being modified by another transaction and reads dirty instances (as well as committed values). The transaction might not be able to read the table (schema) or instance twice (non-repeatable read).

Once a new isolation level is set, the rest of the transaction (until another isolation level is set) uses the new one. The changed isolation level applies to subsequent transactions until another isolation level is set. It is recommended to change the isolation level, if necessary, at the start of a transaction (after commit, rollback or system restart). If a certain isolation level is set during a transaction, some resources (locks) obtained by the transaction can be released while a new isolation level is being set. In this case, you must not assume that one isolation level will apply to the entire transaction.

For example, if you change the initial isolation level **READ COMMITTED CLASS** with **READ UNCOMMITTED INSTANCES** to **REPEATABLE READ CLASS** with **REPEATABLE READ INSTANCES** (**SERIALIZABLE**) during the transaction, the entire transaction may not be **SERIALIZABLE** due to some reads that can occur in the transaction before the new isolation level is set. Generally, it is safe to suppose that the least restrictive isolation level affects the entire transaction.

Getting Isolation Levels

Description

You can get the current isolation level by using the **GET TRANSACTION** statement. The following is a statement that acquires the isolation level. In the following statement, the isolation level is retrieved and assigned to the given *variable*.

Syntax

```
GET TRANSACTION ISOLATION LEVEL [ { INTO | TO } variable ] [ ; ]
```

SERIALIZABLE

In this isolation level, 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:

- A transaction does not overwrite an object being modified by another transaction.
- A transaction does not commit an object until the end of the transaction.
- A transaction does not read an object being modified by another transaction.
- Another transaction cannot add objects that belong to the objects (instances of a certain table or instances satisfying a certain condition) read by the current transaction until the end.

This isolation level uses a two-phase locking protocol.

Example

The following is an example where one transaction inserts an instance into a certain table and another transaction retrieves data with a specific search condition. Transaction T1 is at any isolation level, and transaction T2 has the **SERIALIZABLE** isolation level. Here, it is assumed that the participant2 table was not created earlier.

- T1 creates the participant2 table, and a unique key for host_year and nation_code attributes, and then inserts some instances. After T1 commits, CUBRID releases all locks obtained by T1. Then T2 retrieves data on the participant2 table with the nation_code = 'AUS' condition.
  - User1 (T1):

```
set transaction isolation level serializable;
;autocommit off
create class participant2 (host_year integer, nation_code char(3));
create unique index on participant2 (host_year, nation_code);
insert into participant2 (host_year, nation_code) values (2008, 'AUS');
commit work;
;xrun
1 rows inserted.
```

- User2 (T2)

```
set transaction isolation level serializable;
;autocommit off
select * from participant2 where nation_code='AUS';
;xrun
1 rows selected.
```

- T1 tries to insert the instance (2004, 'AUS') into the participant2 table, but it stands by because T2 hasn't executed the **COMMIT WORK** statement yet after retrieving with the nation_code = 'AUS' condition. Once T2 commits and releases all locks, the waiting T1 is executed. If T2 retrieves again with the same condition later on (below ------ 2nd), it enters into a standby state because T1 still holds the lock on the instance inserted.
  - User1 (T1):

```
insert into participant2 (host_year, nation_code) values (2004, 'AUS');
;xrun
------ 1st
1 rows inserted.
```

  - User2 (T2):

```
select host_year, nation_code from participant2 where nation_code='AUS';
;xrun
=== <Result of SELECT Command in Line 1> ===
    host_year  nation_code
=====================================
         2008  'AUS'
1 rows selected.
commit work;
;xrun
1 command(s) successfully processed.
------ 2nd
select host_year, nation_code from participant2 where nation_code='AUS';
;xrun
```

- If T1 commits, the locks are released and T2 is executed, which returns the result of the **SELECT** statement.
  - User1 (T1):

```
commit work;
;xrun
```

  - User2 (T2):

```
=== <Result of SELECT Command in Line 1> ===
    host_year  nation_code
=====================================
         2008  'AUS'
         2004  'AUS'
2 rows selected.
```

- T1 enters a standby state when it tries to change the name of the participant2 table to nation_medals because T2 did not commit yet.
  - User 1 (T1):

```
rename class participant2 as nation_medals;
;xrun
```

- If T2 commits (a), T1 resumes and the participant2 table is renamed to nation_medals table. If T1 commits (b) and T2 executes the **SELECT** statement on the participant2 table, an error occurs because the participant2 table does not exist any more.
  - User 2 (T2):

```
commit work;
;xrun
-------------------------------------------------------------------------------(a)
select * from participant2 where nation_code = 'AUS';
;xrun
-------------------------------------------------------------------------------(c)
```

```
In line 1, column 1,
ERROR: Unknown class "participant2".
```

- User 1 (T1):

```
commit work;
;xrun
---------------------------------------------------------------------------(b)
```

REPEATABLE READ CLASS with REPEATABLE READ INSTANCES

In this isolation level, a dirty or non-repeatable read does not occur. The transaction may, however, experience phantom reads.

The following are the rules of this isolation level:

- A transaction does not overwrite an object being modified by another transaction.
- A transaction does not commit an object until the end of the transaction.
- A transaction does not read an object being modified by another transaction.
- A transaction does not release any shared/exclusive lock until the end of the transaction, to ensure repeatable reads.
- Another transaction cannot add objects that belong to the objects (record of a certain table or records satisfying a certain condition) read by the current transaction until the end.

This isolation level uses a two-phase locking protocol.

Example

The following is an example where one transaction in this isolation level inserts a record into a certain table and another transaction retrieves data with a specific search condition. Transaction T1 is at any isolation level, and transaction T2 has the **REPEATABLE READ CLASS** with **REPEATABLE READ INSTANCES** isolation level. Here, it is assumed that the participant2 table was not created earlier.

- T1 creates the participant2 table, and a unique key for host_year and nation_code columns, and then inserts some records. After T1 commits, CUBRID releases all locks obtained by T1. Then T2 retrieves data on the participant2 table with the nation_code = 'AUS' condition.
- User1 (T1):

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ CLASS, REPEATABLE READ
INSTANCES;
;xrun
;autocommit off
create table participant2 (host_year integer, nation_code char(3));
create unique index on participant2 (nation_code, host_year);
insert into participant2 (host_year, nation_code) values (2008, 'AUS');
insert into participant2 (host_year, nation_code) values (2008, 'JPN');
insert into participant2 (host_year, nation_code) values (2008, 'BRZ');
insert into participant2 (host_year, nation_code) values (2008, 'CHN');
insert into participant2 (host_year, nation_code) values (2004, 'AUS');

commit work;
;xrun
1 rows inserted.
1 rows inserted.
```

- User2 (T2):

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ CLASS, REPEATABLE READ
INSTANCES;
;xrun
;autocommit off
select host_year, nation_code from participant2 where nation_code='AUS' and
host_year=2008;
;xrun
=== <Result of SELECT Command in Line 1> ===
```

```
    host_year  nation_code
==================================
         2008  'AUS'
1 rows selected.
```

- If T1 tries to insert records represented as (2004, 'KOR') and (2000, 'NED') into the participant2 table, it can proceed right away because T1 and T2 don't hold locks on the same record. Here, records represented as (2004, 'KOR') and (2000, 'NED') may be seen as phantom records to T2. T2 can see the records T1 inserted if it executes the same query again.
- User1 (T1):

```
insert into participant2 (host_year, nation_code) values (2004, 'KOR');
insert into participant2 (host_year, nation_code) values (2000, 'NED');
commit work;
;xrun
1 rows inserted.
1 rows inserted.
```

- User2 (T2):

```
select * from participant2 where nation_code='AUS' and host_year=2008;
;xrun
1 rows selected.
```

- T1 stands by after it executes a query that deletes the record represented as (2008, 'AUS') because there are uncommitted records on which T2 still holds shared locks. If T2 commits, all locks are released and T1 is executed. T2 stands by after it executes the same query again because of the records deleted by T1.
- User1 (T1):

```
delete from participant2
where nation_code = 'AUS' and
host_year=2008;
;xrun
```

- User2 (T2):

```
commit work
;xrun
------
select * from participant2
where nation_code= 'AUS';
;xrun
```

- If T1 commits, the locks are released and T2 is executed, which returns the result of the **SELECT** statement.
- User1 (T1):

```
commit work
;xrun
```

- User2 (T2):

```
=== <Result of SELECT Command in Line 1> ===
    host_year  nation_code
==================================
         2004  'AUS'
```

- T1 enters a standby state when it tries to change the name of the participant2 table to nation_medals because T2 did not commit yet.
- User1 (T1):

```
rename table participant2 as nation_medals;
;xrun
```

- If T2 commits, T1 resumes and the participant2 table is renamed to nation_medals table. A syntax error occurs if T1 commits and T2 is executed again because the participant2 does not exist any more.
- User2 (T2):

```
commit work;
select * from participant2 where nation_code = 'AUS';
```

```
;xrun
In line 2, column 16,
ERROR: Class participant2 does not exist.
```

- User1 (T1):

```
commit work;
;xrun
```

REPEATABLE READ CLASS with READ COMMITTED INSTANCES

This isolation level does not allow a transaction to read an object being modified by another transaction. It also does not allow another transaction to change the table (part of schema) accessed by the transaction. However, the transaction might not be able to read a record twice (non-repeatable read). This means that the transaction may read two different values if it reads the same object twice. This can happen because another transaction may start, change and commit the object between the two reads. Similarly, the results can be different if a single query is executed twice. The difference in the results is caused by another transaction inserting, updating or deleting records committed between the execution of the two queries.

You can use the **`CURSOR STABILITY`** keyword as another name of the isolation level when you execute the **`SET TRANSACTION`** statement.

The following are the rules of this isolation level:

- A transaction does not overwrite an object being modified by another transaction.
- A transaction does not commit an object until the end of the transaction.
- A transaction does not read an object being modified by another transaction.
- Another transaction does not modify a table being read or changed by the current transaction until the end of the current transaction.

This isolation level uses a two-phase locking protocol for an exclusive lock. However, a shared lock on a record is released right after it is read. An intention lock on a table is released at the end of the transaction because schema reads are repeatable. On the initial installation of CUBRID, the isolation level set in **`cubrid.conf`** is **`REPEATABLE READ CLASS, READ UNCOMMITTED INSTANCES`**. If an isolation level is not specified, this will be the default isolation level.

Example

One transaction inserts or deletes data for a table while another transaction queries the table in various aspects. Transaction T1 has an optional isolation level, and transaction T2 has the **`REPEATABLE READ CLASS`** with **`READ COMMITTED INSTANCES`** isolation level. In this example, it is assumed that the participant2 table was not created earlier.

- Transaction T1 creates the participant2 table and inserts a record into the table. As soon as transaction T1 commits, CUBRID releases the lock obtained by transaction T1. Then transaction T2 obtains the lock on the participant2 table. The lock on the record is released after the query is executed, and another transaction is permitted to modify the record.
- User1 (T1):

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ CLASS, READ COMMITTED
INSTANCES;
;xrun
;autocommit off
create table participant2 (host_year integer, nation_code char(3));
insert into participant2 (host_year, nation_code) values (2008, 'AUS');
commit work;
;xrun
```

- User2 (T2):

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ CLASS, READ COMMITTED
INSTANCES;
;xrun
```

```
;autocommit off
select * from participant2;
;xrun
```

- Transaction T1 inserts another record into the participant2 table. When transaction T2 tries to query the participant2 table, it stands by the query execution because transaction T1 is trying to insert another record into the table without releasing the lock. Transaction T2 can see only committed values, so it has to wait until transaction T1 commits.
- User1 (T1):

```
insert into participant2 (host_year, nation_code) values (2000, 'NED');
;xrun
```

- User2 (T2):

```
select host_year, nation_code from participant2;
;xrun
```

- Once transaction T1 inserts another record into the participant2 table and commits, locks are released. Transaction T2 resumes and gets the query result. The participant2 table includes the record committed by the current transaction T1. Note that in the same transaction, the **SELECT** statement may return a different result from the first query result (non-repeatable read on a given record).
- User1 (T1):

```
insert into participant2 (host_year, nation_code) values (2004, 'AUS');
commit work;
;xrun
```

- User2 (T2):

```
=== <Result of SELECT Command in Line 1> ===
    host_year  nation_code
=====================================
        2008  'AUS'
        2000  'NED'
        2004  'AUS'
3 rows selected.
```

- Transaction T1 inserts another record into the participant2 table. Transaction T2 tries to query the participant2 table again, but stands by because transaction T1 is inserting another record into the table.
- User1 (T1):

```
insert into participant2 (host_year, nation_code) values (1994, 'FRA');
;xrun
```

- User2 (T2):

```
select host_year, nation_code from participant2;
;xrun
```

- Transaction T1 deletes the first record (2008, 'AUS') of the participant2 table and commits without releasing the lock. Transaction T2 resumes and sees the new record as well as some records inserted earlier. However, the first record (2008, 'AUS') has been deleted.
- User1 (T1):

```
delete from participant2 where host_year=2008 and nation_code='AUS';
commit work;
;xrun
```

- User2 (T2):

```
=== <Result of SELECT Command in Line 1> ===
    host_year  nation_code
=========================
        2000  'NED'
        2004  'AUS'
        1994  'FRA'
3 rows selected.
```

- Transaction T1 tries to change the name of the participants2 table to nation_medals, but stands by because transaction T2 holds a lock on the table (schema is repeatable). Transaction T2 queries the

table again, commits and releases the lock. Transaction T2 tries to query the table again, but stands by because transaction T1 has not committed yet.

- User1 (T1):

```
rename table participant2 as nation_medals;
;xrun
```

- User2 (T2):

```
select host_year, nation_code from participant2;
;xrun
=== <Result of SELECT Command in Line 1> ===
    host_year  nation_code
===================================
         2000  'NED'
         2004  'AUS'
         1994  'FRA'

commit work;
select * from participant2;
;xrun
```

- Transaction T1 commits the operation and releases the lock, then transaction T2 resumes. However, the query fails because the name of T2 has been changed to nation_medals. Then, transaction T2 gets a syntax error saying that the participant2 table does not exist any more.

- User1 (T1):

```
commit work;
;xrun
1 command(s) successfully processed.
```

- User2 (T2):

```
In line 2, column 16,
ERROR: Class participant2 does not exist.
```

REPEATABLE READ CLASS with READ UNCOMMITTED INSTANCES

This isolation level does not allow a transaction to read a table being modified by another transaction. It also does not allow another transaction to update the table (part of schema) accessed by the transaction. This isolation level allows the transaction to read dirty records - which can be committed or rolled back later - as well as committed records. The transaction may read two different values if it reads the same record twice. This is caused by another transaction updating records between the two reads. Similarly, the results can be different if the same query is executed twice. This situation is caused by another transaction inserting, updating or deleting records, which may not be committed, between the execution of the two queries. Unlike in the isolation levels explained in the previous section, the transaction does not wait until another transaction with an update lock is completed when a record is being updated. In this isolation level, the transaction reads a new record being updated by another transaction. (It reads updated but uncommitted values.) On the initial installation of CUBRID, the isolation level set by default in **`cubrid.conf`** is **`REPEATABLE READ CLASS`** and **`READ UNCOMMITTED INSTANCES`**.

The following are the rules of this isolation level:

- A transaction does not overwrite an object being modified by another transaction.
- A transaction does not commit an object until the end of the transaction.
- A transaction does not read a table being modified by another transaction. This rule does not apply to a record.
- Another transaction does not make any table currently being read or updated dirty until the end of the current transaction. This rule does not apply to a record.

This isolation level uses a two-phase locking for an exclusive lock. However, a shared lock cannot be obtained for a record. An intention lock on a table is maintained until the end of the transaction to prevent the table definition from being updated by another transaction (repeatable read on a table).

Example

In this example, one transaction inserts a record and updates a table while the other transaction selects the table in various aspects. Transaction T1 is at any level, and transaction T2 has the **REPEATABLE READ CLASS** with **READ UNCOMMITTED INSTANCES** isolation level. In this example, the participant2 table is created with the assumption that it did not exist before.

- Transaction T1 creates the participant2 table and inserts a record into the table. As soon as transaction T1 commits, CUBRID releases the lock held by transaction T1. Transaction T2 queries the participant2 table.
  - User1 (T1):

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ CLASS, READ UNCOMMITTED
INSTANCES;
;xrun
;autocommit off
create table participant2 (host_year integer, nation_code char(3));
insert into participant2 (host_year, nation_code) values (2008, 'AUS');
commit work;
;xrun
```

  - User2 (T2):

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ CLASS, READ UNCOMMITTED
INSTANCES;
;xrun
;autocommit off
select host_year, nation_code from participant2;
;xrun
=== <Result of SELECT Command in Line 2> ===
    host_year  nation_code
===================================
         2008  'AUS'
1 rows selected.
```

- Transaction T1 inserts another record into the participant2 table. Suppose that a new record (2012, 'KOR') added by transaction T1 is not stored before the query of transaction T2 is executed. This record cannot be seen while transaction T2 is executing the query.
  - User1 (T1):

```
insert into participant2 (host_year, nation_code) values (2012, 'KOR');
;xrun
```

  - User2 (T2):

```
select host_year, nation_code from participant2;
;xrun
=== <Result of SELECT Command in Line 2> ===
    host_year  nation_code
===================================
         2008  'AUS'
1 rows selected.
```

Note Unlike the isolation levels explained in the previous section, transaction T2 does not wait when it attempts to query the participant2 table again. Instead, transaction T2 can see committed and uncommitted records. The uncommitted records that transaction T2 sees depend on the output to the database. CUBRID flushes dirty instances of the workspace to the database in various situations. For more information, see the How to Handle Dirty Instances in CUBRID section.

- Then transaction executes the **SELECT** query on the participant2 table. During this transaction, the system flushes the uncommitted record (2012, 'KOR'). Transaction T2 sees the record when it re-execute the previous query.
  - User1 (T1):

```
select host_year, nation_code from participant2;
;xrun
=== <Result of SELECT Command in Line 1> ===
    host_year  nation_code
===================================
         2008  'AUS'
         2012  'KOR'
2 rows selected.
```

- User2 (T2):

```
select host_year, nation_code from participant2;
;xrun
=== <Result of SELECT Command in Line 2> ===
    host_year  nation_code
===================================
         2008  'AUS'
         2012  'KOR'
2 rows selected.
```

- Transaction T1 rolls back the update so that the last record is not committed. When transaction T2 re-executes the query, it cannot see the record (2012, 'KOR'). Note that the query results change due to non-repeatable reads on committed and uncommitted records.

  - User1 (T1):

```
rollback work;
;xrun
1 command(s) successfully processed.
```

  - User2 (T2):

```
select host_year, nation_code from participant2;
;xrun
=== <Result of SELECT Command in Line 1> ===
    host_year  nation_code
===================================
         2008  'AUS'
1 rows selected.
```

- Transaction T1 inserts another record into the participant2 table, and deletes the first record. Transaction T2 may still see the first record when it queries the participant2 table again, but in this example the delete operation has been applied to the database. As you can see, the inserted records are outputted. Note that both are dirty records.

  - User1 (T1):

```
insert into participant2 (host_year, nation_code) values (2012, 'KOR');
delete from participant2 where host_year=2008 and  nation_code='AUS';
;xrun
1 rows inserted.
1 rows deleted.
```

  - User2 (T2):

```
select host_year, nation_code from participant2;
;xrun
=== <Result of SELECT Command in Line 1> ===
    host_year  nation_code
===================================
         2012  'KOR'
1 rows selected.
```

- Transaction T1 commits, making changes permanent in the database. Transaction T2 queries the participant2 table again, and gets the same result as before.

  - User1 (T1):

```
commit work;
;xrun
1 command(s) successfully processed.
```

  - User2 (T2):

```
select host_year, nation_code from participant2;
;xrun
```

```
=== <Result of SELECT Command in Line 1> ===
    host_year  nation_code
==================================
         2012  'KOR'
1 rows selected.
```

- Transaction T1 tries to change the name of the location table to place, but aborts because transaction T2 sees the location table (repeatable schema). This happens because an intention shared lock on the table is held until the end of the transaction. Transaction T2 commits, and transaction T1 resumes. Transaction T2 tries to execute the query again but pauses because transaction T1 has not yet committed after renaming the table.
  - User1 (T1):

```
rename table participant2 as nation_medals;
;xrun
```

  - User2 (T2):

```
commit work;
select host_year, nation_code from participant2;
;xrun
```

- Once transaction T1 commits the operation, transaction T2 can query the table. Transaction T2 cannot proceed because the name of the location table has been changed to place. Instead, transaction T2 gets a syntax error saying that the location table does not exist any more.
  - User1 (T1):

```
commit work;
;xrun
```

  - User2 (T2):

```
In line 2, column 37,
ERROR: Class participant2 does not exist.
```

READ COMMITTED CLASS with READ COMMITTED INSTANCES

This isolation level does not allow a transaction to read an object (table or record) being updated by another transaction. However, it does not prevent tables and records read by the transaction from being updated by another transaction. The transaction might not be able to read the table or record twice (non-repeatable read). This means that the transaction may read two different committed values if it reads the same object (table or record) twice. Similarly, the results can be different if a single query is executed twice. The difference in the results is caused by another transaction inserting, updating or deleting records committed between the execution of the two queries, as well as the update of the queried table. The table being queried may not exist any more, or part of its attributes may have been added or deleted. This is not possible in the **REPEATABLE READ CLASS** isolation level because the transaction holds a lock on the table until the end.

The following are the rules of this isolation level:

- A transaction does not overwrite an object being modified by another transaction.
- A transaction does not commit an object until the end of the transaction.
- A transaction does not read an object being modified by another transaction.

This isolation level uses a two-phase locking protocol for an exclusive lock. The read lock on tables and records are released when the reading is completed.

Example

One transaction inserts a record into a table, deletes one from the table, and renames it while another transaction queries the table in various aspects. Transaction T1 is at any level, and transaction T2 has the **READ COMMITTED CLASS** with **READ COMMITTED INSTANCES** isolation level. In this example, the participant2 table is created with the assumption that it did not exist before.

- Transaction T1 creates the participant2 table and inserts a record into the table. As soon as transaction T1 commits, CUBRID releases the lock held by transaction T1. Transaction T2 queries the participant2 table.
- User1 (T1):

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED CLASS, READ COMMITTED
INSTANCES;
;xrun
;autocommit off
create table participant2 (host_year integer, nation_code char(3));
insert into participant2 (host_year, nation_code) values (2008, 'AUS');
commit work;
;xrun
1 rows inserted.
```

- User2 (T2):

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED CLASS, READ COMMITTED
INSTANCES;
;xrun
;autocommit off
select host_year, nation_code from participant2;
;xrun
=== <Result of SELECT Command in Line 2> ===
    host_year  nation_code
===================================
         2008  'AUS'
1 rows selected.
```

- Transaction T1 inserts another record into the participant2 table. Transaction T2 tries to query the participant2 table after a few seconds, but it pauses because transaction T1 holds a lock on the record of the participant2 table (new record).
- User1 (T1):

```
insert into participant2 (host_year, nation_code) values (2012, 'KOR');
;xrun
1 rows inserted.
```

- User2 (T2):

```
select host_year, nation_code from participant2;
;xrun
```

- Once transaction T1 inserts another record into the participant2 table and commits, all locks are released. Transaction T2 resumes and finds a new record.
- User1 (T1):

```
insert into participant2 (host_year, nation_code) values (2012, 'JPN');
commit work;
;xrun
1 rows inserted.
```

- User2 (T2):

```
=== <Result of SELECT Command in Line 1> ===
    host_year  nation_code
===================================
         2008  'AUS'
         2012  'KOR'
         2012  'JPN'
3 rows selected.
```

- Transaction T1 deletes the first record of the participant2 table. Transaction T2 tries to query the participant2 record, but it enters a standby state because transaction T1 is maintaining the lock on the deleted record and at least one intention exclusive lock on the table.
- User1 (T1):

```
delete from participant2
where host_year=2008;
;xrun
1 rows deleted.
```

- User2 (T2):

```
select * from participant2;
;xrun
```

- Transaction T1 commits an operation and releases the lock. Transaction T2 resumes and finds the results of the second and third **INSERT** statements.
- User1 (T1):

```
commit work;
;xrun
1 command(s) successfully processed.
```

- User2 (T2):

```
=== <Result of SELECT Command in Line 1> ===
    host_year  nation_code
==================================
         2012  'KOR'
         2012  'JPN'
2 rows selected.
```

- Transaction T1 succeeds when it tries to change the name of the participant2 table to nation_medals because transaction T2 does not have any read locks on any records or tables (non-repeatable schema). However, if transaction T2 tries to execute a write operation such as update or insert, transaction T1 enters a standby state because transaction T2 holds the lock until the end. When transaction T2 queries the table again, it pauses because transaction T1 is updating the table name. Note that transaction T2 can see only the committed updates. In this case, **RENAME** is committed or canceled.
- User1 (T1):

```
rename table participant2 as nation_medals;
;xrun
1 command(s) successfully processed.
```

- User2 (T2):

```
commit work;
select * from participant2;
;xrun
```

- Transaction T1 commits an operation, and the lock is released. Transaction T2 re-executes the query. However, it does not proceed because the name of the location table has been changed to place. Instead, transaction T2 receives a syntax error message saying that the location table does not exist.
- User1 (T1):

```
ccommit work;
;xrun
1 command(s) successfully processed.
```

- User2 (T2):

```
In line 2, column 2,
ERROR: Unknown class "participant2".
1 command(s) successfully processed.
```

READ COMMITTED CLASS with READ UNCOMMITTED INSTANCES

This isolation level does not allow a transaction to read a table being modified by another transaction, but allows it to read a dirty record that may be updated or rolled back later. The transaction might not be able to read the table or record twice (non-repeatable read). This means that the transaction may read two different committed values if it reads the same table twice. This also means that the transaction may read two different values, committed or not, if it reads the same record twice. An uncommitted record can be rolled back by the transaction which updates them because of another transaction updating the object between the two reads. Similarly, the results can be different if a single query is executed twice. This situation is caused by another transaction inserting records, committed or not, between the execution of the two queries. The table being queried may not exist any more, or part of its attributes may have been added or deleted. Updating tables is not possible in the **REPEATABLE READ** isolation level because the transaction holds a lock on the table until the end.

The following are the rules of this isolation level:

- A transaction does not overwrite an object being modified by another transaction.
- A transaction does not commit an object until the end of the transaction.
- A transaction does not read a table being modified by another transaction. This rule does not apply to a record.

This isolation level uses a two-phase locking protocol for an exclusive lock. A shared lock cannot be obtained for a record, and the shared lock or intention shared lock on a table is released as soon as the table is read.

Example

One transaction inserts a record and updates a table while another transaction queries the table in various aspects. Transaction T1 is at any level, and transaction T2 has the **READ COMMITTED CLASS** with **READ UNCOMMITTED INSTANCES** isolation level. In this example, the participant2 table is created with the assumption that it did not exist before.

- Transaction T1 creates the participant2 table and inserts a record into the table. As soon as transaction T1 commits, CUBRID releases the lock held by transaction T1. Transaction T2 queries the participant2 table.
- User1 (T1):

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED CLASS, READ UNCOMMITTED
INSTANCES;
;xrun
;autocommit off
create table participant2 (host_year integer, nation_code char(3));
insert into participant2 (host_year, nation_code) values (2008, 'AUS');
commit work;
;xrun
1 rows inserted.
```

- User2 (T2):

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED CLASS, READ UNCOMMITTED
INSTANCES;
;xrun
;autocommit off
select host_year, nation_code from participant2;
;xrun
=== <Result of SELECT Command in Line 2> ===
    host_year  nation_code
===================================
         2008  'AUS'
1 rows selected.
```

- Transaction T1 inserts another record into the participant2 table. Transaction T2 sees only the first record when it queries the participant2 table because the second record has not been outputted in the database in this example.
- User1 (T1):

```
insert into participant2 (host_year, nation_code) values (2004, 'FRA');
;xrun
1 rows inserted.
```

- User2 (T2):

```
select * from participant2;
;xrun
=== <Result of SELECT Command in Line 1> ===
    host_year  nation_code
===================================
         2008  'AUS'
1 rows selected.
1 command(s) successfully processed.
```

- Transaction T1 updates the participant2 table and adds the gold column. Then if transaction T2 queries the table, it enters into a standby state because of the **ALTER** statement. This can also happen in other isolation levels because the transaction maintains a lock on the table until the end.
- User1 (T1):

```
alter table participant2
add attribute gold integer;
;xrun
1 command(s) successfully processed.
```

- User2 (T2):

```
select * from participant2;
;xrun
```

- Transaction T1 inserts another record with the gold column into the participant2 table. Transaction T1 commits the operation and transaction T2 resumes. Note that transaction T2 will see a totally different schema (table).
- User1 (T1):

```
insert into participant2 (host_year, nation_code, gold)
values (2012, 'KOR', 20);
commit work;
;xrun
1 command(s) successfully processed.
```

- User2 (T2):

```
=== <Result of SELECT Command in Line 1> ===
    host_year  nation_code                gold
=======================================
         2008  'AUS'                      NULL
         2004  'FRA'                      NULL
         2012  'KOR'                        20
3 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.

Unsupported Combinations of Isolation Levels

The **SET TRANSACTION ISOLATION LEVEL** statement does not allow the following combinations of isolation levels even though it could define other combinations. The system sends a warning message and selects an isolation level closest to the one specified. The following are unsupported isolation levels:

- **READ COMMITTED CLASS** with **REPEATABLE READ INSTANCES**
- **READ UNCOMMITTED CLASS** with **REPEATABLE READ INSTANCES**

If the structure of a table is updated, reading records of the table is not repeatable. Therefore, the above combinations are not appropriate.

- **READ UNCOMMITTED CLASS** with **READ COMMITTED INSTANCES**

- **READ UNCOMMITTED CLASS** with **READ UNCOMMITTED INSTANCES**

The system is unable to interpret the above isolation levels and may fail. For example, many problems will occur if a transaction queries a table and another transaction deletes the same table in the middle of the query execution.

How to Handle Dirty Instances

Dirty instances are flushed from the client workspace to the database server in the following cases. And they can also be flushed in some other cases as well.

- All dirty instances are flushed during the commit.
- Some dirty instances are flushed when the client workspace is over-loaded with many instances.
- Dirty instances of a table are flushed when the table schema is changed.
- Dirty instances of a table are flushed when the table is queried.
- Some dirty instances can be flushed when a server function is called.

Restore

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 a 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 How to Use the CUBRID Backup and Restore Utilities.

Trigger

Creating Triggers

Guidelines 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.

CREATE TRIGGER (Syntax)

Description

A trigger is created by defining a trigger target, condition and action to be performed in the **CREATE TRIGGER** statement.

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 CLASS attribute_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 Types" 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 Targets" 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

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 PDATE 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 Types

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 Types

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

- **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 Targets

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)
...
```

Combinations 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 attribute 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

- **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 Names

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.

Changing Triggers

Renaming Triggers

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.

Altering Trigger Definition

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.

Dropping Triggers

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.

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 Logs

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 Triggers

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".
```

Deferred Actions/Conditions

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 Actions/Conditions

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 Actions/Conditions

Description

Cancels 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.

Applications Using Triggers

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.

Triggers created by the user in the own database can be as powerful as applications created by the user.

Triggers Created in the Demo Database

This section explains triggers created by using the tables in the demo database. 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.

- **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.

  ```
  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 does not have to start at the exact time of the update execution.

```
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.

Methods

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 Types

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 superclass. The subclass has only the name of a class or instance method definition inherited from the superclass.

The rules for resolving method name conflicts are same as those for attribute name conflicts. For more information about attribute/method inheritance conflicts, see Inheriting Attributes and Methods in Class Inheritance.

Calling Methods

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.

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.

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
• See Expressions for additional expression syntax
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.

glo Classes

Overview

A **glo** (Generalized Large Object) class is used to handle large-sized objects. This class works as a base for multimedia support in CUBRID. If you define an attribute of a user-defined class as a **glo** class (or its subclass), you can store unstructured data of other types. Bitmap, digital audio data, large document data and binary data are examples of an instance of a **glo** class (or its subclass).

Instances of the **glo** class can be divided into two types. The first type is LO (Large Object) that can be accessed or stored to the database only by CUBRID. The second type is FBO (File Based Object) that can be stored in the host file system. Most **glo** instances are LO because it is more convenient for CUBRID to manage data. However, if data is used in a non-database application, there are advantages, for example, not having to compress data by allowing the host file system to manage the FBO data storage.

Attributes Defined in Classes

All attributes in a **glo** class are pre-defined by CUBRID. The value related to each attribute is defined when one or more **glo** methods are called. Therefore, multimedia data cannot be entered by the same way general data is entered into a user-defined class, but is linked to the related **glo** instance by a dedicated **glo** method.

Attributes defined by the **glo** class are listed in the table below.

glo Attribute	Attribute Type	Definition
unit_size	integer	Represents the number of bits per each **glo** data unit. Generally, **8** is used. The maximum number of bits allowed for a data unit is 2,147,483,647.
header_size	integer	Represents the size of the unit that configures the header (starting from position 0) away from the **glo** data. The default value is **0**.
holder_obj	glo_holder	An inner class cannot be accessed by general users.

The following figure shows an example of a **glo** instance and its related values. The value of the **unit_size** attribute of this instance is the default value. However, the value of **header_size** is determined by the **glo** method.



Methods Defined in Classes

- Description
- glo Class Method

- glo Instance Method

Description

You can use **glo** classes by calling them with the **CALL METHOD** statement. Class methods and instance methods operate on the **glo** class.

These two types of methods defined on the **glo** class are used to operate class objects rather than class instances.

The **glo** instance method provides access and update operations so that the **glo** data can be manipulated. Most instance methods follow the programming language interfaces such as Java and C because arguments of the method (e.g. data buffer, etc.) cannot be specified in the CSQL processor.

glo Class Method

new_fbo Class Method

This method creates the File Based Object (FBO) type of **glo** instance. The argument value is given as a single string for the file path name. If the **new_fbo**( ) method is executed successfully, a new instance of the requested class is returned. The syntax for the **new_fbo**( ) method to create a FBO is as follows:

```
CALL new_fbo(:path name)
    ON CLASS glo INTO :write_object;
```

You can also use an environment variable as the file path name for the **new_fbo**( ) class method. This can be used to allow clients with different file systems to access the same FBO.

For example, if the following FBO file path is given, one user can configure FBO_ROOT to”/usr/local” and other user can configure it to ”/net/edsel/usr/local”.

```
csql>CALL new_fbo('$FBO_ROOT/test/fbo1.txt')
            ON CLASS glo INTO :fbo;
```

The file path can also be modified in the CUBRID environment variable **CUBRID_FBO_PREFIX**. Once the **CUBRID_FBO_PREFIX** variable is configured to a value, the value prefixes every FBO file path name when the file is accessed. However, the variable value is not prefixed to a file path name which already contains an environment variable reference in the file path name.

In the following example, the path name to access the FBO file is "h:/remote1/usr/local/test/fbo1.txt."

```
%setenv CUBRID_FBO_PREFIX ”h:/remote1”
csql>CALL new_fbo('/usr/local/test/fbo1.txt')
            ON CLASS glo INTO :fbo;
```

If the NFS is mounted to a root file system or a specific sub directory and the file path names are the same, both UNIX and PC clients can access the contents of the FBO. In the example above, the **CUBRID_FBO_PREFIX** variable value would have no value on a UNIX system, and the NFS volume identifier, ”h:/remote1” should be specified on a PC client.

new_lo Class Method

This method creates the Large Object (LO) of **glo** instance. The method has no argument. The **new_lo**( ) method returns a new instance of the desired class.

```
csql>CALL new_lo()ON CLASS glo INTO :labeled_object;
```

new_lo_import( ) Class Method

This method creates a new LO instance and initializes its contents from an external file. If the argument of the method is **NULL**, an empty LO is created. If it is a character string containing a file name, a new instance is created and then the contents of the file are copied.

```
csql>CALL new_lo_import('./import_data.txt')
            ON CLASS glo INTO :labeled_object;
```

glo Instance Method

initialize_data( ) Method

This method is automatically called by the **new_lo**( ) / **new_fbo**( ) class method every time a new instance of the **glo** class is created. The attribute is configured in the **glo** class to the default value. Any class that inherits from the **glo** class can redefine the **initialize_data**( ) method to configure other default attributes, or to perform other operations. This method is ignored without argument because it is not normally called by users.

data_pos( ) Method

This method returns the starting position in the **glo** data area used by the **glo** instance method. The following returns the current position from the **glo** instance.

```
csql>CALL data_pos() ON :labeled_object INTO :location;
```

data_seek( ) Method

This method configures the current position to the value in the first argument. If the **data_seek**( ) method is executed successfully, a new position is returned. The following is an example of configuring the current position in the **glo** data area to 42 by using the **data_seek**( ) method.

```
csql>CALL data_seek(42) ON :labeled_object;
```

read_data( ) Method

This method transfers data from the **glo** to the user data buffer. The data is read starting from the current position in the **glo** data. The **read_data**( ) method requires two arguments. The first argument is a constant that specifies the number of units to be read from the **glo**, and the second argument is a character string that specifies the buffer to store the data. If the method is executed successfully, the number of actual read units and the data read from the **glo** are returned. The current position is automatically updated. The following is an example of a C API that uses the **read_data**( ) method. It reads 1024 units starting from the current position in the **glo** data, and transfers the data to a string array buffer.

```
...
DB_INT32 rc, length = 1024;
DB_OBJECT *read_object;
char buffer[1024];
DB_VALUE val1, val2, val3;
...
DB_MAKE_INTEGER(&val1, length);
DB_MAKE_VARCHAR(&val2, DB_MAX_VARCHAR_PRECISION, buffer, length);
rc = db_send(read_object, GLO_METHOD_READ, &val3, &val1, &val2);
...
```

Note The **read_data**( ) method is not supported in CSQL because its return value must be saved in a character string buffer. Therefore, it can be used only through JDBC or C APIs.

---

write_data( ) Method

This method transfers data from the user data buffer to the **glo**. The write operation is performed by overwriting starting from the current position in the **glo** data, if data exists. The **write_data**( ) method requires two arguments. The first argument is the number of units to be saved in the **glo** (integer), and the second argument is the buffer (character string) that contains the data to be saved. It returns the number of units that have been actually saved, and the current position in the **glo** data is automatically updated. The following is an example of using the **write_data**( ) method. A LO is created and the data is stored in this example.

```
csql>CALL new_lo() ON CLASS glo INTO :wr_lo
csql>CALL write_data(8,'raw data') TO :wr_lo;
```

insert_data( ) Method

This method inserts data from the user data buffer into the **glo**. The data is inserted starting from the position defined by the current position. The **insert_data**( ) method requires two arguments. The first argument is an integer that corresponds to the unit to be inserted into the **glo**, and the second argument is the buffer that contains the data to be inserted. The current position is automatically updated to reflect the position where the next operation begins. If **insert_data**( ) is executed successfully, the amount of data inserted into the **glo** is returned.

```
csql>CALL insert_data(3,'end') ON :wr_lo;
```

delete_data( ) Method

This method deletes the **glo** data that starts from the current position. The method deletes the number of units specified by the first argument. If the units are successfully deleted, the number of deleted units is returned. If the number of units to be deleted exceeds the end of the **glo** data, the system performs truncation and returns the number of units actually deleted. The current position does not change when the data is deleted. The number of bytes allocated is displayed when **delete_data**( ) is called after **data_size**( ) is called. After multiple delete and insert operations, the **compress_data**( ) method must be called to defragment the **glo**.

```
csql>CALL delete_data(3) ON :wr_lo;
```

truncate_data( ) Method

This method removes all data starting from the current position to the end of the **glo** data. No arguments are required for the **truncate_data**( ) method. If the truncation is executed successfully, the number of actually truncated units is returned.

```
csql> call truncate_data() on :wr_lo into :num_deleted
```

append_data( ) Method

This method appends data to the end of the **glo** data. The method requires two arguments. The first argument specifies the number of units to be appended. The second argument is the buffer that has the data to be appended. If the append operation is executed successfully, the number of appended units is returned. The current position is updated automatically to the end of the appended **glo** data.

```
csql>call append_data( 14, 'appended_data' ) on :wr_lo
```

data_size( ) Method

This method returns the total number of units contained in the **glo**. The amount or size of data can change because update methods allow you to add or subtract data from the **glo**. The following is an example of using this method. The total number of units is saved to the **total_units** variable and returned.

```
csql>call data_size () on :wr_lo
```

compress_data( ) Method

This method repacks LO data to be more space-efficient. It is recommended to call the **compress_data**( ) method after executing multiple **delete**( ) or **insert**( ) methods. If the method is executed successfully, 0 is returned; if it fails, a value less than 0 is returned.

```
csql>CALL compress_data() ON :wr_lo;
```

The **glo** must be defragmented by executing **compress_data**( ) after multiple delete and insert operations are performed.

destroy_data( ) Method

This method removes the **glo** data. The method is completed after the **size** attribute, the current position, and the **header_size** attributes are all set to 0. If the method is executed successfully, 0 is returned; otherwise, -1 is returned.

```
csql>CALL destroy_data() ON :wr_lo;
```

copy_from( ) Method

This method copies data from the data source (file or another **glo** instance) to a **glo**. For the argument of the **copy_from**( ) method, you can use a character string that contains the file path name or another **glo** instance. If the method is executed successfully, the existing **glo** instance is overwritten and the size of the new **glo** is returned.

```
csql>CALL new_lo() ON CLASS glo INTO :lob;
csql>CALL copy_from('test.txt')
            ON :lob INTO :size;
```

Note When the **glo** is overwritten by a new file that contains smaller amount of data, the resulting **glo** object retains its original size unless **truncate_data**( ) is not called to delete unwanted data at the end of the **glo**.

---

copy_to( ) Method

This method copies data from the **glo** data to a file or another **glo**. The **copy_to**( ) method requires a character string representing the file path name or another **glo** instance as its argument. If the method is executed successfully, the size of the copied data is returned.

```
csql>CALL copy_to('extract.txt') ON :lob  INTO :size_value;
```

glo_pathname( ) Method

This method returns the path name if the **glo** instance is an FBO. If the **glo** instance is an LO, **NULL** is returned. No arguments are required for this method. In the following example, a file path name is returned if the **glo** is an FBO, and **NULL** is stored in the **file_name** variable and returned if it is an LO.

```
csql> CALL glo_pathname() ON :fbo INTO :file_name;
```

get_error( ) Method

This method returns the error conditions when an error occurs. The following are error values returned by **get_error**( ).

```
INVALID_STRING_INPUT_ARGUMENT -2    /* bad input argument   */
INVALID_INTEGER_INPUT_ARGUMENT -3   /* bad input argument   */
INVALID_STRING_OR_OBJ_ARGUMENT -4   /* bad input argument   */
INVALID_OBJECT_INPUT_ARGUMENT -5    /* bad input argument   */
UNABLE_TO_FIND_GLO_STRUCTURE -6     /* internal error       */
COULD_NOT_ACQUIRE_WRITE_LOCK -7     /* object already locked*/
ERROR_DURING_TRUNCATION -8          /* truncation error     */
ERROR_DURING_DELETE -9              /* delete error         */
ERROR_DURING_INSERT -10             /* insert error         */
ERROR_DURING_WRITE -11              /* write error          */
ERROR_DURING_READ -12               /* read error           */
ERROR_DURING_SEEK -13               /* seek error           */
ERROR_DURING_APPEND -14             /* append error         */
ERROR_DURING_MIGRATE -15            /* migrate error        */
COPY_TO_ERROR -16                   /* copy to error        */
COPY_FROM_ERROR -17                 /* copy from error      */
COULD_NOT_ALLOCATE_SEARCH_BUFFERS -18      /* search error  */
COULD_NOT_COMPILE_REGULAR_EXPRESSION -19   /* reg exp error */
COULD_NOT_RESET_WORKING_BUFFER -20         /* search error  */
```

The following is an example of the **get_error**( ) method.

```
csql>CALL get_error() ON :fbo INTO :error_value;
```

set_error( ) Method

If you expand the class hierarchy of the **glo**, the error value returned by the **get_error**( ) method can be added by using the **set_error**( ) method.

```
csql> CALL set_error(:user_method_error) ON :fbo;
```

Concurrency

Concurrency rules that apply to **glo** instances are the same way as other rules apply to other CUBRID class instances. Many users may read **glo** data concurrently by sharing read locks on the **glo** instance. A read lock can be granted only if other users do not have a write lock on the given **glo** instance. If no other users have a read or write lock on a certain instance, only one user may own a read lock to update the **glo** data during a transaction.

A read and write lock on the **glo** data applies to all **glo**s. A partial lock on the **glo** data is not supported.

A special concurrency condition applies to FBOs. The CUBRID database system allows the creation of **glo** instances that refer to the same file.

If all **glo** instances are created in the same path name, the same concurrency rules apply. However, if two **glo** instances using an absolute path name and another using a relative path name refer to the same file, consistent update access between these two instances cannot be guaranteed. Likewise, if **glo** instances using an absolute path name and another that specifies a link refer to the same file, consistent update access between two instances cannot be guaranteed.

CUBRID supports concurrency control for indirect access of FBO data through the database. However, direct access of FBO by outside users or applications cannot be prevented.

Restore

glo restore rules are mandatory as for other class instance. If a transaction is rolled back, any updates to **glo** (LOs and FBOs) are rolled back. When a transaction is committed, all updated **glo**s are permanently stored.

If a database crashes during the execution of **ROLLBACK WORK** or **COMMIT WORK** statement, the restore procedures for **glo**s are enforced when the database restarts. The restore procedures for LOs are performed internally by the database system when the database restarts. The restore procedures for FBO updates are completed next time the user who updated FBO when the system crashed reconnects to the database. Read and update locks on the **glo** are owned by the database until the restore is completed.

To support the FBO restore, the database uses the temporary shadow file for updates to FBOs. This means that the user who updates FBO must have update authorization on the directory in which FBO exists (in addition to enough space to create a working copy of FBO data). An update operation on FBO instance may fail if the user does not have write authorization on the directory where FBO is located. Likewise, an update operation may fail if the user does not have database authorization by the database to update instances of the **glo** classes or subclasses.

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 Partitions

Defining Range Partitions

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 Expressions."
- *partition_name* : Specifies the partition name.
- *range_value* : Specifies the partition-by value.

Example

- **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).

Redefining Range Partitions

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

- **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 Partitions

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 "Redefining Range Partitions").

Dropping Range Partitions

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 "Redefining Range Partitions").
- 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 Partitions

Defining Hash Partitions

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

- **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.

Redefining Hash Partitions

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 Partitions" for more information).
- There must be at least one partition remaining after redefining partitions.

List Partitions

Defining List Partitions

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 Expressions."
- *partition_name* : Specifies the partition name.
- *partition_value_list* : Specifies the list of the partition by values.

Example

- **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.

Redefining List Partitions

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 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

- **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 Partitions

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;
```

Partition Management

Retrieving and Manipulating Data in Partitions

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 Partition Key Values

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 updated.
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 Tables into Partitioned Tables

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 Partitioned Tables into Regular Tables

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

- **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';
```

  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 Partition Expressions

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
NCHARN	DOUBLE
CHAR	BIT
VARYING	BIT VARYING
INTEGER	NUMERIC OR DECIMAL
SMALLINT	MONETARY
DATE	SET
TIME	LIST OR SEQUENCE
TIMESTAMP	MULTISET
	OBJECT

The following operator functions can be used in partition expressions to be applied to partition keys.

Number Operation Functions			
+	-	*	/
MOD	STRCAT	FLOOR	CEIL
POWER	ROUND	ABS	TRUNC
String Operation Functions			
POSITION	SUBSTRING	OCTEC_LENGTH	BIT_LENGTH
CHAR_LENGTH	LOWER	UPPER	TRIM
LTRIM	RTRIM	LPAD	RPAD
REPLACE	TRANSLATE		
Date Operation Functions			
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 by Using Partitioned Tables

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 Partitioned 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 classes.

The following is an example of the **ANALYZE PARTITION** statement.

```
ALTER TABLE t1 ANALYZE PARTITION p3;
```

Serials

Creating Serials

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 a multi-user environment.
- Generated serial numbers are not related with table so, you can use the same serial in multiple tables.

Description

You can create a serial object in the database by using the **CREATE SERIAL** statement.

Syntax

```
CREATE SERIAL serial_identifier
[ START WITH initial ]
[ INCREMENT BY interval]
[ MINVALUE min | NOMINVALUE ]
[ MAXVALUE max | NOMAXVALUE ]
[ CYCLE | NOCYCLE ]
```

- *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 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; $-(10)^{36}$ 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. 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**.

Example

- **Example 1**

  The following is an example of creating the serial object *order_no* whose initial, increment and maximum values are specified as 1, 1, and 10^{37} respectively.

  ```
  CREATE SERIAL order_no;
  ```

- **Example 2**

  The following is an example of creating the serial object *idx_no* whose initial, increment and maximum values are specified as 10,000, 2, and 20,000 respectively.

  ```
  CREATE SERIAL idx_no START WITH 10000 INCREMENT BY 2 MAXVALUE 20000;
  ```

Accessing and Updating Serials

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 of using serial.
First, the athlete_idx table which includes athlete codes and names is created, and one instance created by the *order_no* serial is inserted.

```
CREATE TABLE athlete_idx
(code INT, name VARCHAR(40) );

INSERT INTO athlete_idx VALUES (order_no.NEXT_VALUE, 'Park Tae-Hwan');
```

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.

Altering Serials

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 ]
[ CYCLE | NOCYCLE ]
```

- *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**.

Caution To change the initial value of Serial, you cannot modify the db_serial table because that is a system catalog. Therefore, the file exported (unloaded) from later than CUBRID 2008 R2.0 cannot be imported (loaddb) in 2008 R1.x.

