DBMaker

Database Administrator's Guide

CASEMaker®

CASEMaker Inc./Corporate Headquarters

1680 Civic Center Drive

Santa Clara, CA 95050, U.S.A.

www.casemaker.com

www.casemaker.com/support

Publication Date: 2017-02-28

For a description of updated functions that do not appear in this manual, read the file named README.TXT after installing the CASEMaker DBMaker software.

Contents

1 Introduction

Welcome to the DBMaker Database Administrator's Guide. DBMaker is a powerful and flexible SQL Database Management System (DBMS) that supports an interactive Structured Query Language (SQL), a Microsoft Open Database Connectivity (ODBC) compatible interface, and Embedded SQL for C (ESQL/C). DBMaker also supports a Java Database Connectivity (JDBC) compliant interface and DBMaker COBOL interface (DCI). The open architecture and native ODBC interface give the user freedom to build custom applications using a wide variety of programming tools, or query a database using existing ODBC, JDBC, or DCI compliant applications.

DBMaker is easily scalable from personal single-user databases to distributed enterprise-wide databases. The advanced security, integrity, and reliability features of DBMaker ensure the safety of critical data for all database configurations. Extensive cross-platform support leverages existing hardware, and enables expansion and upgrading to more powerful hardware as needs grow.

DBMaker provides superior multimedia-handling capabilities, providing storage, searching, retrieval, and manipulation of all multimedia data types. Binary Large Objects (BLOBs) by take full advantage of the advanced security and crash recovery mechanisms in DBMaker for ensuring the integrity of multimedia data. File Objects (FOs) provide multimedia data management while maintaining the capability to edit individual files in a source application.

This guide book is for database administrators who are not familiar with the concepts and principles of the DBMaker DBMS or the syntax and grammar of the DBMaker query language (SQL). However, this resource is most successfully used when you already posses a general working knowledge of computers and are comfortable using the operating system that you are using for hosting DBMaker. Information about the operating system is beyond the scope of this manual; please consult your operating system documentation when necessary.

This guide book contains general information about the concepts and principles that a database administrator must understand when using the DBMaker DBMS. An overview of the DBMaker SQL commands for creating, maintaining, and optimizing databases are introduced and demonstrated. Throughout the manual, examples and illustrations are provided To help present the information more clearly.

The implementation of a DBMS can greatly affect the performance of database operations. Many database performance decisions about database optimization and tuning are required, including: data storage location and access, index configuration, and data protection. This manual provides a background to help database administrators and application developers make careful choices based on their understanding. SQL commands are used to illustrate most of DBMaker's supported functions. References to other database administration tools are also provided.

Most of the concepts, commands, and examples herein are presented in dmSQL, the command-line tool provided with DBMaker. In a few cases, database administration functions can only be performed using one of the other DBMaker application tools or utilities. For more information about using the application tools and utilities provided with DBMaker, please refer to section 1.1, *Other Sources of Information*.

1.1 Other Sources of Information

DBMaker provides many other user's guides and reference manuals in addition to this reference.

For more information on a particular subject, consult one of these books:

- The *SQL Command and Function Reference* provides more information about the SQL language implemented by DBMaker.
- The *ESQL/C Programmer's Guide* is an excellent resource on the ESQL/C language implemented by DBMaker.

- The *dmSQL User's Guide* offers detailed information on using dmSQL.
- The *Error Reference and Message* provides detailed information about error and warning messages.
- The *JDBA Tool User's Guide*, *JServer Manager User's Guide,* and *JConfiguration Tool Reference* each offer information on configuring and managing databases using DBMaker's JTools.
- The *DBMaker SQL Stored Procedure User's Guide* provides detailed information about the SQL stored procedure language implemented in DBMaker.
- The *ODBC Programmer's Guide* and *JDBC Programmer's Guide* provides detailed information about the native ODBC API and JDBC API.