Example

The following is an example of altering the increment of the *order_no* serial to 2.

```
ALTER SERIAL order_no INCREMENT BY 2;
```

Dropping Serials

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;
```

Authorizing Access to Serial

Any user including **PUBLIC** can create a serial object. Once a serial is created, all users can retrieve the serial number by using **CURRENT_VALUE** or **NEXT_VALUE**. Only the owner and **DBA** can alter or drop the serial object. However, if the owner is **PUBLIC**, anyone can alter or drop the serial.

Syntax for Granting Access Authorization

Database Users

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 Users

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

- **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:

Operations	Granted Authorization
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 create an index for an attribute to improve the search speed.
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

- **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

- **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 revoked 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;
```

Java Stored Functions/Procedures

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 Functions/Procedures

To use Java stored functions/procedures in CUBRID, you must have Java installed in the environment where CUBRID is installed. By default, CUBRID locates and runs SUN's Java Virtual Machine. To run Java stored functions/procedures in CUBRID, you must have a Java Virtual Machine implementing JNI version 1.4 or higher. Furthermore, you should check if the version of CUBRID is correspondingly matched to that of the Java Virtual Machine. That is, if the 64-bit CUBRID 2008 R2.x is running on the 32-bit Java environment, an error message appears, meaning that the JVM libraries can not be found.

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. The following is an example of adding the path of the **jvm.dll** file to the **PATH** environment variable.

```
set JAVA_HOME=C:\jdk1.5.0
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. The following is an example of setting up the environment variable in the configuration file (e.g. **.profile**, **.cshrc**, **.bashrc**, **.bash_profile**, etc.). For Linux, only glibc 2.3.4 or higher is supported.

- For bash

```
JAVA_HOME=/home/cubrid/jdk1.5.0_10 ;export JAVA_HOME
PATH=.:$JAVA_HOME/bin:$PATH ;export PATH
LANG=korean ;export LANG
```

```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$JAVA_HOME/jre/lib/i386:$JAVA_HOME/jre/lib/i
386/client ;export LD_LIBRARY_PATH
```

- For csh

```
setenv JAVA_HOME $HOME/jdk1.5.0_10
setenv LD_LIBRARY_PATH $HOME/jdk1.5.0_10/jre/lib/i386:
$HOME/jdk1.5.0_10/jre/lib/i386/client:$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 Functions/Procedures

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 Internal JDBC Driver on Server-Side

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 Databases

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).

Registering Loaded Java Classes

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, OUBLE,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''SpCubrid.HelloCubrid() return java.lang.String'  'DBA'

'sp_int'              'FUNCTION'           'INTEGER'                       1
  'JAVA''SpCubrid.SpInt(int) return int'  'DBA'

'athlete_add'         'PROCEDURE'          'void'                          4
  'JAVA''Athlete.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_name*s or *procedure_name*s 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 Phone_Info
```

Calling Java Stored Functions/Procedures

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 Statements

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 Applications

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());
  }
}
```

List of SQL Statements

List of SQL Statements

List of SQL Statements

```
<statement>     <create statement>
                      |<alter statement>
                      |<drop statement>
                      |<insert statement>
                      |<delete statement>
                      |<update statement>
                      |<rename statement>
                      |<call statement>
                      |<evaluate statement>
                      |<execute statement>
                      |<query statement>
                      |<authorization statement>
                      |<commit statement>
                      |<rollback statement>
                      |<savepoint statement>
                      |<transaction statement>
                      |<trigger statement>
                      |<optimization statement>
                      |<update statistics statement>
                      |<user statement>
                      |<index statement>
                      |<partition statement>
```

CREATE Statement

<create class statement>

```
CREATE { CLASS | TABLE } <class name>
[ <subclass definition> ]
[ ( <class element comma list> ) ]
[ CLASS ATTRIBUTE ( <attribute definition comma list> ) ]
[ ( <class constraint or attribute definition comma list> ) ]
```

<create virtual class statement>

```
CREATE { VCLASS | VIEW } <class name>
[ <subclass definition> ]
[ CLASS ATTRIBUTE ( <attribute definition comma list> ) ]
[ ( <view attribute definition comma list> ) ]
[ INHERIT <resolution comma list> ]
```

```
[ AS <query statement> ]
[ WITH CHECK OPTION ]
```

<create trigger statement>

```
CREATE TRIGGER <trigger name>
[ STATUS { ACTIVE | INACTIVE } ]
[ PRIORITY <trigger key> ]
<trigger event time> <event specification>
[ IF <trigger condition> ]
EXECUTE [ <trigger action time> ] <trigger action>
```

<create serial statement>

```
CREATE SERIAL <serial name>
[ START WITH <integer literal> ]
```

<subclass definition>

```
AS SUBCLASS OF <class name comma list>
```

<class constraint or attribute definition>

```
{ <class constraint definition> | <attribute definition> }
```

<class constraint definition>

```
[ CONSTRAINT <constraint name> ] UNIQUE ( <attribute name comma list> )
| [ PRIMARY KEY ( <attribute name comma list> ) ]
| [referential constraint]
```

<attribute definition>

```
<general attribute name> <attribute type>
[ <default or shared> ]
[ <attribute constraint list> ]
```

<view attribute definition>

```
<attribute definition> | <attribute name>
```

<class element>

```
<attribute definition> | <class constraint>
```

<resolution comma list>

```
<resolution> [ { , <resolution> }... ]
```

<resolution>

```
<general attribute name> OF <class name> [ AS <attribute name> ]
```

<view attribute definition comma list>

```
<view attribute definition> [ { , <view attribute definition> }... ]
```

<view attribute definition>

```
<attribute definition> | <attribute name>
```

<trigger key>

```
<unsigned real literal>
```

<trigger event time>

```
BEFORE | AFTER | DEFERRED
```

<event specification>

```
INSERT ON <event class target>
| STATEMENT INSERT ON <event class target>
| UPDATE ON { <event class target> | <event class attribute target> }
| STATEMENT UPDATE ON { <event class target> | <event class attribute
target> }
| DELETE ON <event class target>
| STATEMENT DELETE ON <event class target>
| COMMIT
| ROLLBACK
```

<event class target>

```
<class name>
```

<event class attribute target>

```
<class name> ( <attribute name> )
```

<trigger condition>

```
<search condition>
```

<trigger action time>

```
AFTER | DEFERRED
```

<trigger action>

```
REJECT
| INVALIDATE TRANSACTION
| PRINT <string literal>
| <call statement>
| <insert statement>
| <delete statement>
| <update statement>
```

<general attribute name>

```
[ CLASS ] <attribute name>
```

<general method name>

```
[ CLASS ] <method name>
```

<attribute type>

```
<domain name>
```

<argument type>

```
<domain name>
```

<result type>

```
<domain name>
```

<referential constraint>

```
FOREIGN KEY
[constraint name] (attribute name comma list)
REFERENCES
[referenced table name] (attribute name comma list)
[ <referential triggered action> ]
```

<referential triggered action>

```
<update rule>
[ <delete rule> [ <cache object rule> ] ]
```

<update rule>

```
ON UPDATE <referential action>
```

<delete rule>

```
ON DELETE <referential action>
```

<cache object rule>

```
ON CACHE OBJECT <cache object column name>
```

<cache object column name>

```
<attribute name>
```

<referential action>

```
CASCADE | RESTRICT | NO ACTION
```

Example

```
CREATE CLASS address (street string, city string);
CREATE CLASS person (
name string,
birthday date,
residence address,
UNIQUE (name))
METHOD get_age () integer
FILE "/p/xsql/current/bin/person.o";
CREATE CLASS album
(id char(10) NOT NULL PRIMARY KEY,
title varchar(100),
artist varchar(100)
);
CREATE CLASS track
(album char(10),
dsk integer,
posn integer,
song varchar(255),
FOREIGN KEY (album) REFERENCES album(id)
);
```

ALTER Statement

<alter statement>

```
<alter class statement>
| <alter trigger statement>
| <alter serial statement>
```

<alter class statement>

```
ALTER [ <class type> ] <class name> <alter clause>
```

<alter clause>

```
ADD <alter add> [ INHERIT <resolution comma list> ]
| DROP <alter drop> [ INHERIT <resolution comma list> ]
| CHANGE <alter change>
| RENAME <alter rename> [ INHERIT <resolution comma list> ]
```

<alter add>

```
CLASS ATTRIBUTE <attribute definition comma list>
| [ ATTRIBUTE | COLUMN ] <class constraint or attribute definition comma
list>
| SUPERCLASS <class name comma list>
| QUERY <query statement>
```

<alter drop>

```
[ ATTRIBUTE | COLUMN ] <attribute name comma list>
| CONSTRAINT <constraint name>
| SUPERCLASS <class name comma list>
```

<alter change>

```
<attribute default comma list>
| QUERY [ <unsigned integer literal> ] <query statement>
```

<attribute default>

```
[ CLASS ] <attribute name> DEFAULT <value specification>
```

<alter rename>

```
[ ATTRIBUTE | COLUMN ] [ CLASS ] <attribute name> AS <attribute name>
```

<alter trigger statement>

```
ALTER TRIGGER <trigger name> <trigger status or priority>
```

<trigger status or priority>

```
STATUS { ACTIVE | INACTIVE }
| PRIORITY <trigger key>
```

<alter serial statement>

```
ALTER SERIAL <serial name> [ INCREMENT BY <integer literal> ] [ START WITH
<integer literal> ]
[ MINVALUE <integer literal> | NOMINVALUE]
[ MAXVALUE <integer literal> | MAXVALUE] [ CYCLE | NOCYCLE ]
```

Example

```
ALTER CLASS procedings ADD SUPERCLASS reports
INHERIT editor OF reports;
ALTER CLASS employee ADD ATTRIBUTE
name string,
age integer DEFAULT 20;
ALTER CLASS employee DROP name, age;
ALTER CLASS employee CHANGE age DEFAULT 30;
ALTER CLASS employee DROP CONSTRAINT unique_age;
```

DROP Statement

<drop statement>

```
<drop class statement>
| <drop trigger statement>
| <drop deferred trigger statement>
| <drop serial statement>
| <drop index statement>
| <drop variable statement>
```

<drop class statement>

```
DROP [ <class type> ] <class specification comma list>
```

<drop trigger statement>

```
DROP TRIGGER <trigger name comma list>
```

<drop deferred trigger statement>

```
DROP DEFERRED TRIGGER <trigger spec>
```

<drop serial statement>

```
DROP SERIAL <serial name>
```

<drop index statement>

```
DROP [REVERSE] [UNIQUE] INDEX [index name] ON <class name>
| ( <attribute name> )
```

<drop variable statement>

```
DROP VARIABLE <variable comma list>
```

<trigger spec>

```
<trigger name comma list>
| ALL TRIGGERS
```

<class specification>

```
<class hierarchy>
| ( <class hierarchy comma list> )
```

<class hierarchy>

```
[ONLY] <class name>
|ALL <class name> [EXCEPT <class specification> ]
```

Example

```
DROP person, employee;
DROP INDEX ON employee (name);
DROP INDEX ON employee (ssn, name);
DROP TRIGGER check_salary;
DROP DEFERRED TRIGGER check_salary, check_age;
DROP DEFERRED TRIGGER ALL TRIGGERS;
```

INSERT Statement

<insert statement>

```
INSERT INTO [ ONLY ] <class name> [ ( [ <attribute name comma list> ] ) ]
| <insert statement value clause>
```

<insert statement value clause>

```
VALUES ( <insert value comma list> )
| <query statement>
| DEFAULT [ VALUES ]
```

<insert value comma list>

```
<insert value> [ { , <insert value> }... ]
```

<insert value>

```
<expression>
| <query specification>
```

Example

```
INSERT INTO person(name, age, residence)
VALUES ('Amy', 20, (INSERT INTO address(street, city)
VALUES ('1 Wolf St. ', 'Austin')));
```

DELETE Statement

<delete statement>

```
DELETE FROM { <class specification> | <meta class specification> }
[ WHERE <search condition> ]
[ <using index clause> ]
```

Example

```
DELETE FROM employee;
DELETE FROM ONLY hotel WHERE name = 'Hilton';
DELETE FROM location
WHERE lodging IN (SELECT name FROM resort);
```

UPDATE Statement

<update statement>

```
UPDATE <class specification> | <meta class specification> [ [ AS ]
<identifier> ]
SET <update assignment comma list>
[ WHERE <search condition> ]
[ <using index clause> ]
| UPDATE OBJECT <oid variable>
SET <update assignment comma list>
```

<update assignment comma list>

```
<update assignment> [ { , <update assignment> }... ]
```

<update assignment>

```
(<attribute comma list>) = {(<query expression>)
| (<expression comma list>)}
```

<attribute comma list>

```
<attribute name> [{, <attribute name>}...]
```

Example

```
UPDATE resort
SET cost = $198.00
WHERE name = 'Tryall Golf, Tennis, and Beach Club';
```

RENAME Statement

<rename statement>

```
<rename class statement>
| <rename trigger statement>
```

<rename class statement>

```
RENAME [ <class type> ] [ ONLY ] <class name> AS [ ONLY ] <class name>
```

<rename trigger statement>

```
RENAME TRIGGER <trigger name> AS <trigger name>
```

Example

```
RENAME CLASS employee AS company_employee;
RENAME TRIGGER check_age AS check_employee_age;
```

CALL Statement

<call statement>

```
CALL <method call> [ ON <call target> ] [ { TO | INTO} <variable> ]
```

<method call>

```
<method name> ( [ <expression comma list>] )
```

<call target>

```
<primary>
```

Example

```
CALL find_employee('smith') ON CLASS employee TO myvariable;
CALL update_status() ON myvariable;
```

EXECUTE Statement

<execute statement>

```
EXECUTE DEFERRED TRIGGER { <trigger name comma list> | ALL TRIGGERS}
```

Example

```
EXECUTE DEFERRED TRIGGER check_age, check_name;
EXECUTE DEFERRED TRIGGER ALL TRIGGERS;
```

EVALUATE Statement

<evaluate statement>

```
EVALUATE <expression> [ { TO | INTO } <variable> ]
```

SELECT Statement

<query state>

```
<query expression>
[ ORDER BY <sort specification comma list> ]
```

<query expression>

```
{ <query term> [ <table operator> [ <qualifier> ] <query term> ]...
```

<query term>

```
<query specification>
| <subquery>
```

<query specification>

```
SELECT [ <qualifier> ] <select list> [ { TO | INTO } <variable comma list> ]
FROM <extended table specification comma list>
[ WHERE <search condition> ]
[ GROUP BY <path expression comma list> ]
[ HAVING <search condition> ]
[ <using index clause> ]
```

<hierarchical query specification>

```
SELECT [ <qualifier> ] <select list> [ { TO | INTO } <variable comma list> ]
FROM <extended table specification comma list>
[ WHERE <search condition> ]
[ START WITH <condition> ]
```

```
CONNECT BY [NOCYCLE] <condition> ]
[ ORDER BY <sort specification comma list> ]
```

<subquery>

```
(<query statement>)
```

<table operator>

```
UNION
| { DIFFERENCE | EXCEPT }
| INTERSECTION
```

<qualifier>

```
ALL
| DISTINCT
| UNIQUE
```

<select list>

```
*
| <aliased expression comma list>
| <aliased counter expression comma list>
counter expression : INCR(path expression)
```

<aliased expression comma list>

```
<aliased expression> [ { , <aliased expression> }... ]
```

<aliased expression>

```
<expression> [ [ AS ] <alias name> ]
```

<sort specification comma list>

```
<sort specification> [ { , <sort specification> }... ]
```

<sort specification>

```
{ <path expression> | <unsigned integer literal> | <alias> } [ ASC | DESC ]
```

<using index clause>

```
USING INDEX { <index name comma list> | NONE }
```

Example

```
SELECT * FROM resort;
SELECT lodging FROM location WHERE country = 'Jamaica';
SELECT country, AVG(cost) FROM hotel
WHERE allows_children = 'yes' AND
number_of_pools > 0
GROUP BY country HAVING AVG(cost) <= $150.00;
```

GRANT/REVOKE Statement

<authorization statement>

```
GRANT <privileges> ON <class specification comma list>
TO <user name comma list> [ WITH GRANT OPTION ]
REVOKE <privileges> ON <class specification comma list>
FROM <user name comma list>
```

<privileges>

```
ALL [ PRIVILEGES ]
| <privilege comma list>
```

<privilege>

```
SELECT
| INSERT
| DELETE
| UPDATE
| INDEX
| ALTER
| EXECUTE
```

Example

```
GRANT SELECT, INSERT, UPDATE ON employee TO jones;
GRANT ALL PRIVILEGES ON person, student, TO smith, brown;
REVOKE INSERT, UPDATE ON employee FROM smith;
REVOKE SELECT ON manufacturing_site FROM jones, brown;
```

COMMIT Statement

<commit statement>

```
COMMIT [WORK]
```

ROLLBACK Statement

<rollback statement>

```
ROLLBACK [ WORK ] [ TO [ SAVEPOINT ] mark ]
```

SAVEPOINT Statement

<savepoint statement>

```
SAVEPOINT mark
```

TRANSACTION Statement

<get transaction statement>

```
GET TRANSACTION ISOLATION LEVEL
<isolation level spec>
| GET TRANSACTION LOCK TIMEOUT
<timeout spec>
```

<set transaction statement>

```
SET TRANSACTION ISOLATION LEVEL
<isolation level spec>
| SET TRANSACTION LOCK TIMEOUT
<timeout spec>
```

<isolation level spec>

```
<expression>
| SERIALIZABLE
| CURSOR STABILITY
| <class isolation> [ , <instance isolation> ]
| <instance isolation> [ , <class isolation> ]
```

<class isolation>

```
REPEATABLE READ { CLASS | SCHEMA }
| READ COMMITTED { CLASS | SCHEMA }
| READ UNCOMMITTED { CLASS | SCHEMA }
```

<instance isolation>

```
REPEATABLE READ INSTANCES
| READ COMMITTED INSTANCES
| READ UNCOMMITTED INSTANCES
```

<timeout spec>

```
INFINITE
| OFF
| <unsigned integer literal>
| <variable>>
```

TRIGGER Statement

<get trigger statement>

```
GET TRIGGER TRACE [ { TO | INTO }
<variable> ]
| GET TRIGGER [ MAXIMUM ] DEPTH [ { TO | INTO } <variable> ]
```

<set trigger statement>

```
SET TRIGGER TRACE <trace spec>
| SET TRIGGER [ MAXIMUM ] DEPTH <depth spec>
```

<trace spec>

```
{ ON | OFF }
| <unsigned integer literal>
| <variable>
```

<depth spec>

```
INFINITE
| <unsigned integer literal>
| <variable>
```

Example

```
CREATE TRIGGER check_age_update
BEFORE UPDATE ON person(age)
IF new.age < obj.age
EXECUTE REJECT;
CREATE TRIGGER emit_message_on_commit
BEFORE COMMIT
EXECUTE PRINT "Committing transaction.";
ALTER TRIGGER check_age_update PRIORITY 10.0;
ALTER TRIGGER emit_message_on_commit STATUS INACTIVE;
```

OPTIMIZATION Statement

<get optimization statement>

```
GET OPTIMIZATION LEVEL [ { TO | INTO } <variable> ]
| GET OPTIMIZATION LIMIT [ { TO | INTO } <variable> ]
| GET OPTIMIZATION COST [ OF ]
<stringliteral> [ { TO | INTO } <variable> ]
```

<set optimization statement>

```
SET OPTIMIZATION LEVEL [ TO | = ] <opt level spec>
| SET OPTIMIZATION LIMIT [ TO | = ] <opt limit spec>
| SET OPTIMIZATION COST [ OF ] <string literal> [ TO | = ] <expression>
```

<opt level spec>

```
{ ON | OFF }
| <unsigned integer literal>
| <variable>
```

<opt limit spec>

```
<unsigned integer literal> | <variable>
```

UPDATE STATISTICS Statement

<update statistics statement>

```
UPDATE STATISTICS ON <update statistics classes>
```

<update statistics classes>

```
[ONLY] <class name>
| ALL CLASSES
```

USER Statement

<create user>

```
CREATE USER <user name>
[PASSWORD password]
[GROUPS user name comma list]
[MEMBERS user name comma list]
```

<drop user>

```
DROP USER user name
```

<alter user>

```
ALTER USER user_name PASSWORD password
```

Example

```
CREATE USER david;
CREATE USER company;
CREATE USER engineering GROUPS company;
CREATE USER engineering MEMBERS david;
ALTER USER david PASSWORD 'passwd';
DROP USER david;
```

INDEX Statement

<index>

```
CREATE [REVERSE] [UNIQUE] INDEX [index_name]
ON <index_specification>
```

<index_specification>

```
<class_name> ( <attribute_name_comma_list> )
```

Example

```
CREATE INDEX ON employee (name);
CREATE INDEX ON person (name, social_security_number);
```

PARTITION Statement

<create partition>

```
<create class> <partition clause>
```

<alter partition>

```
ALTER {CLASS | TABLE} <class name>
<alter partition clause>
```

<create class>

```
CREATE {CLASS | TABLE} <class name>
[ <subclass definition>]
[ ( <class element comma list> ) ]
[ CLASS ATTRIBUTE ( <attribute definition comma list> ) ]
[ METHOD <method definition comma list> ]
[ FILE <method file comma list> ]
[ INHERIT <resolution comma list> ]
```

<alter partition clause>

```
<alter clause>
| DROP PARTITION <alter partition name comma list>
| REMOVE PARTITIONING
| ADD PARTITION <partition options>
| REORGANIZE PARTITION
<alter partition name comma list> INTO ( <partition definition comma list> )
| ANALYZE PARTITION { ALL | <alter partition name comma list> }
| COALESCE PARTITION <unsigned integer>
```

<partition clause>

```
PARTITION BY { HASH | RANGE | LIST }
( <expression> )
<partition options>
```

<partition options>

```
( <partition definition comma list> )
| PARTITIONS <unsigned integer>
```

<partition definition>

```
PARTITION <partition name>
<partition value options>
```

<partition value options>

```
<partition range> | <partition list>
```

<partition range>

```
VALUES LESS THAN <partition range values>
```

<partition range values>

```
MAXVALUE | ( <partition value>
```

<partition list>

```
VALUES IN ( <partition value comma list> )
```

Example

```
CREATE CLASS person (
name string,
birthday date,
residence string,
UNIQUE (name))
PARTITION BY RANGE ( name ) (
PARTITION atoh VALUES LESS THAN ('I'),
PARTITION itor VALUES LESS THAN ('S'),
PARTITION etcname VALUES LESS THAN MAXVALUE );
```

```
CREATE CLASS person (
name string,
birthday date,
residence string,
UNIQUE (name))
PARTITION BY HASH ( name )
PARTITIONS 20;
ALTER CLASS person
PARTITION BY LIST ( SUBSTRING(residence from 1 for 1) ) (
PARTITION atoh VALUES IN ('A','B','C','D','E','F','G','H'),
PARTITION itoo VALUES IN ('I','J','K','L','M','N','O'));
ALTER CLASS person REMOVE PARTITIONING;
```

Search Conditions

<search condition>

```
<boolean term> [ OR <boolean term> ]...
```

<boolean term>

```
<boolean factor> [ AND <boolean factor> ]...
```

<boolean factor>

```
[ NOT ] <boolean primary>
```

<boolean primary>

```
<predicate>
```

<predicate>

```
<exists predicate>
| <comparison predicate>
| <multi column predicate>
| <between predicate>
| <in predicate>
| <like predicate>
| <is null predicate>
| <set predicate>
```

<exists predicate>

```
EXISTS <expression>
```

<comparison operator>

```
<expression> <comparison operator> [ <quantifier> ] <expression>
```

<quantifier>

```
=
| < >
| <
| <=
| >
| >=
```

<multi column predicate>

```
ALL
| SOME
| ANY
```

<multi in predicate>

```
{<multi in predicate> | <multi comparison predicate>}
```

<between predicate>

```
(<expression comma list>)
[NOT] IN ( <query expression> )
```

<in predicate>

```
( <expression comma list> ) { = | < > }
( { <expression comma list> | <query expression> } )
```

<like predicate>

```
<expression> [NOT] LIKE <expression> [
ESCAPE <string literal> ]
```

<is null predicate>

```
<expression> IS [NOT] NULL
```

<set predicate>

```
<expression> <set operator> <expression>
```

<set opera>

```
SETEQ
| SETNEQ
| SUPERSET
| SUPERSETEQ
| SUBSET
| SUBSETEQ
```

Expressions

<expression>

```
<term> [ { + | - | '||' } term ]...
```

<term>

```
<factor> [ { * | / } <factor> ]
```

<factor>

```
[ + | - ] <primary>
```

<primary>

```
<value specification>
| <path expression>
| <aggregate function>
| <set expression>
| <subquery>
| <insert statement>
| ( <expression> )
| <arithmetic function>
| <character function>
| <scalar function>
| <class function>
| <cast function>
| <extract function>
| <instance number function>
| <orderby number function>
```

<path expression>

```
<path element> [ . <path element> ]...
| CLASS <class name> . <path element> [ . <path element> ]...
```

<path element>

```
<attribute name>
| <attribute name> <identifier>
| <method call>
```

<aggregate function>

```
COUNT(*)
| COUNT ( [ <qualifier> ] <path expression> )
| { AVG | MAX | MIN| SUM | STDDEV | VARIANCE } ( { DISTINCT | UNIQUE } <path
expression> )
| { AVG | MAX | MIN | SUM | STDDEV | VARIANCE } ( [ ALL ] <expression> )
| GROUPBY_NUM()
```

<set expression>

```
[ <set specifier> ] <subquery>
| [ <set specifier> ] '{' [ <expression comma list> ] '}'
```

<set specifier>

```
SET
| MULTISET
| { SEQUENCE | LIST }
```

<arithmetic function>

```
MOD ( <expression> , <expression> )
```

<character function>

```
POSITION ( <expression> IN <expression> )
| SUBSTRING (<expression> FROM <expression> [ FOR <expression> ] )
| OCTET_LENGTH ( <expression> )
| BIT_LENGTH ( <expression> )
| CHAR_LENGTH ( <expression> )
| LOWER ( <expression> )
| UPPER ( <expression> )
| TRANSLATE ( <expression> , <expression> , <expression> )
| REPLACE ( <expression> , <expression> [ , <expression>] )
| UPPER ( <expression> , <expression> [ , <expression>] )
| RPAD ( <expression> , <expression> [ , <expression>] )
| LTRIM ( <expression> [ , <expression>] )
| RTRIM ( <expression> [ , <expression>] )
| TRIM ( [ [ LEADING | TRAILING | BOTH ] [ <expression> ] FROM ]
<expression> )
```

<scalar function>

```
ADD_MONTHS ( <expression> , <expression> )
| LAST_DAY ( <expression> )
| MONTHS_BETWEEN ( <expression> , <expression> )
| SYS_DATE
| SYS_DATETIME
| SYS_TIME
| SYS_TIMESTAMP
| TO_CHAR ( <expression> [ , <expression> [, {'en_US' | 'ko_KR'}] ] )
| TO_DATE ( <expression> [ , <expression> [, {'en_US' | 'ko_KR'}] ] )
| TO_DATETIME ( <expression> [ , <expression> [, {'en_US' | 'ko_KR'}] ] )
| TO_TIME ( <expression> [ , <expression> [, {'en_US' | 'ko_KR'}] ] )
| TO_TIMESTAMP ( <expression> [ , <expression> [, {'en_US' | 'ko_KR'}] ] )
| TO_NUMBER ( <expression> [ , <expression> ] )
```

<class function>

```
CLASS ( <class name> )
```

<cast function>

```
CAST ( <expression> AS <data type> )
```

<case expression>

```
CAST <expression> <simple when clause> [ <case else clause> ] END
| CASE <searched when clause> [ <case else clause> ] END
| NULLIF ( <expression> , <expression> )
| COALESCE ( <expression comma list> )
```

<simple when clause>

```
WHEN <expression> THEN <result>
```

<searched when clause>

```
WHEN <search condition> THEN <result>
```

<case else clause>

```
ELSE <result>
```

<result>

```
<expression>
| NULL
```

<extract function>

```
EXTRACT ( <extract field> FROM <expression> )
```

<extract field>

```
YEAR
| MONTH
| DAY
| HOUR
| MINUTE
| SECOND
| MILLISECOND
```

<instance number function>

```
ROWNUM
| INST_NUM()
```

<orderby number function>

```
ORDERBY_NUM()
```

Table Specifications

<extended table specification comma list>

```
<table specification> [ {, <table specification>
| <qualified join table specification> }... ]
```

<table specification>

```
<class specification>
[ <correlation> ]
| <meta class specification>
[ <correlation> ]
| <subquery> <correlation>
| TABLE ( <expression> ) <correlation>
```

<correlation>

```
[ AS ] <identifier>
[ ( <identifier comma list> ) ]
```

<qualified join table specification>

```
[ INNER | { LEFT | RIGHT [ OUTER ] ] JOIN <table specification>
[ <join condition> ]
```

<join condition>

```
ON <search condition>
```

<class type>

```
CLASS
| TABLE
| VCLASS
| VIEW
```

<class specification>

```
<single class specification>
| ( <single class specification comma list> )
```

<single class specification>

```
[ ONLY ] <class name>
| ALL <class name>
[ ( EXCEPT <class specification comma list> ) ]
```

<meta class specification>

```
CLASS <class name>
```

Data Types

<data type>

```
<primitive type>
| <set of> <primitive type>
| <set of> <primitive type comma list>
```

<primitive type comma list>

```
<primitive type> [ { , <primitive type }... ]
```

<primitive type>

```
[ ALL ] <class name>
| BIGINT
| BIT [ ( <unsigned integer literal> ) ]
| BIT VARYING [ ( <unsigned integer literal> ) ]
| { CHAR | CHARACTER } [ ( <unsigned integer literal> ) ]
| { CHAR | CHARACTER } VARYING | VARCHAR [ ( <unsigned integer literal> ) ]
| DATE
| DATETIME
| {  DEC | DECIMAL } [( <precision> [ , <scale> ] ) ]
| DOUBLE [ PRECISION ]
| { FLOAT | REAL } [( <precision> )]
| { INT | INTEGER }
| MONETARY
| { NATIONAL { CHAR | CHARACTER } | NCHAR } [ ( <unsigned integer literal> ) ]
| { NATIONAL { CHAR | CHARACTER } | NCHAR } VARYING [ ( <unsigned integer
literal> ) ]
| NUMERIC [ ( <precision> [, <scale> ] ) ]
| OBJECT
| {  SMALLINT | SHORT }
```

```
| STRING
| TIME
| {  TIMESTAMP | UTIME }
```

<precision>

```
<unsigned integer literal>
```

<scale>

```
<unsigned integer literal>unsigned integer literal>
```

<set of>

```
{ SET OF | SET [ OF ] }
| {  MULTISET_OF | MULTISET [ OF ] }
| { SEQUENCE_OF | { SEQUENCE | LIST } [ OF ] }
```

Identifier, Name, Variable

<class name comma list>

```
<class name> [ { , <class name> }... ]
```

<class name>

```
[ <user name>. ] <identifier>
```

<user name>

```
<identifier>
| PUBLIC
```

<constraint name>

```
<identifier>
```

<attribute name comma list>

```
<attribute name> [ { , <attribute name> }... ]
```

<attribute name>

```
<identifier>
```

<method name comma list>

```
<method name> [ { , <method name> }... ]
```

<method name>

```
<identifier>
```

<method file path comma list>

```
<method file path> [ { , <method file path> }... ]
```

<method file path>

```
<string literal>
```

<method function name>

```
<identifier>
```

<trigger name comma list>

```
<trigger name> [ { , <trigger name> }... ]
```

<trigger name>

```
<identifier>
```

<serial name>

```
<identifier>
```

<index name>

```
<identifier>
```

<alias name>

```
<identifier>
```

<variable comma list>

```
<variable> [ { , <variable> }... ]
```

<variable>

```
[ : ] <identifier>
```

<oid variable>

```
<variable>
```

<value specification>

```
<literal>
| <variable>
```

Literal

<literal>

```
<string literal>
| <number literal>
| <date literal>
| <time literal>
| <monetary literal>
| <set literal>
| USER
| NULL
```

<string literal>

See "String Data Type."

<number literal>

```
<exact number literal>
| <approx number literal>
```

<exact number liter>

```
[ + | - ] <unsigned real literal>
```

<unsigned real literal>

```
<unsigned integer literal> [.<unsigned integer literal> ]
| <unsigned integer literal>
| <unsigned integer literal>
```

<approx number literal>

```
<exact number literal> [E|e] <integer literal>
```

<integer literal>

```
[ + | - ] <unsigned integer literal>
```

<unsigned integer literal>

```
One or more decimal digits.
```

<date literal>

```
<string literal> of form mm/dd/[yyyy].
```

<datetime literal>

```
<string literal> of form 'mm/dd/[yyyy] hh:mm[:ss[.ff] ] [ampm]'.
```

<time literal>

```
<string literal> of form 'hh[:mm[:ss] ] [ampm]'.
```

<monetary literal>

```
$<unsigned real literal>
```

<set literal>

```
[<set specifier> {[value specification comma list>]}
```

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;

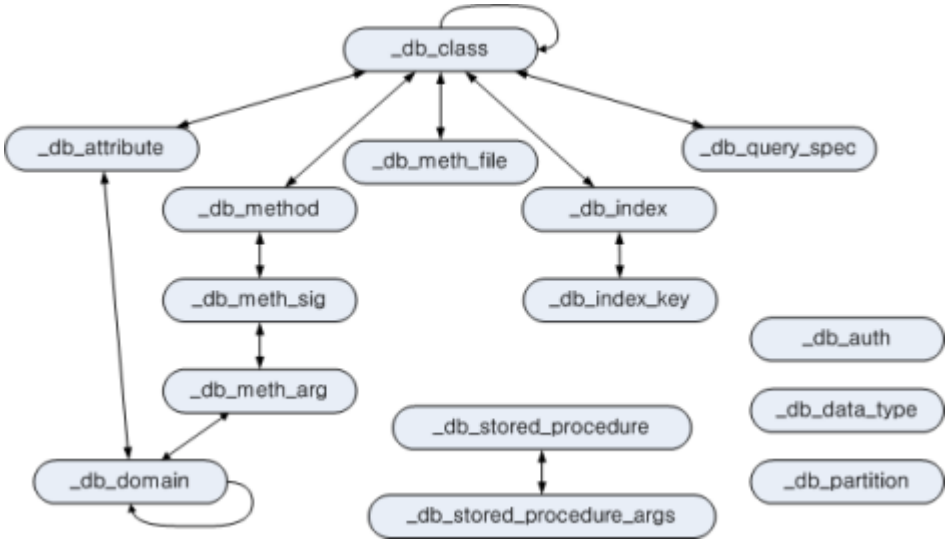
-- Attribute of the 'db_user' class
SELECT attr_name, data_type
FROM db_attribute
WHERE class_name = 'db_user';

-- Attribute inherited from the superclass, among attributes of the glo class
SELECT attr_name
FROM db_attribute
WHERE class_name = 'glo' AND
class_name <> from_class_name;
```

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 Superclass).

```
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 superclass. 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 superclass 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 superclass 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, ‘volume id | page id | slot id’ is used. If the data type is a set, ‘{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
1	INTEGER
2	FLOAT
3	DOUBLE
4	STRING
5	OBJECT
6	SET
7	MULTISET
8	SEQUENCE
9	ELO
10	TIME
11	TIMESTAMP
12	DATE
13	MONETARY
18	SHORT
20	OID
22	NUMERIC
23	BIT
24	VARBIT
25	CHAR
26	NCHAR
27	VARNCHAR
31	BIGINT
32	DATETIME

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 (a value in the "Value" column of the "Character Sets Supported by CUBRID" table in _db_attribute) if the data type is a character, and 0 otherwise.
set_domains	SEQUENCE OF _db_domain	Domain information about the data type of the element of the set if the data type is a set, and **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 superclass 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 superclass 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 m
in c.class
_meths))
============================================
  'db_authorization'    {'check_authorization'}
  'db_authorizations'   {'add_user', 'drop_user', 'find_user',
'print_authorizations',
'info', 'change_owner', 'change_trigger_owner', 'get_owner'}
  'db_root'             {'add_user', 'drop_user', 'find_user',
'print_authorizations',
'info', 'change_owner', 'change_trigger_owner', 'get_owner',
'change_sp_owner'}
  'db_user'             {'add_user', 'drop_user', 'find_user', 'login'}
  'glo'                 {'new', 'new_lo', 'new_lo_import', 'new_fbo'}
  'glo_holder'          {'new'}
```

_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 superclass 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.

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**
- **DROP QUERY**
- **INDEX**
- **EXECUTE**

Example

The following is an example of retrieving the authorization information defined in the class 'glo'.

```
csql> select grantor.name, grantee.name, auth_type
csql> from _db_auth
csql> where class_of.class_name = 'glo';
csql> ;xrun

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

  grantor.name          grantee.name          auth_type
============================================================
  'DBA'                 'PUBLIC'              'SELECT'
  'DBA'                 'PUBLIC'              'INSERT'
  'DBA'                 'PUBLIC'              'UPDATE'
  'DBA'                 'PUBLIC'              'DELETE'
  'DBA'                 'PUBLIC'              'EXECUTE'


5 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

Method 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.

System Catalog Virtual Classes

System Catalog Virtual Classes

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.

Definition

```
CREATE VCLASS db_class (class_name, owner_name, class_type, is_system_class,
partitioned)
AS
SELECT c.class_name, CAST(c.owner.name AS VARCHAR(255)),
       CASE WHEN c.class_type = 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'));
```

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'
  'glo'
```

DB_DIRECT_SUPER_CLASS

Represents names of superclasses (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)	Superclass 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

- **Example 1**

  The following is an example of retrieving superclasses of the 'female_event' class (see Adding a Superclass).

```
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'
```

- **Example 2**

  The following is an example of retrieving superclasses 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 superclass.
from_class_name	VARCHAR(255)	If the attribute is inherited, the superclass 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 superclass 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 (a value in the 'Value' column of the 'Character Sets Supported by CUBRID' table in _db_attribute) if the data type is a character, and 0 otherwise.
domain_class_name	VARCHAR(255)	Domain class name if the data type is an object; otherwise, **NULL**
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

- **Example 1**

  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
```

- **Example 2**

  The following is an example of retrieving attributes of the ‘female_event’ class and its superclass.

```
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'
```

- **Example 3**

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)	'INSTANCE' 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
‘set_attr’	‘D’	‘INSTANCE’	‘SET’	0	0	0	‘A’
‘set_attr’	‘D’	‘INSTANCE’	‘SET’	0	0	0	‘B’
‘set_attr’	‘D’	‘INSTANCE’	‘SET’	0	0	0	‘C’

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_clas
s_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 superclass 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 superclass 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 ‘glo’ class.

```
csql> select meth_name, meth_type, func_name
csql> from db_method
csql> where class_name = 'glo'
csql> order by meth_type, meth_name;
csql> ;xrun

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

  meth_name             meth_type             func_name
============================================================================
  'new'                 'CLASS'               'esm_Glo_create'
  'new_fbo'             'CLASS'               'esm_Glo_create_fbo'
  'new_lo'              'CLASS'               'esm_Glo_create_lo'
  'new_lo_import'       'CLASS'               'esm_Glo_import_lo'
  'append_data'         'INSTANCE'            'esm_Glo_append'
  'binary_search'       'INSTANCE'            'esm_Glo_binary_search'
  'compress_data'       'INSTANCE'            'esm_Glo_compress'
  'copy_from'           'INSTANCE'            'esm_Glo_copy_from'
  'copy_to'             'INSTANCE'            'esm_Glo_copy_to'
  'data_pos'            'INSTANCE'            'esm_Glo_position'
  'data_seek'           'INSTANCE'            'esm_Glo_seek'
  'data_size'           'INSTANCE'            'esm_Glo_size'
  'delete_data'         'INSTANCE'            'esm_Glo_delete'
. . .
```

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 ‘append_data’ method of the ‘glo’ class.

```
csql> select meth_name, data_type, prec
csql> from db_meth_arg
csql> where class_name = 'glo' and
csql>       meth_name = 'append_data' and
csql>       index_of > 0
csql> order by index_of;
csql> ;xrun

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

  meth_name             data_type                     prec
============================================================
  'append_data'         'INTEGER'                        0
  '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 superclass 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.

Definition

```
CREATE VCLASS db_index_key (index_name, class_name, key_attr_name, key_order)
AS
SELECT k.index_of.index_name, k.index_of.class_of.class_name, k.key_attr_name,
k.key_order
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_a
thlete_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

- **Example 1**

  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'));
```

- **Example 2**

  The following is an example of retrieving trigger information of the class.

```
csql> select target_class_name, trigger_name
csql> from db_trig
csql> order by 1;
csql> ;xrun


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

target_class_name     trigger_name
============================================
  'glo'               'glo_delete_contents'
```

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         pa
rtition_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 Classes/Virtual Classes 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 class case_insensitive(name varchar(255));
Select * from db_class where class_name = 'case_insensitive'; -- Success
Select * from db_class where class_name = 'case_insensitive'; -- Failure
```

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:

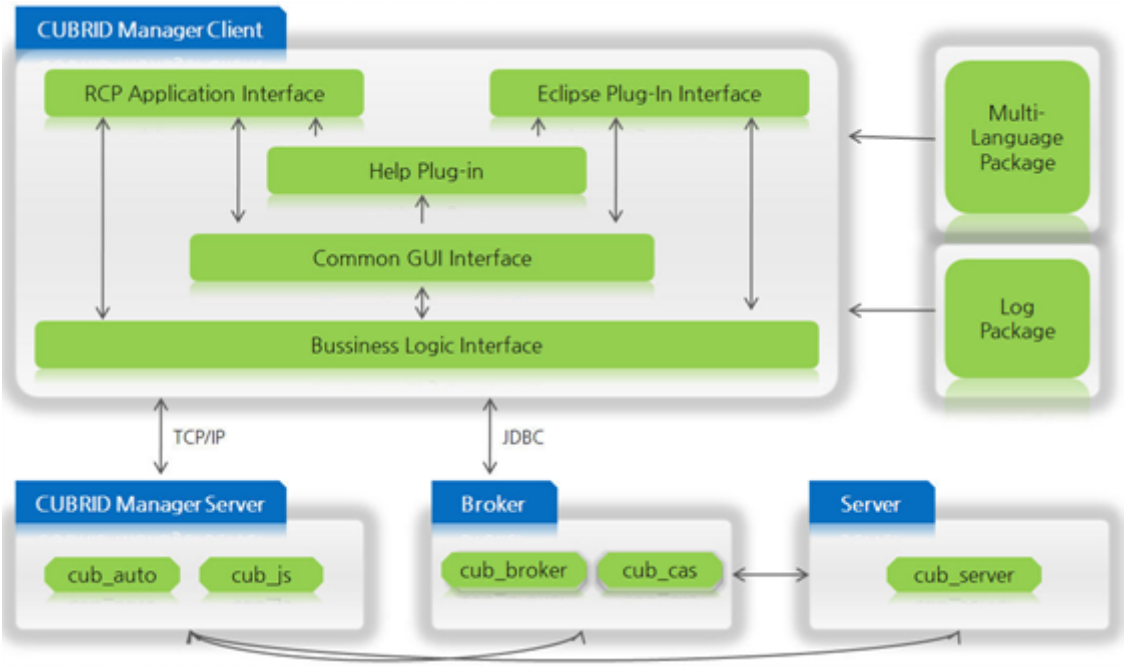
- Introduction to the CUBRID Manager
- Running the CUBRID Manager
- Client Features of the CUBRID Manager

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.



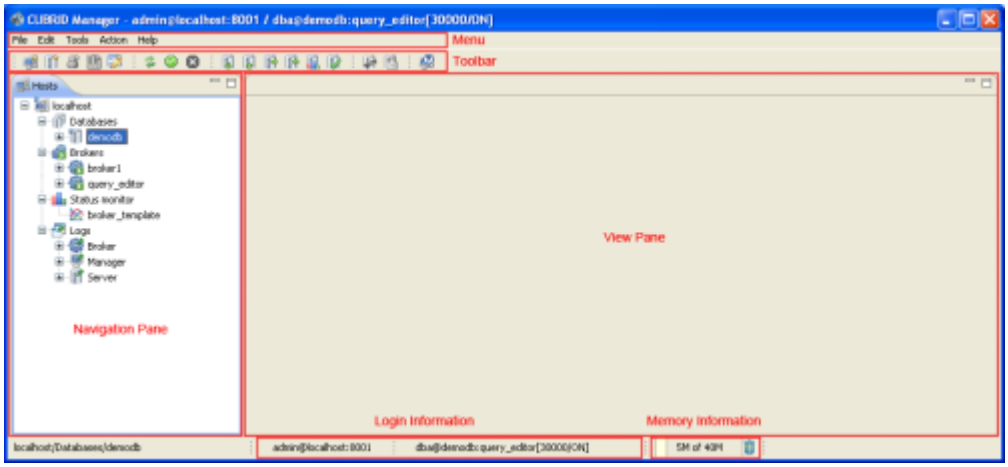
For more information on running the server, see Configuring the CUBRID Manager Server.

CUBRID Manager Client

RCP Application

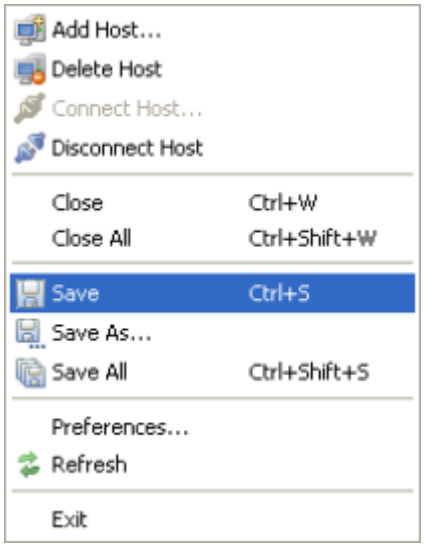
Query Editor Layout

The Client consists of 4 areas - the menu, search, view, and status bar. Each area provides unique information.



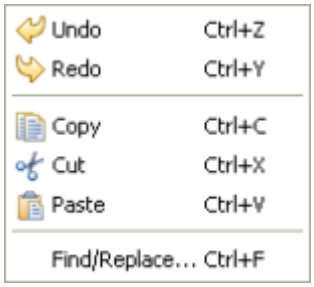
File Menu

The [File] menu consists of host menu, file menu, CUBRID Manager preference menu and others.



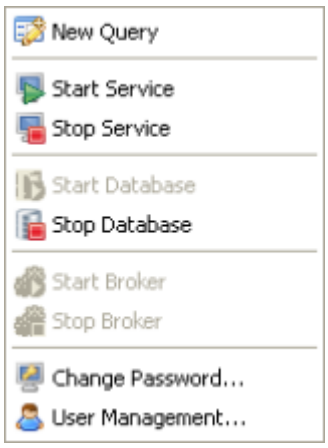
Edit Menu

The [Edit] menu consists of options used in the CUBRID Manager for editing.



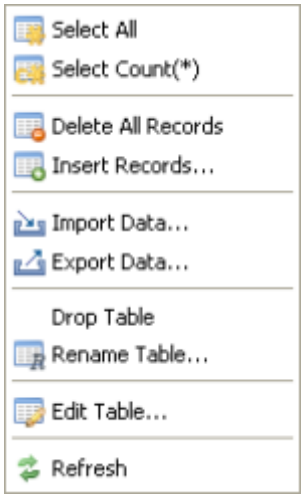
Tools

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

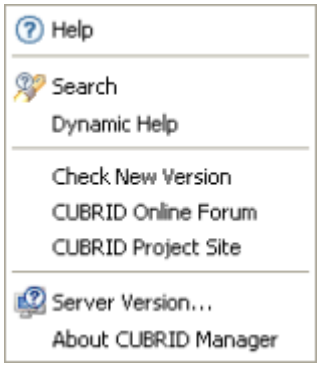
The [Action] menu is a dynamic menu, showing available actions by a right-click in the navigation pane. The following shows the options of the [Action] menu, displayed by a right-click of a table.



Help Menu

The [Help] menu consists of Help, which allows you to browse CUBRID Database Manual, Dynamic Help, and Search options.

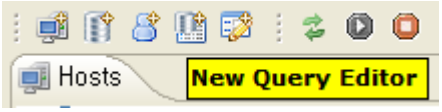
To check CUBRID tips or development news, select [CUBRID Online Forum] or [CUBRID Project Site].



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.



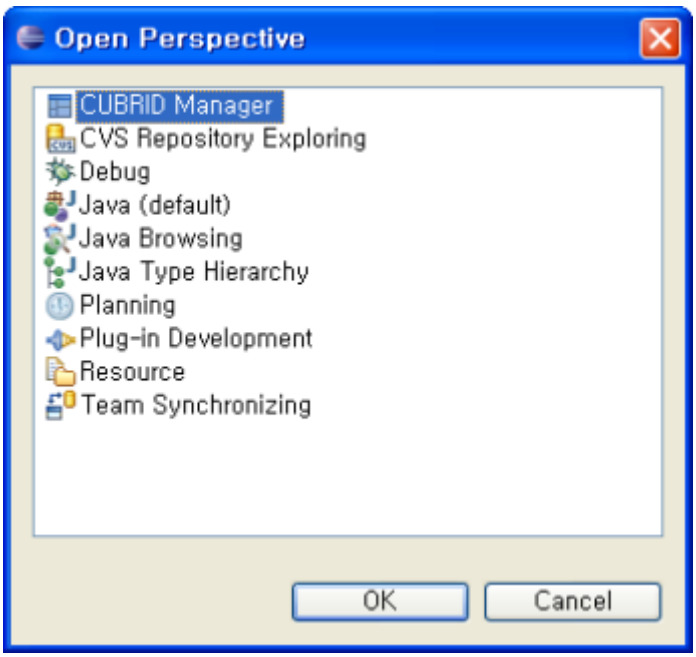
The toolbar provides the following functions.