1.2 Technical Support

CASEMaker provides thirty days of complimentary email and phone support during the evaluation period. When software is registered, the support period is extending an additional thirty days for a total of sixty days. However, CASEMaker will continue to provide email support (free of charges) for bugs reported after the complimentary support or registered support expires.

For most products, support is available beyond sixty days and may be purchased for twenty percent of the retail price of the product. Please contact sales@casemaker.com for details and prices.

CASEMaker support contact information, by post mail, phone, or email, for your area is at: www.casemaker.com/support. We recommend searching the most current database of FAQ's before contacting CASEMaker support staff.

Please have the following information available when phoning support for a troubleshooting enquiry or include this information in your correspondence:

- Product's name and version number
- Registration number

- Registered customer's name and address
- Supplier/distributor where the product was purchased
- Platform and computer system configuration
- Specific action(s) performed before error(s) occurred
- Error message and number, if any
- Any additional information deemed pertinent

1.3 Document Conventions

This guide book uses a standard set of typographical conventions for clarity and ease of use. The NOTE, Procedure, Example, and Command Line conventions also have a second setting used with indentation.

CONVENTION	DESCRIPTION
Italics	Italics indicate placeholders for information that must be supplied, such as user and table names. The word in italics should not be typed, but is replaced by the actual name. Italics also introduce new words, and are occasionally used for emphasis in text.
Boldface	Boldface indicates filenames, database names, table names, column names, user names, and other database schema objects. It is also used to emphasize menu commands in procedural steps.
KEYWORDS	All keywords used by the SQL language appear in uppercase when used in normal paragraph text.
SMALL CAPS	Small capital letters indicate keys on the keyboard. A plus sign (+) between two key names indicates to hold down the first key while pressing the second. A comma (,) between two key names indicates to release the first key before pressing the second key.
NOTE	Contains important information.
➲ **Procedure**	Indicates that procedural steps or sequential items will follow. Many tasks are described using this format to provide a logical sequence of steps for the user to follow .
➲ **Example**	Examples are given to clarify descriptions, and commonly include text, as it will appear on the screen. Other forms of this convention include Prototype and Syntax.
`CommandLine`	Indicates text, as it should appear on a text-delimited screen. This format is commonly used to show input and output for dmSQL commands or the content in the dmconfig.ini file.

Table 1-1 Document Conventions

2 Overview

The physical organization of data spanning the files that comprise a database can become quite complex. A DBMS, such as DBMaker, isolates a view of the data from the database's implementation on a computer. The database is viewed as a collection of two-dimensional tables containing rows and columns of data values . These tables are easy to visualize and provide flexibility for data modeling.

DBMaker provides many methods for retrieving data from tables. The interactive dmSQL line command tool is useful for daily transaction processing or ad-hoc queries, and the DBMaker *application programming interface (API)* is ideally suited for developing applications quickly and easily. DBMaker also includes easy-to-use graphical-based tools that are consistent across platforms.

2.1 Features

As a SQL database management system, DBMaker has all of the features traditionally found in a relational database management system. DBMaker is also enhanced with many powerful and advanced features. These enhanced features increase performance and provide DBMaker with capabilities not normally found in traditional database management systems, especially in the area of multimedia support.

Multimedia Support

Powerful multimedia management capabilities built into the database engine provide efficient storage and manipulation of large amounts of multimedia data including graphics, audio, video, animation, and text. These multimedia management capabilities provide significant flexibility, allowing multimedia data to be stored in different ways to best satisfy the needs of the user.

Multimedia features include:

- Binary Large Objects (BLOBs) and File Objects (FOs).
- Multiple BLOB and FO columns in a table.
- Edit File Objects with existing multimedia tools.
- Built-in full-text search engine.

Multimedia data can be stored directly in the database as Binary Large Objects (BLOBs). This data is fully protected by the same security, reliability, and integrity features used for conventional data types. In addition, multimedia data can be stored as file objects. The file objects provide full access to third-party multimedia tools while the multimedia data remains under database control.

64 Bit Support

DBMaker supports 64 bit porting for the Windows x64 and Linux x64 operating systems. Users must install the appropriate DBMaker 64 bit version on an x86-64 architecture CPU with a 64 bit Windows or Linux OS environment.

The 64 bit version has the following limitations:

- The 64/32 bit database server cannot start databases created with the 32/64 bit database server. Additionally, stored procedures and user defined functions are incompatible between the two versions.
- User must migrate their database if they want to use in different OS architecture. However, 32/64 bit client can connect to 64/32 bit's database server.
- User must use 64bit C compiler to compile and build UDF, ESQL/C and Stored procedure as 64bit. For .NET application, user must use VS2005 or above to compile and link as 64bit application program. For JDBC or java sp, user must use 64bit JVM to compile the java program.

- DBMaker's shared memory size is 2^{31} pages (2^{31} × PAGE SIZE bytes) for 64 bit environments. For 32 bit environments shared memory size is 2GB bytes.

JDBC Support

DBMaker supports features of JDBC 3.0 and Java Transaction API (JTA) functions. JDBC JTA facilitates connections to popular Java AP servers such as BEA WebLogic™.

To learn about implementing JDBC and JDBA, please refer to the product documentation. Information about the JDBC specification is available at: http://java.sun.com/products/jdbc/.

Information about the JTA specification is available at http://java.sun.com/products/jta/.

Microsoft Transaction Server (MTS) Support

Microsoft Transaction Server (MTS) is an integral part of Windows NT, and is installed by default as part of the Windows operating system. MTS evolved as a Transaction Processing (TP) system given Windows NT the same kinds of features available on other platforms like CICS and Tuxedo. These are specifically designed for creating stable environments for data sources.

DBMaker supports transactional operations via MTS.

The following are required to use DBMaker with MTS:

- Microsoft Data Access Components (MDAC) version 2.6 or higher to run with MTS. The latest version of MDAC is available for downloading from http://www.microsoft.com/data.
- When using MDAC 2.5, the **DM_DifEn = 0** option must be added to the **DM_COMMON_OPTION** section of the **dmconfig.ini** file.

Open Interface

High-performance applications are quickly created using ODBC 3.0's native compatible interface and ANSI SQL-99 support. Applications can be built using a wide variety of popular development tools, including Visual C++, Visual Basic, Delphi, and AcuBench. DBMaker does not restrict developers to a proprietary development environment. Developer and administrators are free to use their existing tools.

Open interface features include:

- ANSI-99 entry-level compliance
- ODBC 3.0 support
- ESQL/C preprocessor
- JDBC 2.0 support

The included ESQL/C preprocessor simplifies the development process for programs written using a traditional C development environments. Database applications written using the powerful high-level Embedded SQL query language are automatically translate it to the appropriate ODBC function calls by the DBMaker preprocessor.

Data Integrity

DBMaker provides a full range of traditional data integrity features. Primary and foreign keys ensure data integrity, with full support for referential actions. User-defined data types, together with domain, column, and table constraints ensure only valid values are entered in each field.

Data integrity features include:

- Primary and foreign key integrity checking
- Referential actions fully supported
- Table and column constraints

- User-defined data types
- Default column values

Data Reliability

Advanced data protection features keep your data safe, always. Features include: automatic crash recovery, database consistency checking, and automatic backups. These features ensure data consistency and safety in the event of operating system or disk failures.

Data reliability features include:

- Online transaction processing
- Online full, differential backupand incremental backups
- Automatic crash recovery
- Automatic incremental backups
- Automatic statistic updates
- Database consistency checking
- Multiple journal files
- Optional BLOB backup

Storage Management

DBMaker's modern storage management facilities provide flexible data storage with simple management and configuration. There is no practical limit on the number of rows in a table, or on the number of tables in a database. A table may even span multiple disks. DBMaker also enhances the development of applications that can dynamically adjust to the user's needs with its support of online table schema editing.