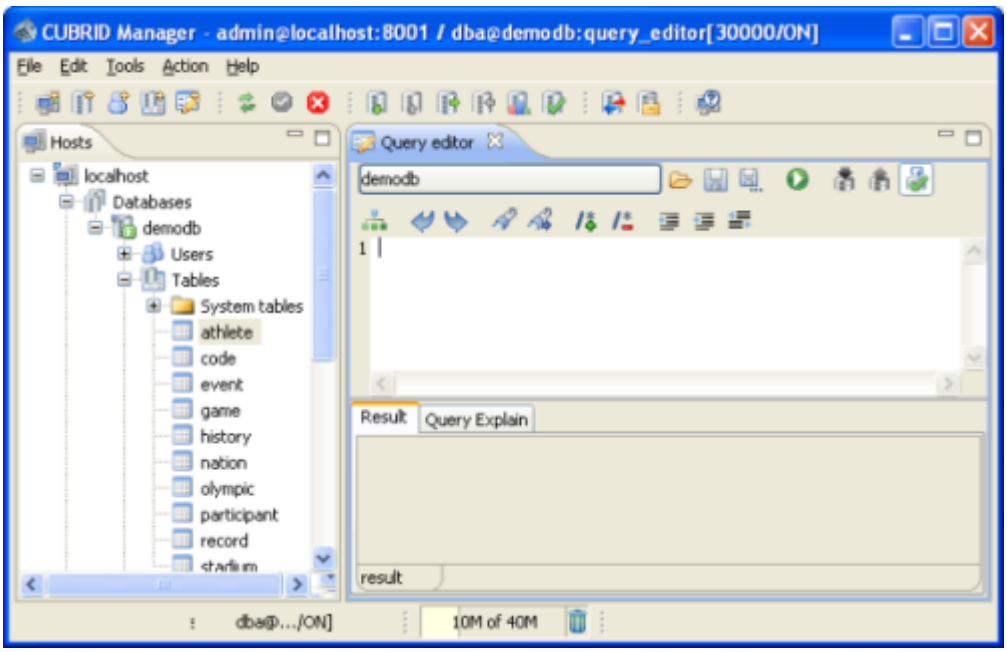


	Add Host		Unload Database
	Create Database		Load Database
	Create User		Backup Database
	Create Table		Restore Database
	New Query		Optimize Database
	Refresh		Check Database
	Start		Transaction Info
	Stop		Lock Information
			Server Version

Eclipse Plug-in

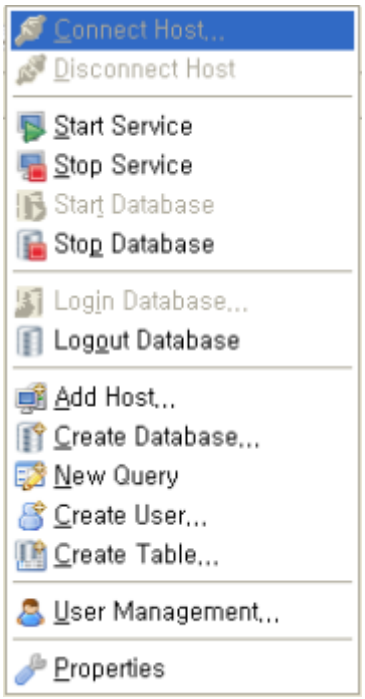
When you connect to the CUBRID Manager client with Eclipse plug-in, the [CUBRID] menu provides the basic menu and the same tree pane for the RCP application. To run the CUBRID Manager plug-in, select [Window] > [Open Perspective] > [Other] in Eclipse, then select [CUBRID Manager].





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.



Configuring the 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 blanks. 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.

execute_diag

A parameter that activates the Database Server diagnosis function provided by the CUBRID Manager. The default value is **OFF**. The CUBRID Broker's diagnosis function is always activated regardless of this parameter.

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.

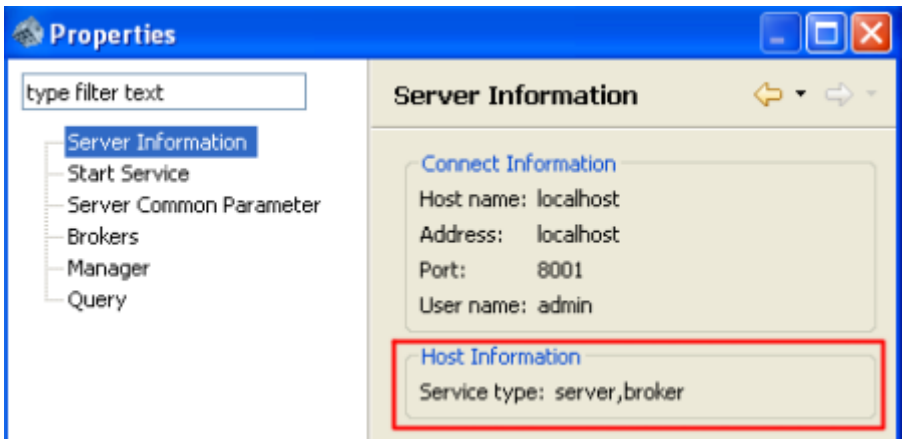
cm_target

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].

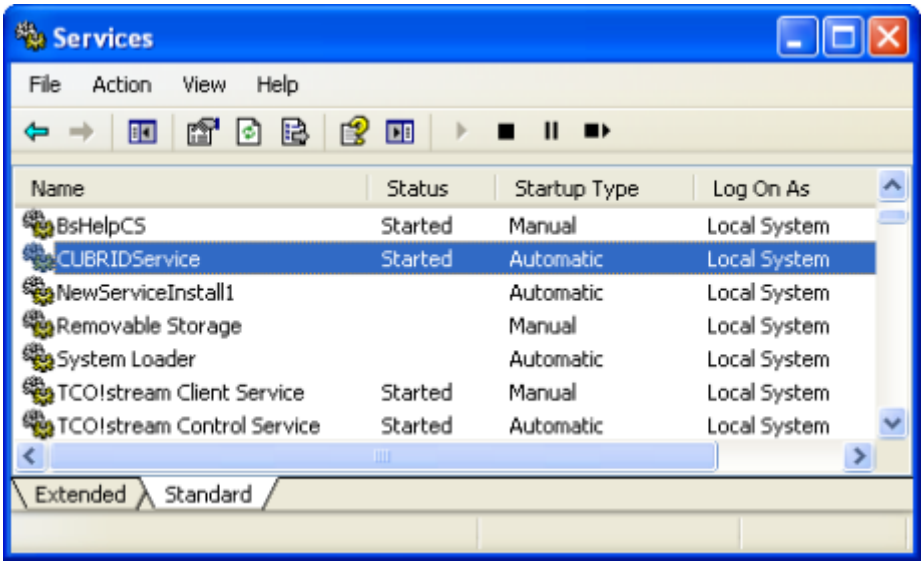


Running the CUBRID Manager Server

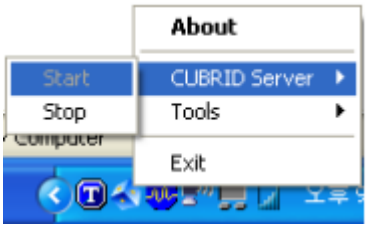
The CUBRID Manager consists of a server and a client. You can run the CUBRID Manager client only when the CUBRID Manager server is running.

Running in Windows

- To start the CUBRID Server, go to [Control Panel] > [Performance and Maintenance] > [Administrative Tools] > [Services] and double-click CUBRIDService.



- 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 the 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.

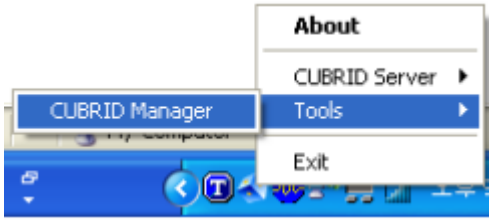
The CUBRID Manager client is a Java application program that runs only on JRE/JDK 1.5 or higher. If the version is below 1.5, an error message appears: "Unsupported JRE version. You must use JRE 1.5 or a later version."

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.

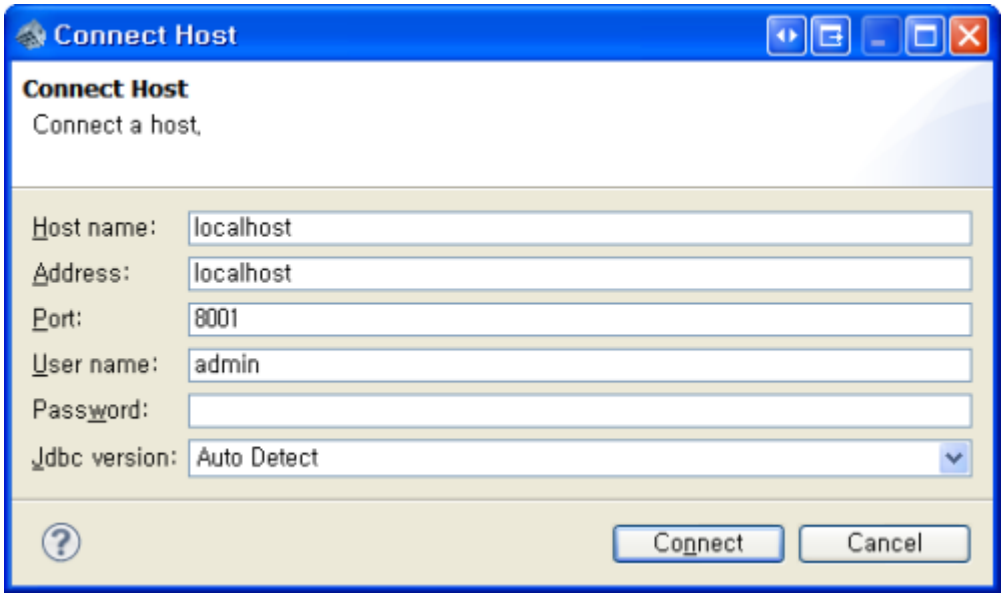
- %cd $CUBRID/cubridmanager
- %./cubridmanager

CUBRID Manager Client Log

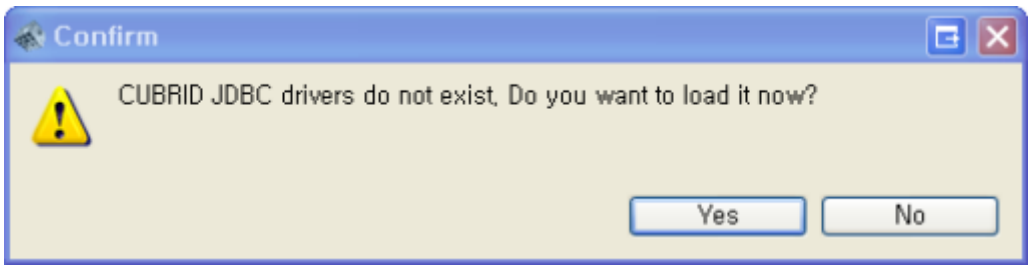
Error logs that occurred while the CUBRID Manager client is running are created as the **`cubridmanager.log`** file in **`$CUBRID/cubridmanager/logs`**. When analyzing CUBRID Manager improvements and errors, you can participate in efforts to improve the CUBRID Manager by registering the error log and the symptom as an issue at the CUBRID Manager development site.

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.



- **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.

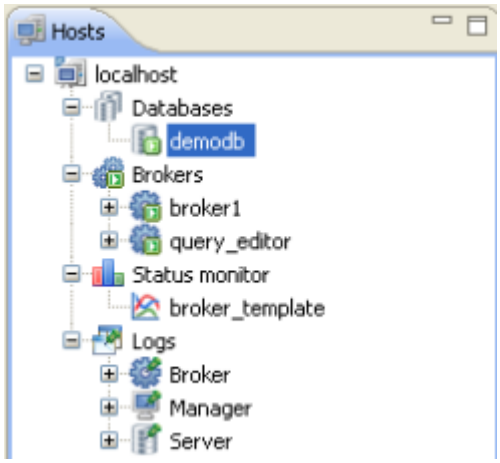


Connect/Disconnect Host

Connect Host

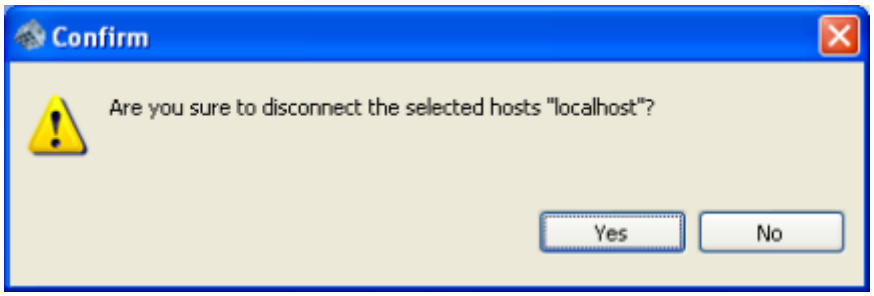
A connection to a specific host is made when you double-click the host registered in the host navigation tree. Or you can right-click a host, select [Connect Host], check the user name, and then click [Connect] to connect to the host.

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].

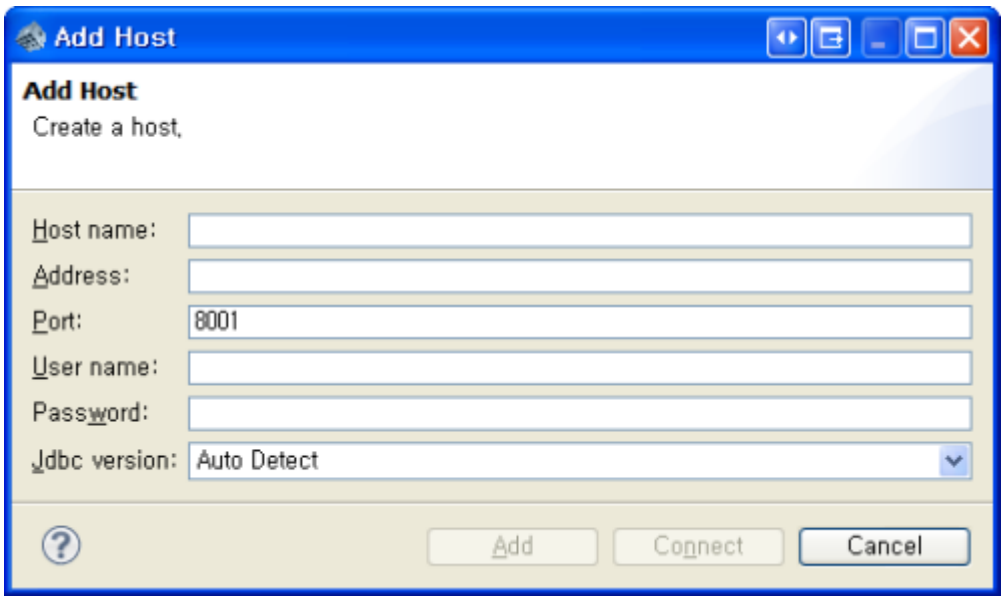


Add/Delete Host

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].

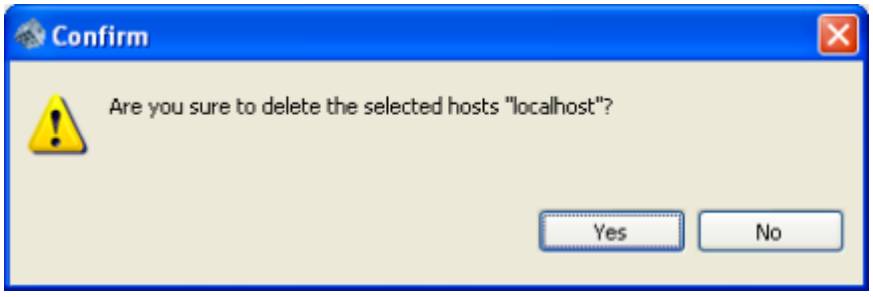


For more information about host name, host address, connection port, user name and password, see Default Host Information.

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.

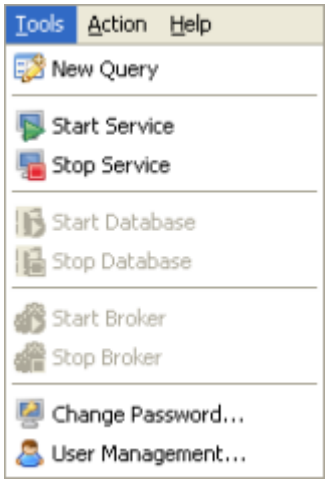
Delete Host

Deletes the information of the host registered in the host navigation tree. When a host is deleted, all information in the subnodes is deleted as well. However, only the CUBRID Manager client configuration, not the CUBRID Manager server configuration, is deleted.



Managing Service

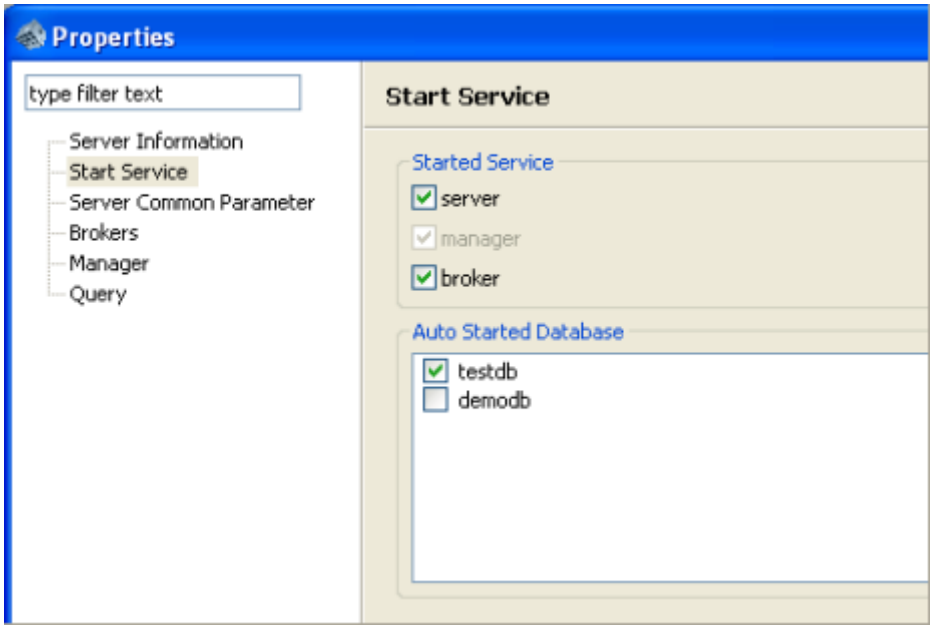
You can start or stop CUBRID related services with the CUBRID Manager. This functionality works the same as using 'cubrid,' a management utility used in the CUBRID database. However, you cannot start or stop the CUBRID Manager server with the CUBRID Manager client.



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.



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:

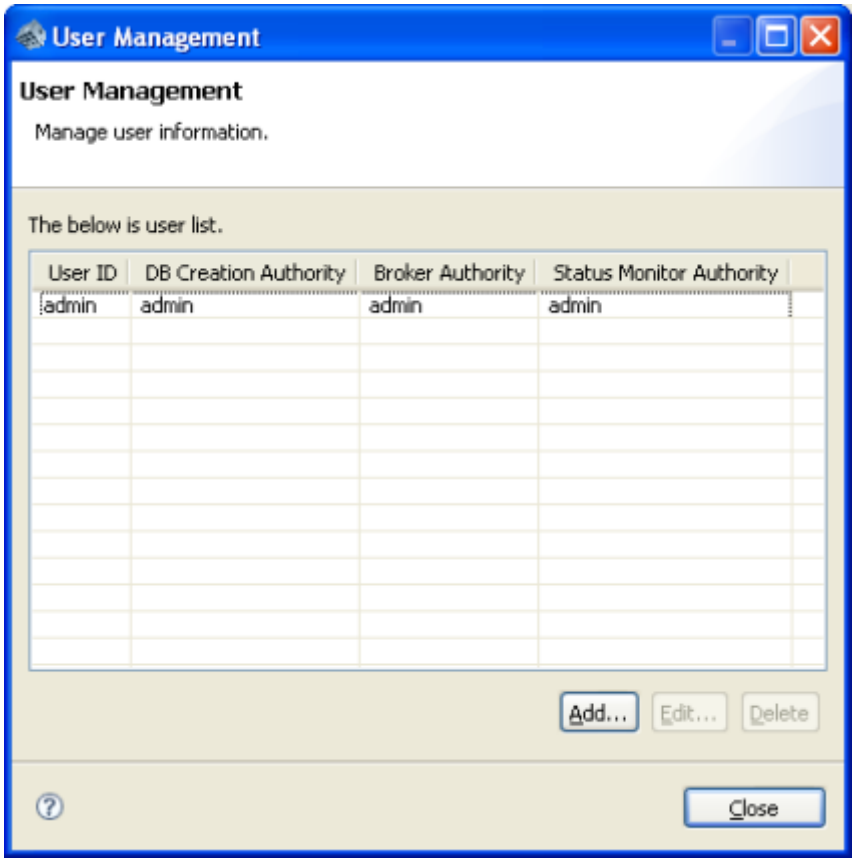
- Click [Start] or [Stop] from the toolbar.
- Right-click a Broker and then select [Start Broker] or [Stop Broker].
- Select [Action] > [Start Broker], or [Action] > [Stop Broker] on the menu.

Setting User Authorization

Multiple users can be registered as CUBRID Manager users, whose information is stored to the CUBRID Manager server. Therefore, you need to 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:



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

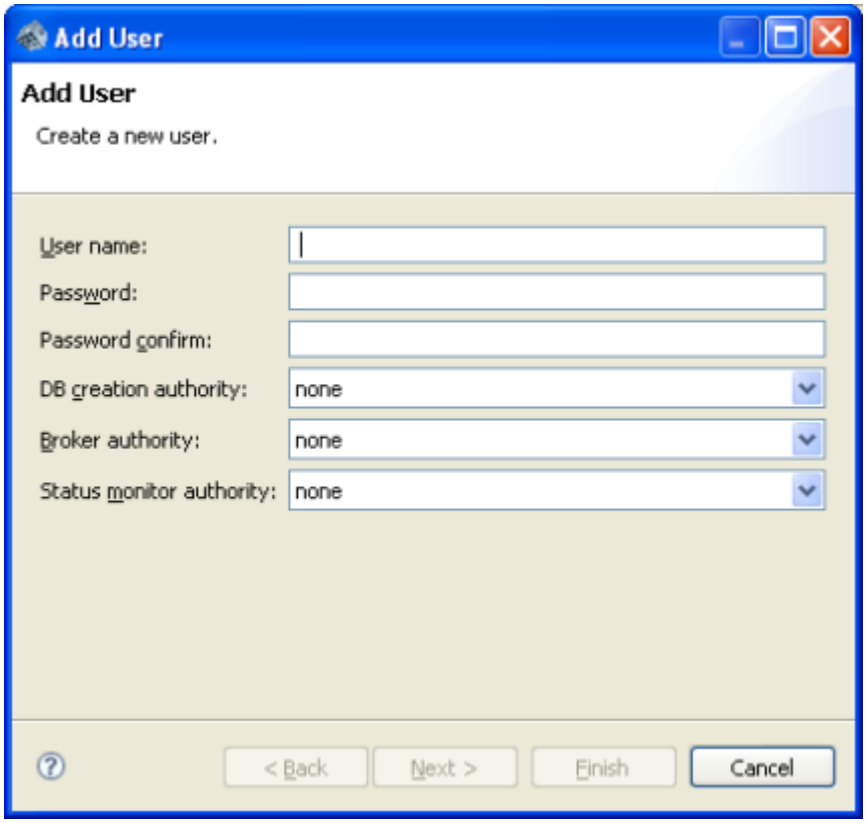
- **admin** : Authorization to perform, add, edit, and delete status monitoring.
- **monitor** : Authorization to monitor the progress of a status monitor.
- **none** : No authorization.

Adding, Editing, and Deleting Users

Add User

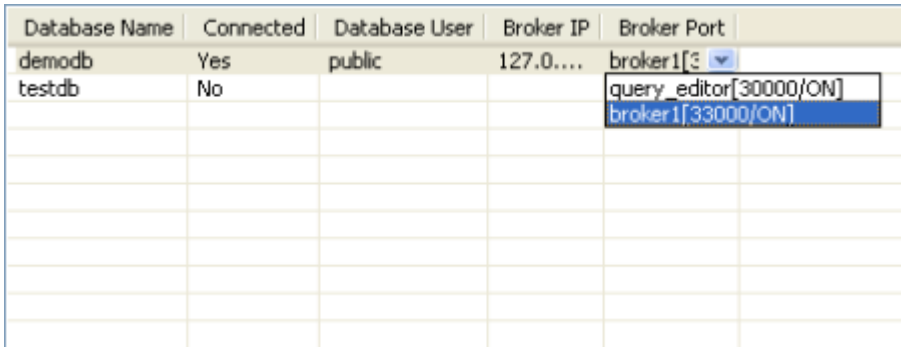
To add a user, you need to specify the user account information and authorization, and database access authority.

Setting User Account and Authority



- **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



- **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

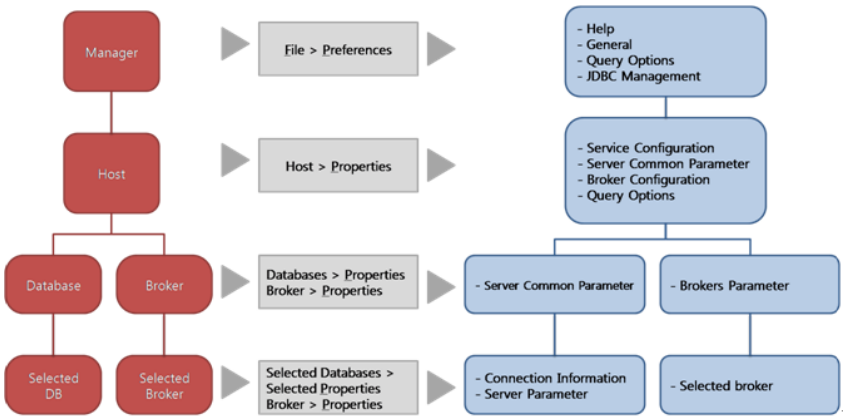
You can select a user from the user list and edit it as you add a user. However, editing the **admin** account is limited to changing its password. Other authorization for the **admin** account cannot be edited.

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

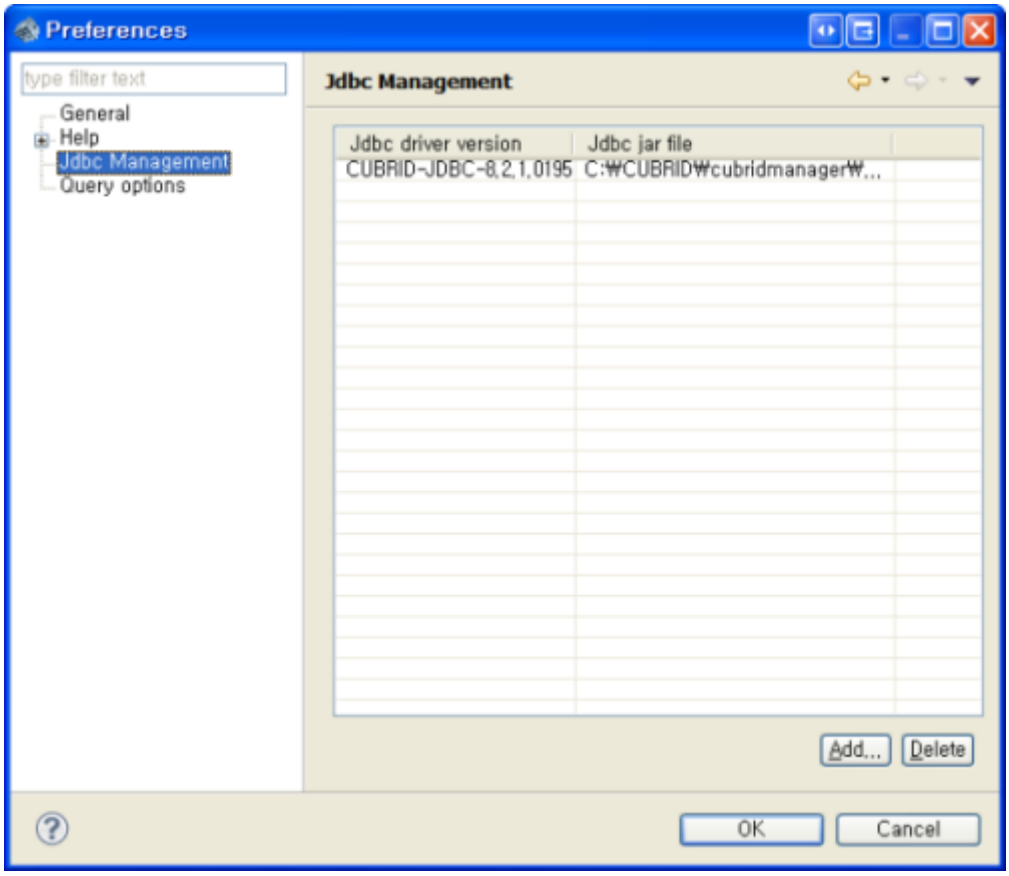
You can set whether the Help is displayed through the internal Help window or the external browser. You can also select the open mode when the Help is executed as an Eclipse plug-in.

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 is open with a single or double click in the navigation tree. (The default value is double click.)

JDBC Driver Setting



The most recent version of the CUBRID JDBC driver is provided in **`cubridmanager/plugins`**. If you want to connect to a previous version of the database to execute queries using the most recent version of the CUBRID Manager, you need to add an another version of CUBRID JDBC driver in Basic Configuration > JDBC Driver Configuration. With the multiple JDBC driver configuration feature available in CUBRID 2008 R2.1 or higher, you can execute queries by connecting to multiple versions of databases through a single manager.

However, depending on the version of the database server that is being connected to, some functions may not work. 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.

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.

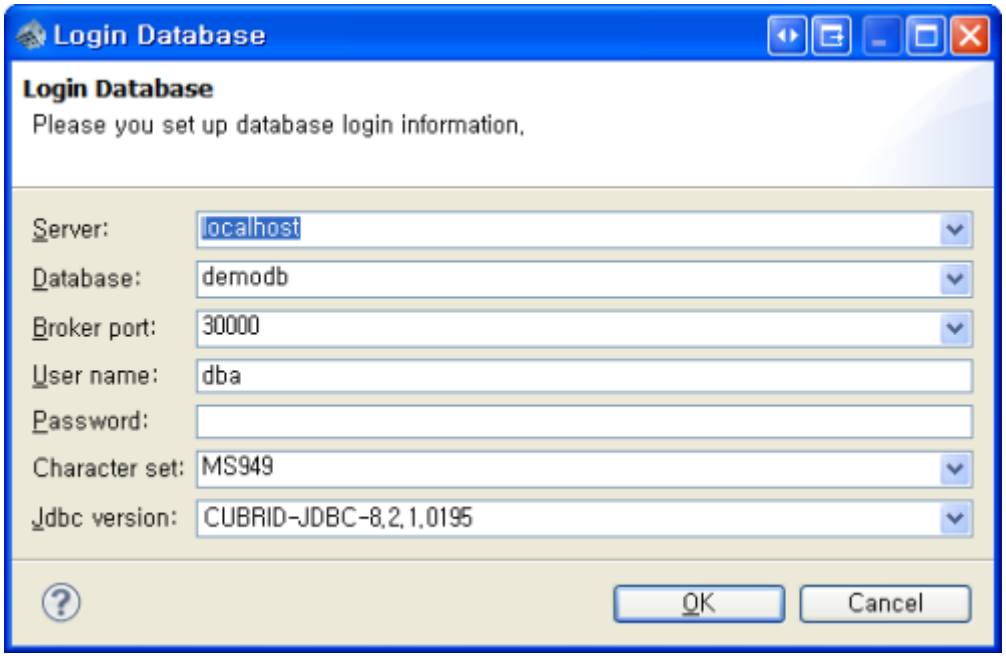
Query Editor

Query Editor Structure

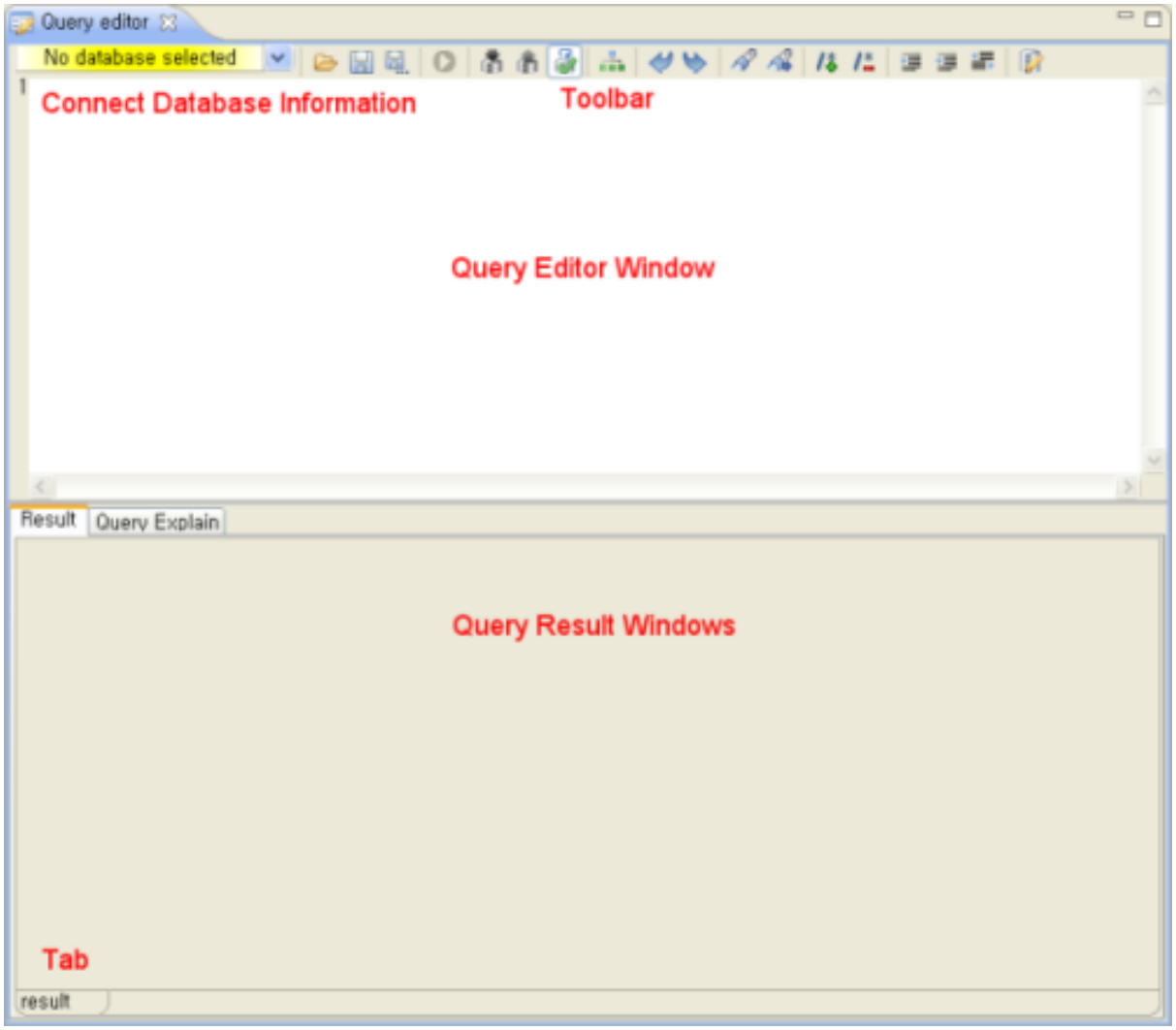
The CUBRID Manager's Query Editor is a query tool that supports execution of all **DML**, **DDL**, **DCL** statements, allowing users to edit and execute queries more easily.

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.



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.

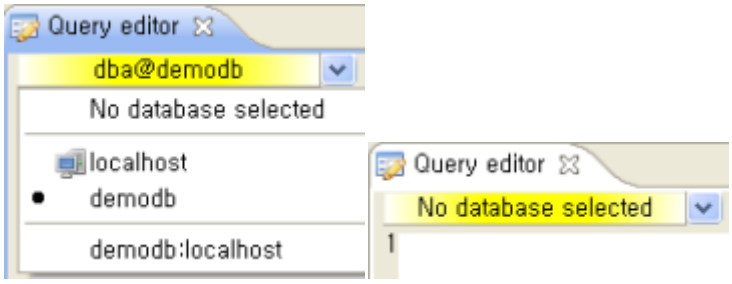


Toolbar Options

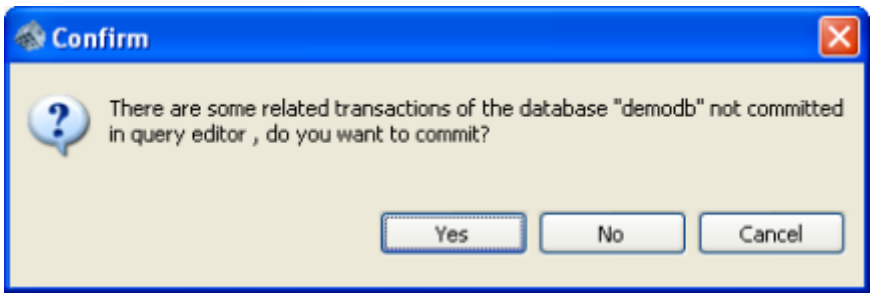
Select Database

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.



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 while auto commit is selected ( ), and activates to allow you to determine whether to commit ( ) when a transaction occurs while auto commit is not selected.
Rollback			Keeps deactivated while auto commit is selected ( ), and activates to allow you to determine whether to rollback ( ) when a transaction occurs while auto commit is not selected.
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		F6	Displays the execution plan of the selected query. For more information, see Display Query 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.

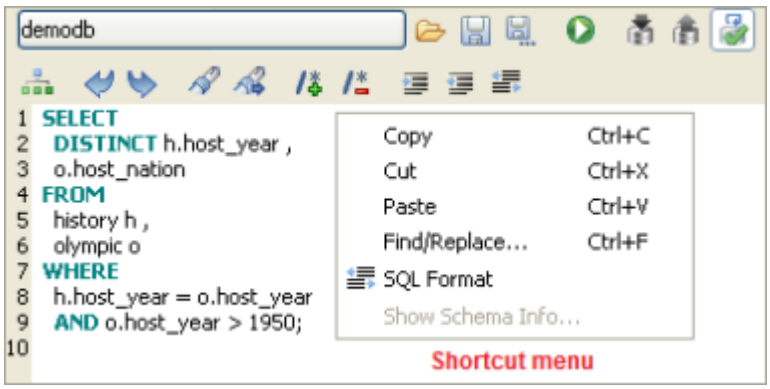
Find next		F3	Provides further search for items that are already searched once.
Convert to comment		Ctrl+/	Adds the comments to the selection area or to the line where the cursor is located in the query edit pane. Comment strings are inserted with a double dash (--).
Delete comment string		Ctrl+/	Deletes 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	Outdents 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.

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.

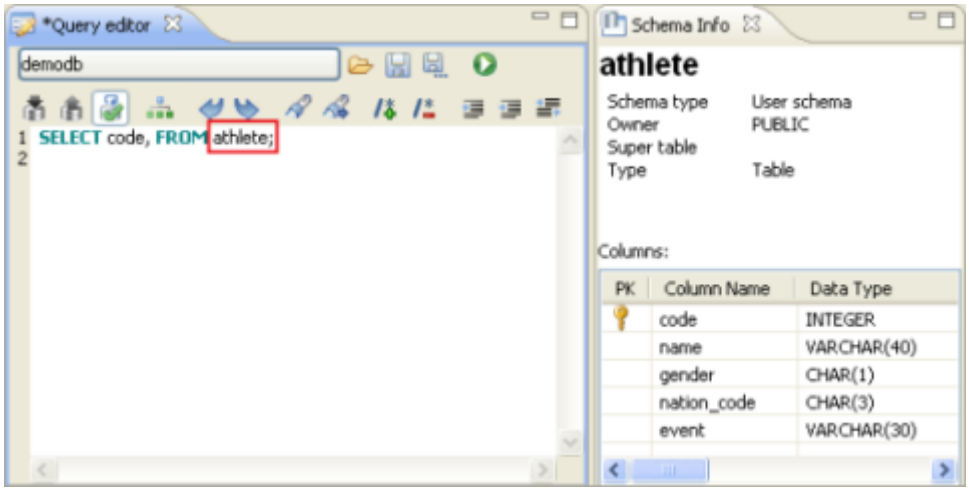
Shortcut Menu

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.



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.



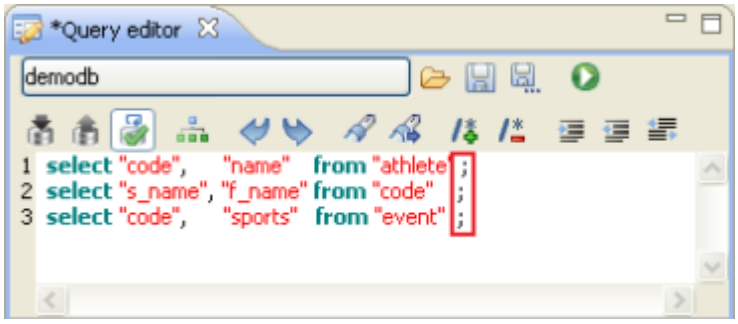
Automatic Statement Completion

If you enter a CUBRID database keyword, CUBRID automatically finds and completes the statement, increasing usability and user accessibility.



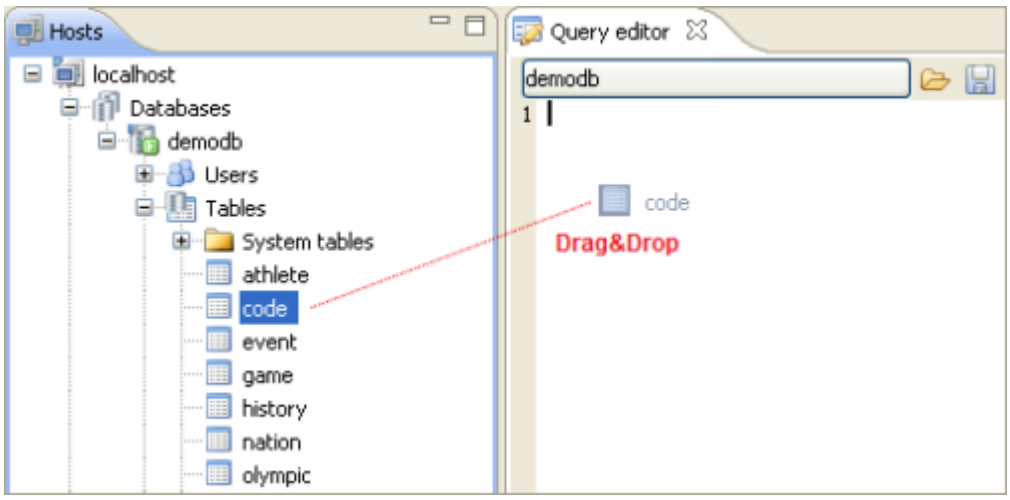
Executing Multiple Queries

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.



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

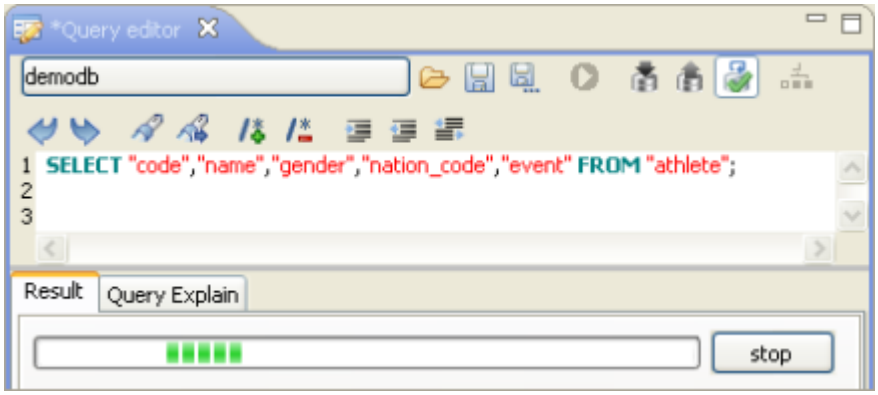
A function that stops the currently running query. This can be divided into two functions.

- **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**

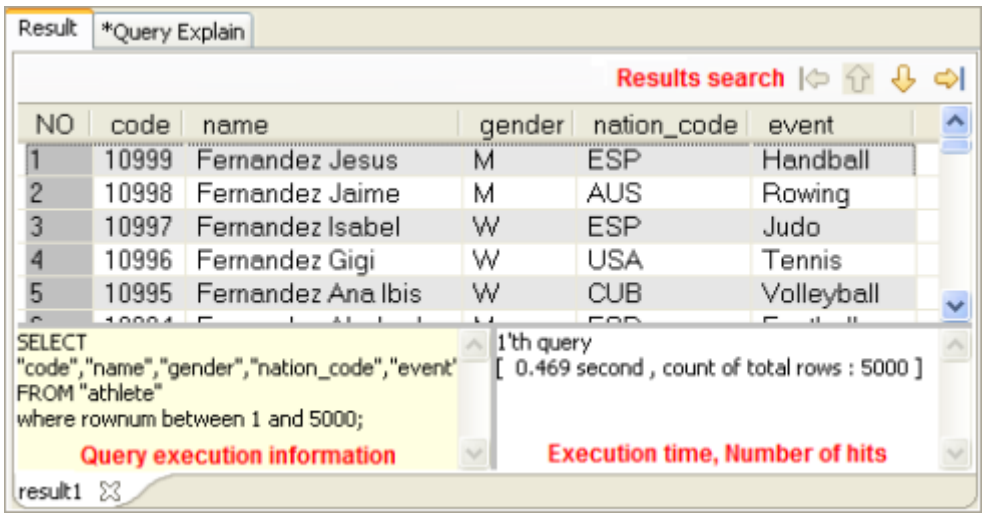
  If you click the Stop button when executing a long transaction, a query stop command is delivered to CUBRID Manager > JDBC > Broker > Server, and the query finally stops in the database server—the query is actually canceled. However, this function does not work if the Broker and the Database Server are running in a Windows system.



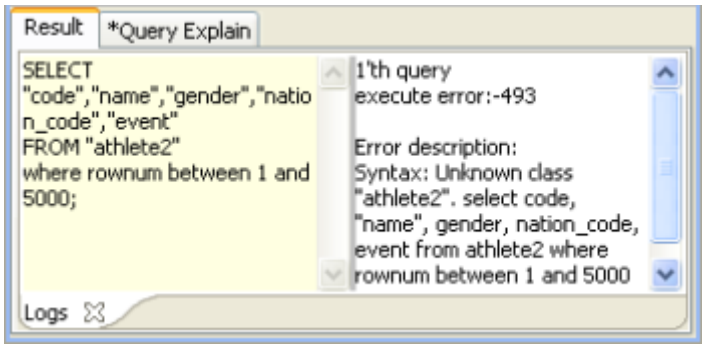
Query Results Pane

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.



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.

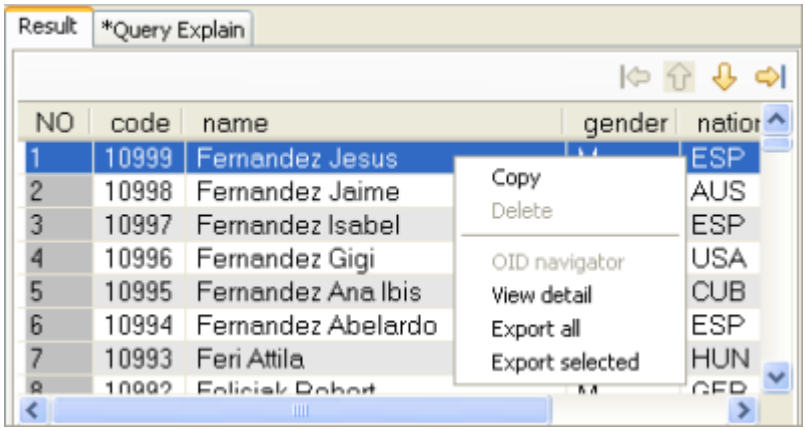


Query Results Pane Structure

- **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.

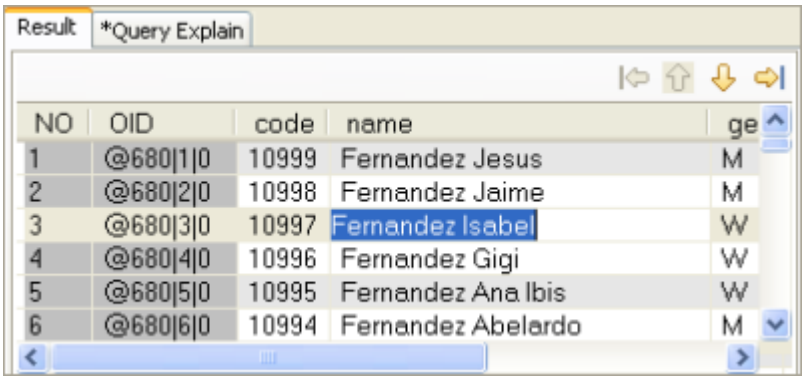
Options in the Query Results Pane

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.

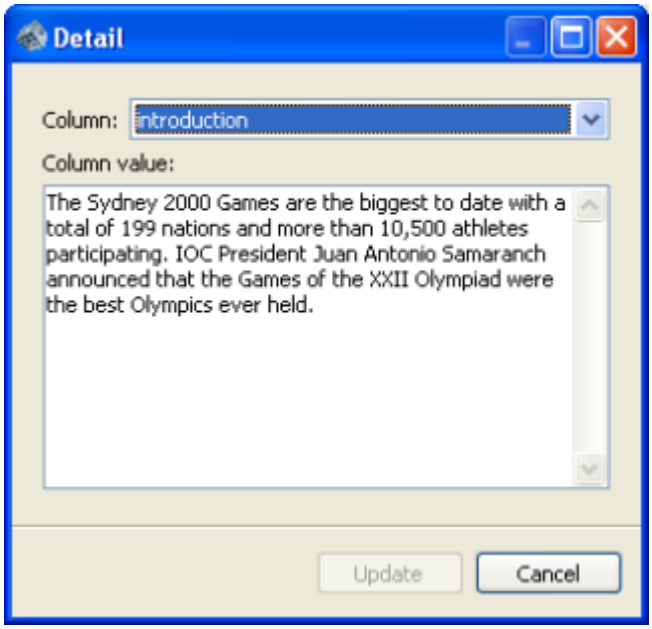


- **Copy** : Copies an entire row. To copy a specific column of the row only, select [View detail].

- **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.