Storage management features include:

- Autoextend and regular tablespaces
- Raw device support on UNIX platforms

- Maximum database size of 256 PB (petabytes: 1 PB = 1,024 TB)
- Unlimited number of tables
- Unlimited number of records
- Online table schema redefinition/editing

DBMaker dynamically extends database storage space up to the available disk space. Storage space may also be fixed then manually adjusted. On UNIX platforms, DBMaker supports raw devices. These achieve maximum performance by bypassing the file system and writing data directly to the raw device.

Security Management

The centralized and multi-user nature of a DBMS requires various forms of security control. This is necessary for preventing unauthorized access and limiting the access of authorized users. User-level and group-level authority is used for implementing these database security controls. Privilege management on tables and individual columns further controls individual user access.

Security management features include:

- User-level and group-level security
- Nested groups
- Privilege management on tables and individual columns
- Privilege management on stored commands and stored procedures
- Encrypted network links

Advanced Language Features

Advanced language features complement traditional database functions. Easily extend and customize the capabilities of DBMaker using stored commands, stored procedures, Triggers, and user-defined functions. Business logic can be written directly into the database engine, centralizing the logic in the database so it is easier to manage and maintain.

Advanced language features include:

- Built-in functions
- User-defined functions
- Stored commands
- Stored procedures
- Triggers

2.2 Database Modes

The database administrator may start a database in one of several different database modes. Each mode provides different options for connecting to and accessing a database. This offers the ability to scale databases from simple single-user systems on one computer to large multi-user systems distributed across several computers.

The database modes available vary according to the database server's platform and network connection. DBMaker's three database modes are: single-user, multiple-connection, and client/server.

Single-User Mode

Single-user mode is only available on the UNIX and Linux platforms. This is a simplified version of DBMaker for non-sharable databases. Single user databases don't require locks, security, or network support. As a result, this mode benefits from, smaller application sizes and sports faster execution speeds for most database operations.

This mode is limited in that since only one single connection can exist to the database, the database cannot run extra servers or daemons (e.g., backup server, replication server, or global transaction server), and the database is not available over the network and must be accessed only from the host machine.

Multiple-Connection Mode

Multiple-connection mode is only available on the Windows platform. In this mode multiple simultaneous connections to a database are supported, with the full range of security and reliability features of DBMaker available. However, the database is not available over the network and must be accessed from the host machine.

In this mode the database does not support extra servers or daemons, such as backup server, replication server, or global transaction server.

Client/Server Mode

Client/server mode is available on all platforms. This mode permits multiple simultaneous connections to a database from any computer connected to the host computer via a TCP/IP network. The full range of security, reliability, and concurrency control features of DBMaker are available In addition, data sent across the network can be encrypted for additional security. This mode supports all of the extra servers and daemons, such as backup server, replication server, and global transaction server.

NOTE *One connection cannot support multi statement execution at the same time.*

2.3 DBMaker Interface and Tools

DBMaker comes complete with an application programming interface (API) and many tools and utilities for database management. These include a command-line based interactive SQL query tool and a graphical interface-based tool for managing multiple servers. Novice database users can enjoy the simple management features and graphical tools that are consistent across platforms.

Application Program Interface

The API is a library of low-level routines that operate directly on the database engine. The API is used when creating software applications with a general-purpose programming languages such as C++ or Visual Basic. DBMaker provides an ODBC 3.0 compatible interface that supports all core-level functions and most of the extended-level functions.

dmSQL Interactive Query Tool

dmSQL has a character-based interactive interface that directly utilizes the full power and functionality of DBMaker. dmSQL manipulates databases, performs ad-hoc queries, and displays result sets immediately. dmSQL is often the only method for exploiting the full power of a database without creating programs using a conventional programming language.

JDBA Tool

JDBA Tool has an interactive graphical interface for maintaining and monitoring databases. JDBA Tool hides the complexity of the DBMS and query language behind an intuitive, easy to understand, and convenient interface. This allows casual users the ability to access the database without learning the query language, and it allows advanced users to quickly manage and manipulate the database without the trouble of entering formal commands using SQL. JDBA Tool also provides statistical data and information on who is using a database with its monitoring functions.