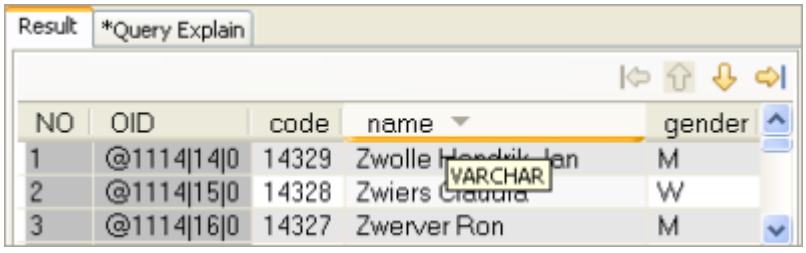


- **Delete** : You can delete data directly in the query results pane. Right-click the row to delete and then select [Delete]. This option is available only when [Get OID info] is set as the modify function does.
- **OID navigator** : This option can be activated by right-clicking an OID data. It provides 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. In addition, you can copy values for each column.



- **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.

When you place the mouse pointer over a column name in the query results pane, you can see the data type of the column.



Query Editor Options

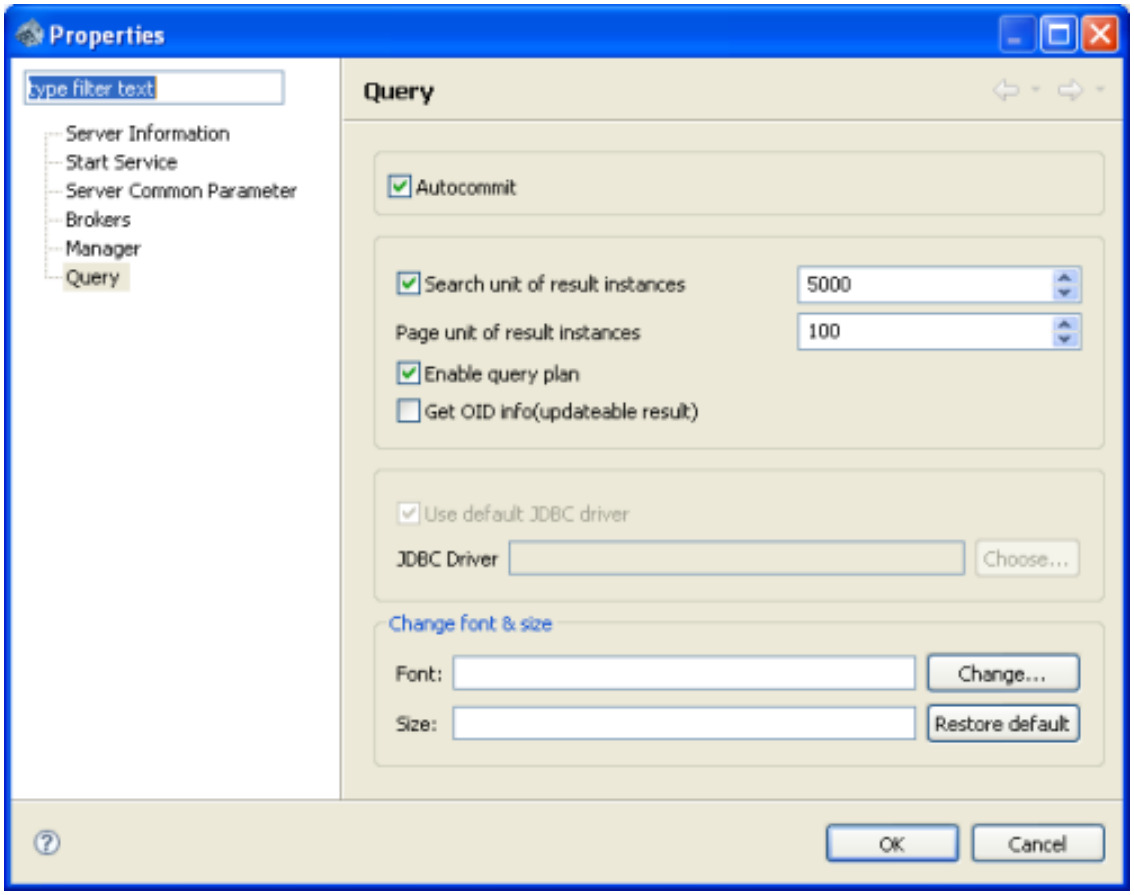
The Query Editor's options can be set in the following way. Options are applied according to the following priority.

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]

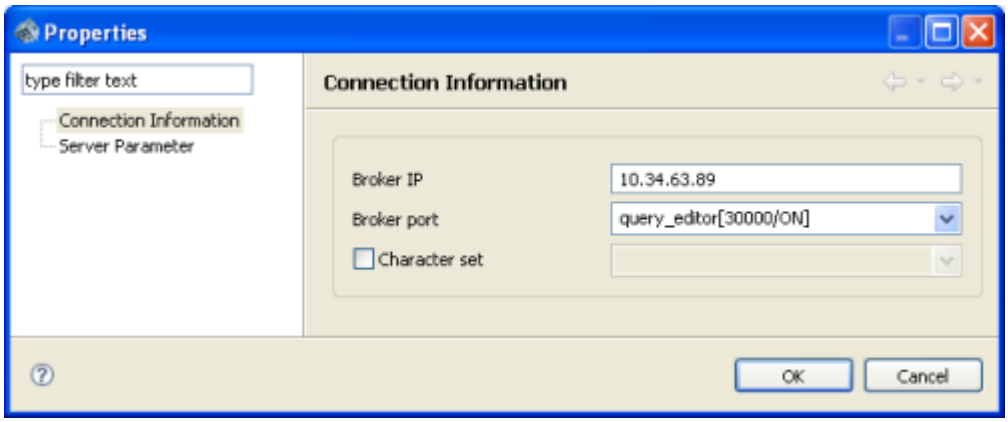


In the [Query] dialog box, you can set the following options:

- **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** : 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.

Display Query 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.

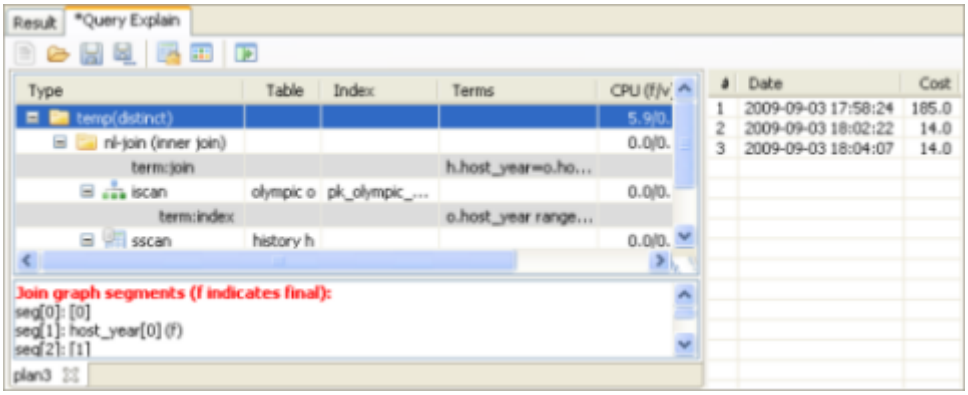
The query plan function retrieves the SQL execution plans; it is used not for a one-time purpose but for a collection purpose to continuously manage and retrieve them. Every time you retrieve a query plan, the query plan history is accumulated. You can save this accumulated data to an .xml file. When 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.



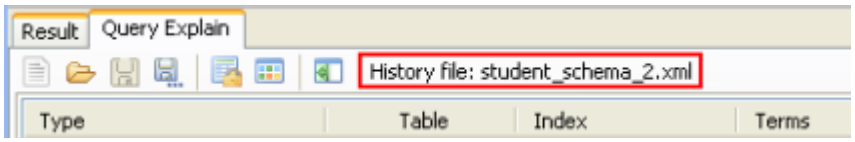
Query Explain Toolbar

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.



- **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.

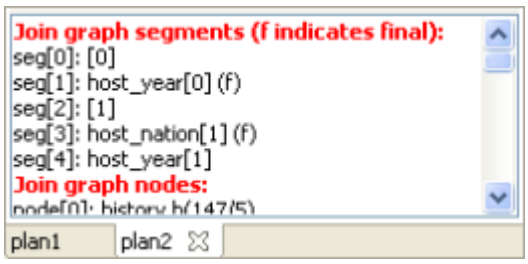
Each item in the vertical axis is called a node. Each node contains different data. You can view the tree moving from the top to the bottom.

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 **+**, subnodes 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.



Query Plan History Pane

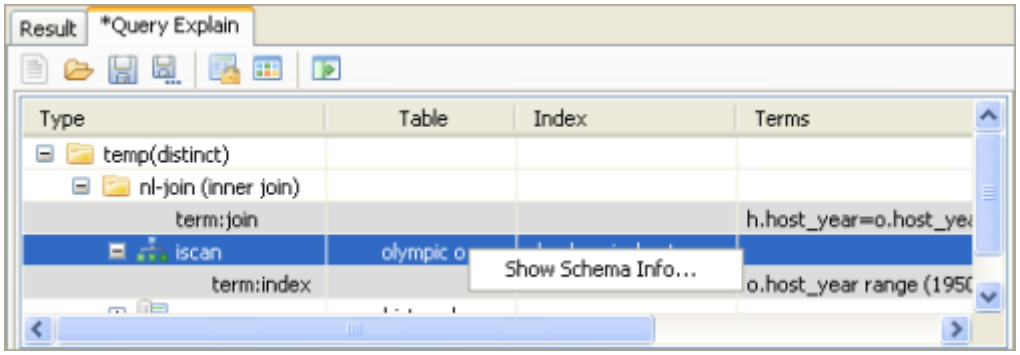
Histories are displayed accumulatively in the query plan history pane every time a query plan is executed. # is the accumulation order and corresponds to the tab number below the query plan display pane. Date indicates the date when a query plan is executed, and Cost is the sum of CPU and Disk I/O costs. 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

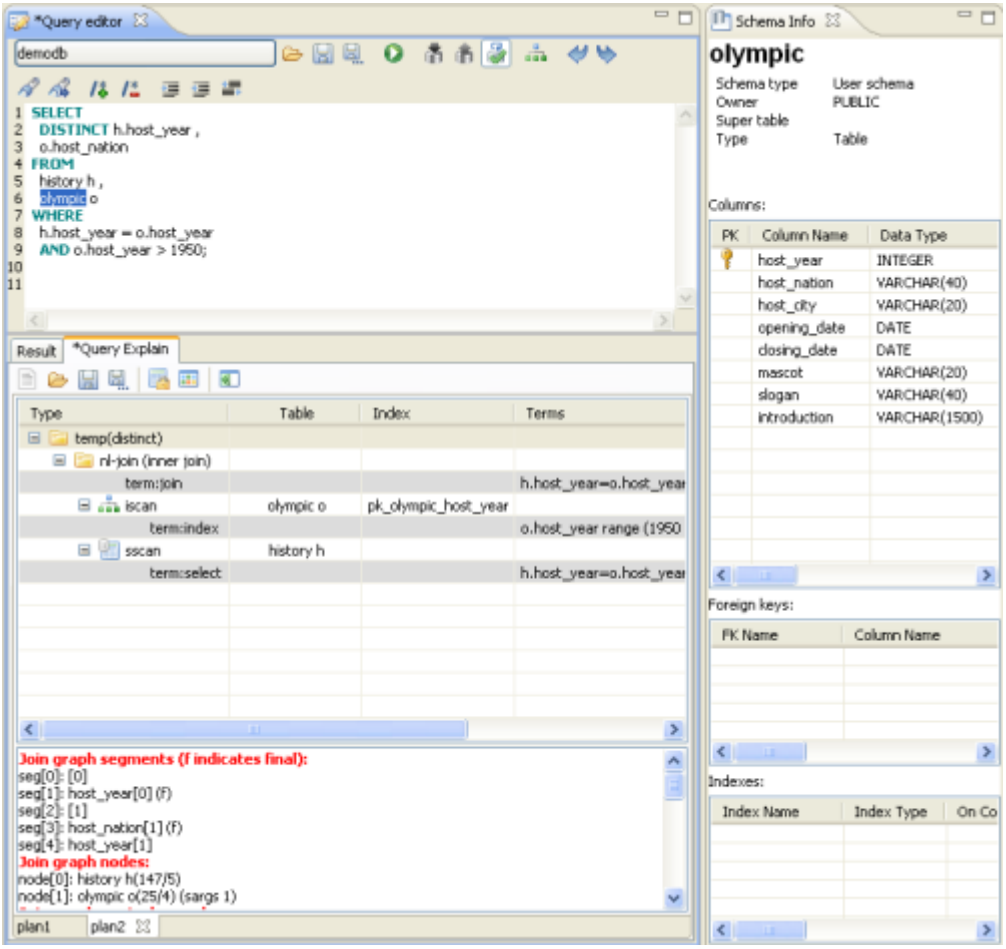
Using Query Plans

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.



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.


*Query editor
demodb
1 SELECT
2 DISTINCT h.host_year ,
3 o.host_nation
4 FROM
5 history h ,
6 olympic o
7 WHERE
8 h.host_year = o.host_year
9 AND o.host_year > 1950;
10
11
Result
*Query Explain
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)
plan1
plan2
Schema Info
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.

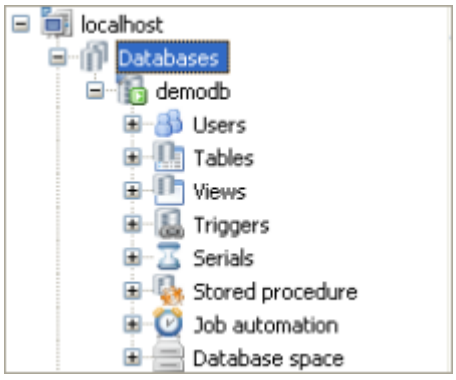
You can create databases, and set common database parameters that affect the entire host by selecting [Properties].



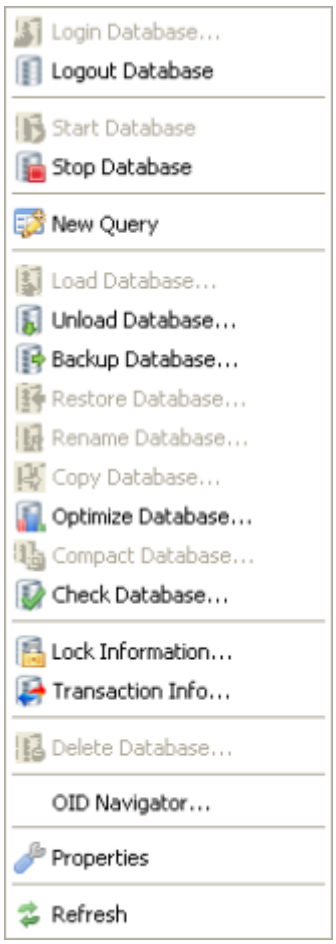
Individual Database Management/Development Function

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.

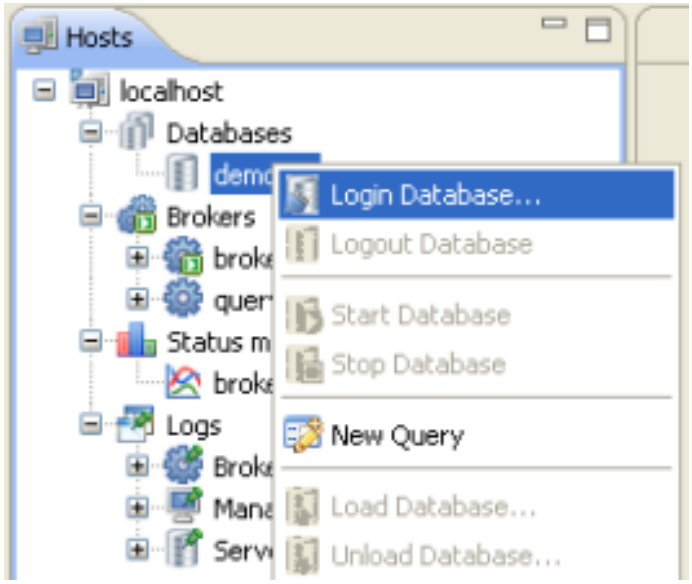


Shortcut menus provide all functions related to the operation. All functions except for the OID navigation function can be used even when the database is not running.

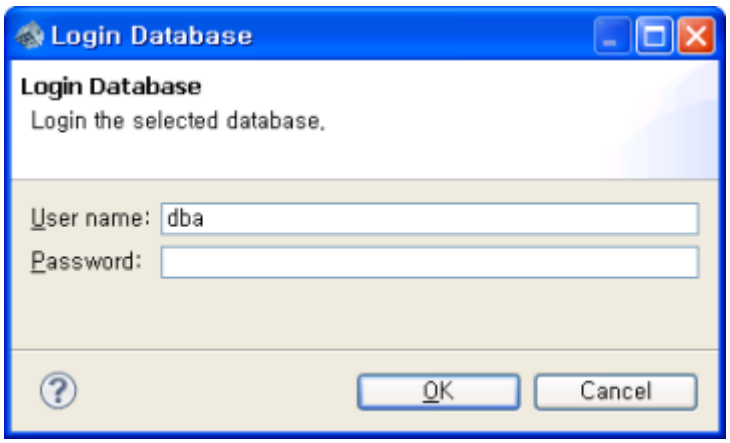


Connecting to a Database

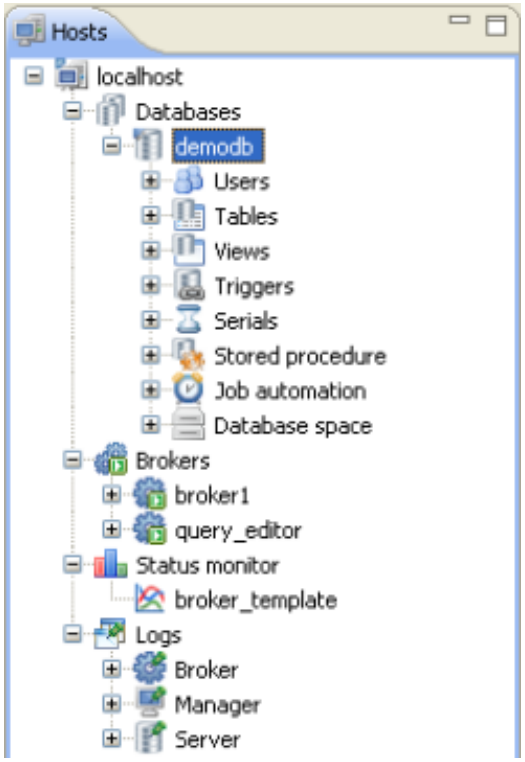
To run a database, you need to log into the database first. In the host navigation tree, right-click a database and select [Login Database] or double-click the database.



Enter a user name and password in the [Login Database] window. The default user name is **dba**; no password is required.



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.

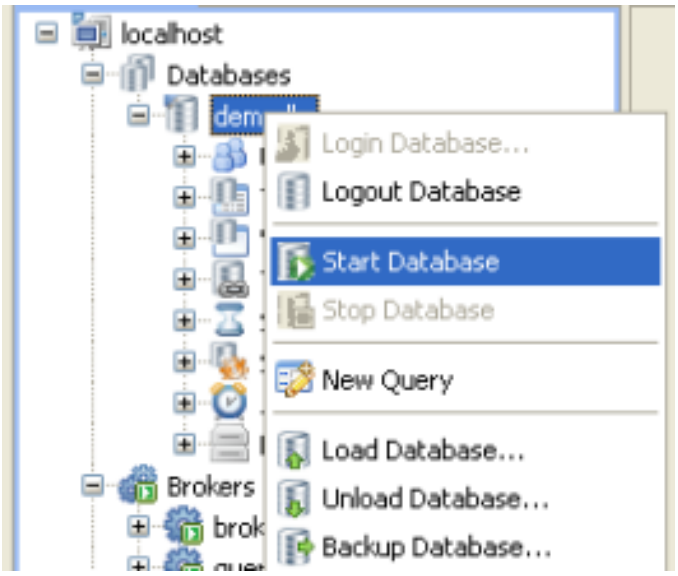


In the CUBRID Manager is not normally running, the following error message will be displayed: Cannot connect to a server. Please check the configuration environment of the CUBRID Manager server and other connection. In this case, you solve the problem by checking the following:

- 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.

Starting a 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.



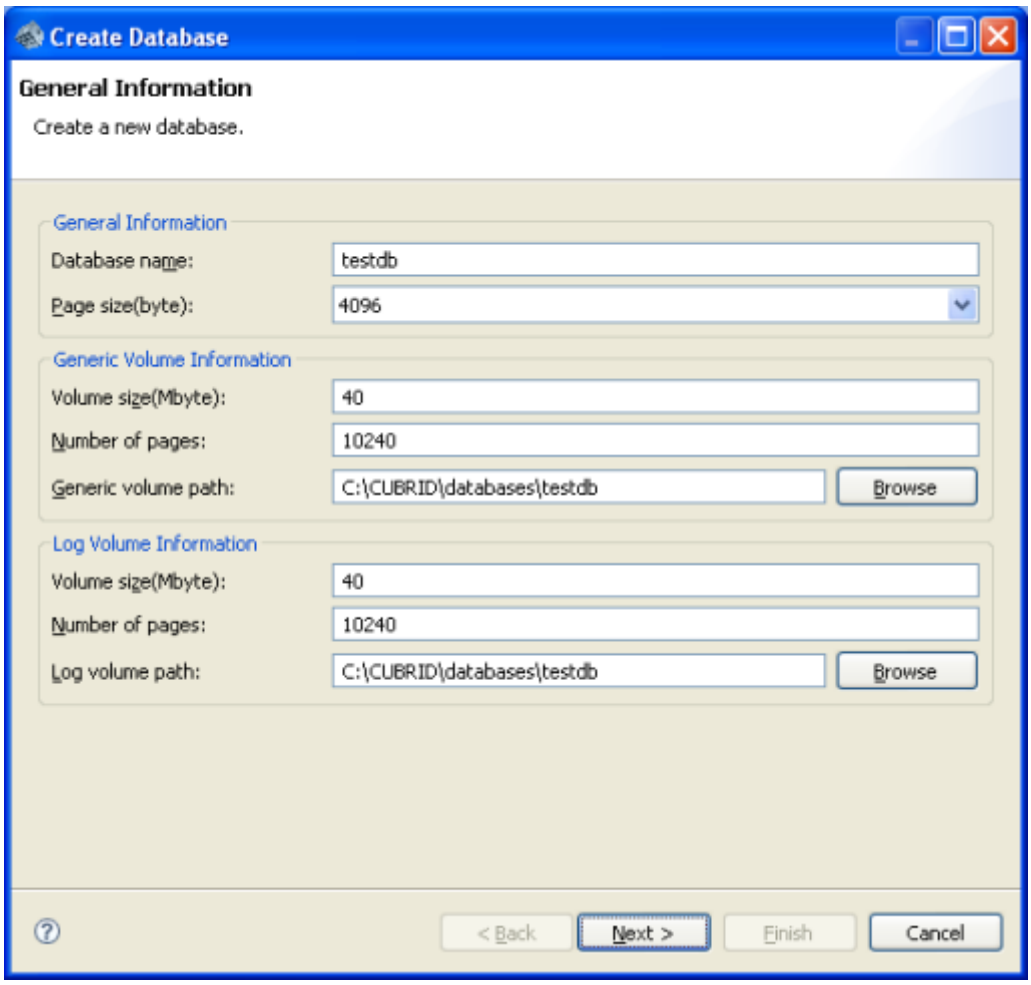
Create Database

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

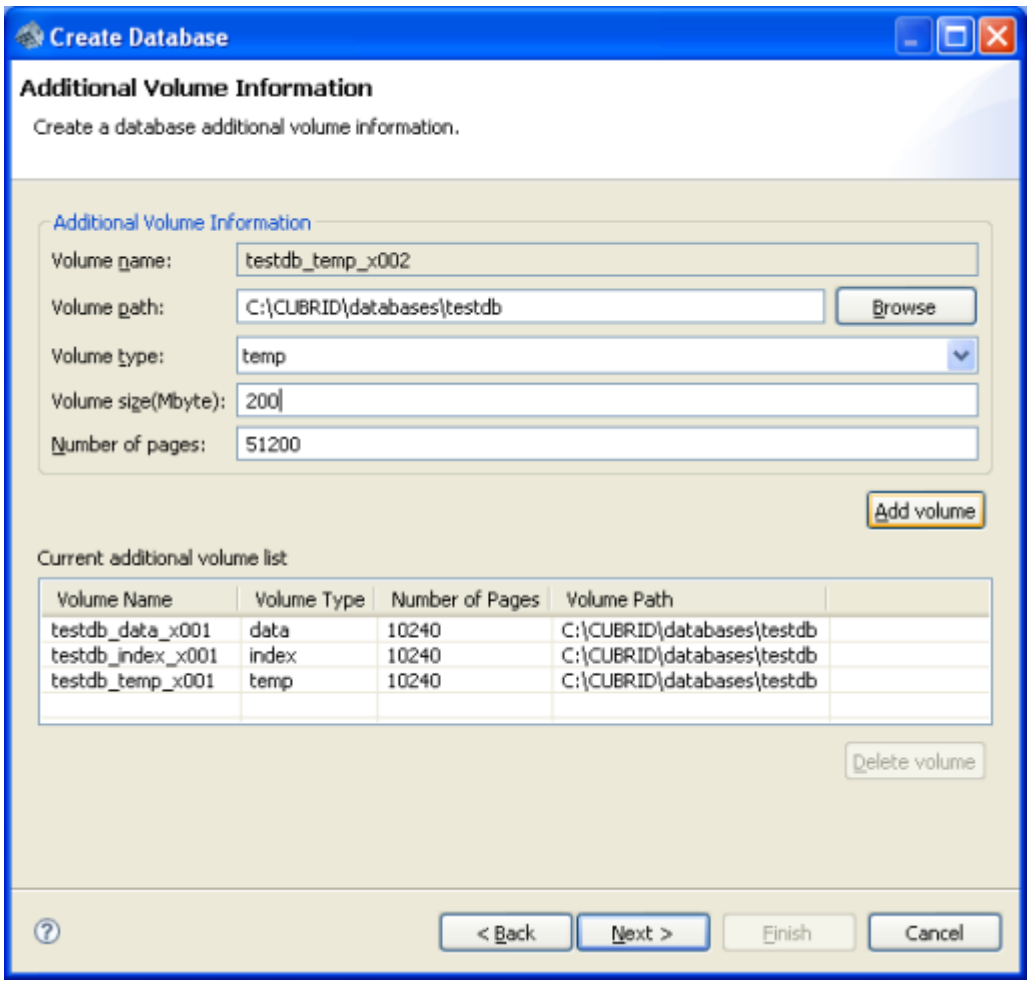
Enter the basic information such as database name, the generic and log volumes.



- **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

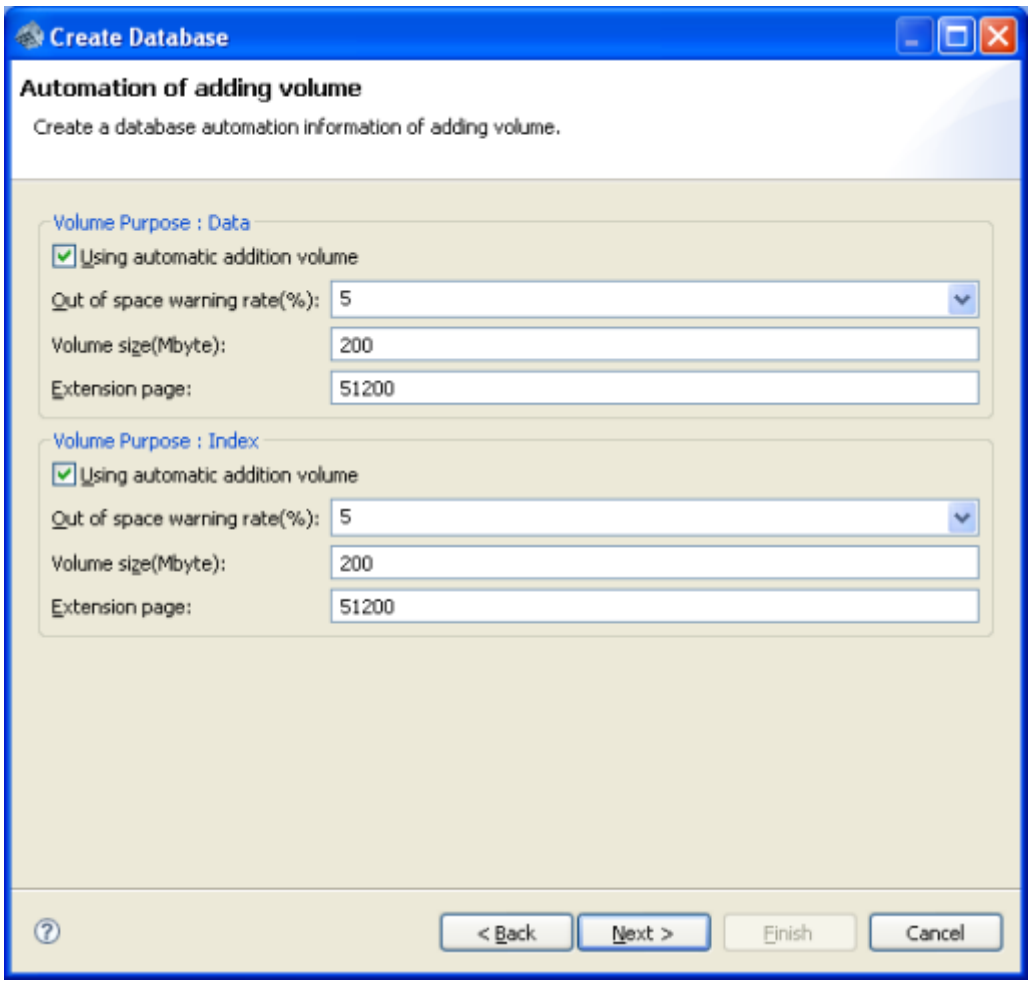
Enter information of additional volume by type such as generic, data, index, temp. In order to use the Automation of adding volume function for each volume type, you must configure the data volume and index volume.



- **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

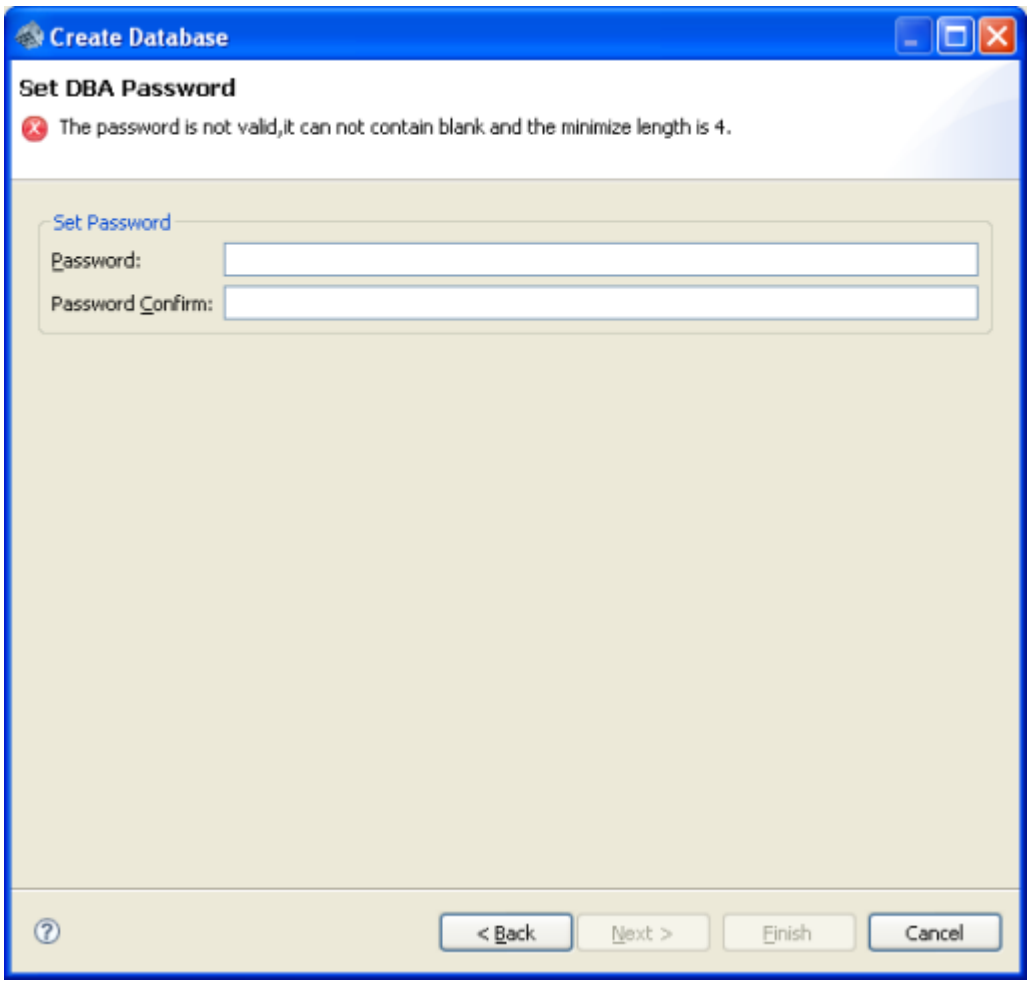
Enter information used for the automatic addition feature when data or index volumes do not have enough space. The automatic addition functionality can be used for a data or an index volume.



- **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.

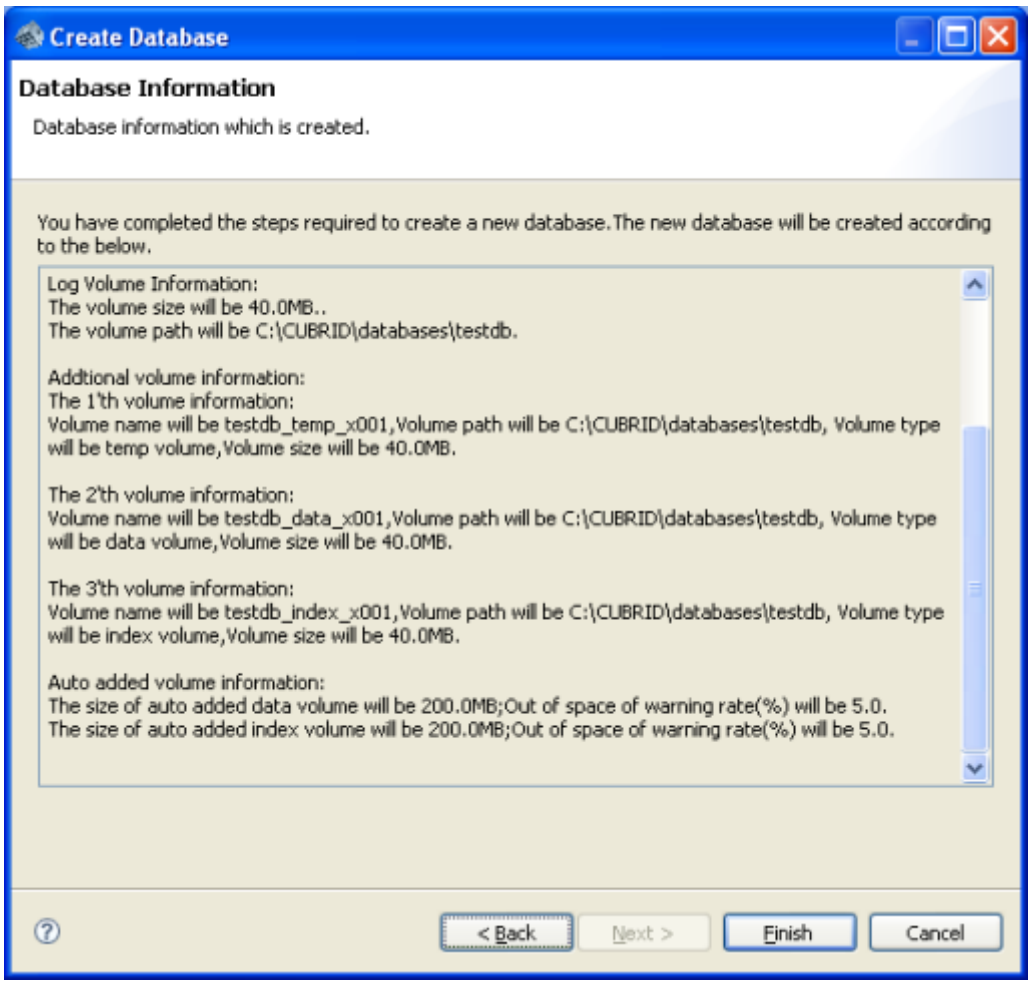
Step 4 : Set DBA Password

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.



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 Configuration

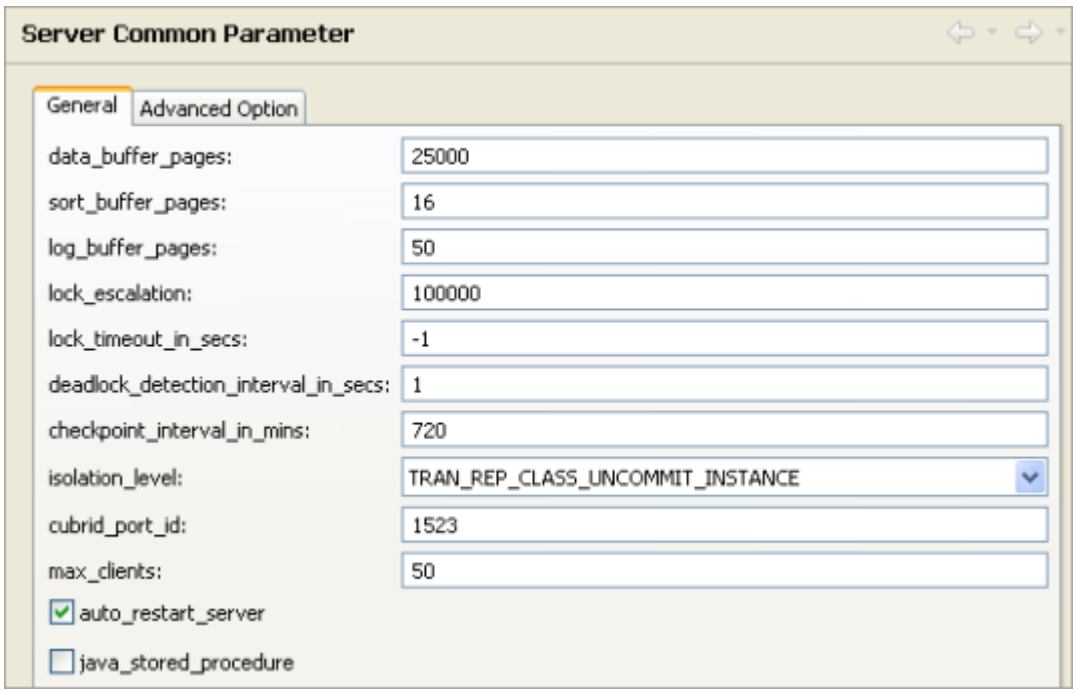
You can set database server parameters as follows:

- 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].

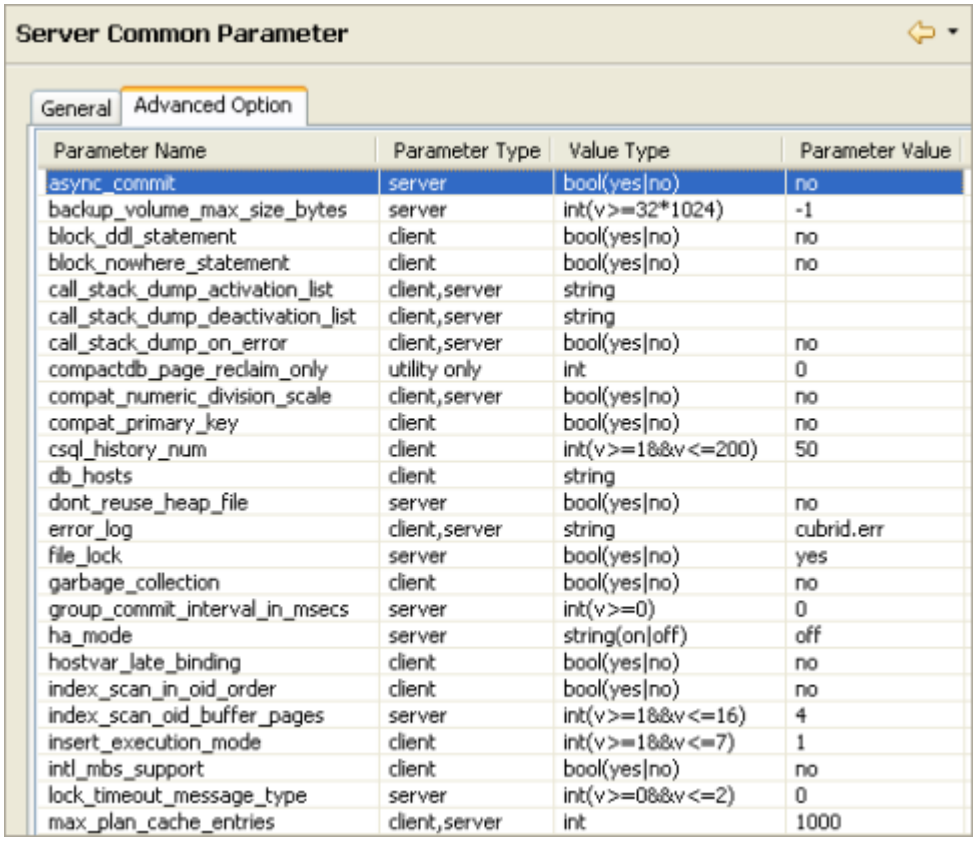
Server Parameter Setting

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.

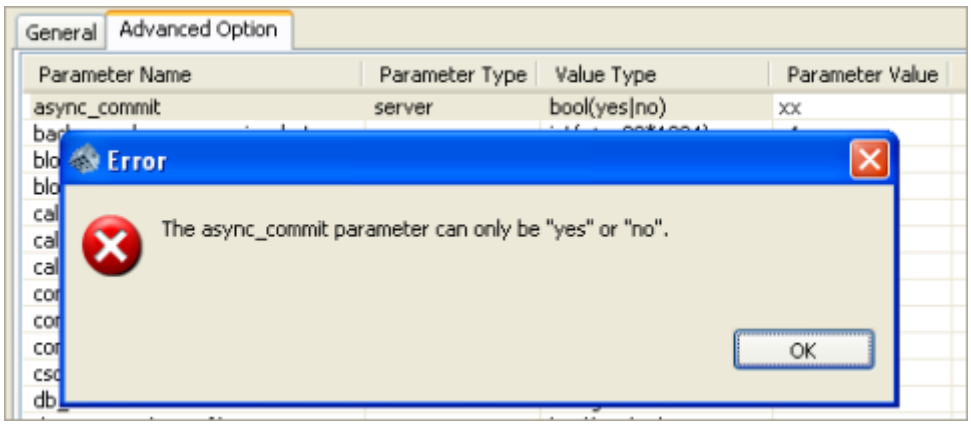
- **General**



- **Advanced Option**



A validation check is performed for all values set in [Server Parameter] immediately upon editing so that the user's mistake can be minimized.

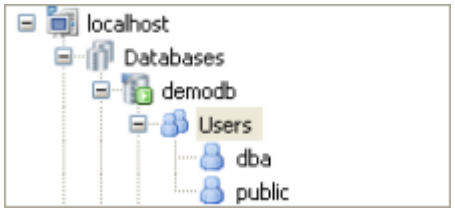


Server-Specific Parameter Setting

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.

Users

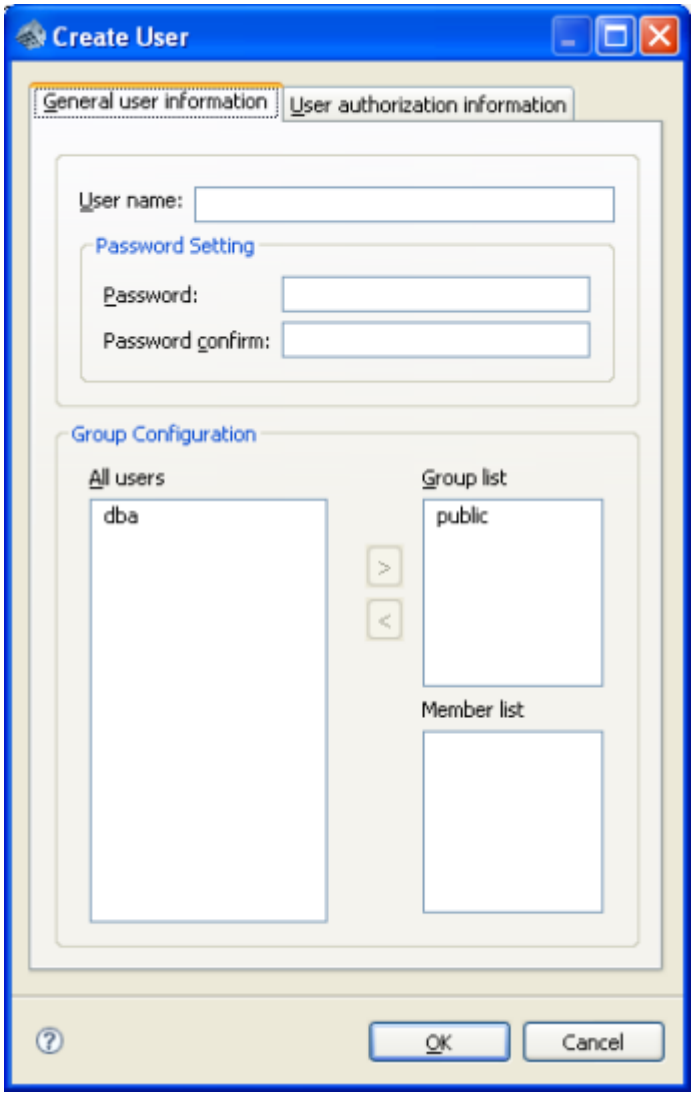
You can add, edit or drop users by right-clicking [Users] in the navigation tree after login to the database. All users of the current database are displayed in the subnode of [Users].



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.

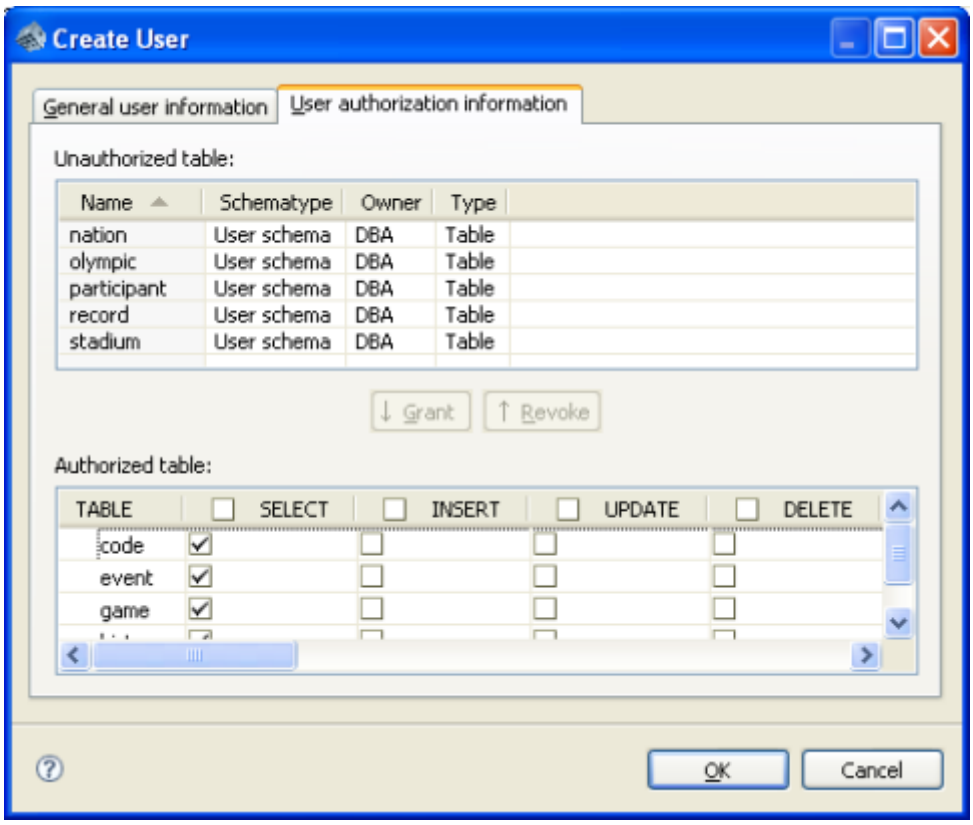
General User Information



- **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.



Edit User

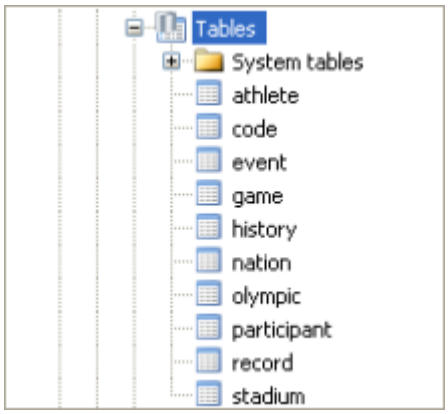
You can edit the settings entered in the [Add User] dialog box. However, the user name cannot be modified.

Drop User

You can drop the selected database user account.

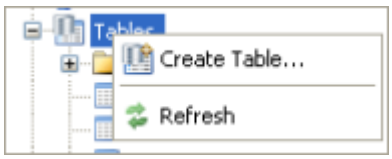
Table

When you login to the database, you can see accessible tables and system tables in the navigation tree.

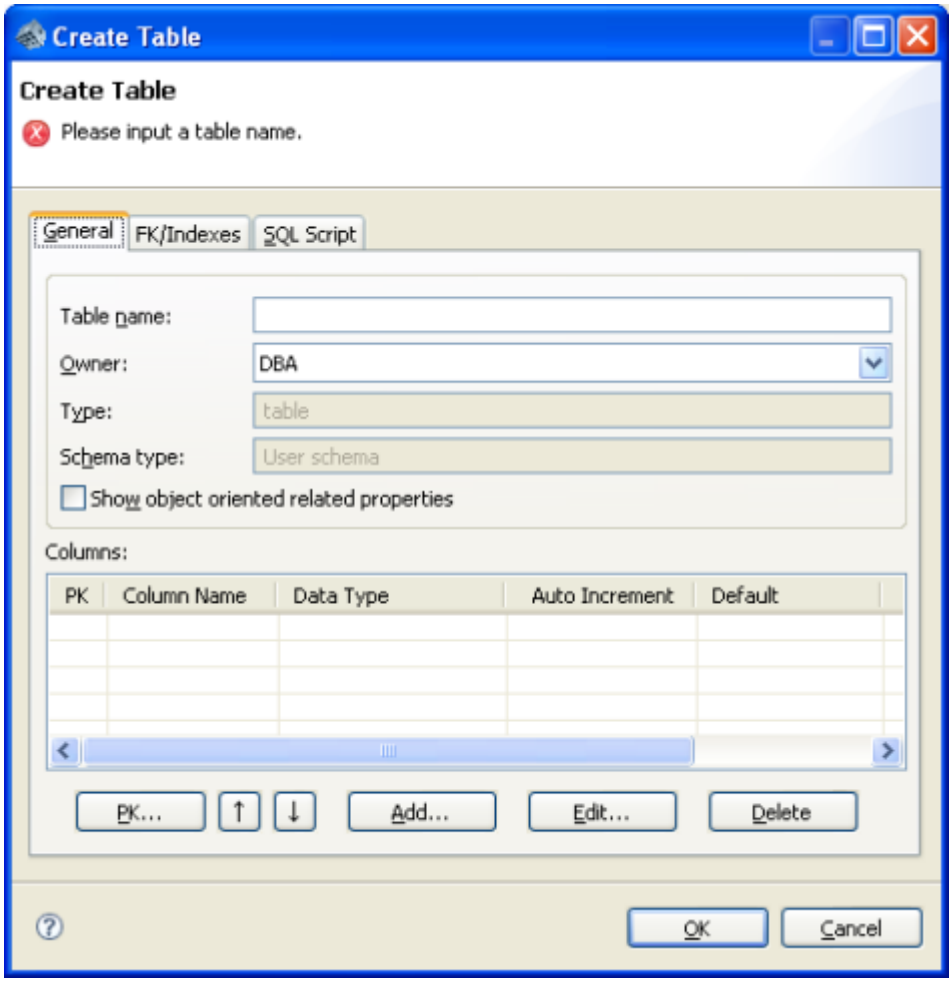


Create Table

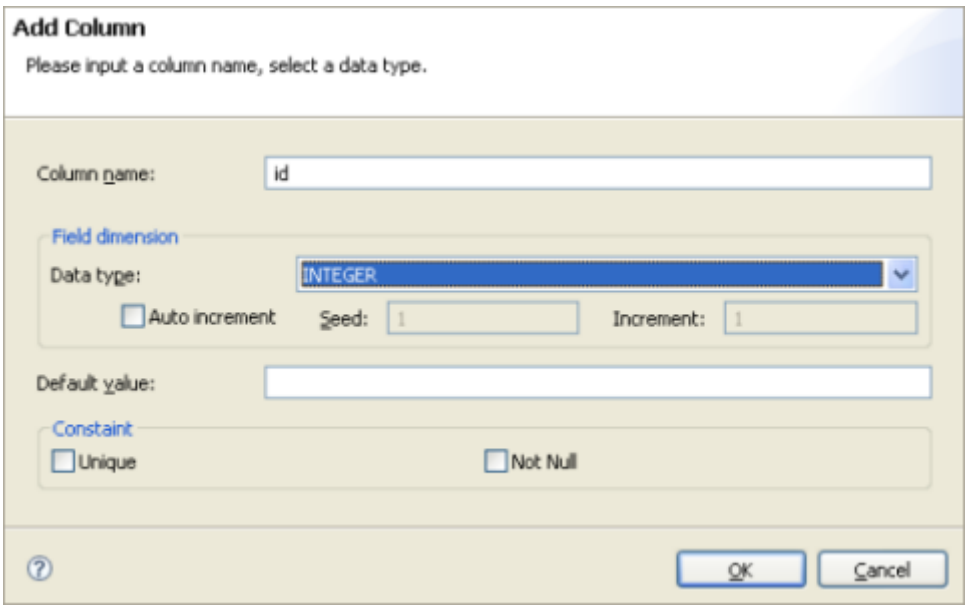
Right-click [Tables] in the navigation tree and then select [Create Table] or click [Create Table ] from the toolbar.



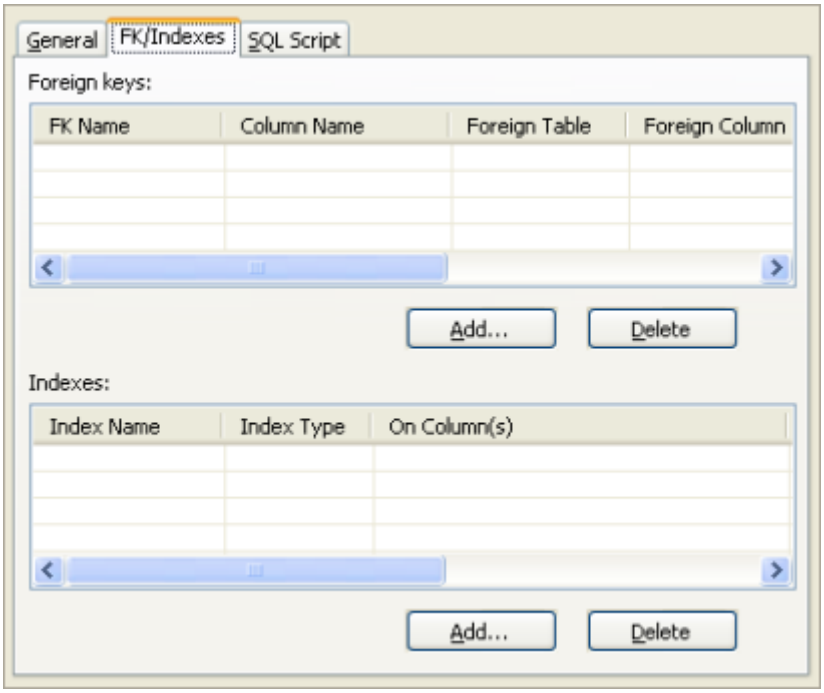
Create Table



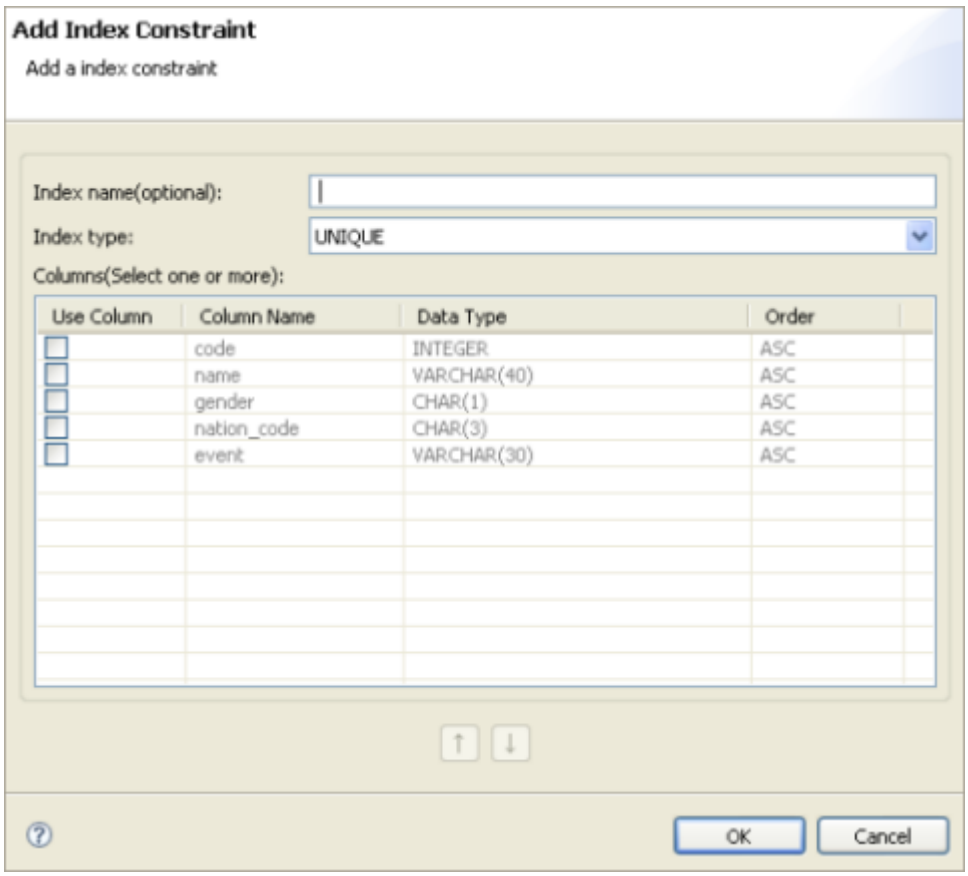
- **[General] tab** : You can define the name of the table to be added and add, edit and drop columns. You can also set the primary key (PK) and, before the table is created, adjust the position of the selected column by using [↑] and [↓].
  When you perform add or edit operations, the Add Column wizard appears as shown below. If what is entered is not grammatically correct, a warning message is displayed, or the selection is disabled so that a grammatically incorrect entry cannot be selected.



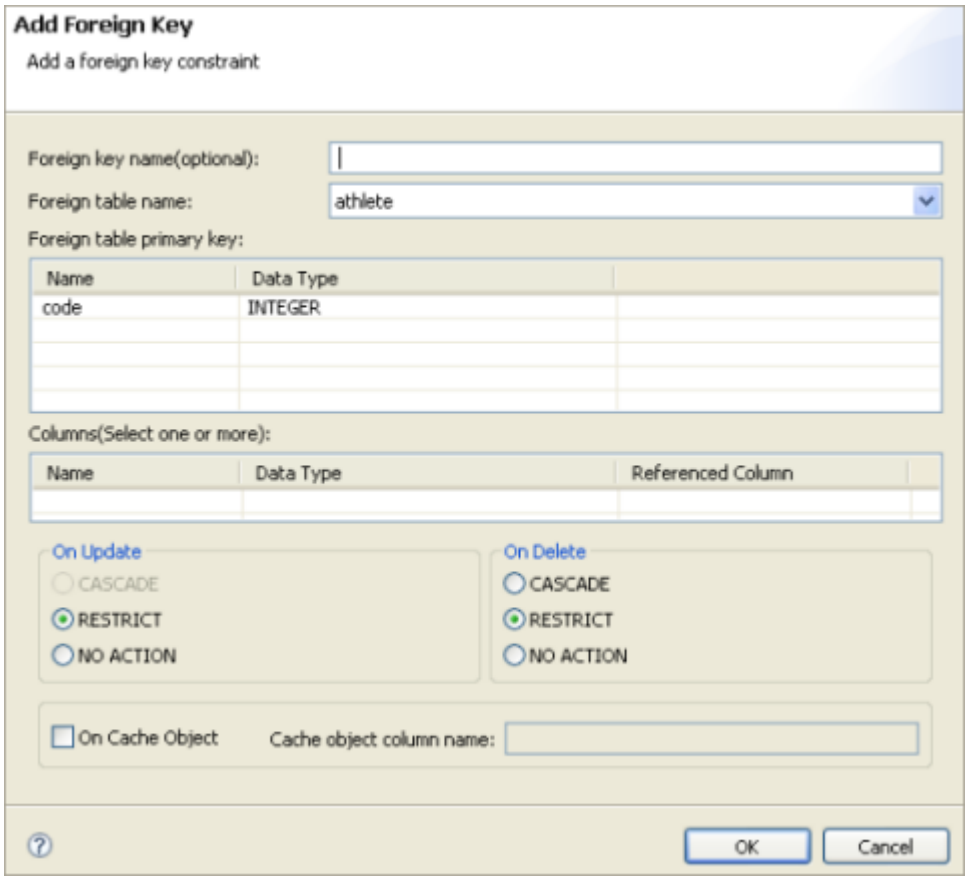
- **[FK/Indexes] tab** : You can set foreign keys and indexes.



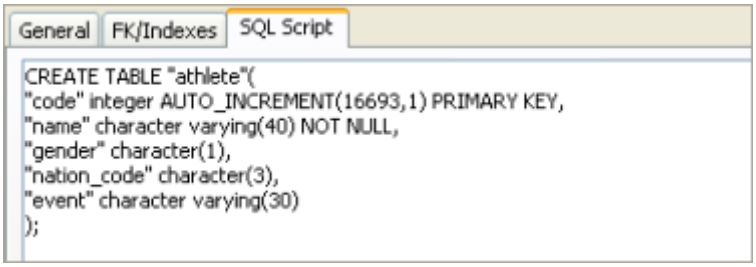
- **Add Index** : The Add Index wizard is available for Index-related features. Ascending (asc) and descending (desc) sorting can be selected within the supported range. For reverse indexes, only descending sorting is supported.



- **Add Foreign Key** : The Add Foreign Key wizard is available within the range supported by the database.

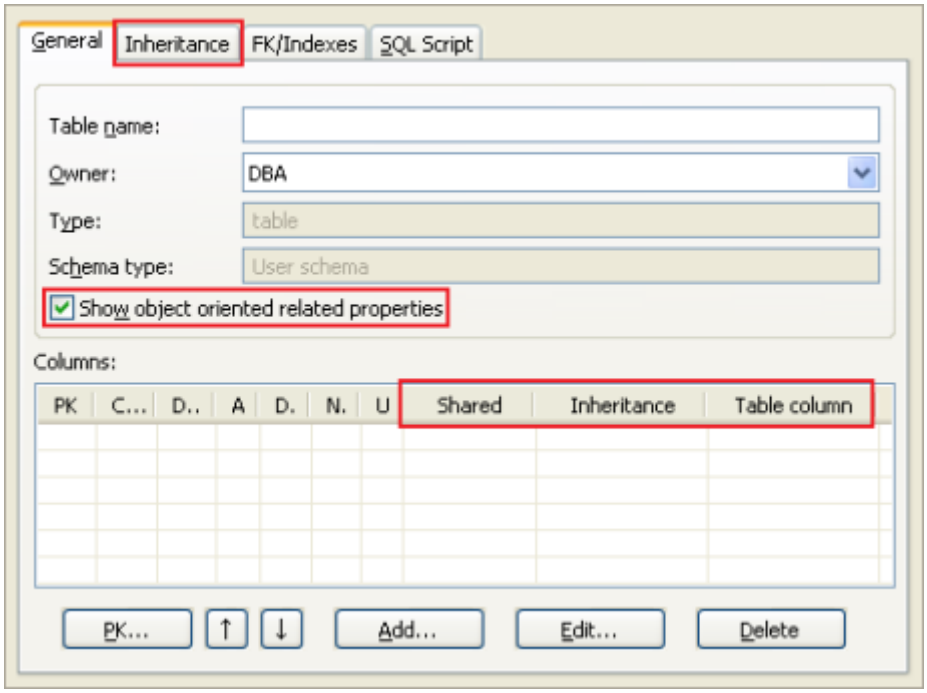


- **[SQL Script] tab** : In the [SQL Script] tab, you can check or copy all SQL statements created in the General and FK/Index tabs.

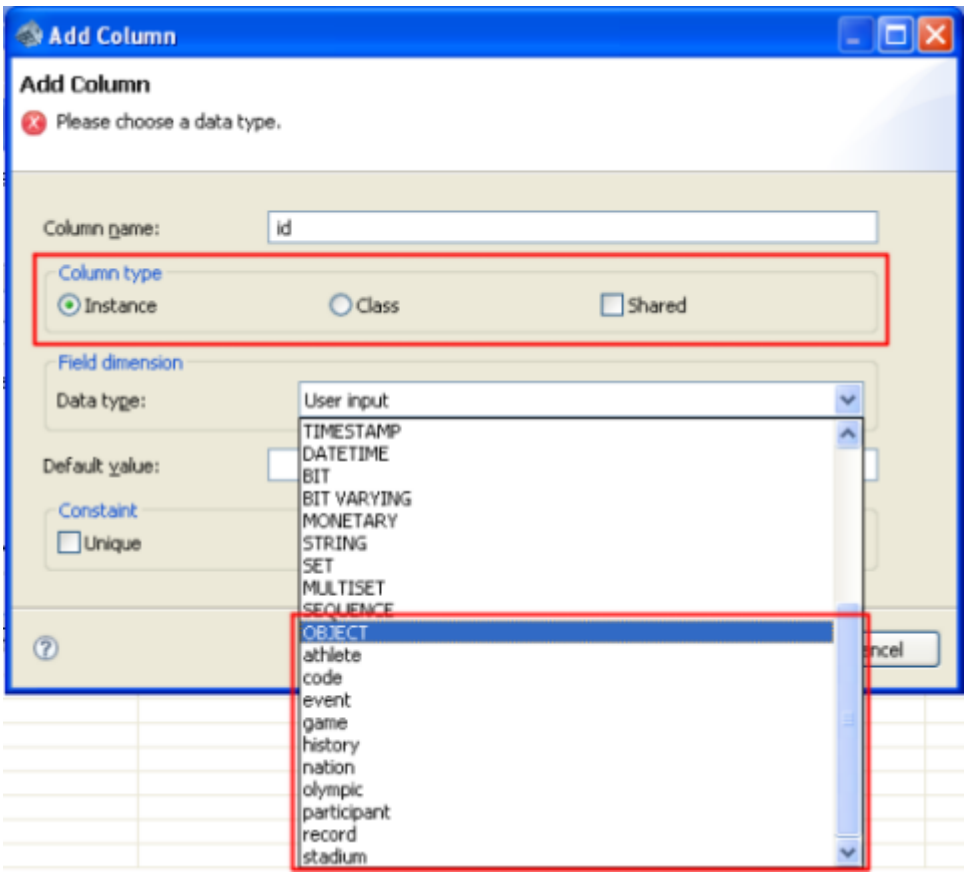


Adding Object Oriented Tables

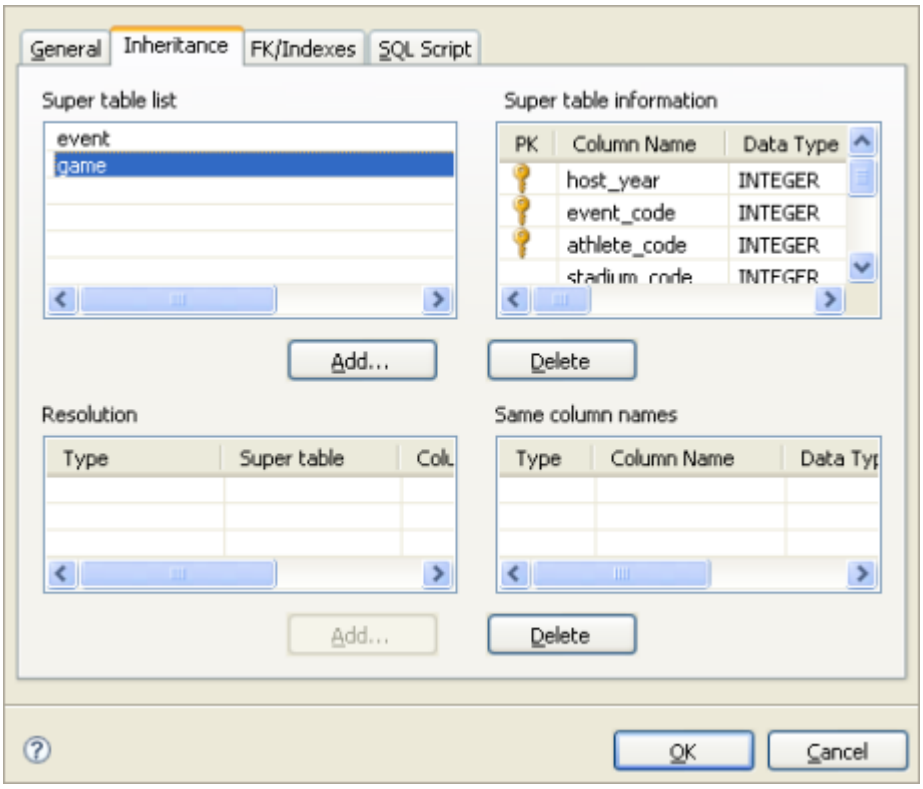
To add a table with object-oriented properties, select [Show object oriented related properties]. When you select [Show object oriented related properties], the Inheritance tab is added.



- **[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.



- **[Inheritance] tab** : You can define the super table to inherit from. If a column name conflict occurs, it can be adjusted.



Select All

Right-click a table in the navigation tree and select [Select All]. Or you can drag and drop the table into the edit or 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
```

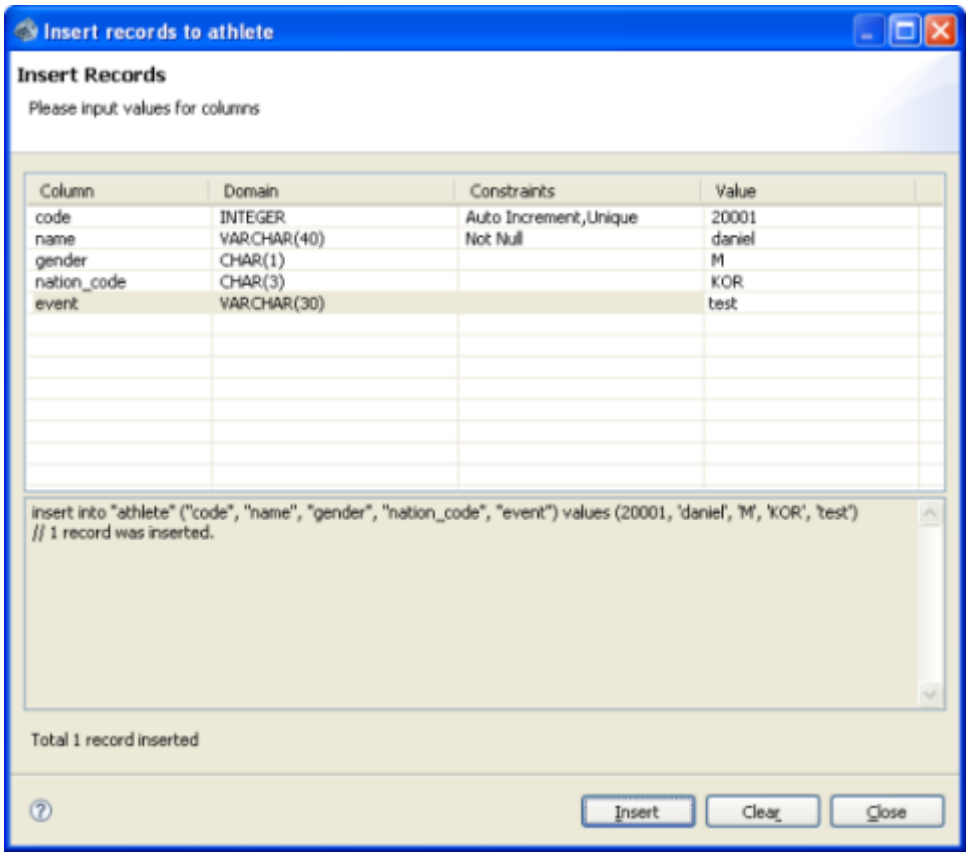
Delete All Records

Deletes all records from the table and performs the same functionality as the following syntax.

```
DELETE FROM table
```

Insert Records

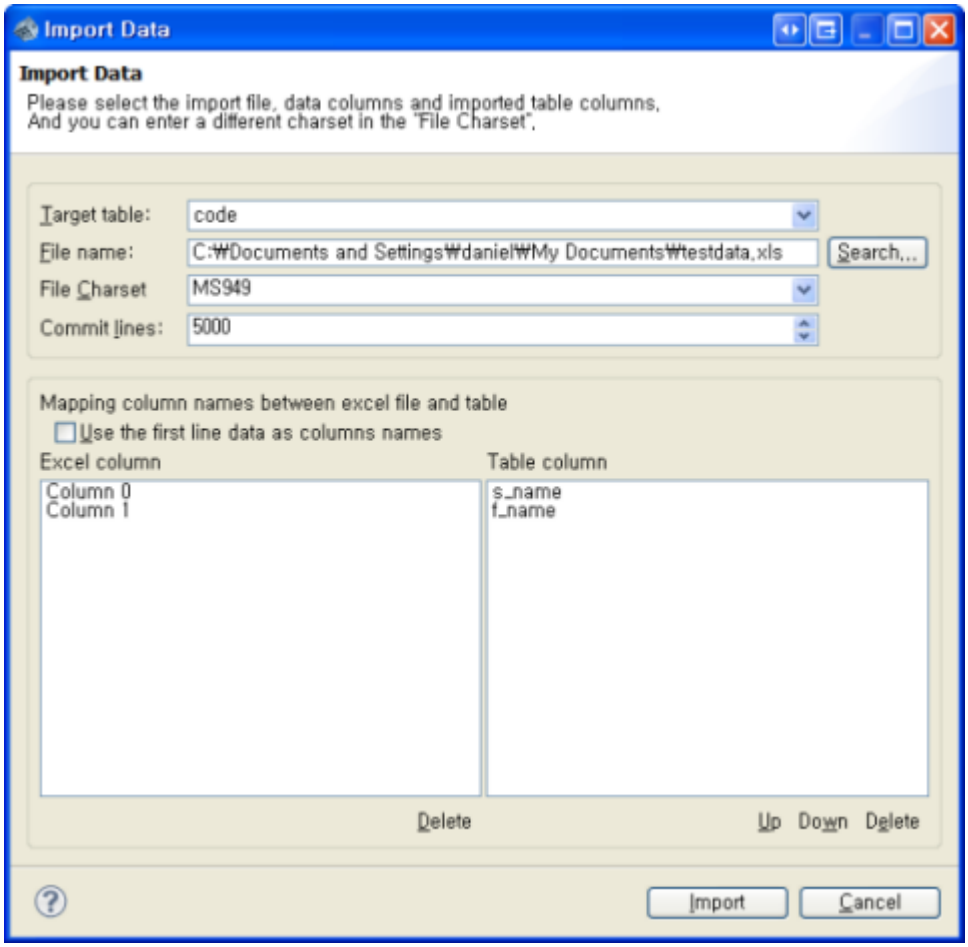
You can insert values for each column while checking its type and constraints.



- 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 table. When character set is different between data to import and a target database, you can configure the character set of the file before importing data into the database.



Export Data

You can export all data of the table to an Excel, CSV, SQL or CUBRID load format.

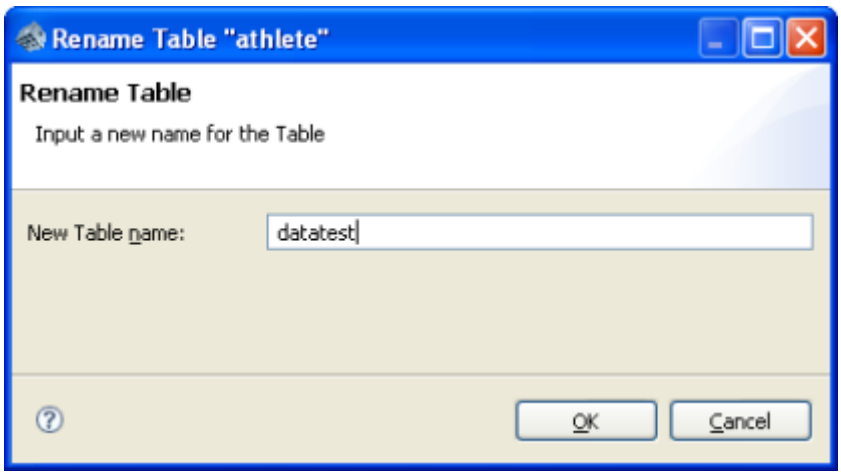
Drop Table

Drops the selected table and performs the same functionality as the following syntax.

```
DROP FROM table
```

Rename Table

You can change the name of the current table.

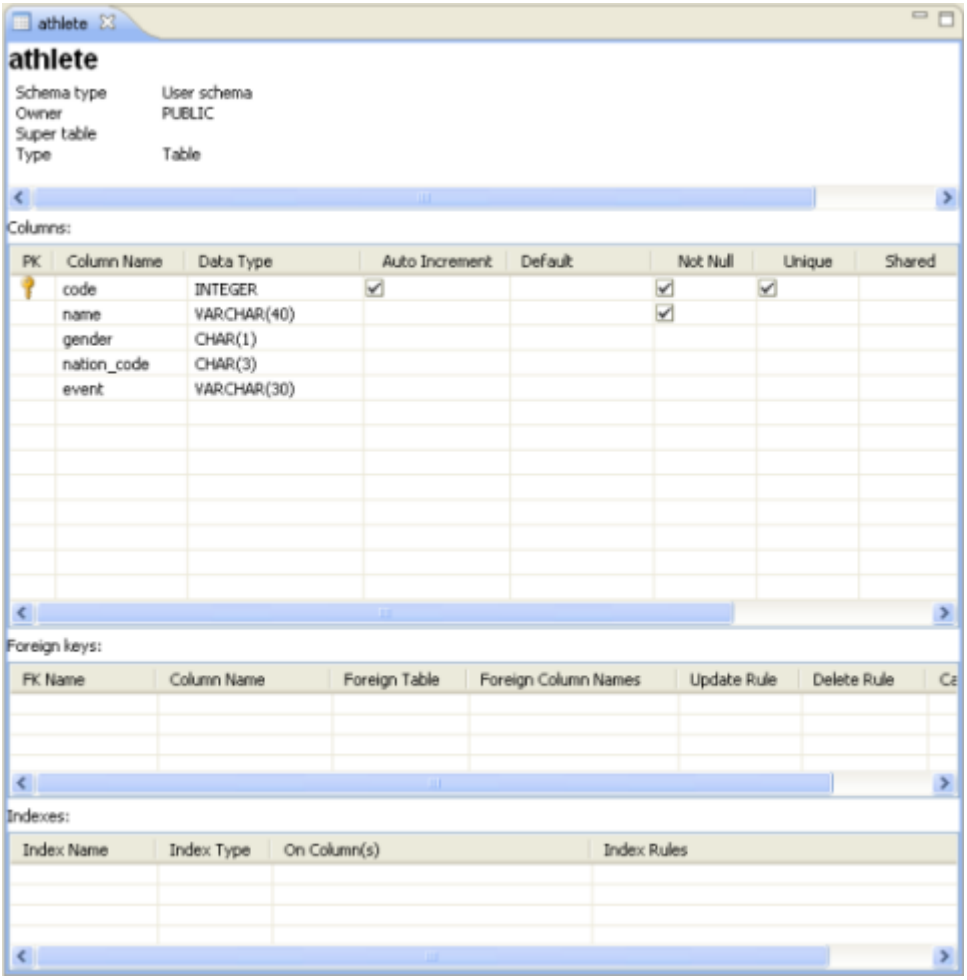


Edit Table

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.



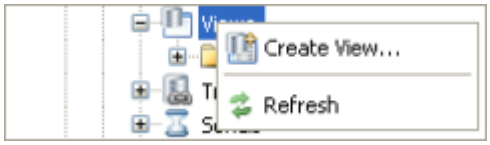
Views

When you login to the database, you can see accessible views and system views in the navigation tree.

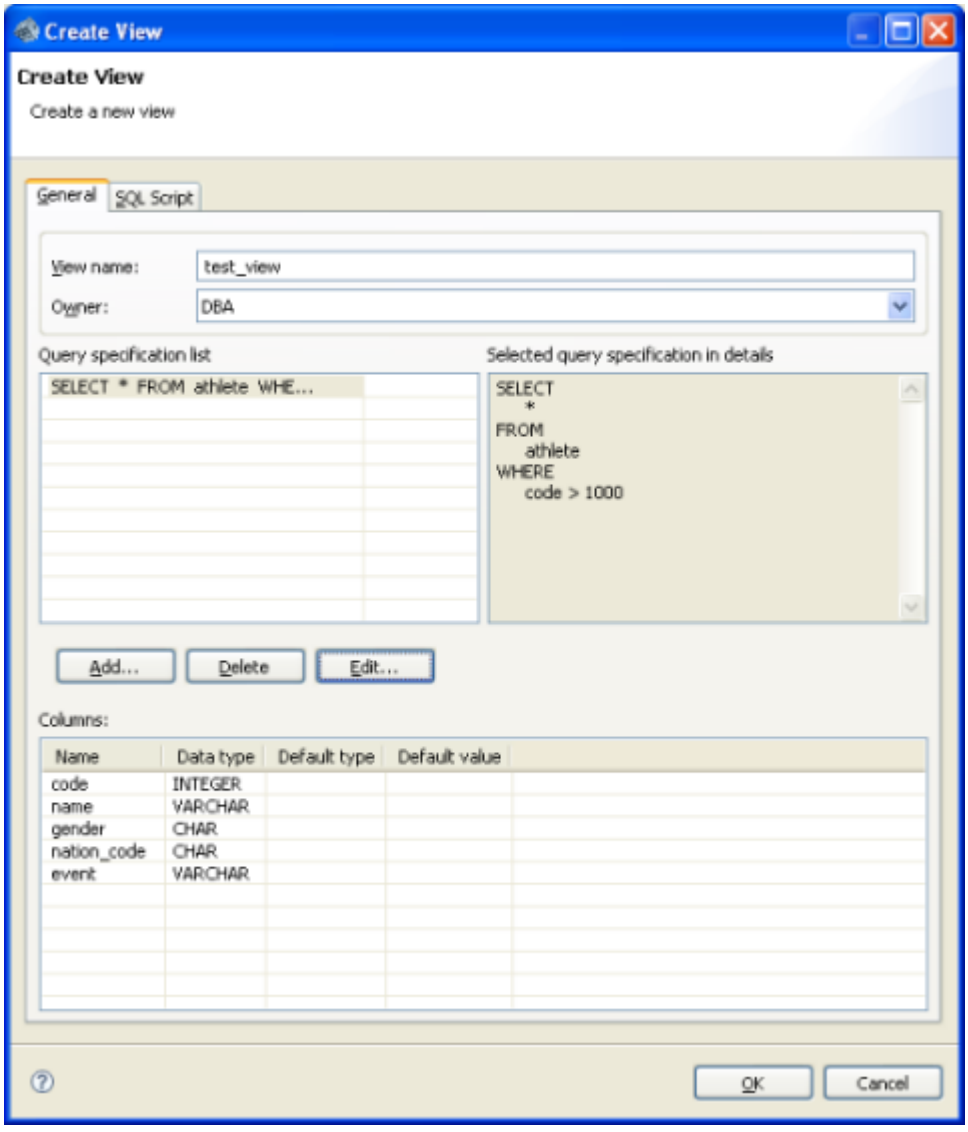


Create View

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.



Select All

Right-click a view in the navigation tree and click [Select All]. Or you can drag and drop the view into the edit or 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.

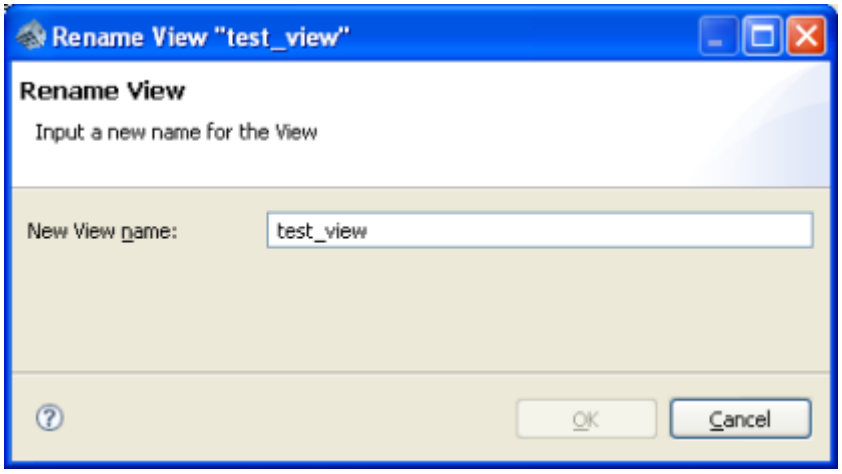
Drop View

Drops the selected view and performs the same functionality as the following syntax.