JServer Manager

JServer Manager has an intuitive graphical interface for creating, starting, stopping, backing up, and restoring databases. JServer Manager provides one central location for creating and managing all database servers simultaneously.

JConfiguration Tool

JConfiguration Tool has a graphical interface for managing database configuration parameters. It provides a simple and direct method for modifying keywords in the DBMaker configuration files. Each configuration parameters is clearly defined within the user interface. This eliminates the need to cross reference the documentation or memorize the definition of keywords.

ESQL for C language

ESQL for C has a graphical interactive for editing and preprocess Embedded SQL/C programs. It provides an easy-to-use interface for managing, editing, and preprocessing multiple ESQL/C programs. Examining any warnings or errors generated during preprocessing is a simple mouse click operation.

2.4 Syntax Diagrams

Syntax diagrams show the syntax for all SQL commands. These diagrams provide assistance when constructing a statement on the command line. An example syntax diagram is shown in Figure 2-1.

To use the syntax diagram, simply follow the line from start to finish. Any element of the command that cannot be bypassed is required. Any elements that can be bypassed are command options, and provide additional functionality for the command at the user's discretion.

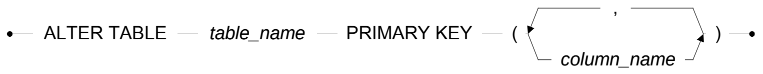


Figure 2-1 Syntax of the ALTER TABLE Statement

Any words that appear in italics are placeholders for the actual names used in a database. The actual names should be substituted for these placeholders. In Figure 2-1, replace the <*table_name*> placeholder with the name of a table in the database. For example, in the tutorial database, replace <*table_name*> with **Customers** to execute this command on the **Customers** table.

Please note the arrow direction. Sometimes it is possible to have a list of items in a command. This is shown in the syntax diagram as a circular arrow path. Both column name fields in Figure 2-1 can include a list of column names, separated by commas, as indicated by the circular path of the arrows.

3 System Architecture

This chapter introduces in detail DBMaker's two architectural models . We will first look at the DBMaker process and the *Database Communication and Control Area (DCCA)*, which store all necessary information for each started database, and then the architecture of both models is explained.

3.1 The DBMaker Process

A DBMaker process handles storage and retrieval of data according to user commands and other database functions. A DBMaker process consists of several layers as shown in Figure 3-1.

Figure 3-1 illustrates the user applications communicating with DBMaker through an Application Programming Interface (API). The API passes user commands (i.e., SQL commands) or function calls to the SQL Engine. The SQL Engine is responsible for analyzing and translating the SQL commands into sequences of function calls that are acceptable to the Database Engine. Next, the SQL Engine passes these calls to the Database Engine, which executes these function calls to store data in tables or retrieve data from tables.

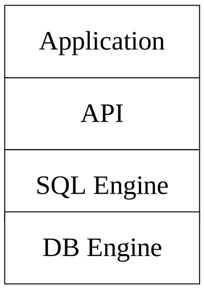


Figure 3-1: A DBMaker Process

The SQL Engine and the Database Engine have different roles. The SQL Engine handles SQL parsing and query optimization. The Database Engine handles space/buffer management, concurrency control, crash recovery, and other related tasks. All modules cooperate to maintain data consistency throughout the entire database. Most performance tuning parameters are related to the Database Engine.

The API and SQL Engines are identical in the single-user and client/server modes. However, the Database Engines in the single-user and client/server modes are different. The single-user mode can handle only one user while the client/server mode can handle multiple users.

In the client/server mode the application and API are tied together and run on client machines while the SQL Engine and the Database Engine are tied together and run on server machines. In this manner, the API can communicate with the SQL engine via network protocols.

3.2 Database Communication and Control Area (DCCA)

When started, DBMaker first allocates a large block of memory for storing database related information such as buffer pools and various types of control information. This memory block is called the Database Communication and Control Area (DCCA). It contains three types of data: *page buffers*, *journal buffers*, and the *System Control Area (SCA)*.