```
DROP FROM view
```

Rename View

You can change the name of the current view.

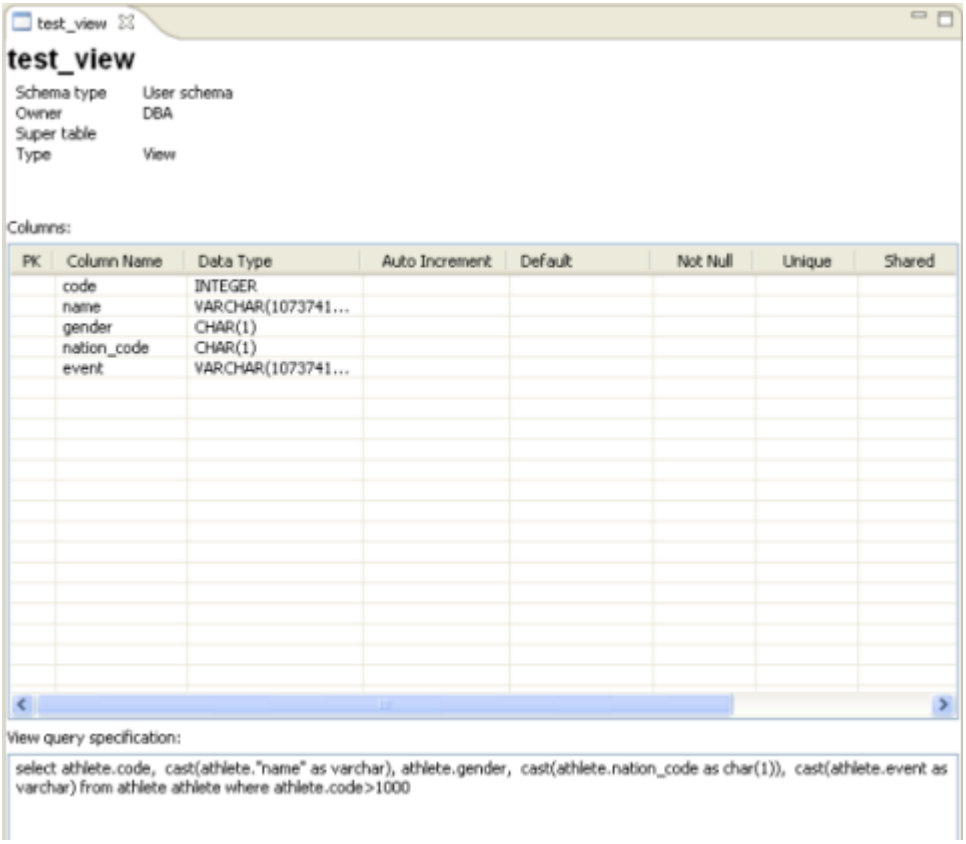


Edit View

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.

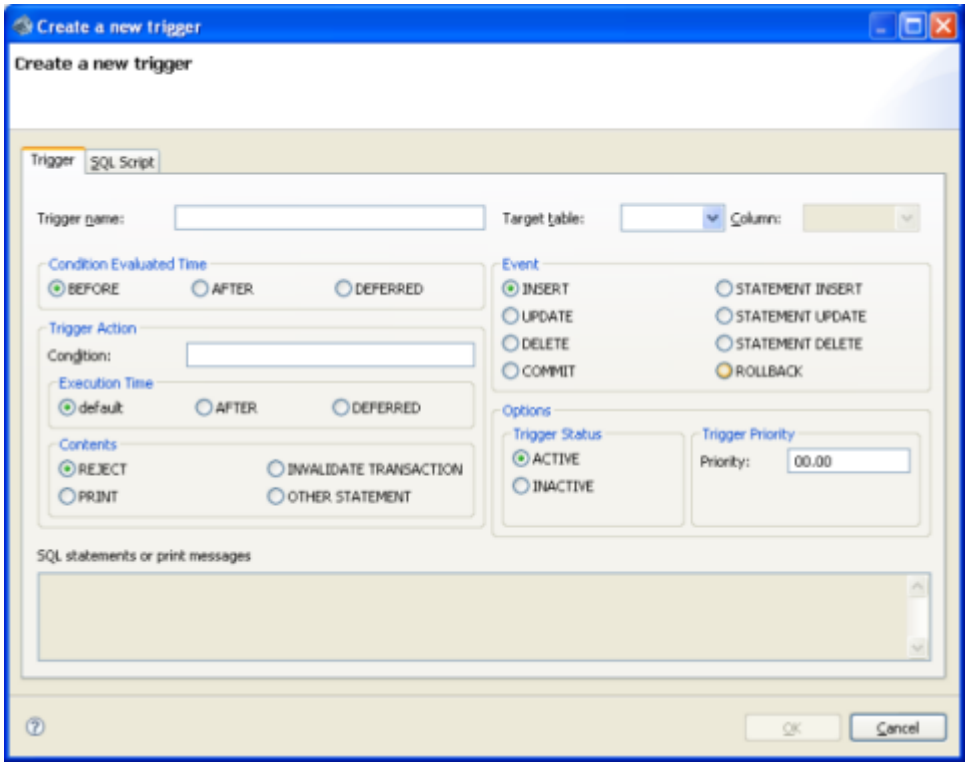


Triggers

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].



- **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.

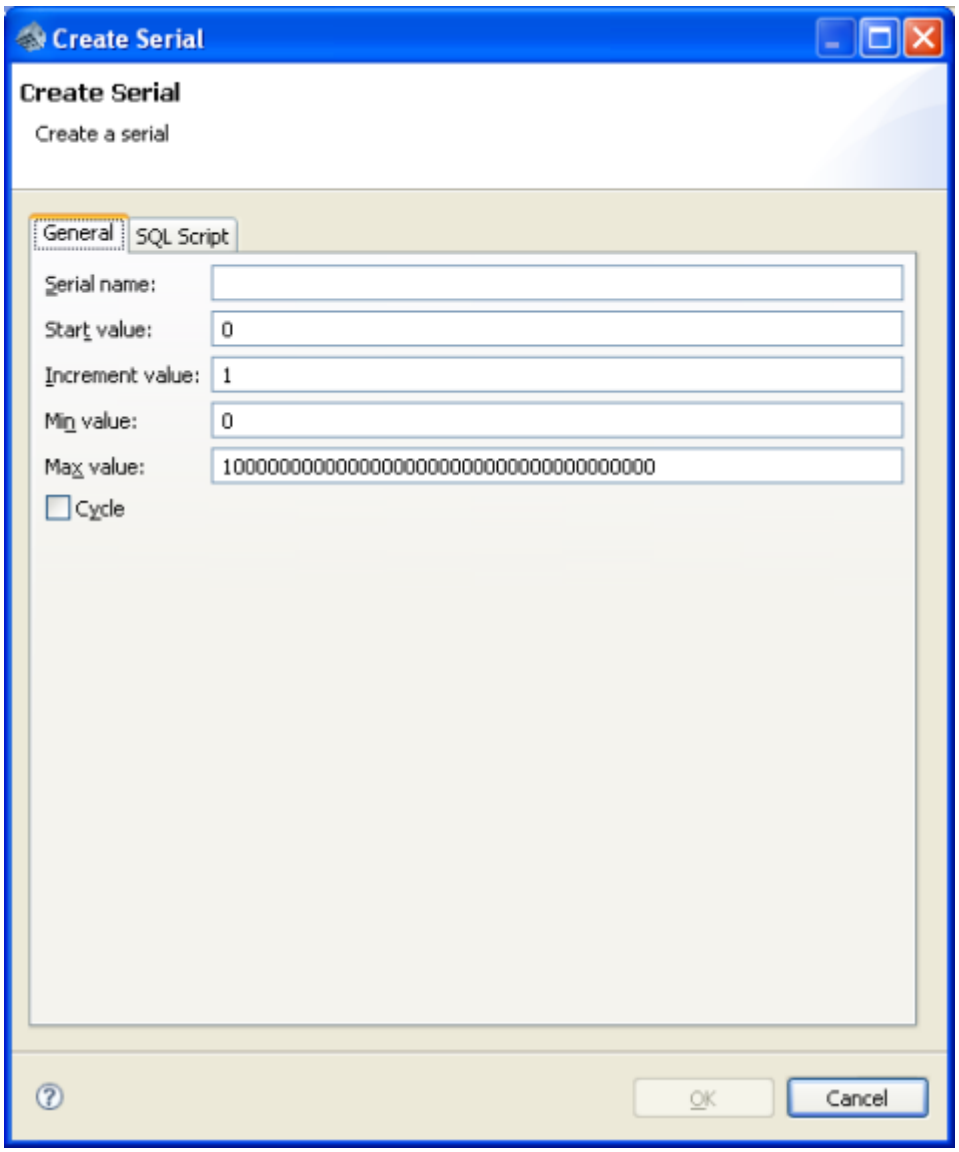
Serials

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.



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 Procedures

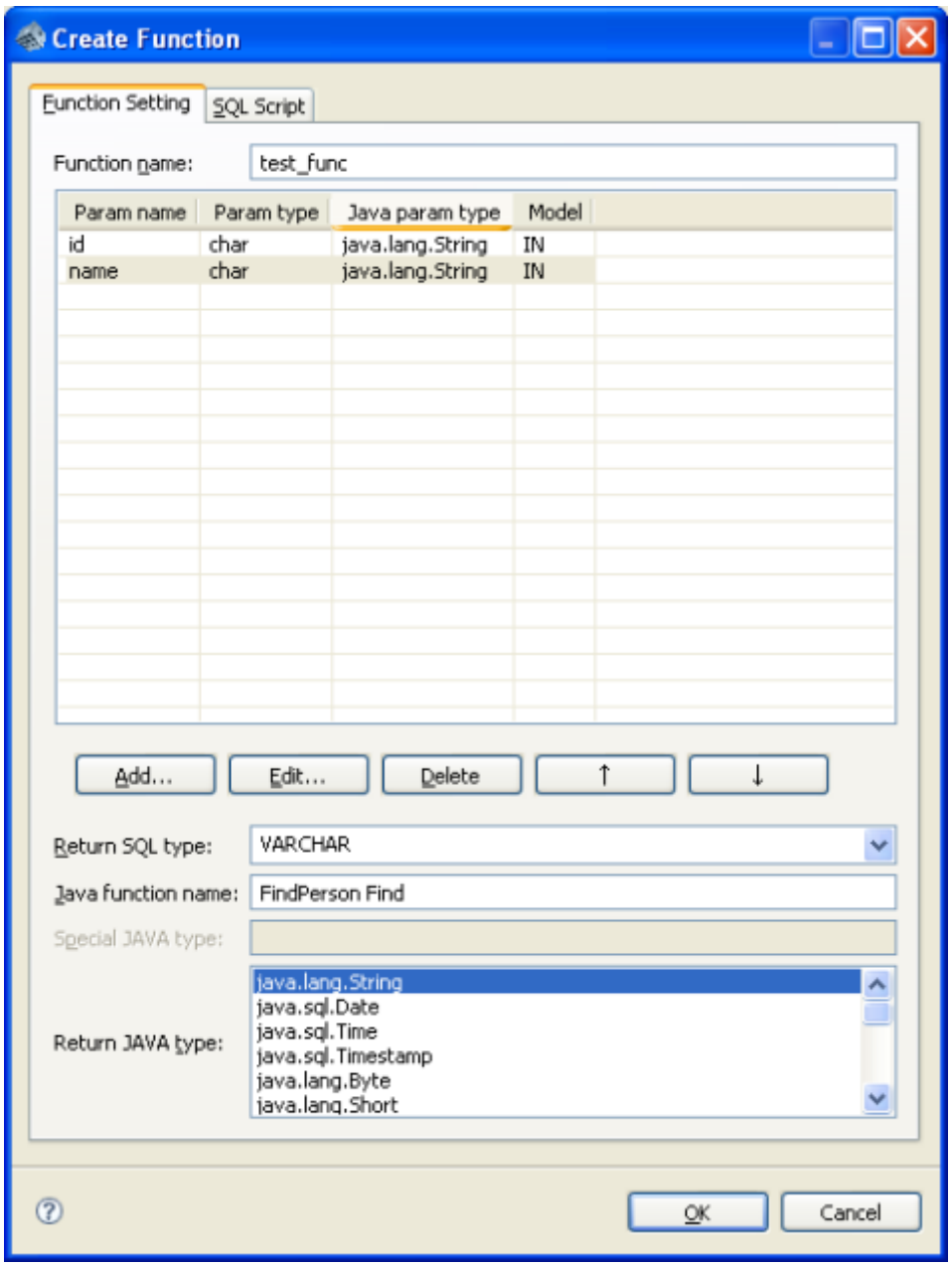
When you login to the database, you can view accessible stored procedures.



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].



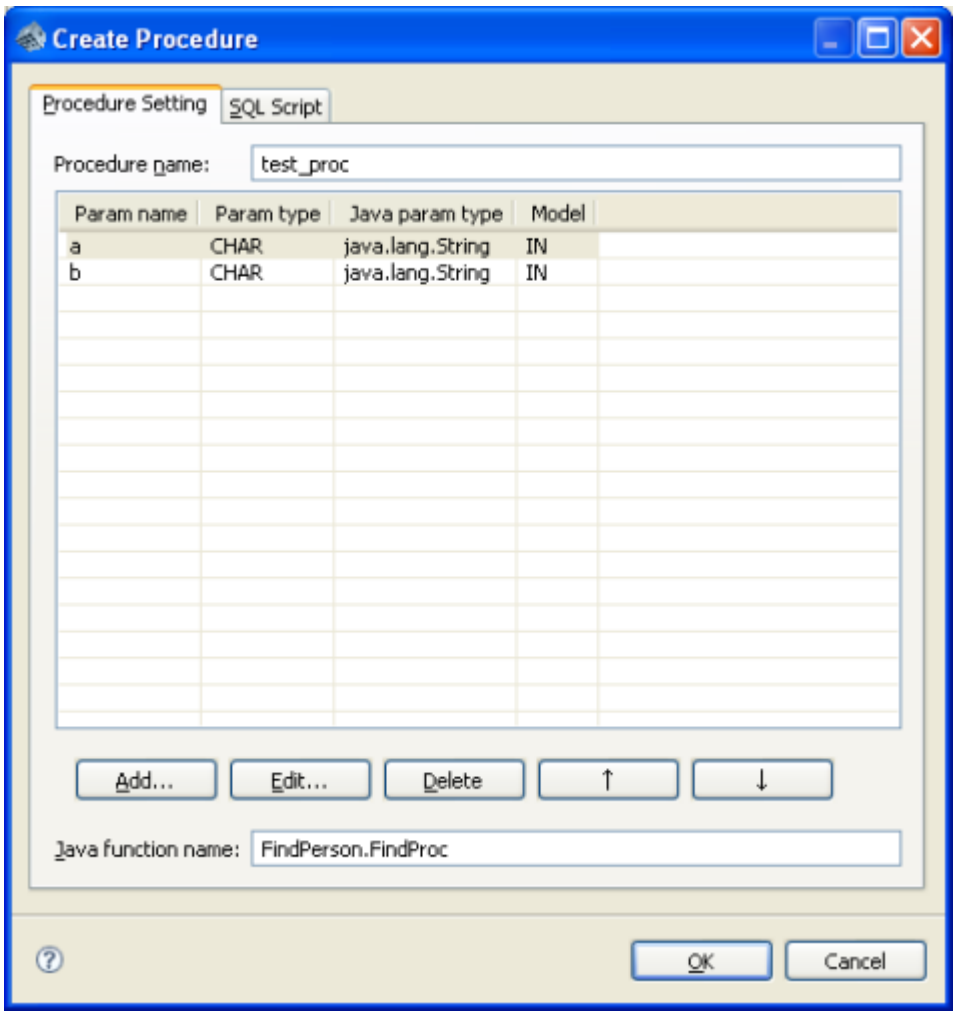
Only Java types compatible with the selected SQL type are displayed so that the user does not make a mistake with type mapping.

For more information, see Java Stored Function/Procedure.

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].



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 Java Stored Function/Procedure.

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

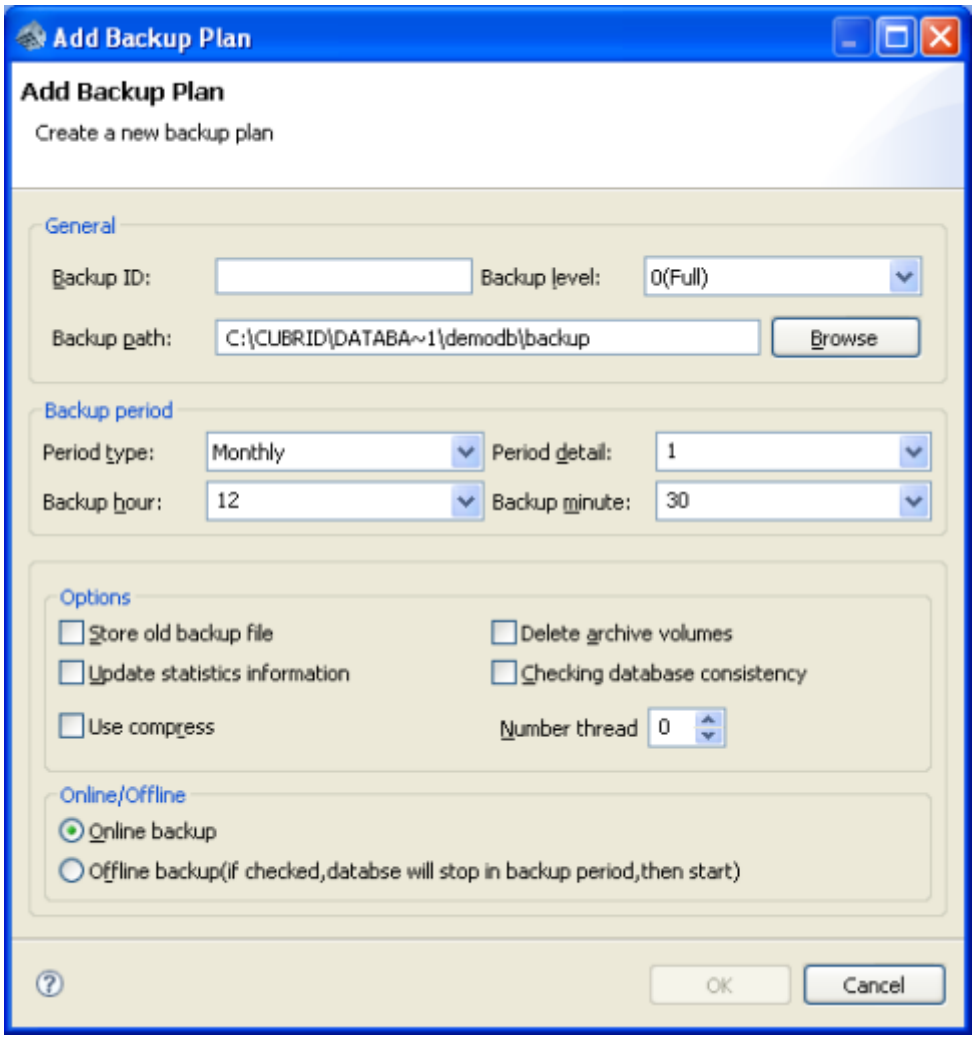
Backup Automation

When you login to the database, you can view [Backup plan] under [Job automation] in the navigation tree.

If you want to execute a backup periodically with the CUBRID Manager, use [Backup plan].



Add Backup Plan

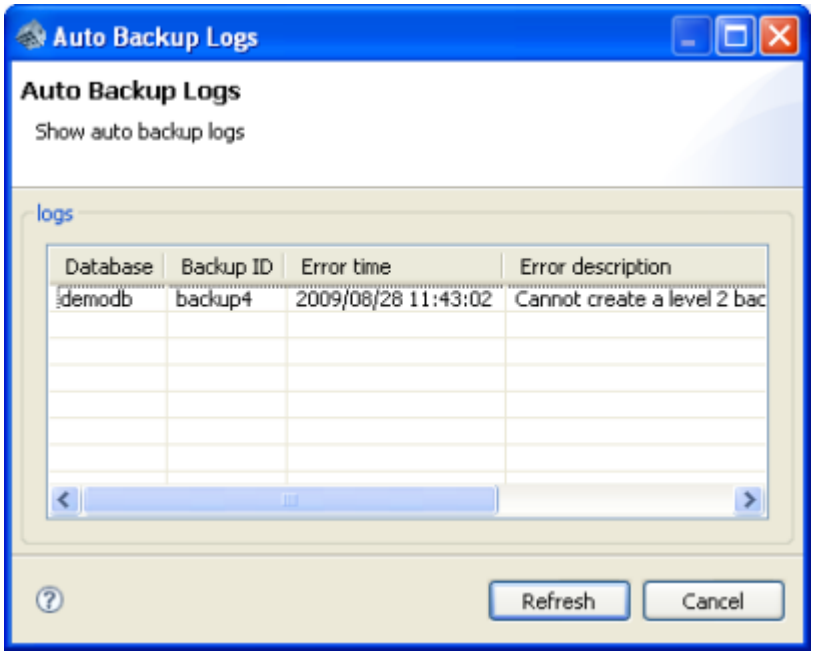


- **Backup ID** : Enter the name of the backup job. The backup ID must be unique in the database because multiple backup plans may exist in the same database.
- **Backup level** : You can choose from 0, 1, and 2. Level0 is a full backup. Level1 is the first incremental backup that backs up changes made 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.
- **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.



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].

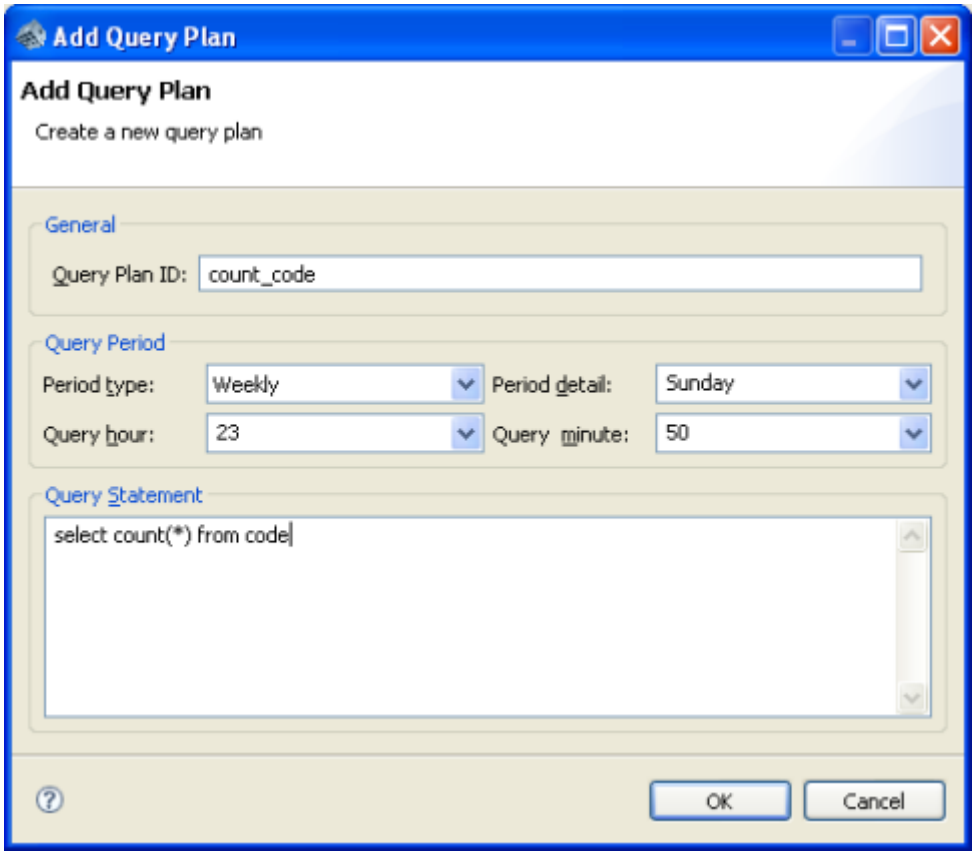
Query Automation

When you login to the database, you can view [Query plan] under [Job automation] in the navigation tree.

If you want to execute a query periodically with the CUBRID Manager, use [Query plan].



Add Query Plan

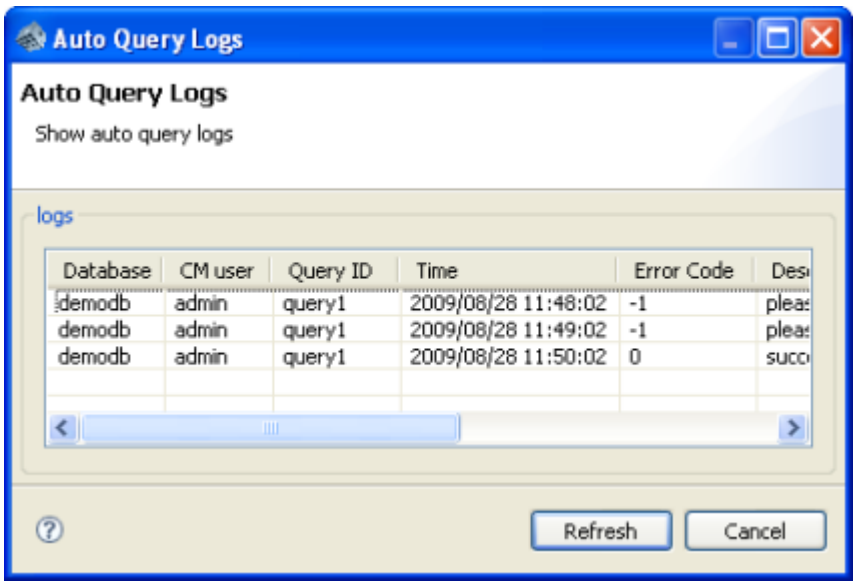


- **Query Plan ID** : Enter the name of the query job. The query plan ID must be unique in the database because multiple query plans may exist in the same database.
- **Period type** : You can select a query automation period from options of Monthly, Weekly, Daily and A specific day.
- **Period detail** : 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.

Because the results are not recorded after query automation, executing the **`SELECT`** statement automatically does not have any significance. However, the query automation function is useful when it is used to update statistics information (**`UPDATE STATISTICS`**) or update instances regularly. The execution results of the automated query are recorded in the **`$CUBRID/log/manager/auto_execquery.log`** file.

Auto Query Log

Provides the logs created during query automation. Logs are recorded regardless of success or failure.



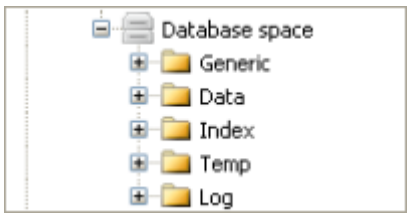
Edit/Delete Query Plan

Edits query automation jobs in the same way for adding query automation, or deletes unnecessary query automation jobs.

Right-click a desired query automation in the navigation tree and then select [Edit Query Plan], or [Delete Query Plan].

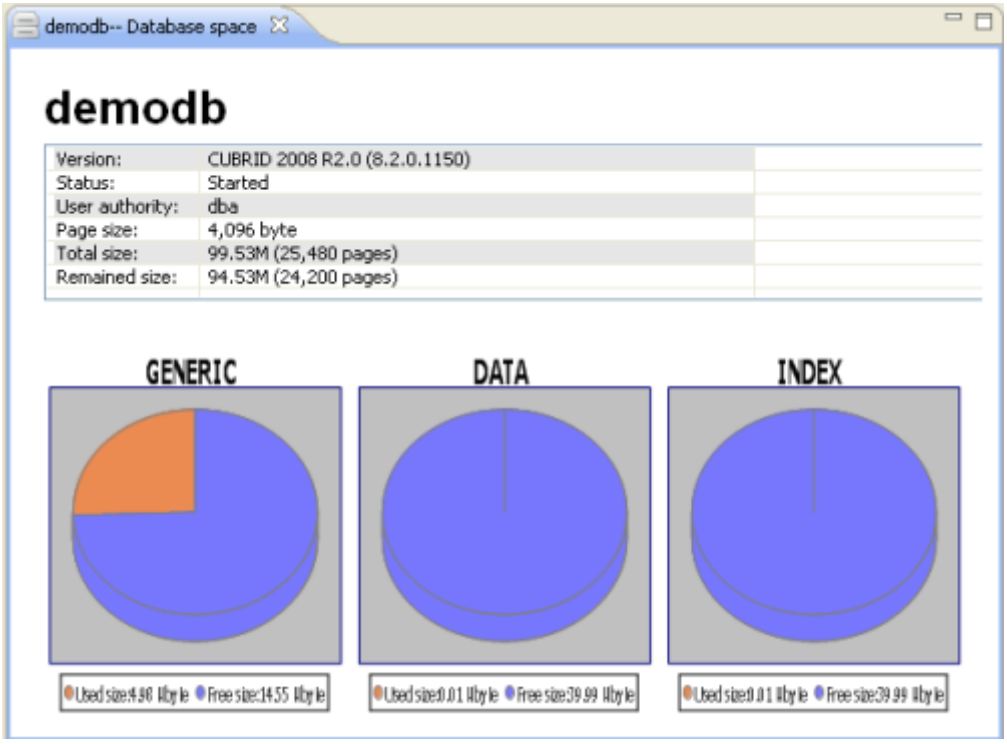
Database Space

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.

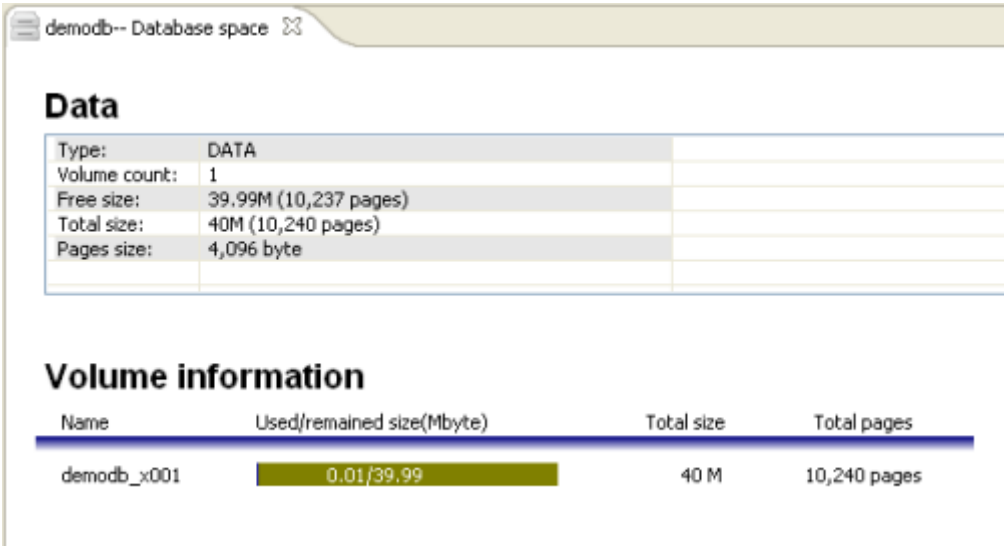


View Database

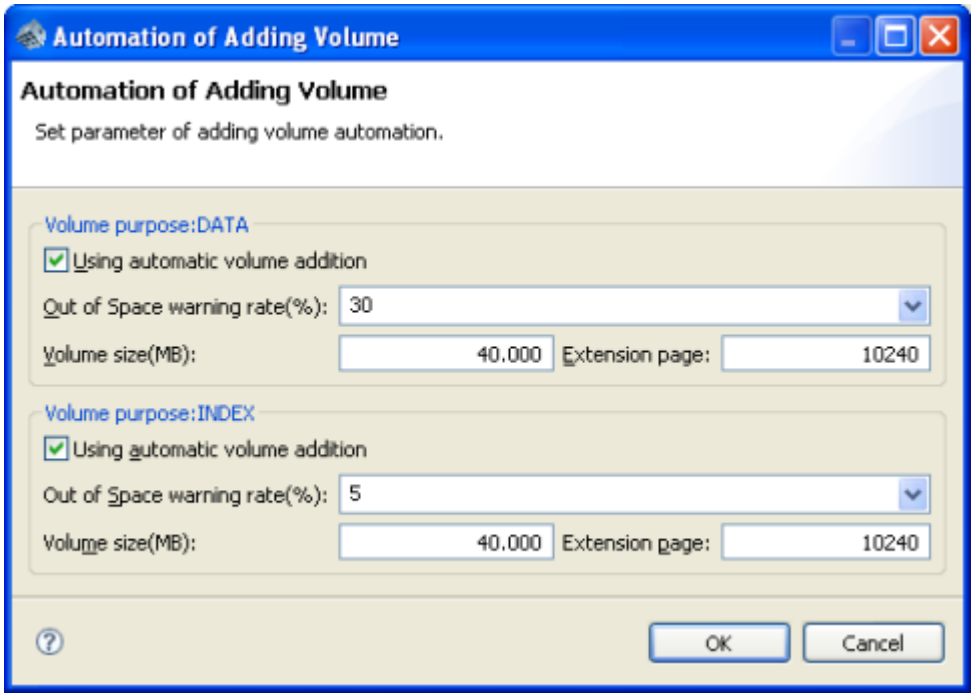
If you double-click the name of a database in the navigation tree or right-click [Database space] then select [View Database], you can check space information of the database.



If you double-click a sub-node of [Database space] in the navigation tree, the volume information appears.

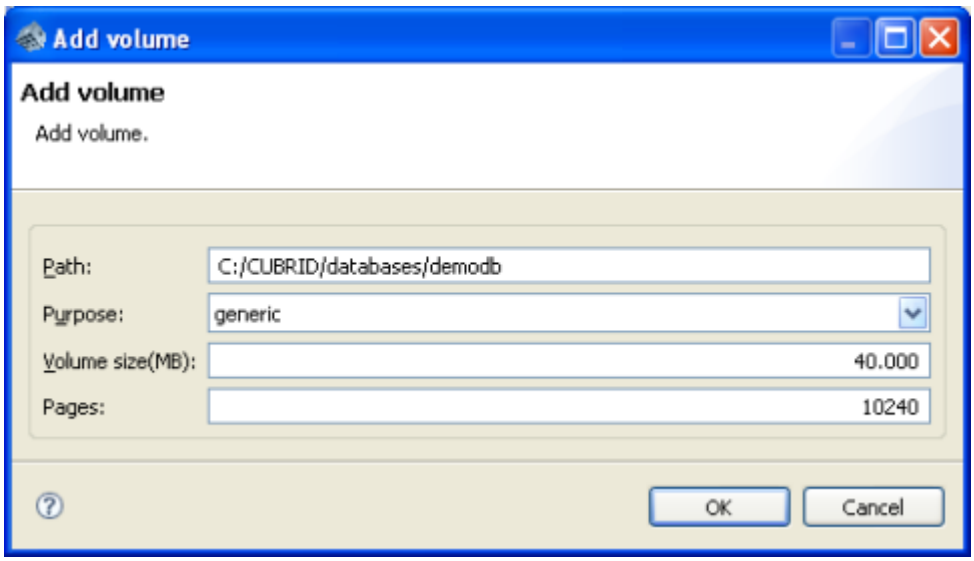


Set Auto Add Volume



- **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



- **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.

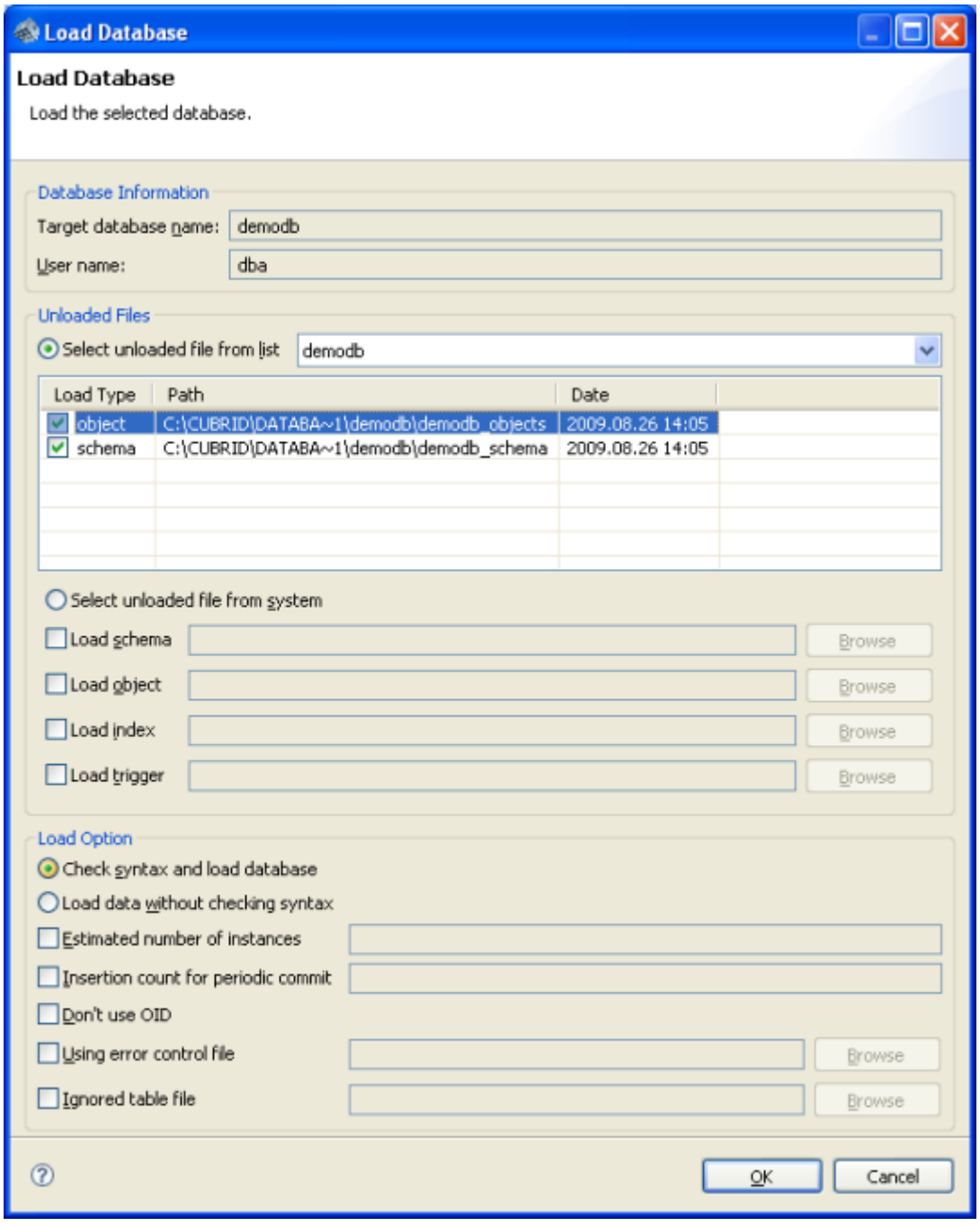
- **Pages** : When the size is entered in Volume size, the number of pages for the volume is automatically calculated and displayed.

Load Database

To load the unloaded data into the currently selected database, perform one of the following after login to the database.

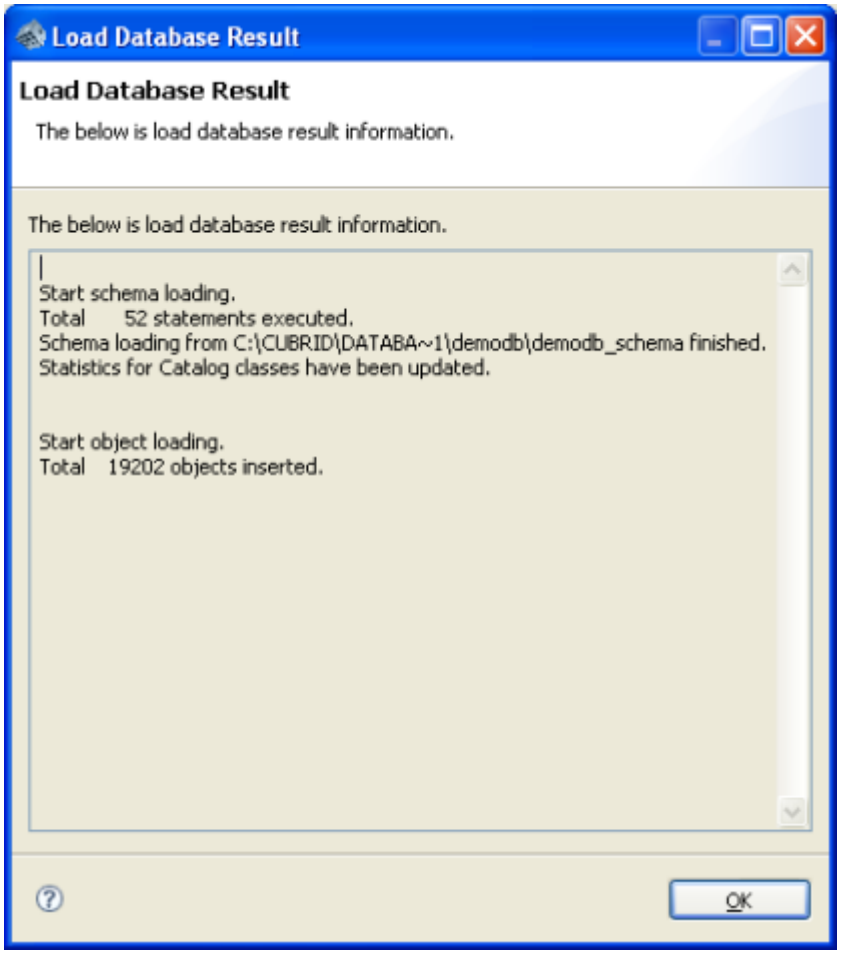
- Click [Load Database] from the toolbar.
- Right-click the database and then select [Load Database].
- Select [Action] > [Load Database] on the menu.

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.



- **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 a Database.

To check the results of the database load operation, set options and then click the [OK] button in the [Database load] dialog box.

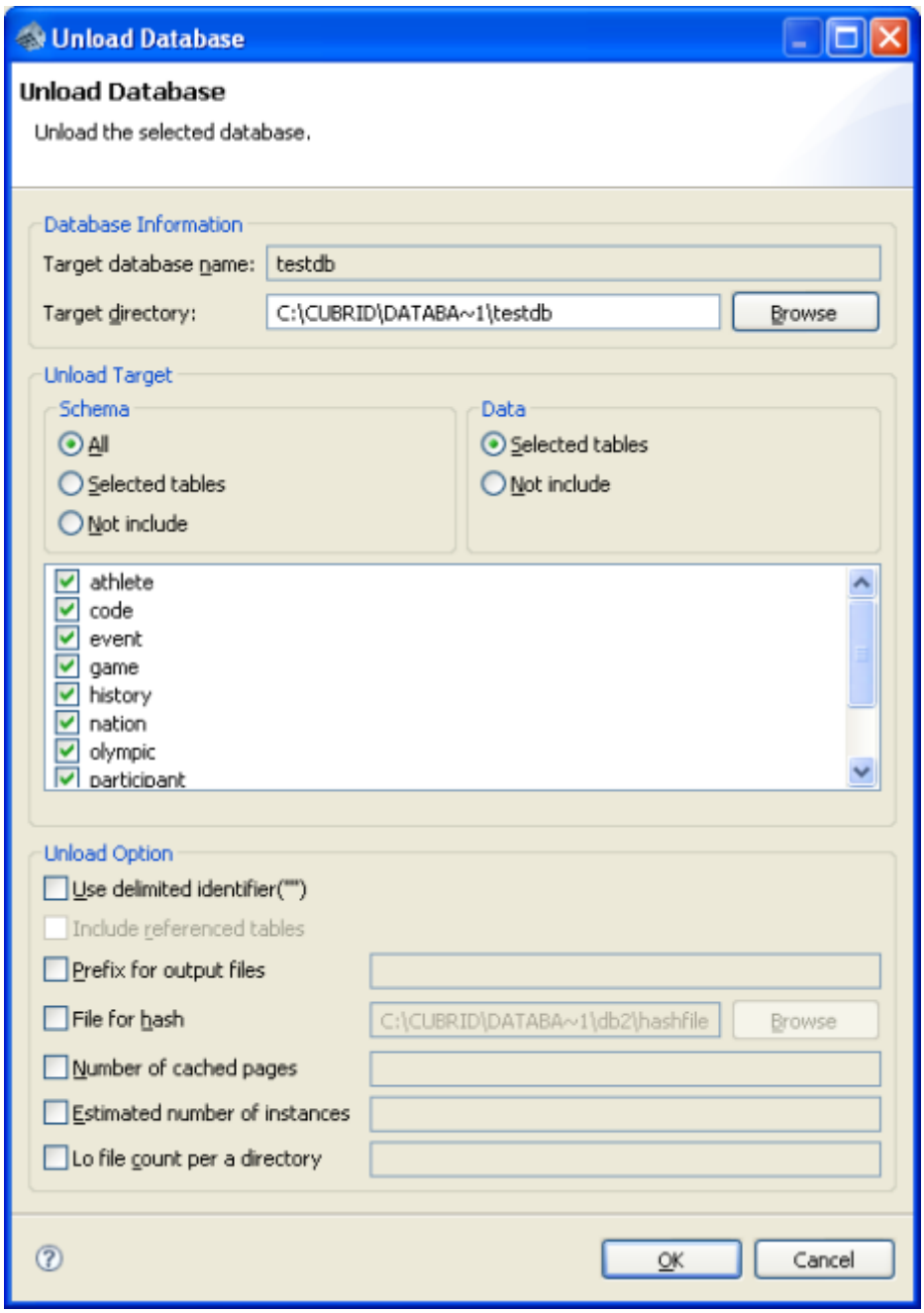


Unload Database

To unload data of the selected database, perform one of the following after login to the database.

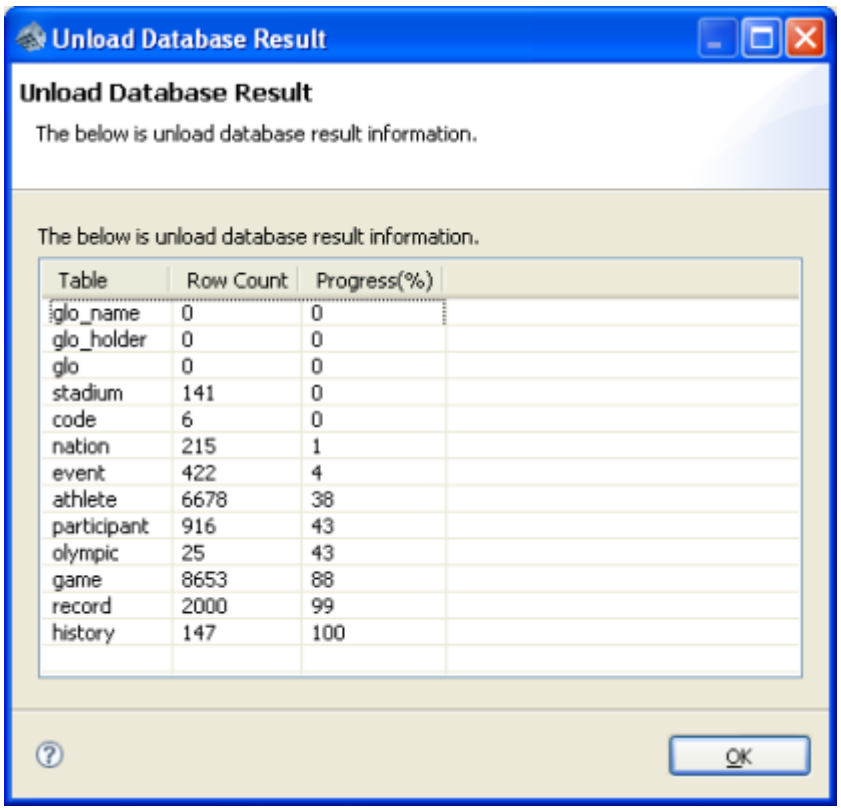
- Click [Unload Database] from the toolbar.
- Right-click the database and then select [Unload Database].
- Select [Action] > [Unload Database] on the menu.

The unload operation can be performed even when the database server is running.



- **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 a Database.

To check the results of the database unload operation, set options and then click the [OK] button in the [Unload Database] dialog box.

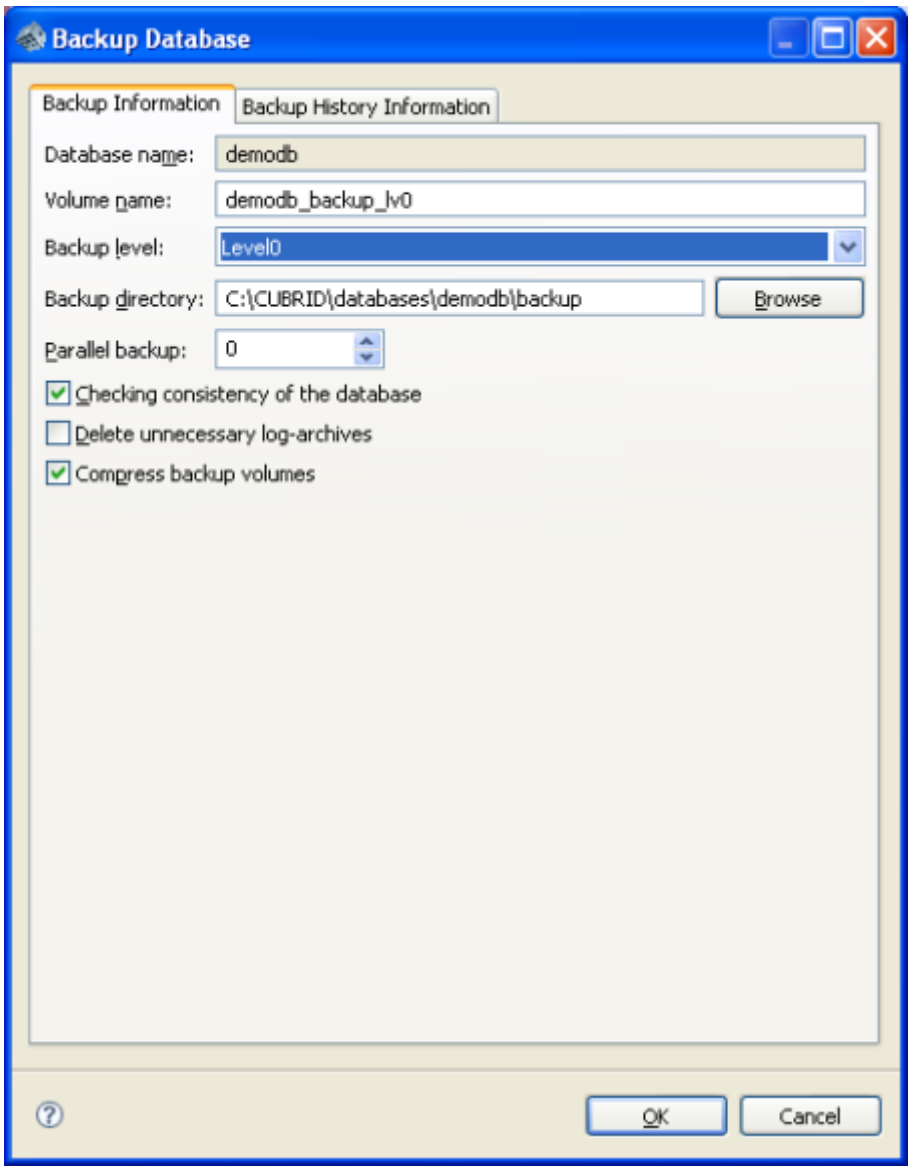


Backup Database

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.

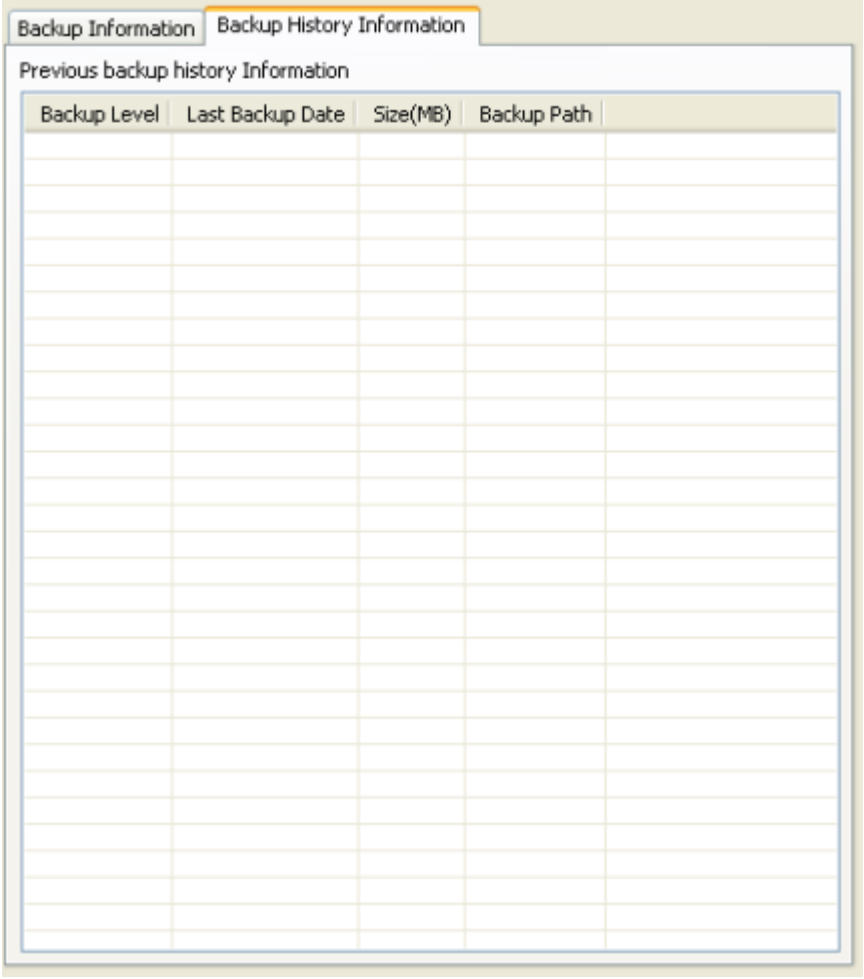
The backup operation can be performed even when the database server is running. For more information on database backup, see Database Backup and Restore.



- **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.



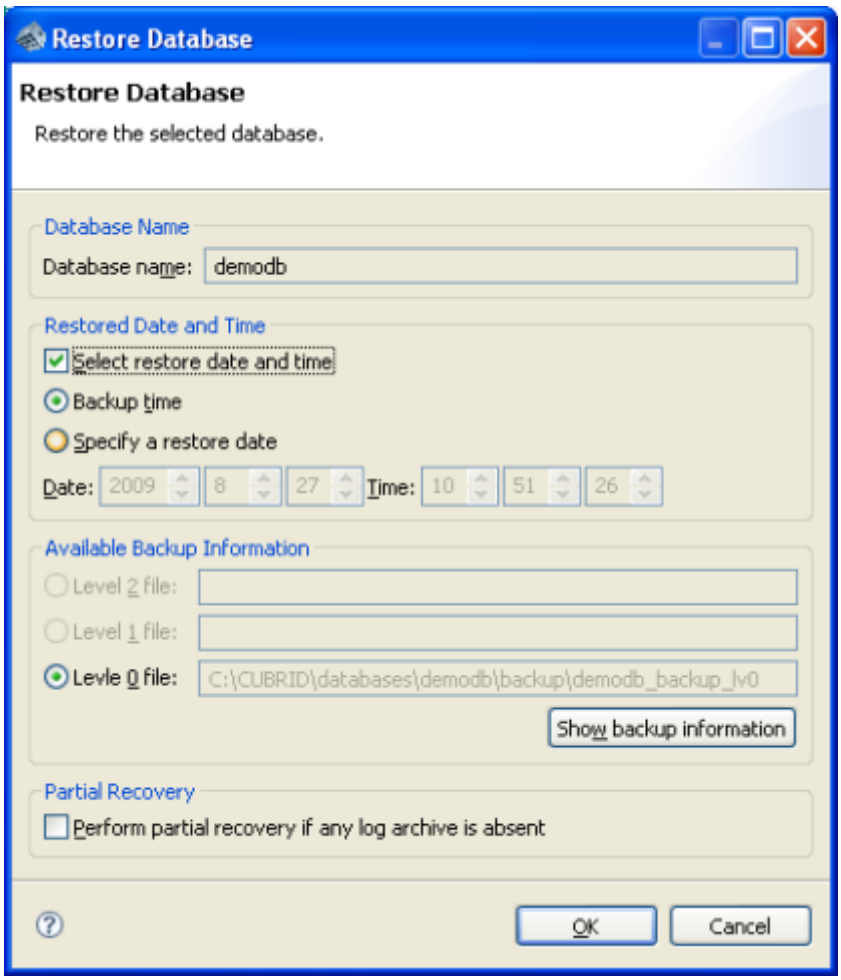
Restore Database

To restore a database, perform one of the following after login to the database.

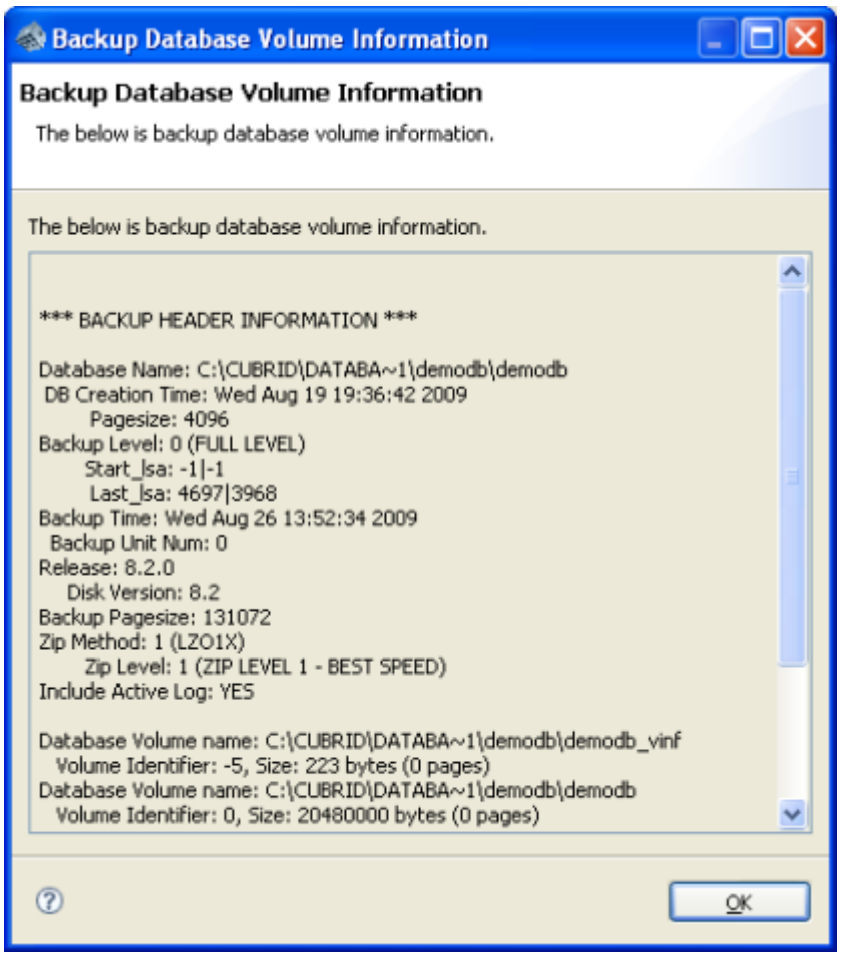
- Click [Restore Database] from the toolbar.
- Right-click the database and then select [Restore Database].
- Select [Action] > [Restore Database] on the menu.

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 and Restore.



- **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.



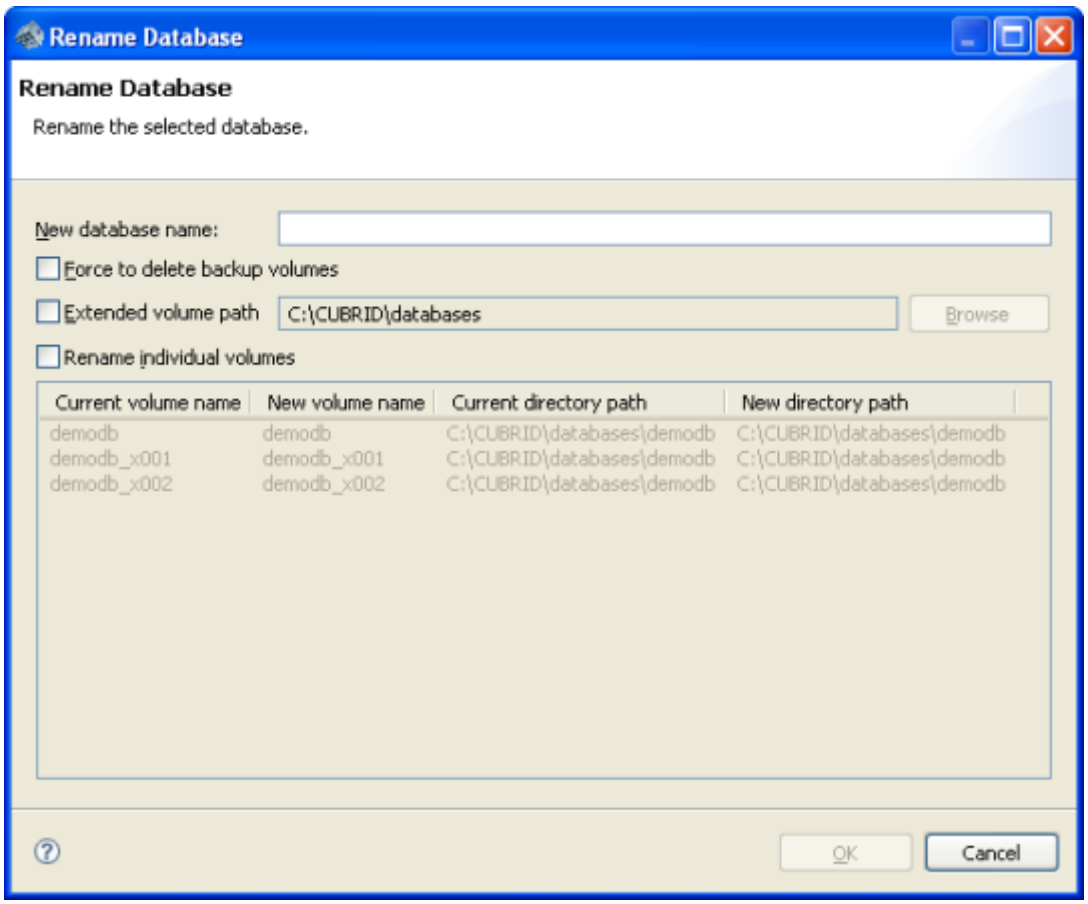
Rename Database

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; The [Rename Database] menu is deactivated while the database is running.

For more information about renaming databases, see Database Backup and Restore.



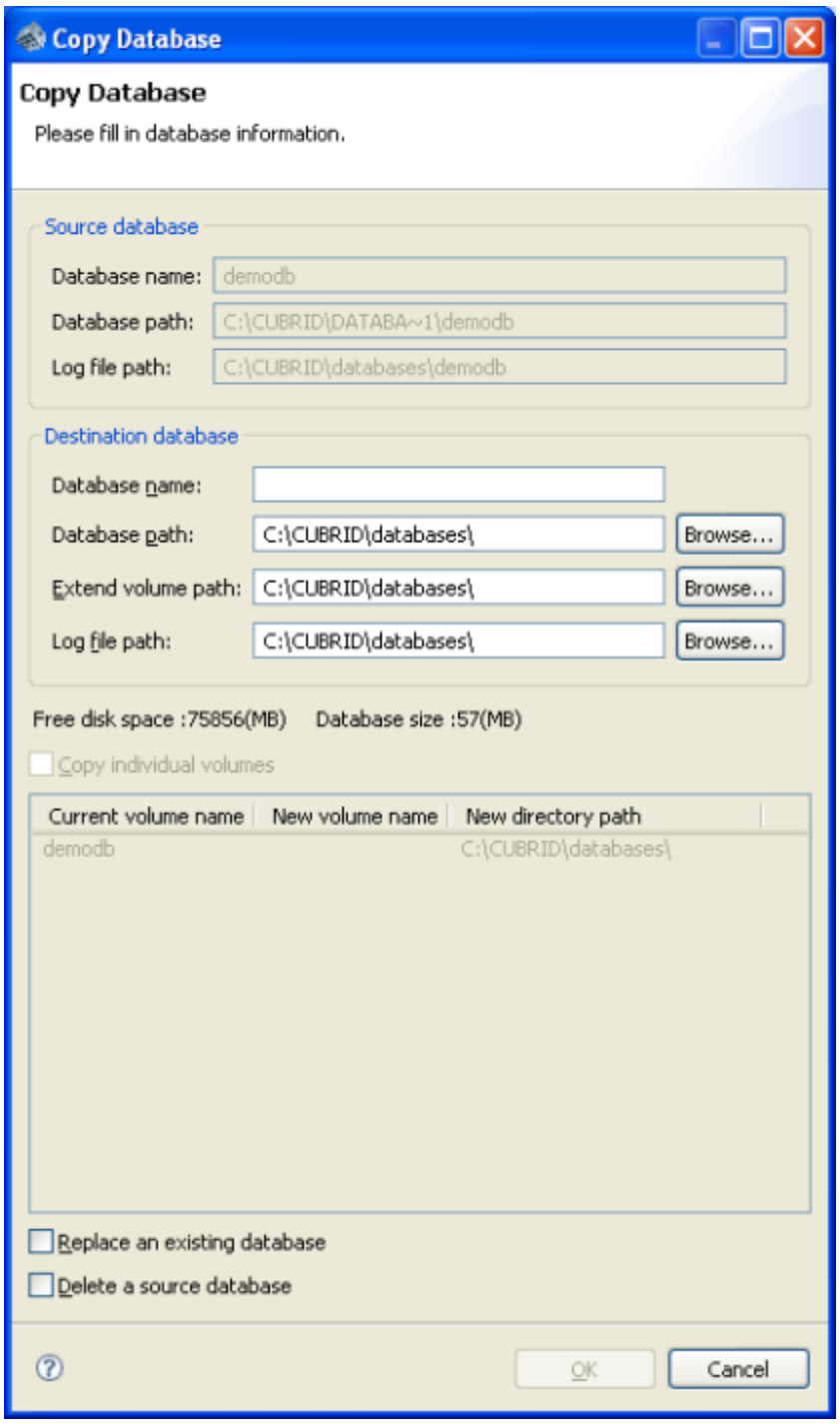
- **New database name** : Enter a new name for the database to be renamed.
- **Force to delete backup volumes** : Delete backup volumes of the database before the rename operation.
- **Extended volume path** : An option that specifies the path in which volumes to be added to the new database are stored.
- **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; The [Copy Database] menu is deactivated while the database is running.



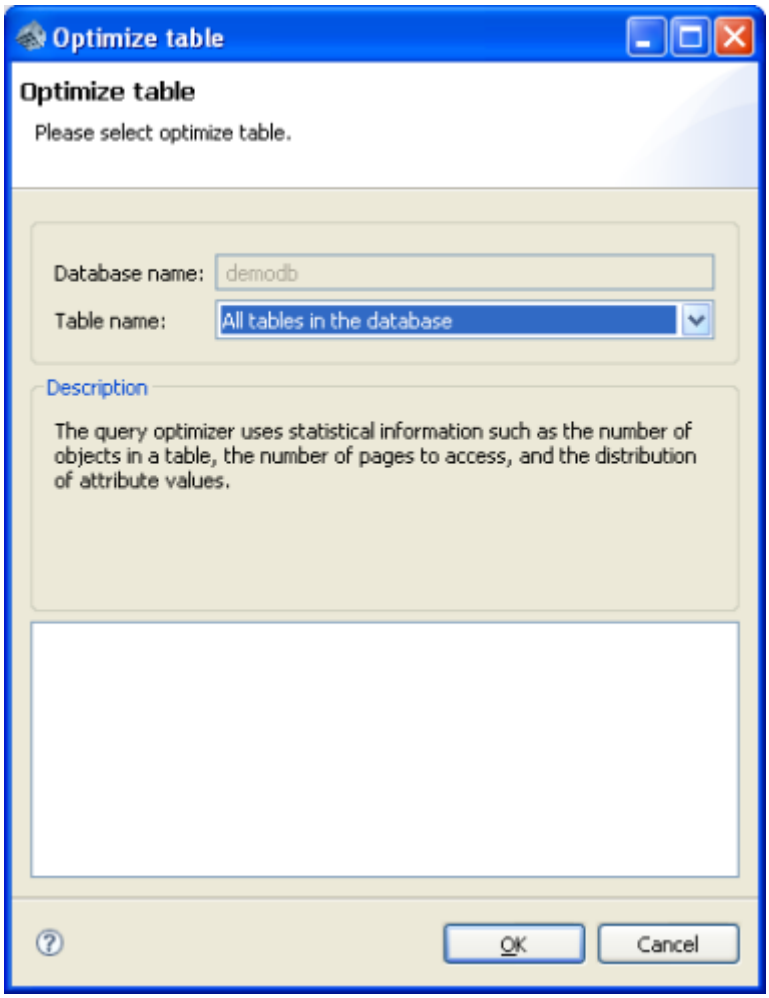
- **Source database** : The name of the database to be copied, the database directory path and the log file directory path are displayed.
- **Destination database** : Enter the name, the save path for database volume and extended volume, and the log files of the database to be created by the copy operation.
- **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.

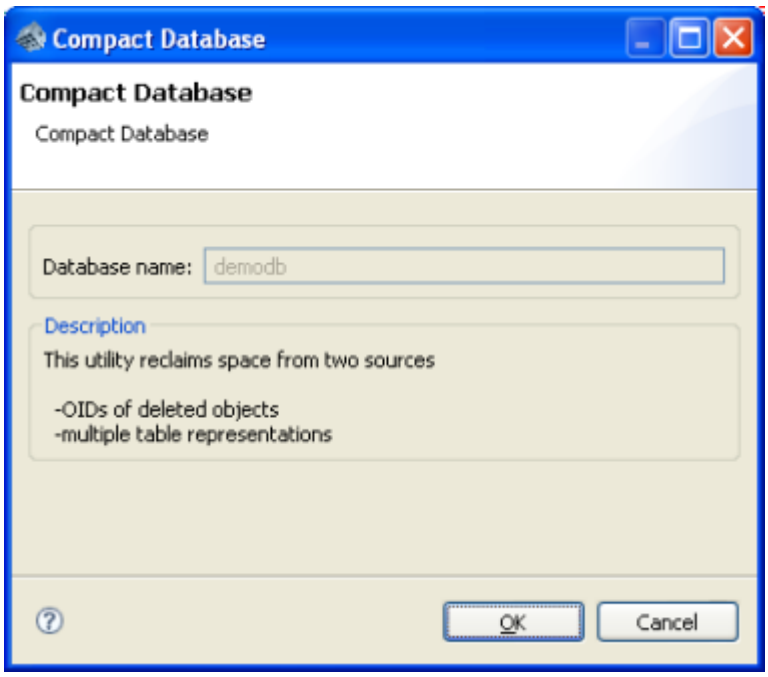


Compact Database

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.

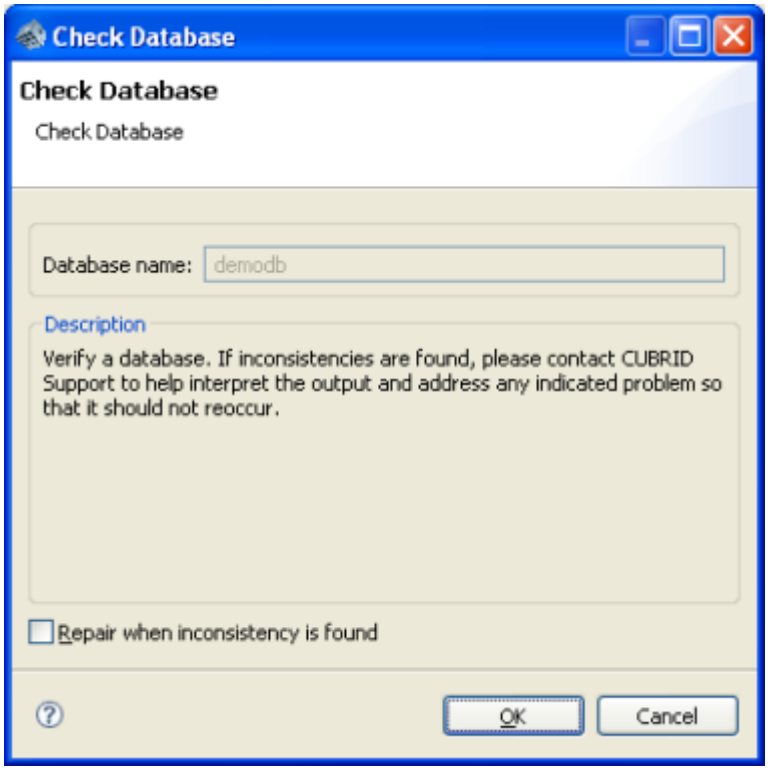
The compact database operation can be performed only when the database server is not running; The [Compact Database] menu is deactivated while the database is running.



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.



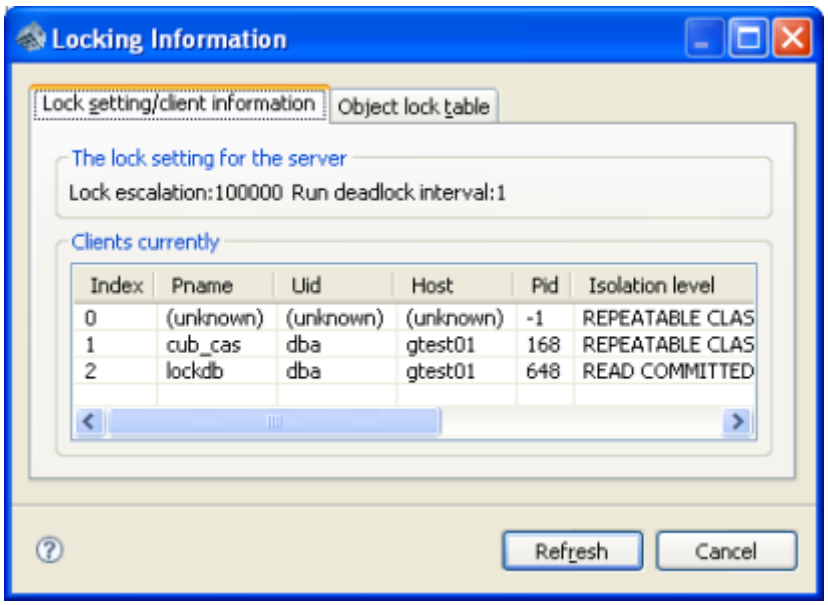
Database Lock Information

To view the lock information of the database, perform one of the following after login to the database.

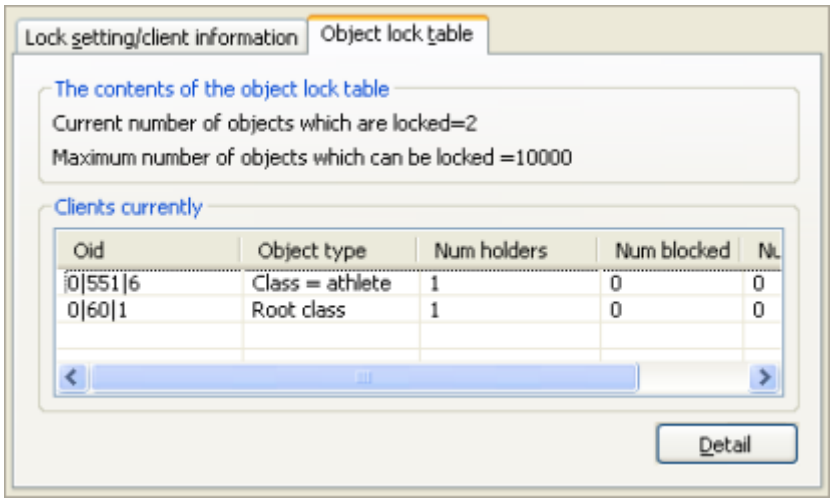
- Click [Lock Information] from the toolbar.
- Right-click the database and then select [Lock Information].
- Select [Action] > [Lock Information] on the menu.

The [Lock Information] dialog box consists of two tabs.

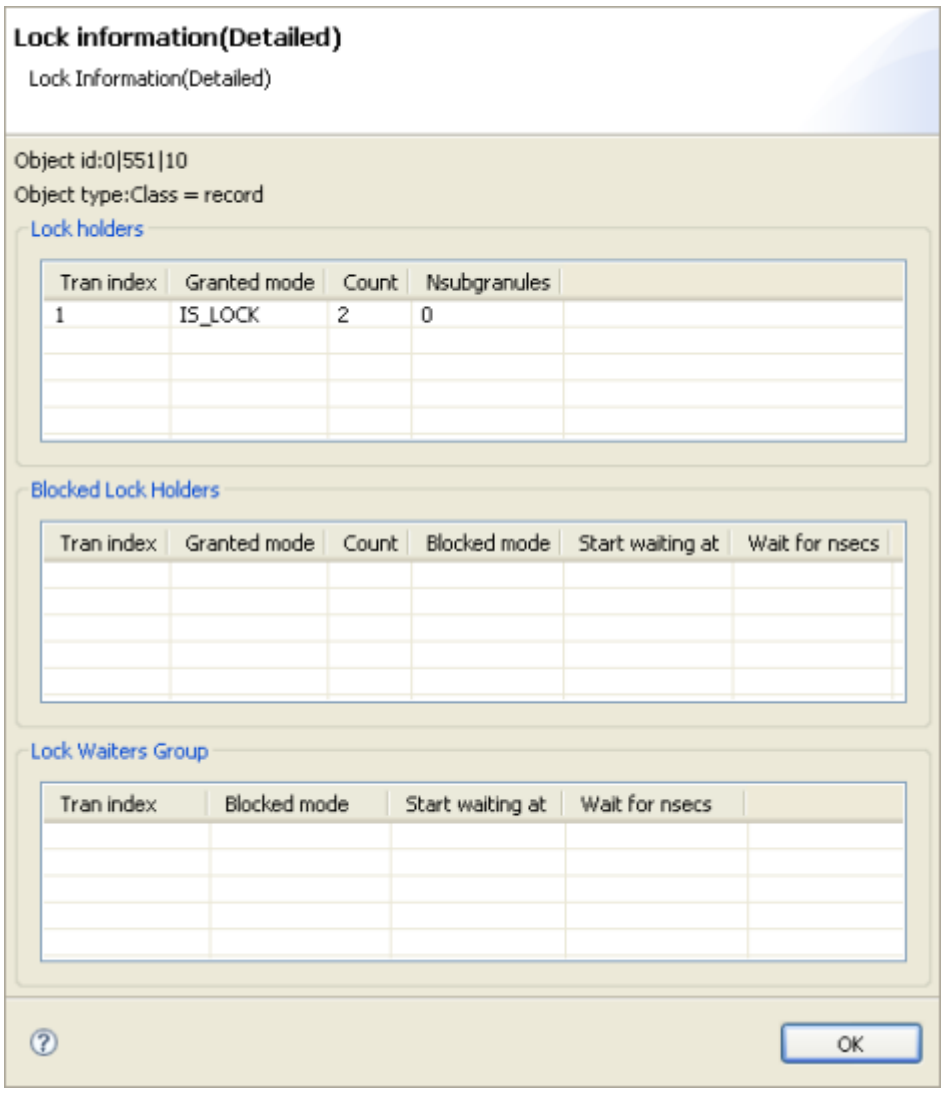
The [Lock setting/client information] tab provides information of the clients currently connected to the database.



In the [Object lock table] tab, you can check the lock information of the database objects.



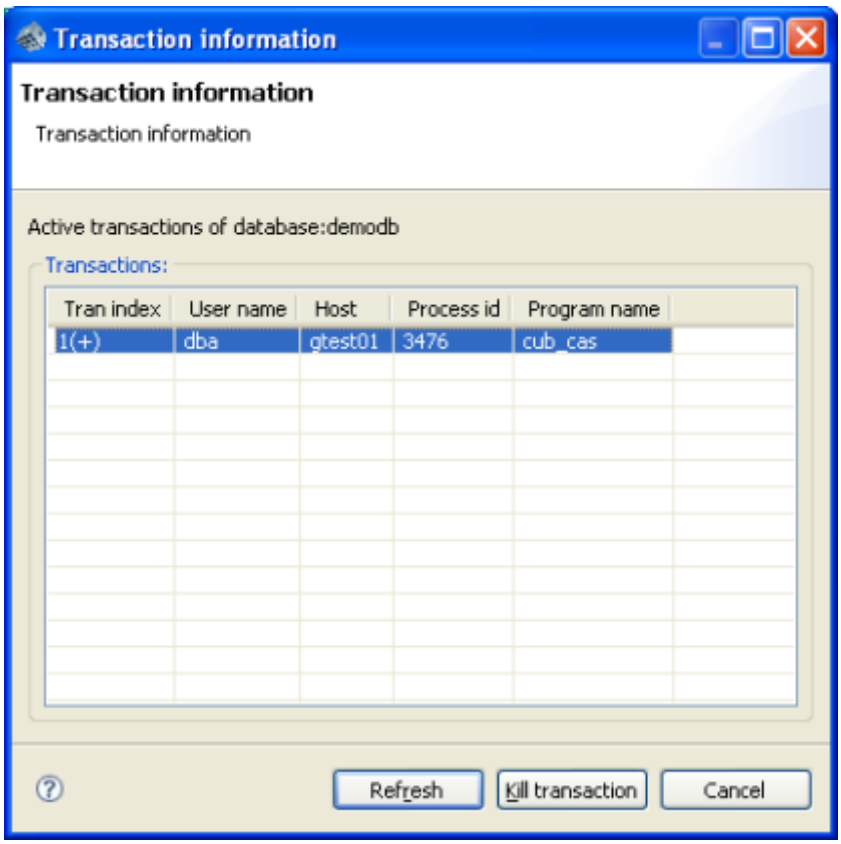
You can view detailed lock information by clicking [Detail] in the [Object lock table] tab.



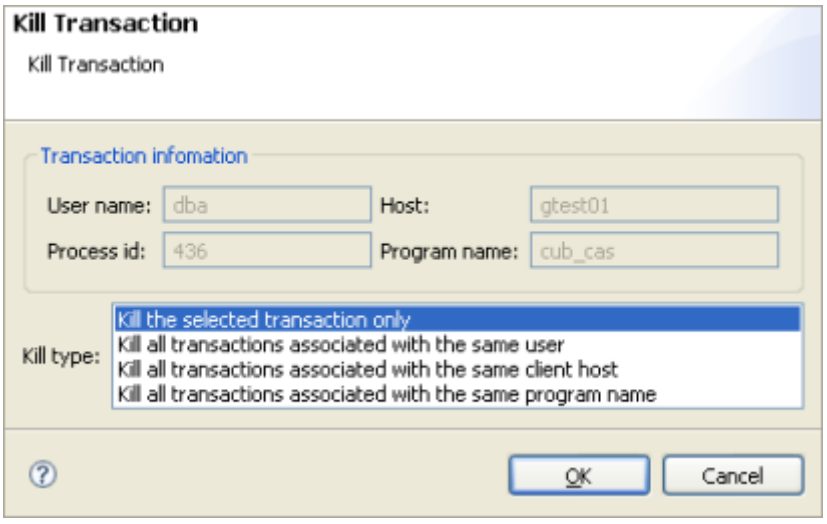
Database Transaction Information

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.



- **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.

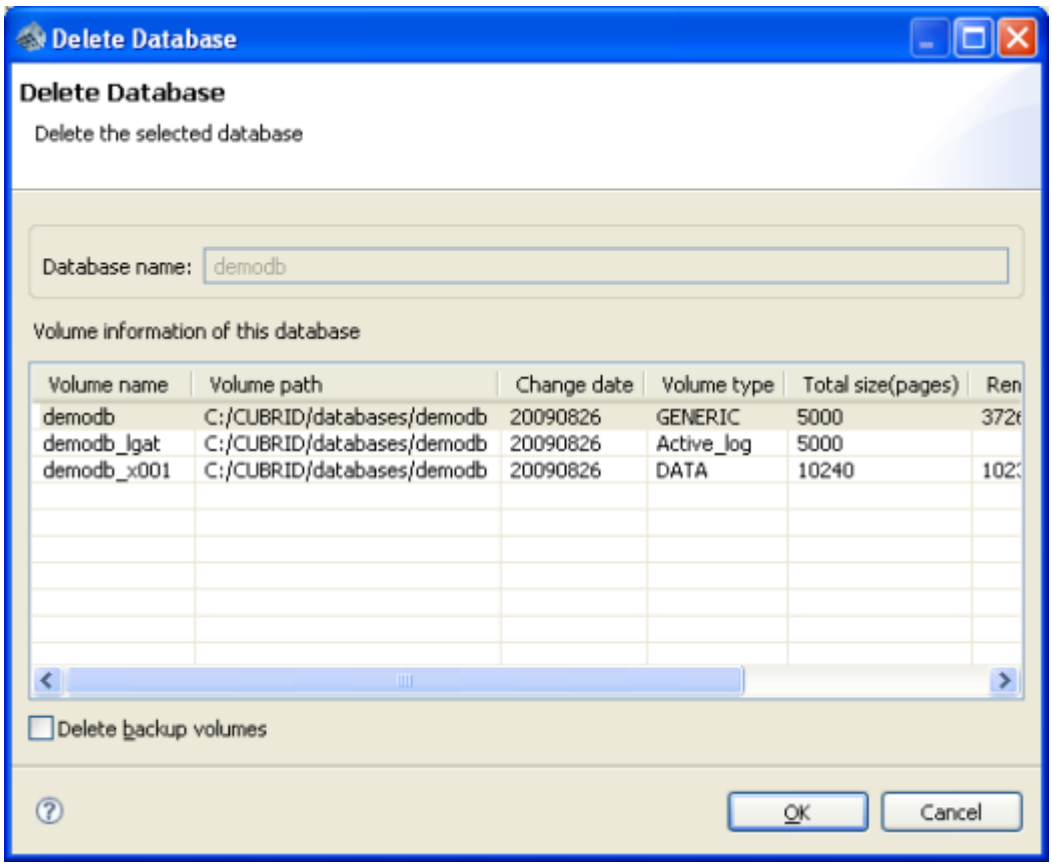


Delete Database

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.

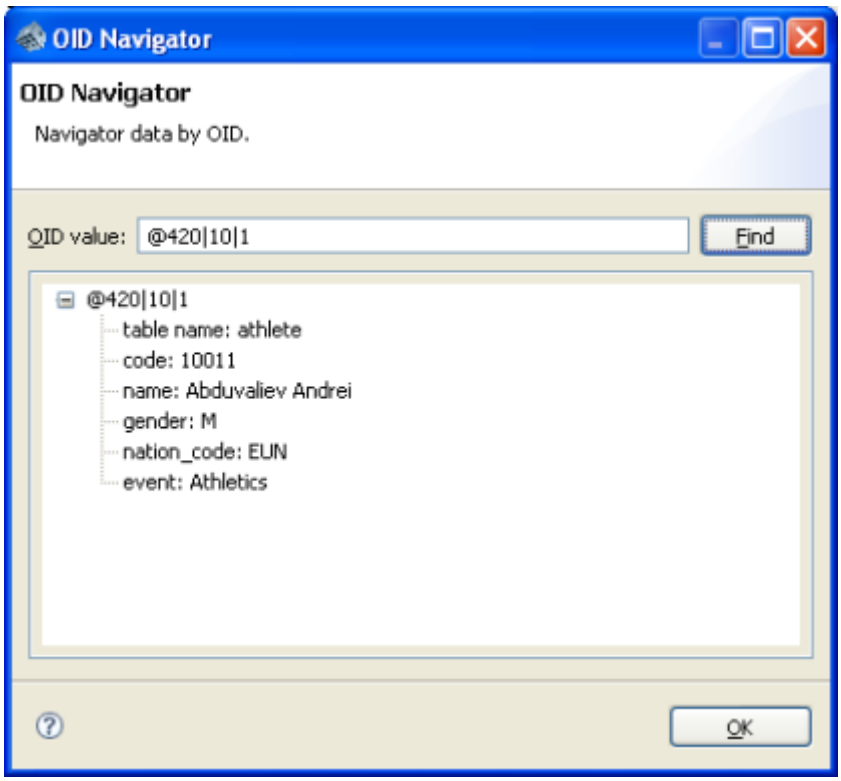


If you click [OK], the database is deleted after checking the **dba** password. Only the **dba** user can delete a database.

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].

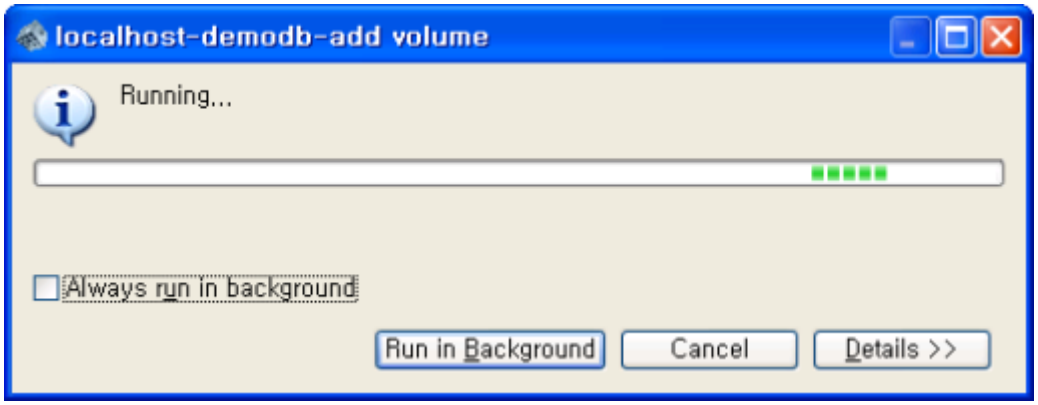


Configuring Background Operations in a Multiple-Host Environment

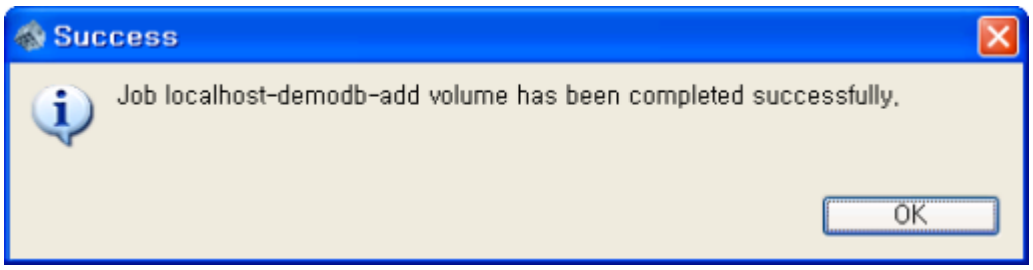
If you manage databases in several hosts by using [Add Host], databases can be configured for background operations. For example, if you want to execute a query to a database in host B while doing backup for a database in host A, the database backup operation, which generally takes a long time, can be 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, Comapct 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.



When a background operation is complete, the dialog shown below appears.



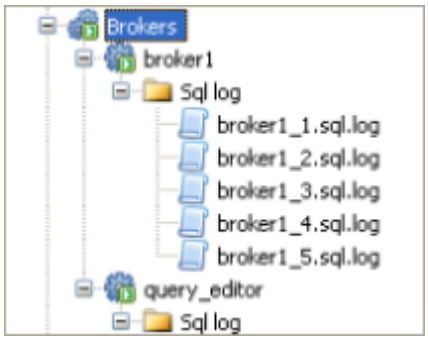
If an administrator stops the database or logs out from the database or host while a database operation is being executed as a background operation, a dialog indicating that a background operation is in progress appears.

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.

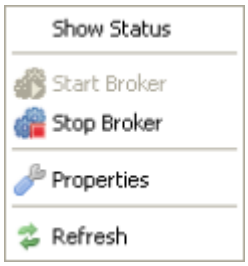
A Broker consists of the names of individually configured Brokers and their SQL logs.



Broker Menu

To execute the Broker functions, right-click [Broker] in the navigation tree. The menu consists of [Show Status], [Start Broker], [Stop Broker], [Properties] and [Refresh].

If the Broker is running, it is displayed as icon; if it has stopped, it is displayed as icon in the navigation tree.

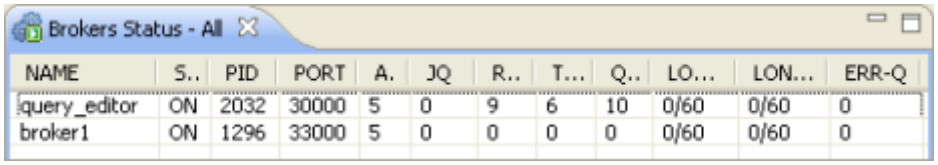


Show Status

[Show Status] executes a function like the following command:

```
cubrid broker status
```

To automatically refresh information, select the [On] check box in the [Refresh Interval Setting] tab in [Properties] and then set a desired interval.



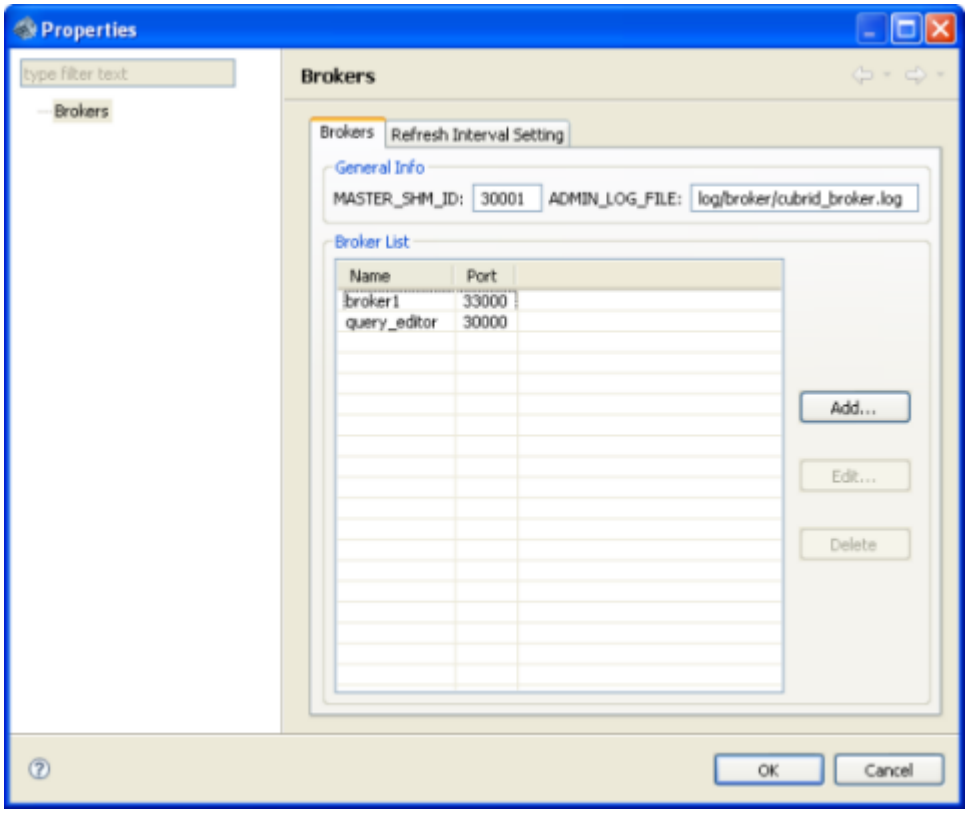
NAME	S..	PID	PORT	A.	JQ	R..	T...	Q..	LO...	LON...	ERR-Q
query_editor	ON	2032	30000	5	0	9	6	10	0/60	0/60	0
broker1	ON	1296	33000	5	0	0	0	0	0/60	0/60	0

Start/Stop Broker

This function starts or stops the entire Brokers based on the broker configuration information in the **`$CUBRID/conf/cubrid_broker.conf`** file.

Properties

To edit the properties of the Broker, right-click a host in the navigation tree and select [Properties], then click [Brokers]. Alternatively, right-click [Broker] and select [Properties].

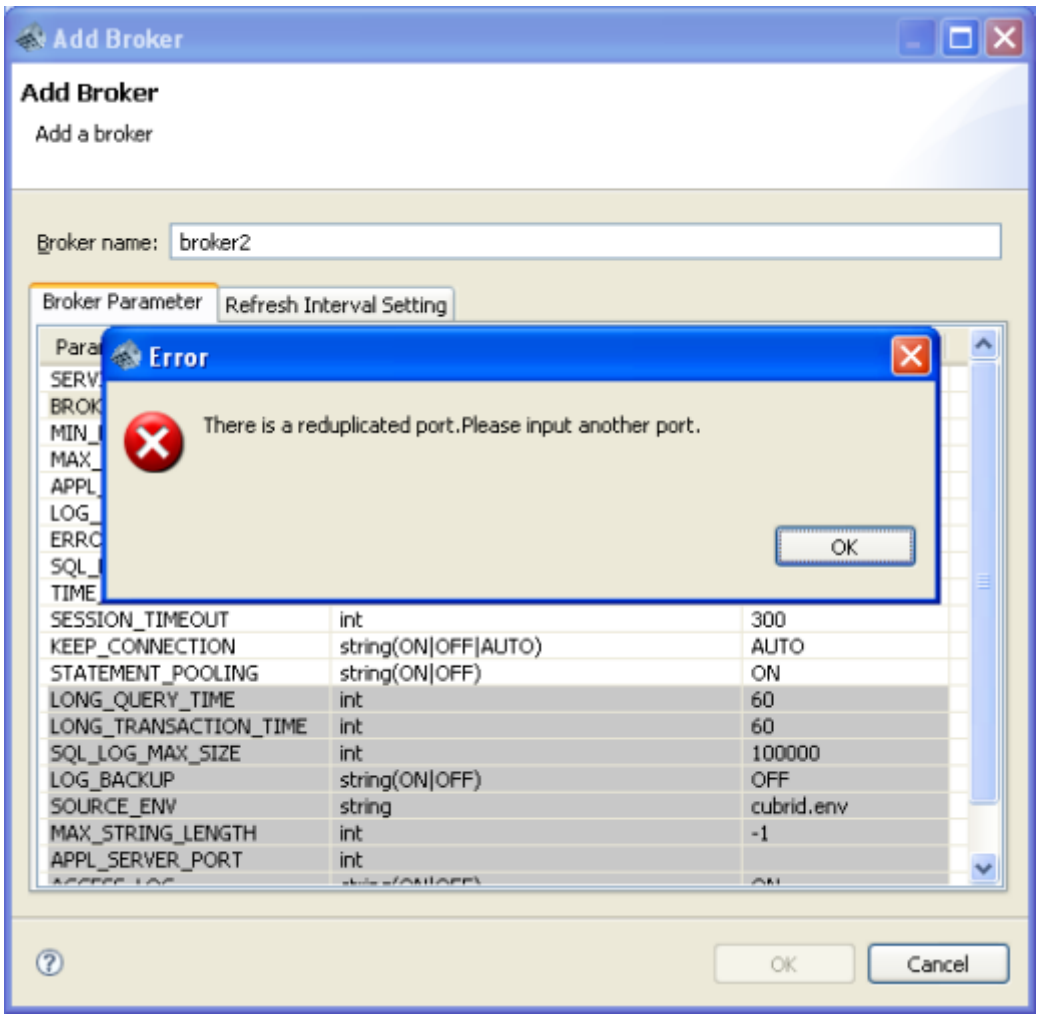


All the configuration information in the **`$CUBRID/conf/cubrid_broker.conf`** file can be added, edited, and deleted with GUI. For more information about each parameter, see Broker Configuration.

Broker Parameter | Refresh Interval Setting

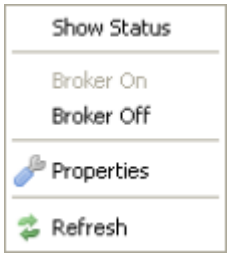
Parameter	Value type	Parameter value
SERVICE	string(ON|OFF)	ON
BROKER_PORT	int(1024~65535)	33000
MIN_NUM_APPL_SERVER	int	5
MAX_NUM_APPL_SERVER	int	40
APPL_SERVER_SHM_ID	int(1024~65535)	33000
LOG_DIR	string	log/broker/sql_log
ERROR_LOG_DIR	string	log/broker/error_log
SQL_LOG	string(ON|OFF|ERROR|NOTICE|TIMEOUT)	ON
TIME_TO_KILL	int	120
SESSION_TIMEOUT	int	300
KEEP_CONNECTION	string(ON|OFF|AUTO)	AUTO
STATEMENT_POOLING	string(ON|OFF)	ON
LONG_QUERY_TIME	int	60
LONG_TRANSACTION_TIME	int	60
SQL_LOG_MAX_SIZE	int	100000
LOG_BACKUP	string(ON|OFF)	OFF
SOURCE_ENV	string	cubrid.env
MAX_STRING_LENGTH	int	-1
APPL_SERVER_PORT	int	33001
ACCESS_LOG	string(ON|OFF)	ON
ACCESS_LIST	string	

The value you input or edit is instantly validated to minimize user errors.



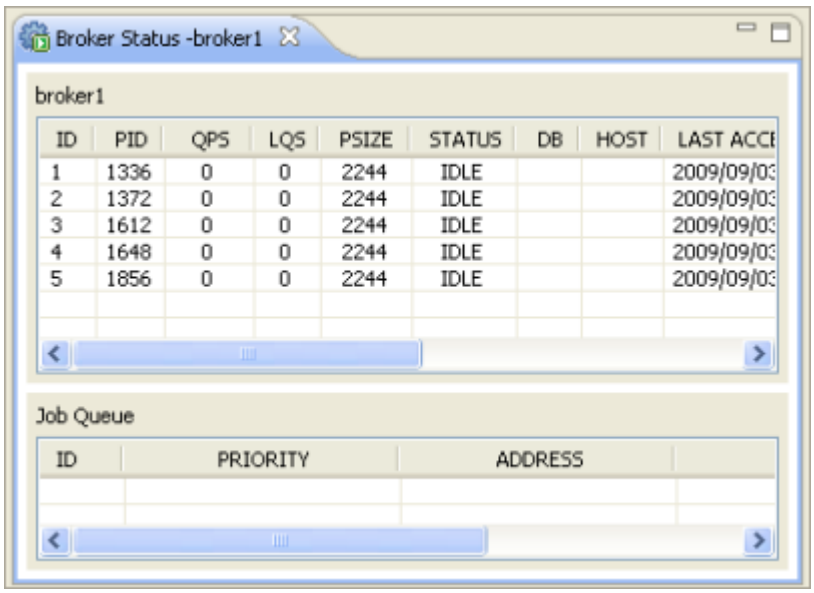
Broker Function

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.



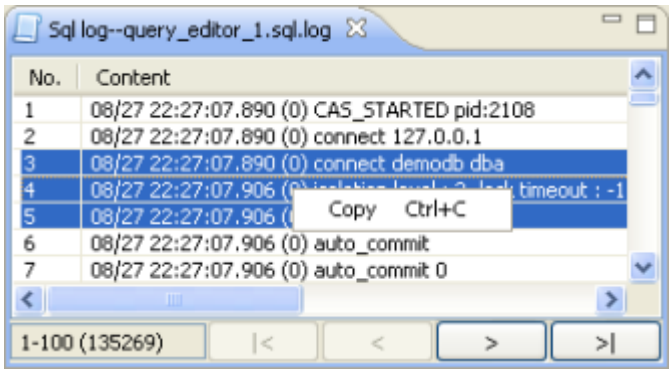
SQL Logs

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.



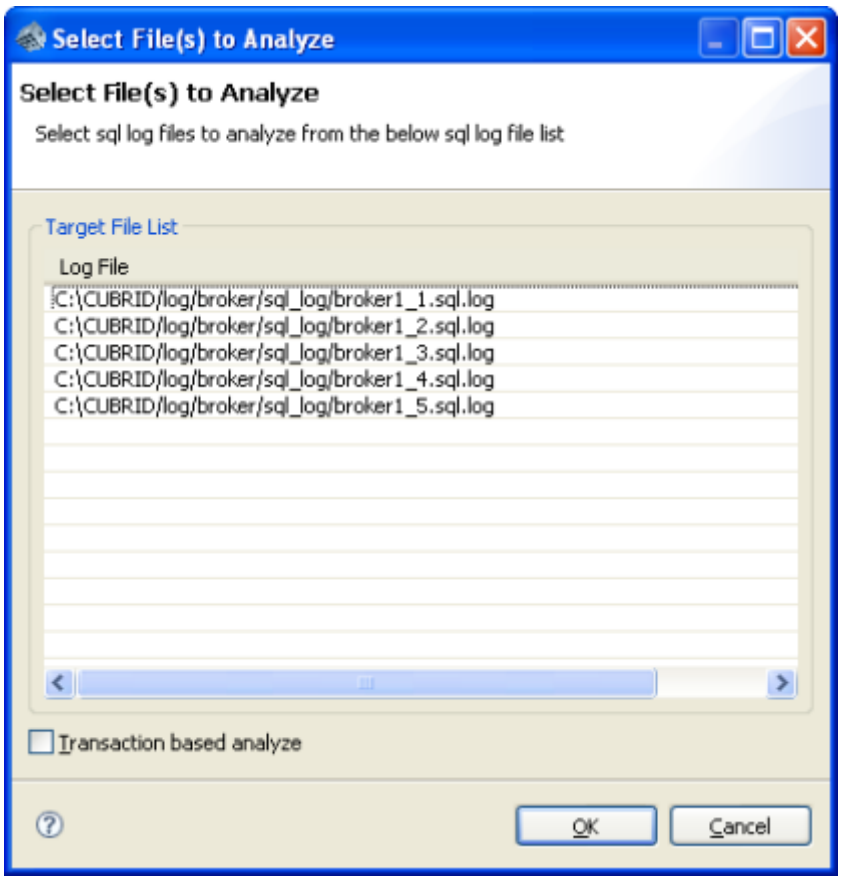
View Log

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.

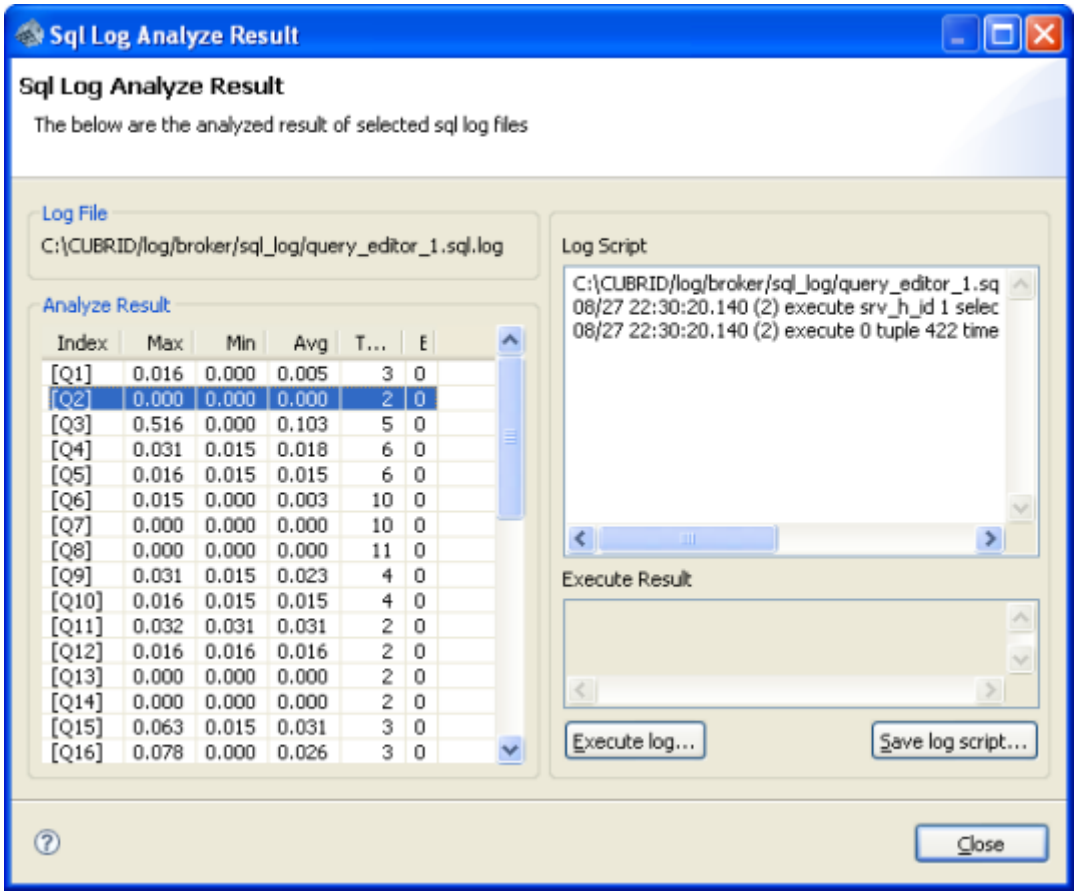


Analyze Log

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.



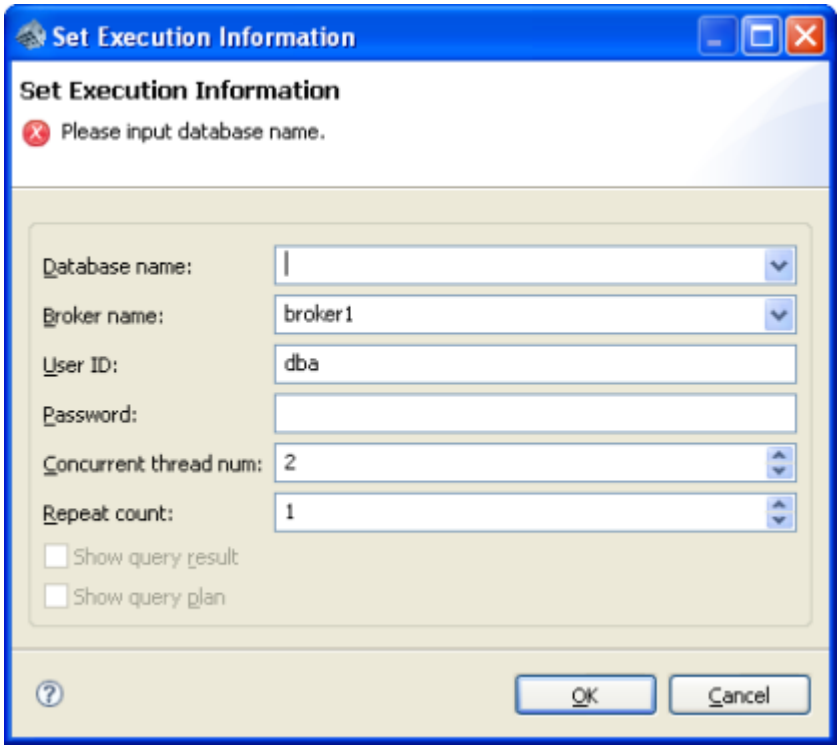
If you select the SQL log file you want and then [OK] in the [Select File(s) to Analyze] dialog box, the [Sql Log Analyze Result] dialog box which shows the results appears.



- **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

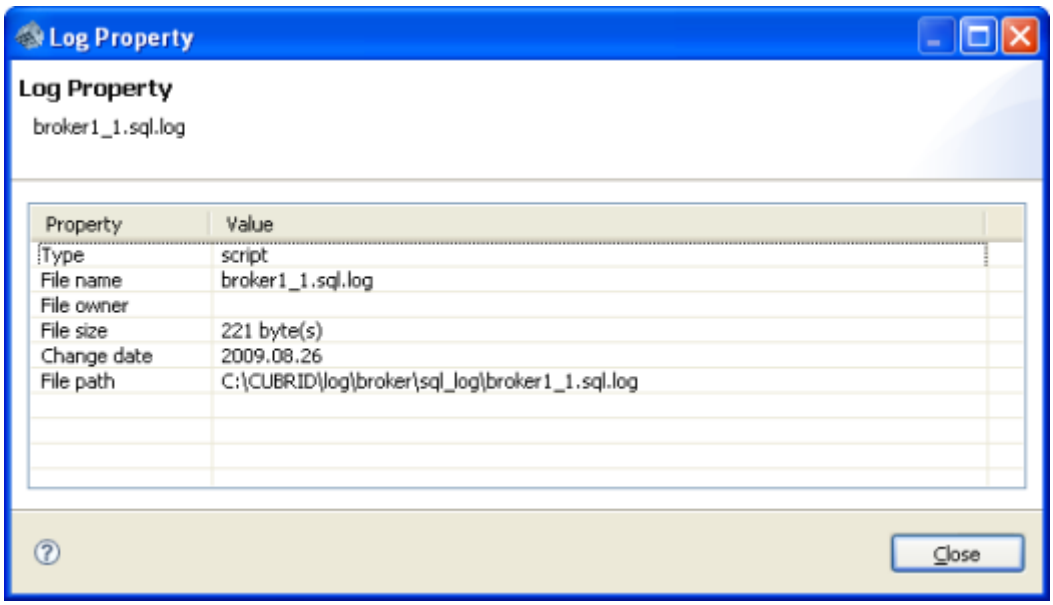
If you select [Execute log], the [Set Execution Information] dialog box appears, in which you can configure the environment.



- **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
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
Close

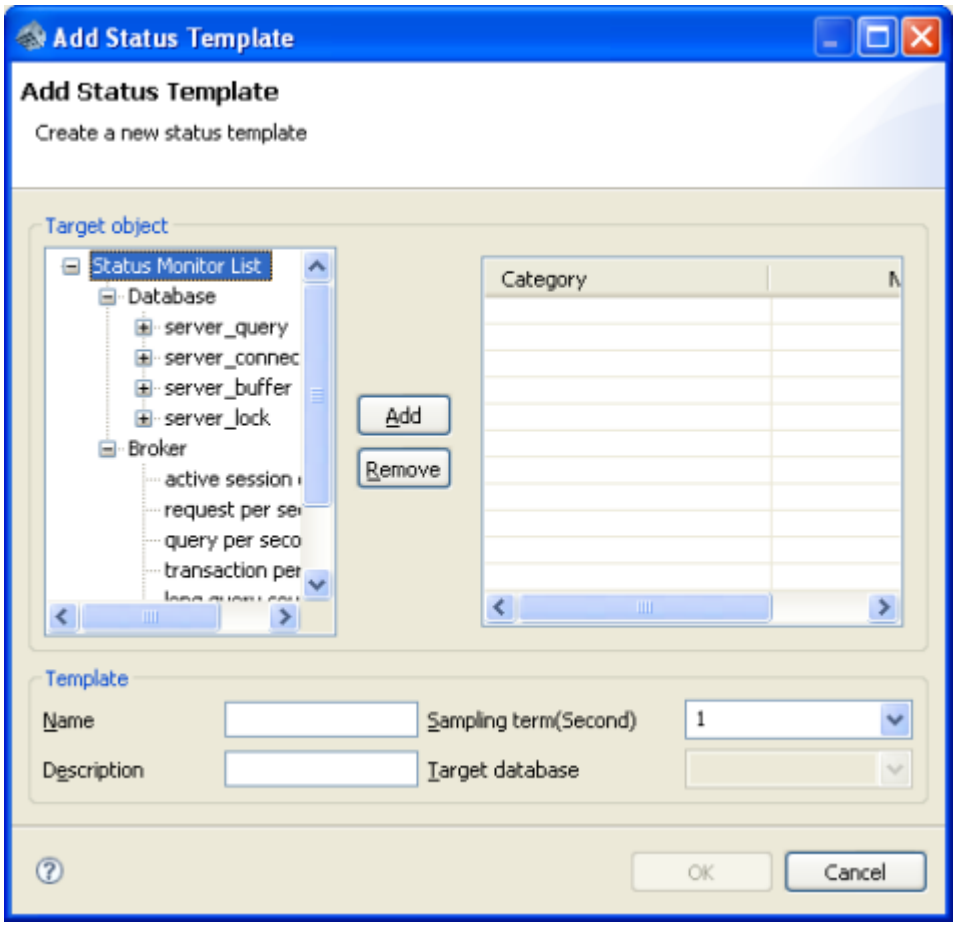
Status Monitor

The Status monitor provides the following functions:

- Adding the Status Monitor
- Executing the Status Monitor
- Editing the Status Monitor
- Deleting the Status Monitor

Adding the Status Monitor

You can set the information for the monitoring target by right-clicking [Status monitor] in the navigation tree and then selecting [Add Template].



Database

Database monitoring items are information that can be collected only when the **execute_diag** parameter is selected (ON). These database items are divided into query, connection, buffer and lock information.

- **server_query**
- **opened_page** : Indicates the number of buffer pages kicked out of the buffer pool. This value increases when the cache hit rate of the server buffer page decreases.
- **slow_query** : Indicates, out of all queries executed on the server, the number of queries whose execution time exceeds the value set by the CUBRID Manager parameter **server_long_query_time**. The **server_long_query_time** parameter can be modified from [Properties] of the CUBRID Manager or in **$CUBRID/conf/cm.conf**.
- **full_scan** : Indicates the number of table scans that is not able to perform an index seek.

- **server_connection**
- **client_request** : Indicates the number of queries requested by the client.
- **aborted_client** : Indicates the number of times when the database terminates abnormally, causing abnormal disconnection.
- **conn_request** : Indicates the number of times when the connection to the database is requested.
- **conn_reject** : Indicates the number of times when the requested connection to the database fails.
- **server_buffer**
- **buffer_page_read** : Indicates the number of database pages read from the disk.
- **buffer_page_write** : Indicates the number of database pages written to the disk.
- **server_lock**
- **dead_lock** : Indicates the number of deadlocks which occur on the server.
- **lock_request** : Indicates the number of lock requests for the database table or row.

Broker

Broker-related information can be collected regardless of CUBRID Manager variable settings. The information consists of the number of requests per second, the number of sessions being executed, the number of transactions processed per second and the number of queries processed per second.

All other object monitoring values except for **active_session** of the Broker represent the number of occurrences that happen during the last sampling interval.

- **request_sec** : Indicates the number of requests per second received by the Broker.
- **active_session** : Indicates the number of sessions currently connected to the Broker. That is, it is the number of BUSY sessions.
- **transaction_sec** : Indicates the number of transactions processed per second by the Broker.
- **query_ sec** : Indicates the number of queries 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.

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 a multi-monitor environment.



Editing the Status Monitor

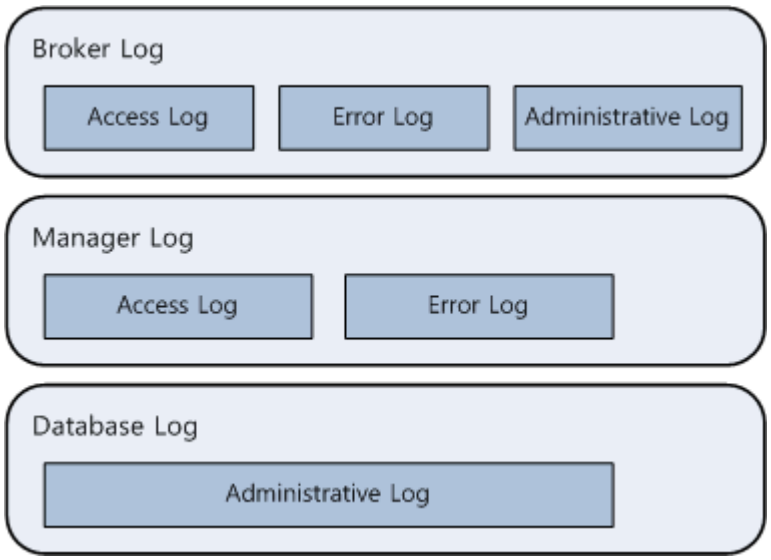
You can edit in the same interface as with [Add Template] by right-clicking the Status monitor template to edit and then selecting [Edit Template].

Deleting the Status Monitor

To delete a registered Status monitor template, right-click the template to delete and then select [Delete Template].

Logs

Logs consist of Broker, Manager and Database logs. There can be subnodes such as Access, Error and Admin logs. Each log consists as follows:



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 ir
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, 'demodt
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, 'demodt
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, 'demodt
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, 'demodt
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, 'demodt
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, 'demodt
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, 'demodt

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 manages 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

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 enables the system to make a connection to the CUBRID database in an application written in Java. The CUBRID JDBC driver v1.0 was developed based on the JDBC 1.2 specification. This driver is the Pure Java Driver for Database Middleware (also known as type 3 driver) that connects to the DBMS through CUBRID CAS. By default, the CUBRID JDBC driver provides compilation output with JDK 1.5. If you use JDK 1.4 or JDK 1.6, see the release note or the JDBC configuration section.

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();
       }
       ...
```

Connection Configuration

The **`DriverManager`** is a basic service 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`*) method. The **`getConnection`** method returns the **`Connection`** object, which is used for query and command executions and transaction commit or rollback. The URL is composed as follows:

`jdbc`:**`cubrid`**:[*`hostname`*|*`ipaddr`*]:[*`port`*]:[*`dbname`*]:[*`userid`*]:[*`password`*]

```
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 method, which requests the transaction rollback, exits when the server completes the work.

Using OIDs and Collections

In addition to the methods defined in the JDBC specification, the CUBRID JDBC driver provides methods that handle OIDs, collections (set, multiset and sequence) and GLOs.

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, collections and GLOs.

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:210.216.33.250:43300: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();
       }
   }
}
```

Using GLOs

To use GLOs, you should use the CUBRIDConnection through the DriverManager. The connection received from the CUBRIDConnection is used to input the FileInputStream object by using the getNewGLO() method. An instance of the CUBRIDOID class is returned as the result.

Example 1

```
import java.sql.*;
import java.sql.*;
import java.io.*;
import java.lang.*;
import cubrid.jdbc.driver.*;
import cubrid.sql.*;

//-------------------------------------------------------------
// create class filetest under glo
//
// assumes that testin1, testin2 files are in the location
// of the execution of this program.
// result of theexecution will create in the location
// of execution,
//-------------------------------------------------------------

class GLO_Sample
{
   public static void main (String args [])
   {
       String url = "jdbc:cubrid:210.216.33.250:43300:demodb:::" ;
       String user = "";
       String passwd = "";
       try {
           Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
       } catch(Exception e){
           e.printStackTrace();
       }
       try {
           CUBRIDConnection con = (CUBRIDConnection)
           DriverManager.getConnection(url,user,passwd);
           FileInputStream inputfile = new FileInputStream("testin");
           CUBRIDOID oid = con.getNewGLO("filetest", inputfile);
           inputfile.close();
           System.out.println("getNewGLO1..");
           FileOutputStream outputfile = new FileOutputStream("testout");
           oid.loadGLO(outputfile);
           outputfile.close();
           System.out.println("loadGLO1..");
           inputfile = new FileInputStream("testin2");
           oid.saveGLO( inputfile );
           inputfile.close();
           System.out.println("saveGLO2..");
           outputfile = new FileOutputStream("testout2");
           oid.loadGLO(outputfile);
           outputfile.close();
           System.out.println("loadGLO2..");
           con.commit();
           con.close();
       } catch(SQLException e) {
           e.printStackTrace();
       } catch(Exception e) {
           e.printStackTrace();
       }
   }
}
```

Example 2

```
import java.sql.*;
import java.lang.*;
```

```
import cubrid.jdbc.driver.*;
import cubrid.sql.*;

class SetOID_Sample
{
   public static void main (String args [])
   {
       String url = "jdbc:cubrid:210.216.33.250:43300:demodb:::" ;
       String user = "";
       String passwd = "";
       String usql = "update location set city='Pusan' where location=?";
       String sql = "select location, city from location" ;

       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);
           CUBRIDOID oid = null;

           rs.next()
           oid = rs.getOID(1);

           CUBRIDPreparedStatement ps = (CUBRIDPreparedStatement)
           con.prepareStatement(usql);

           ps.setOID(1, oid); // OID of the last tuple
           ps.executeUpdate();
           rs.close();
           stmt.close();
           con.close();
       } catch(SQLException e) {
           e.printStackTrace();
       }
    }
}
```

Retrieving 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 Columns 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**, **Statement.executeUpdate** or **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 classes of the database server that supports the auto-increment columns. Each method form can be applied only to a single-row **INSERT** statement.
  - To insert a row using the **PreparedStatement.executeUpdate** method, create a **PreparedStatement** object by calling one of the forms of the

Connection.prepareStatement method.
Connection.prepareStatement(*sql statement*, **Statement.RETURN_GENERATED_KEYS**);

- To insert a row using the **Statement.executeUpdate** method, call one of the following forms of the **Statement.executeUpdate** method.
  Statement.executeUpdate(*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.
  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.

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 class with the auto-increment feature, entering data into the class 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
```

CUBRIDConnection

Overview

The **CUBRIDConnection** class extends the standard **Connection** class and contains the following additional methods.

Return Type	Method Name
CUBRIDOID	getNewGLO(String className, InputStream stream)
CUBRIDOID	getNewGLO(String className, InputStream stream, int length)

getNewGLO

Description

Creates a new GLO with the specified *`className`* and the *`stream`* to be saved. You can specify the length of the stream to be saved with *`length`*.

Syntax

```
CUBRIDOID getNewGLO(String className, InputStream stream)
CUBRIDOID getNewGLO(String className, InputStream stream, int length)
```

Return Value

- **`CUBRIDOID`** of the object where the glo is saved.

Example

```
// Schema used in the example:
//
// create class filetest under glo;
//

CUBRIDConnection con = (CUBRIDConnection)
                    DriverManager.getConnection(url,user,passwd);

// Input file name : "testin1"
FileInputStream inputfile = new FileInputStream("testin1");
CUBRIDOID oid = con.getNewGLO("filetest", inputfile);
inputfile.close();

con.commit();
// Because you cannot use the auto commit method while using CUBRIDOID,
// you need to specify a commit to terminate this transaction.
```

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	loadGLO(OutputStream stream)

void	putIntoSequence(String attrName, int index, Object value)
void	remove()
void	removeFromSequence(String attrName, int index)
void	removeFromSet(String attrName, Object value)
void	saveGLO(InputStream stream)
void	saveGLO(InputStream stream, int length)
void	setReadLock()
void	setValues(String[] attrNames, Object[] values)
void	setWriteLock()

addToSequence

Description

This method inserts 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 method inserts 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 method converts 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 method converts 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 method 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 method returns 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 method returns 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
}
```

loadGLO

Description

This method loads the glo corresponding to the **CUBRIDOID** within the given *stream*.

Syntax

```
void loadGLO(OutputStream stream)
```

Example

```
CUBRIDConnection con = (CUBRIDConnection)
                    DriverManager.getConnection(url,user,passwd);

// filename : testin1, classname : filetest
FileInputStream inputfile = new FileInputStream("testin1");
CUBRIDOID oid = con.getNewGLO("filetest", inputfile);
inputfile.close();

// filename : testout1, classname : filetest
FileOutputStream outputfile = new FileOutputStream("testout1");
System.out.println("loadGLO..");
oid.loadGLO(outputfile);
outputfile.close();

con.commit();
// You should commit this transaction.
// You cannot use the Auto commit feature when using CUBRIDOID.
```

putIntoSequence

Description

This method modifies 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 method removes 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 method removes 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 method removes 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}
}
```

saveGLO

Description

This method saves the given *stream* into the glo held by this **CUBRIDOID**. This method stores a given *stream* in the glo corresponding to the CUBRIDOID as much as *length* (bytes).

Syntax

```
void saveGLO(InputStream stream)
void saveGLO(InputStream stream, int length)
```

Example

```
CUBRIDConnection con =
(CUBRIDConnection)DriverManager.getConnection(url,user,passwd);

// filename : testin1, classname : filetest
FileInputStream inputfile = new FileInputStream("testin1");
CUBRIDOID oid = con.getNewGLO("filetest", inputfile);
inputfile.close();

// filename : testin2, classname : filetest
FileInputStream  inputfile = new FileInputStream("testin2");
System.out.println("saveGLO..");
oid.saveGLO( inputfile );
inputfile.close();

con.commit();
// You should commit this transaction.
// You cannot use the Auto commit feature when using CUBRIDOID.
CUBRIDConnection con =
(CUBRIDConnection)DriverManager.getConnection(url,user,passwd);

// filename : testin1, classname : filetest
FileInputStream inputfile = new FileInputStream("testin1");
CUBRIDOID oid = con.getNewGLO("filetest", inputfile);
inputfile.close();
```

```
// filename : testin2, classname : filetest
FileInputStream  inputfile = new FileInputStream("testin2");
System.out.println("saveGLO..");
oid.saveGLO( inputfile, 100 );  // save just 100 byte
inputfile.close();

con.commit();
// You must commit this transaction
// NOT AUTO COMMIT on CUBRIDOID
```

setReadLock

Description

This method sets 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 method replaces 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 method sets 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 method executes 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 method specifies 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 method specifies 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

These methods 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

These methods 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 method returns 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 method returns 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 method returns 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 Programming

CUBRID ODBC Driver

The CUBRID ODBC driver supports ODBC core and some of Level 1 and Level 2 APIs. The CUBRID ODBC driver is based on ODBC Spec 3.x. Therefore, backward compatibility is not completely ensured for programs written using ODBC Spec 2.x. As CUBRID is an object-relational DBMS, some ODBC features implemented for the relational DBMS are not supported.

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}."

CUBRID/ODBC Data Type Mapping

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)

Backward Compatibility

The following table shows, among ODBC 2.x functions, the ones that can be used in ODBC 3.x instead of the ones not supported, according to the backward compatibility.

ODBC 2.x function

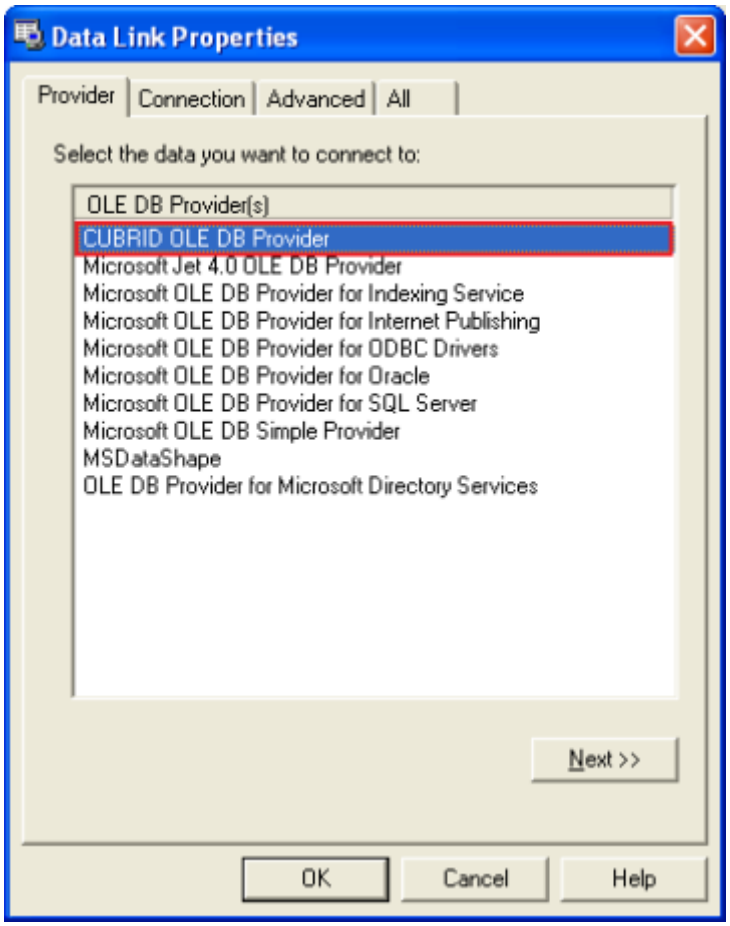
ODBC 2.x function	**ODBC 3.x function**
SQLAllocConnect	SQLAllocHandle
SQLAllocEnv	SQLAllocHandle
SQLAllocStmt	SQLAllocHandle
SQLBindParam	SQLBindParameter
SQLColAttributes	SQLColAttribute
SQLError	SQLGetDiagRec
SQLFreeConnect	SQLFreeHandle
SQLFreeEnv	SQLFreeHandle
SQLFreeStmt with SQL_DROP	SQLFreeHandle
SQLGetConnectOption	SQLGetConnectAttr
SQLGetStmtOption	SQLGetStmtAttr
SQLParamOptions	SQLSetStmtAttr
SQLSetConnectOption	SQLSetConnectAttr
SQLSetParam	SQLBindParameter
SQLSetScrollOption	SQLSetStmtAttr
SQLSetStmtOption	SQLSetStmtAttr
SQLTransact	SQLEndTran

OLE DB Programming

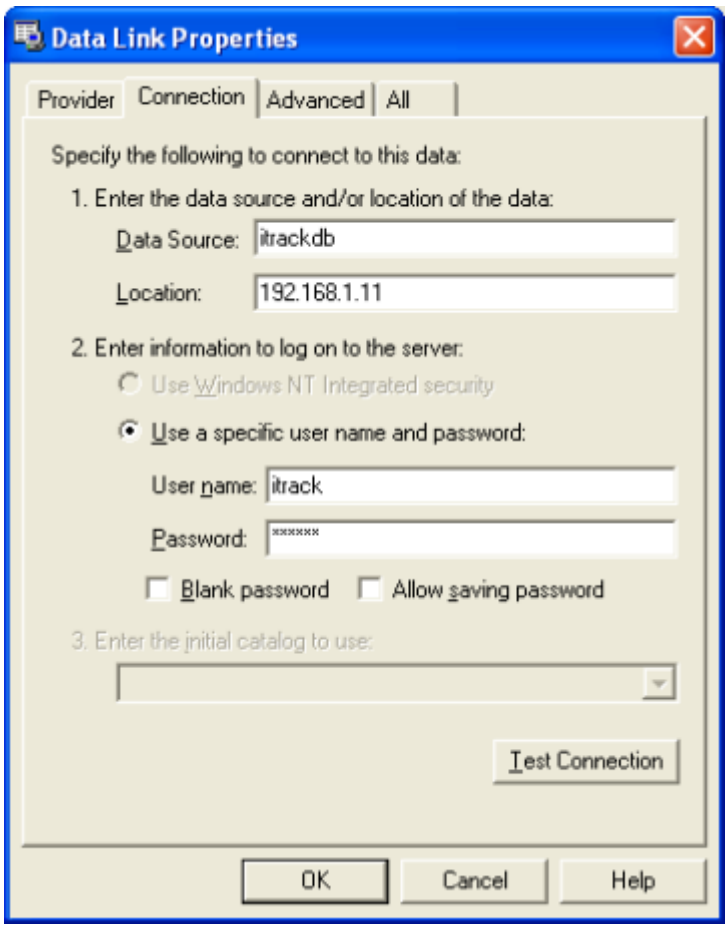
Using Data Link Property Dialog Box

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.

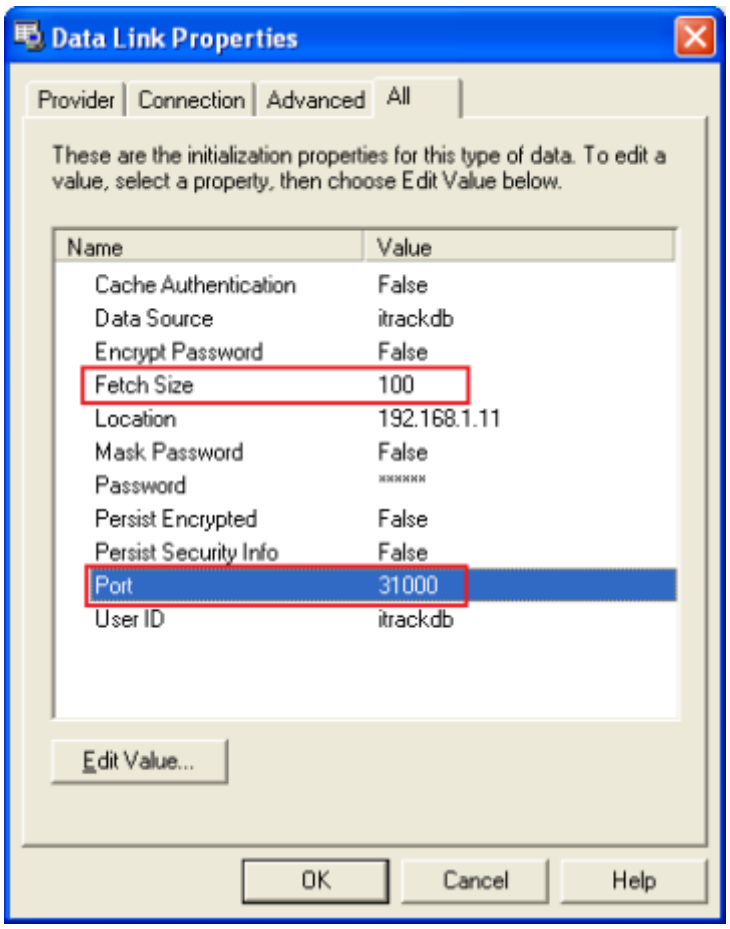


If you click the [Next] button after selecting 'CUBRID OLE DB Provider', the [Connection] tab appears as shown below. Set the desired link properties in the [Connection] tab.



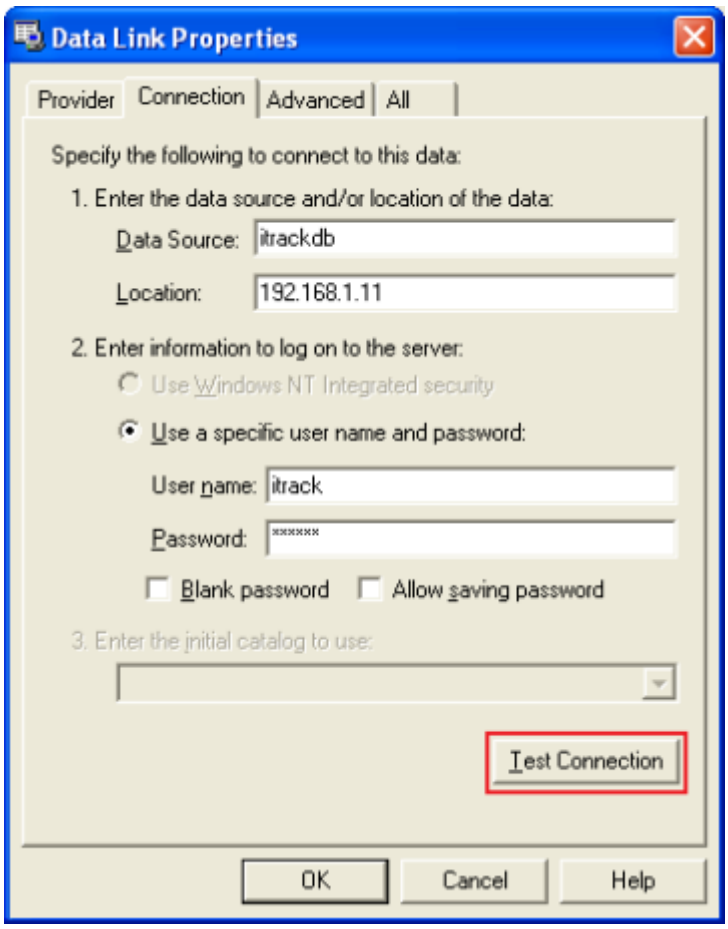
- **`Data source`** : Enter the name of the CUBRID database.
- **`Location`** : Enter the IP address or name of the server where the given database exists.
- **`User name`** : Enter the name of the user who will log on to the server.
- **`Password`** : Enter the password to be used for the server logon.

Select all connection properties and then click the [All] tab.



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.



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	Server IP
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 Programming

General Features

Connection/Transaction

- **Connecting to a database**

  The first step of a database application is to use the cubrid_connect() method which provides a database connection. Once the cubrid_connect() method is executed successfully, you can use any methods available in the databse. It is very important to call the cubrid_disconnect() method before the application is terminated completely. The cubrid_disconnect() method terminates the current transaction as well as the connection handle and all request handles created by the cubrid_connect() method.

  Note Executing the cubrid_connect() method does not create the connection with the database server automatically. The actual connection is made when the necessary method for the database server connection is called.

- **Database transaction**

  The cubrid_commit() or cubrid_rollback() method is used to commit or roll back a transaction. The cubrid_disconnect() method terminates the transaction and rolls back uncommitted ones.

Processing Queries

- **Executing queries**

  The following are basic steps of query execution.

  a. Creating a connection handle

  b. Creating a request handle for an SQL query request

  c. Fetching the result

  d. 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() method is used to get an array containing column types, and the cubrid_column_names() method 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() method 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() method. The type of the array can be determined when the cubrid_fetch() method 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.

  a. Numeric array

```
while (list($id, $name) = cubrid_fetch($req, CUBRID_NUM)) {
   echo $id;
   echo $name;
}
```

  b. 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, methods, triggers and constraints can be obtained by calling the cubrid_schema() method. The return value of the cubrid_schema() method 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() methods.

The return value of the cubrid_error_code_facility() method 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() method, you can get the OID value of the current row updated by the cubrid_current_oid() method.

```
$req = cubrid_execute($con, "select * from person",
CUBRID_INCLUDE_OID);
if ($req) {
while ($row = cubrid_fetch ($req)) {
echo cubrid_current_oid ($req);
echo $row[°8id°±];
echo $row[°8name°±];
}
cubrid_close_request ($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() method, 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 methods that support array data types. The following is an example of fetching the query result with the cubrid_fetch() method.

```
$row = cubrid_fetch ($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() method.

```
$tels = cubrid_col_get ($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() methods.

```
$tels = cubrid_col_get ($con, $oid, "tels");
while (list ($key, $tel) = each ($tels)) {
$res = cubrid_set_drop ($con, $oid, "tel", $tel);
}
cubrid_commit ($con);
```

Using GLOs

The GLO data type is used to save files in the database.

The cubrid_new_glo() method is used to create a glo instance. Files are saved in the created instance. The cubrid_save_to_glo() method is used to save files to the specified GLO. The cubrid_load_from_glo()

method is used to read data from an GLO instance. The cubrid_ send_glo() method is used to send GLO data as PHP standard output.

```
$oid = $cubrid_new_glo ($con, "glo", "input.txt");
$cubrid_load_from_glo ($con, $oid, "output.txt");
Header ("Content-type: image/jpeg");
cubrid_send_glo ($con, $oid);
```

cubrid_affected_rows

Description

The **cubrid_affected_rows** method 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

The **cubrid_bind** method 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

The **cubrid_close_request** method 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

The **cubrid_col_get** method 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

The **cubrid_col_size** method 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

The **cubrid_column_names** method 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

The **cubrid_column_types** method 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

The **cubrid_commit** method 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** method 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

The **cubrid_connect** method 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* : CAS server host name
- *port* : CAS server port number
- *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);
}
```

- *host* : CAS server host name
- *port* : CAS server port number
- *dbname* : Database name
- *userid* : Database user name
- *passwd* : Database user password

See Also

- **cubrid_disconnect**

cubrid_current_oid

Description

The **cubrid_current_oid** method 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_disconnect

Description

The **cubrid_disconnect** method 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

The **cubrid_drop** method 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

The **cubrid_error_code** method 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

The **cubrid_error_code_facility** method 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

The **cubrid_error_msg** method 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

The **cubrid_execute** method 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 a prepared query 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

The **cubrid_fetch** method 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_get

Description

The **cubrid_get** method 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_class_name

Description

The **cubrid_get_class_name** method 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_is_instance

Description

The **cubrid_is_instance** method 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 : If the instance exists
- 0 : If the instance does not exist
- -1 : If 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_load_from_glo

Description

The **cubrid_load_from_glo** method is used to read data from a glo instance and saves it to the desired file.

Syntax

```
int cubrid_load_from_glo (int conn_handle, string oid, string file_name)
```

- *conn_handle* : Connection handle
- *oid* : OID of an instance from which data is to be read
- *file_name* : Name of the file to which data is to be saved

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
$req = cubrid_execute ($con, "select image from person where id=1");
if ($req) {
   list ($oid) = cubrid_fetch($req);
   cubrid_close_request($req);
```

```
   $res = cubrid_load_from_glo ($con, $oid, "output.jpg");
   if ($res) {
      echo "image changed successfully";
   }
}
```

See Also

- cubrid_new_glo
- cubrid_save_to_glo
- cubrid_send_glo

cubrid_lock_read

Description

The **cubrid_lock_read** method 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

The **cubrid_lock_write** method 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

The **cubrid_move_cursor** method 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_new_glo

Description

The **cubrid_new_glo** method is used to create a glo instance in the requested class (GLO class). The type of the glo to be created is LO, and saved in the *file_name* file.

Syntax

```
mixed cubrid_new_glo (int conn_handle, string class_name, string file_name)
```

- *conn_handle :* Connection handle
- *class_name :* Name of the class in which a glo is to be created

- *file_name :* Name of the file in which the created GLO is to be saved

Return Value

- Success : OID of the created instance
- Failure : FALSE

Example

```
$oid = cubrid_new_glo ($con, "glo", "input.jpg");
if ($oid){
   $req = cubrid_execute ($con,
                          "insert into person(image) values ($oid)");
   if ($req) {
      echo "image inserted successfully";
      cubrid_close_request ($req);
      cubrid_commit($con);
   }
}
```

See Also

- **cubrid_save_to_glo**
- **cubrid_load_from_glo**
- **cubrid_send_glo**

cubrid_num_cols

Description

The **cubrid_num_cols** method 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_rows

Description

The **cubrid_num_rows** method is used to return the number of rows in the query result. This method 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

The **cubrid_prepare** method 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

The **cubrid_put** method 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, float-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_save_to_glo

Description

The **cubrid_save_to_glo** method is ued to save a desired file to a glo instance.

Syntax

```
int cubrid_save_to_glo (int conn_handle, string oid, string file_name)
```

- *conn_handle* : Connection handle
- *oid* : OID of a glo instance to which the file is to be saved
- *file_name* : Name of the file to be saved

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
$req = cubrid_execute ($con, "select image from person where id=1");
if ($req) {
   list ($oid) = cubrid_fetch($req);
   cubrid_close_request($req);
   $res = cubrid_save_to_glo ($con, $oid, "input.jpg");
   if ($res) {
      echo "image changed successfully";
   }
}
```

See Also

- **cubrid_new_glo**
- **cubrid_load_from_glo**
- **cubrid_send_glo**

cubrid_schema

Description

The **cubrid_schema** method 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** method 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_SUPERCLASS	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"];
}
```

See Also

- **cubrid_new_glo**
- **cubrid_load_from_glo**
- **cubrid_send_glo**

cubrid_send_glo

Description

The **cubrid_send_glo** method is used to obtain data from an GLO instance and sends it as PHP standard output.

Syntax

```
int cubrid_send_glo (int conn_handle, string oid)
```

- *conn_handle* : Connection handle
- *oid* : OID of an instance from which data is to be obtained

Return Value

- Success : TRUE
- Failure : FALSE

Example

```
$req = cubrid_execute ($con, "select image from person where id =1");
if ($req) {
  list ($oid) = cubrid_fetch($req);
  cubrid_close_request($req);
  Header ("Content-type: image/jpeg");
  cubrid_send_glo ($con, $oid);
}
```

See Also

- **cubrid_new_glo**
- **cubrid_save_to_glo**
- **cubrid_load_from_glo**

cubrid_seq_drop

Description

The **cubrid_seq_drop** method 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

The **cubrid_seq_insert** method 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

The **cubrid_seq_put** method 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

The **cubrid_set_add** method is ued 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

The **cubrid_set_drop** method 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_rollback

Description

The **cubrid_rollback** method 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_version

Description

The **cubrid_version** method 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 Overview

Overview

The Broker is a generic application server that executes queries received from the client and sends the results to the client. CCI is a C interface for the client that is necessary for the Broker CAS application server to execute queries. It is used as an infrastructure for making tools that utilizes CAS (e.g. PHP and ODBC).

CCI is composed of header and library files.

- Header file : **cas_cci.h**
- Library file : **libcascci.a** (Windows : **cascci.lib**)

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, time type
		short mon	
		short day	
		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			
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	
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_connect

Description

This method is used to create a connection handle. Only the information of the given connection is saved. The actual connection occurs at the first server request.

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) Server port
- *db_name* : (IN) Database name
- *db_user* : (IN) Database user name
- *db_passwd* : (IN) Database user password

Return Value

- Success : Connection handle (int)
- Failure : Error code

cci_disconnect

Description

This method 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 : no error)

cci_end_tran

Description

This method is used to perform a commit or rollback on the current transaction. All open request handles are closed and then the transaction and the connection with the server is terminated. The connection handle is valid even after the connection with the server is terminated. This is similar to the state in which a single connection handle is assigned by the **cci_connect** method.

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 : no error)

cci_prepare

Description

This method is used to prepare an SQL statement. For the prepared SQL statement, input binding information and, if the SQL statement is **SELECT**, column information are provided. If the SQL statement is composed of multiple queries, the preparation is performed only for the first query. **CCI_PREPARE_INCLUDE_OID** is automatically configured if the **CCI_PREPARE_UPDATABLE** flag has been configured.

To create updatable (sensitive) results, **CCI_PREPARE_UPDATABLE** must be configured in the *flag*. Because you cannot create updatable results for all queries even when **CCI_PREPARE_UPDATABLE** has been configured, you should check if the results are updatable (sensitive) by using **cci_is_updatable** after the preparation.

To create updatable (sensitive) results, 1) the query must be **SELECT**, 2) OID must be allowed in the query results, and 3) 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**
  - **CCI_PREPARE_UPDATABLE**
- *err_buf* : (OUT) Database error buffer

Return Value

- Success : Request handle (int)
- Failure : Error code

cci_get_bind_num

Description

This method 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

cci_is_updatable

Description

This method is used to inform whether the prepared query is updatable (or sensitive).

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

cci_get_result_info

Description

If a prepared SQL statement is **SELECT**, this method gets the pointer for the column information. If the handle is not for **SELECT**, **NULL** is returned and the value of *num* becomes 0. In this case, methods such as **CCI_GET_RESULT_INFO_TYPE**() must not be called.

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**

CCI_GET_RESULT_INFO_TYPE

Description

This macro is used to get the *index* -th column type of a prepared **SELECT** statement. It is not checked whether *res_info* of the parameter is **NULL**, or 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_GET_RESULT_INFO_SCALE

Description

This macro is used to get the *index* -th column's scale of a prepared **SELECT** statement. It is not checked whether *res_info* of the parameter is **NULL**, or 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_PRECISION

Description

This macro is used to get the *index* -th precision of a prepared **SELECT** statement. It is not checked whether *res_info* of the parameter is **NULL**, or 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_NAME

Description

This macro is used to get the *index* -th column name of a prepared **SELECT** statement. It is not checked whether *res_info* of the parameter is **NULL**, or 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 by **cci_get_result_info**
- *index* : (IN) Column index

Return Value

- Column name (char*)

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 is not checked whether *res_info* of the parameter is **NULL**, or 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 is not checked whether *res_info* of the parameter is **NULL**, or 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 is not checked whether *res_info* of the parameter is **NULL**, or 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_bind_param

Description

This method is used to bind parameters for a prepared request handle. Converts *value* of the given *a_type* to an actual binding type and saves it. Subsequently, whenever **cci_execute**() occurs, 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*, *value* pointers to **CCI_A_TYPE_STR**, **CCI_A_TYPE_BIT** and **CCI_A_TYPE_SET** are copied, but no values are copied. If **CCI_BIND_PTR** is not configured, memory is assigned and values are copied. If multiple columns are bound by using the same memory buffer, **CCI_BIND_PTR** must not be configured.

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)

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) Binding location
- *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 : no error)

cci_bind_param_array

Description

This method 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.

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)

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 : no error)

cci_bind_param_array_size

Description

This method 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 SQL statement
- *array_size* : (IN) Binding array size

Return Value

- Error code (0 : no error)

cci_execute

Description

This method is ued to execute a query by using the data bound to a prepared SQL statement. *max_col_size* specifies the maximum size of a column when it is fetched by the **SELECT** statement. If the type of the column is **CHAR**, **VARCHAR**, **NCHAR**, **VARNCHAR**, **BIT** or **VARBIT**, the maximum size is the size of the column to be transferred to the client. If *max_col_size* is 0, all data is retrieved.

If the SQL statement used during preparation is composed of multiple queries, all queries are executed when **CCI_EXEC_QUERY_ALL** *flag* has been set, and only the first query is executed if *flag* has not been set.

If **CCI_EXEC_QUERY_ALL** *flag* has been set,

- The return value is the result of the first query.
- If an error occurs in one of the queries, the execution is considered as a failure.
- For a query composed of q1; q2; q3, if q1 succeeds and q2 occurs an error, the result of q1 is valid. That is, when an error occurs, rollback is not executed for previously successful queries.
- If the query is executed successfully, you can get the result of the second query with **cci_next_result**.

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** : Async query
  - **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** (sync mode) : The number of results, (async mode) : 0

    - **INSERT**, **UPDATE** : The number of tuples reflected
    - Others : 0
- Failure : Error code

cci_execute_result

Description

This method 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

- The number of queries (< 0 : error code)

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_next_result

Description

The method 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 SQL 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 tuples reflected
- Others : 0

- Failure : Error code

cci_execute_array

Description

This method is used to execute a query using the data bound to a prepared SQL statement. Uses multiple binding data to get the same results as when executing a single query multiple times. The results of each query are sent by *query_result*. The results of each query are retrieved by **CCI_QUERY_RESULT_STMT_TYPE**, **CCI_QUERY_RESULT_RESULT** and **CCI_QUERY_RESULT_ERR_MSG**.

The query results used must be deleted by **cci_query_result_free**.

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
- *err_buf* : (OUT) Database error buffer

Return Value

- The number of queries executed (< 0 : error code)

Example

```
T_CCI_QUERY_RESULT  *qr;
char  **sql_stmt;


cci_prepare( ... );
cci_bind_param( ... );

...

res = cci_execute_array(conn, &qr, &err_buf);
if (res < 0) {
  /* error */
}
else {
for (i=1 ; i <= res ; i++) {
  result_count = CCI_QUERY_RESULT_RESULT(qr, i);
  if (resul_count < 0)
    err_msg = CCI_QUERY_RESULT_ERR_MSG(qr, i);
}
cci_query_result_free(qr, res);
}
```

cci_execute_batch

Description

This method is used to execute the number of *sql_stmt*s by *num_sql_stmt*. If *num_sql_stmt* queries are all executed successfully, the return value is 0 (no error); otherwise, the error code is returned. The results of each query are transferred by the *query_result*. The results of each query are retrieved by **CCI_QUERY_RESULT_RESULT** and **CCI_QUERY_RESULT_ERR_MSG**. The query results used must be deleted by **cci_query_result_free**.

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

- Error code (0 : no error)

Example

```
T_CCI_QUERY_RESULT *qr;
char **sql_stmt;


res = cci_execute_batch(conn, num_sql, sql_stmt, &qr, &err_buf);
if (res < 0) {
  /* error */
}
else {
for (i=1 ; i <= num_sql ; i++) {
  result_count = CCI_QUERY_RESULT_RESULT(qr, i);
  if (resul_count < 0)
    err_msg = CCI_QUERY_RESULT_ERR_MSG(qr, i);
}
cci_query_result_free(qr, num_sql);
}
```

CCI_QUERY_RESULT_RESULT

Description

This macro is used to get the result count of the **cci_execute_batch** query. It is not checked whether *query_result* of the parameter is **NULL**, or 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_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 is not checked whether *query_result* of the parameter is **NULL**, or 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_STMT_TYPE

Description

This macro is used to get the statement type of the **cci_execute_batch** query. It is not checked whether *query_result* of the parameter is **NULL**, or 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_query_result_free

Description

This method 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 : no error)

Example

```
T_CCI_QUERY_RESULT *qr;
char **sql_stmt;


res = cci_execute_array(conn, &qr, &err_buf);


cci_query_result_free(qr, res);
```

cci_get_db_parameter

Description

This method is used to get a parameter set 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

Return Value

- Error code (0 : no error)

cci_set_db_parameter

Description

This method 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 : no error)

cci_close_req_handle

Description

This method is used to close the specified request handle.

Syntax

```
int cci_close_req_handle(int req_handle)
```

- *req_handle* : (IN) Request handle

Return Value

- Error code (0 : no error)

cci_cursor

Description

This method is used to move the cursor set in the request handle. **cci_cursor** (*req_handle*, 1, **CCI_CURSOR_FIRST**, *err_buf*) always moves the cursor to the first position; **cci_cursor** (*req_handle*, 1, **CCI_CURSOR_LAST**, *err_buf*) moves it to the last position. **cci_cursor** (*req_handle*, 1, **CCI_CURSOR_CURRENT**, *err_buf*) moves the cursor the the next position from the current one. If the cursor position 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) Cursor position
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : no error)

cci_cursor_update

Description

This method 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)

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)

cci_fetch_size

Description

This method is used to determine the number of tuples 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 : no error)

cci_fetch_sensitive

Description

This method 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 : no error)

cci_fetch

Description

This method is used to fetch results in the client buffer. Fetched results can be seen by **cci_get_data**().

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 : no error)

cci_fetch_buffer_clear

Description

This method is used to clear the tuples 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 : no error)

cci_get_data

Description

This method is used to get the *col_no*-th results from the fetched ones. The results are returned to *value* for *type*. The data type of *value* for *type* is shown in the table below.

type	value Type	
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
CCI_A_TYPE_SET	**T_CCI_SET***	alloc and value copy
CCI_A_TYPE_DATE	**T_CCI_DATE**	value copy
CCI_A_TYPE_BIGINT	int64_t (For Windows : __int64)	value copy

Pointer copy returns a pointer in the application client library; it becomes invalid at the next **cci_get_data**() is called. Value copy returns a value to the out parameter; memory needs to be assigned to the pointer used as the parameter.

In the case of **CCI_A_TYPE_BIT**, pointer copy occurs for the member. In the case of **CCI_A_TYPE_BIT**, you must not free the pointer of the returned *value*. In the case of **CCI_A_TYPE_SET**, you must free the returned set with **cci_set_free**() when it is not used any more.

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) Column index
- *type* : (IN) Data type
- *value* : (OUT) Result
- *indicator* : (OUT) **NULL** indicator (0 : not **NULL**, -1 : **NULL**)
  - if *type* is **CCI_A_TYPE_STR**:
    - -1 : **NULL**

    - >= 0 : **strlen**(*value*)
- if *type* is **CCI_A_TYPE_STR**:
    - -1 : **NULL**
    - 0 : not **NULL**

Return Value

- Error code (0 : no error)

cci_schema_info

Description

This method is ued to get schema information. If 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 flags by using the OR operator (|). Performance may significantly decrease if pattern matching is used.

The following table shows tuple composition of each *type*.

Tuple 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_SUPERCLASS	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*

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
- c*lass_name* : (IN) Class name or NULL
- *attr_name* : (IN) Attribute name of NULL

- *flag* : (IN) Pattern matching flag
  CCI_CLASS_NAME_PATTERN_MATCH
  CCI_ATTR_NAME_PATTERN_MATCH
- *err_buf* : (OUT) Database error buffer

Return Value

- Success : Request handle
- Failure : Error code

cci_get_cur_oid

Description

If CCI_INCLUDE_OID is configured in execution, this method gets the OID of the currently fetched tuple. 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 : no error)

cci_oid_get

Description

This method 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

cci_oid_put

Description

This method is used to configured 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 : no error)

cci_oid_put2

Description

This method 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 : no error)

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_glo_new

Description

This method is used to create a glo object in a glo class of class_name with *filename*. If *filename* is NULL, only the glo object is created (cci 2.3 or higher).

Syntax

```
intcci_glo_new(int conn_handle, char *class_name, char *filename, char
*oid_str, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *class_name* : (IN) glo class name
- *filename* : (IN) Source file
- *oid_str :* (OUT) glo object (OID string)
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : no error)

cci_glo_save

Description

This method is used to save the file in a glo object.

Syntax

```
intcci_glo_save(int conn_handle, char *oid_str, char *filename, T_CCI_ERROR
*err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str :* (IN) glo object (OID string)
- *filename* : (IN) Source file
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : no error)

cci_glo_load

Description

This method is used to export data of the given glo object to *out_fd*.

Syntax

```
intcci_glo_load(int conn_handle, char *oid_str, int out_fd, T_CCI_ERROR
*err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str :* (IN) glo object (OID string)
- *out_fd :* (IN) Result file descriptor
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code (0 : no error)

cci_get_db_version

Description

This method 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 : no error)

cci_get_class_num_objs

Description

This method 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

```
intcci_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 : no error)

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 : no error)

cci_oid_get_class_name

Description

This method 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

cci_col_get

Description

This method 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 tuples.

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

cci_col_size

Description

This method 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 : no error)

cci_col_set_drop

Description

This method 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_set_add

Description

This method 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

cci_col_seq_drop

Description

This method 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

cci_col_seq_insert

Description

This method 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

cci_col_seq_put

Description

This method 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

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_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_set_isolation_level

Description

This method 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

cci_set_free

Description

This method 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_size

Description

This method 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

cci_set_element_type

Description

This method 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_get

Description

This method 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_make

Description

This method 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.

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* (For Windows : __int64*)
CCI_U_TYPE_DATETIME	**T_CCI_DATE***

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 method is used to configured the maximum number of tuples 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_savepoint

Description

This method is used to configured a savepoint or performs transaction rollback to a 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_glo_read_data

Description

call data_seek (*start_pos*) on <glo object>

call read_data (*length*, *buffer*) on <glo object>

Syntax

```
int cci_glo_read_data(int conn_handle, char *oid_str, int start_pos, int
length, char *buf, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str :* (IN) glo object (OID string)
- *start_pos* : (IN) Starting offset
- *length* : (IN) The number of bytes to read
- *buf* : (OUT) Data buffer
- *err_buf* : (OUT) Database error buffer

Return Value

- The number of bytes read

cci_glo_write_data

Description

call data_seek (start_pos) on <glo object>

call write_data (length, buffer) on <glo object>

Syntax

```
int cci_glo_write_data(int conn_handle, char *oid_str, int start_pos, int
length, char *buf, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str :* (IN) glo object (OID string)
- *start_pos* : (IN) Starting offset
- *length* : (IN) The number of bytes to write
- *buf* : (IN) Data buffer
- *err_buf* : (OUT) Database error buffer

Return Value

- The number of bytes written

cci_glo_insert_data

Description

call data_seek (start_pos) on <glo object>

call insert_data (length, buffer) on <glo object>

Syntax

```
intcci_glo_insert_data(int conn_handle, char *oid_str, int start_pos, int
length, char *buf, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str :* (IN) glo object (OID string)
- *start_pos* : (IN) Starting offset
- *length* : (IN) The number of bytes to be inserted
- *buf* : (IN) Data buffer
- *err_buf* : (OUT) Database error buffer

Return Value

- The number of bytes inserted

cci_glo_delete_data

Description

call data_seek (start_pos) on <glo object>

call delete_data (length) on <glo object>

Syntax

```
int cci_glo_delete_data(int conn_handle, char *oid_str, int start_pos, int
length, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str* : (IN) glo object (OID string)
- *start_pos* : (IN) Starting offset
- *length* : (IN) The number of bytes to be deleted
- *err_buf* : (OUT) Database error buffer

Return Value

- The number of bytes deleted

cci_glo_truncate_data

Description

call data_seek (start_pos) on <glo object>

call truncate_data() on <glo object>

Syntax

```
int cci_glo_truncate_data(int conn_handle, char *oid_str, int start_pos,
T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str :* (IN) glo object (OID string)
- *start_pos* : (IN) Starting offset
- *err_buf* : (OUT) Database error buffer

Return Value

- The number of bytes deleted

cci_glo_append_data

Description

call append_data (length, buf) on <glo object>

Syntax

```
int cci_glo_append_data(int conn_handle, char *oid_str, int length, char *buf,
T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str :* (IN) glo object (OID string)
- *length :* (IN) The number of bytes to be appended
- *buf* : (IN) Data buffer
- *err_buf* : (OUT) Database error buffer

Return Value

- The number of bytes appended

cci_glo_data_size

Description

call data_size() on <glo object>

Syntax

```
int cci_glo_data_size(int conn_handle, char *oid_str, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str :* (IN) glo object (OID string)
- *err_buf* : (OUT) Database error buffer

Return Value

- Data size

cci_glo_compress_data

Description

call compress_data() on <glo object>

Syntax

```
int cci_glo_compress_data(int conn_handle, char *oid_str, T_CCI_ERROR
*err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str :* (IN) glo object (OID string)
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code

cci_glo_destroy_data

Description

call destroy_data() on <glo object>

Syntax

```
int cci_glo_destroy_data(int conn_handle, char *oid_str, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) Connection handle
- *oid_str :* (IN) glo object (OID string)
- *err_buf* : (OUT) Database error buffer

Return Value

- Error code