The DCCA is very important to DBMaker's operation, especially when run in client/server mode. The DCCA is allocated from the private heap for Microsoft Windows and UNIX single-user environments. In a UNIX client/server environment, the DCCA must be shared among all DBMaker processes that access the same database, so it cannot be allocated from the private heap. Instead, the UNIX shared memory mechanism is used to allocate the DCCA. All DBMaker processes that run in client/server mode communicate with each other via the DCCA.

The size and usage of the DCCA are easily tuned in DBMaker. This will greatly affect the overall performance of DBMaker. The DCCA is described in more detail in Chapter 18, *Performance Tuning*.

3.3 Architecture of the Single-User Model

The DBMaker single-user mode is a DBMS that supports only one user or application. It is smaller and faster than other modes, in part, because concurrency control is unnecessary. DBMaker's single-user mode is a good choice when one user or application owns a database. Figure 3-2 illustrates the system architecture of DBMaker's single-user mode.

Since only one user or application can simultaneously connect to a single-user DBMaker database, the DCCA is obtained from the private heap and is not sharable. Please note, DBMaker does not support a locking mechanism when in single-user mode. The DBMaker engine increases performance by maintaining all database data in memory while running, and writes the modified pages back to disk files, including data files and journal files, at the proper points in time. The **dmconfig.ini** file text file defines many parameters required for DBMaker configuration.

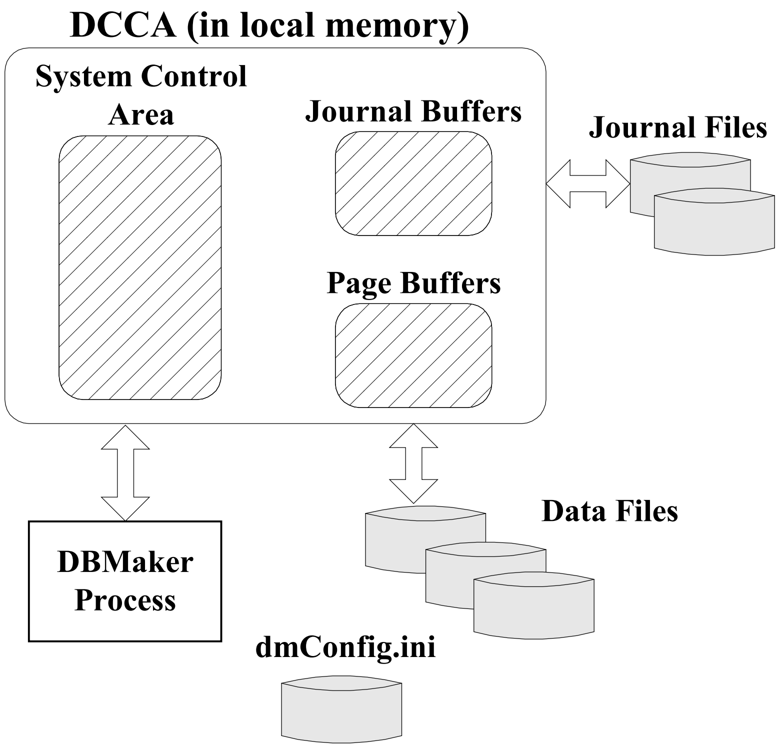


Figure 3-2: System architecture of the DBMaker single-user model

3.4 Architecture of the Client/Server Model

DBMaker also supports a client/server mode. In this mode, two processes comprise the application program: the client application process and the database server process (also called the server process). Typically, the client process resides on a front-end PC or workstation and uses DBMaker's API library routines to communicate with the server process. The server process is located elsewhere as part of a local area network.

Please note, in a client/server configuration, all of the computers involved, including the servers and the clients, can be different platform types.

DBMaker's client/server version includes a network management module. This must be installed for both the client and the server. Network managers are responsible for sending data between the clients and the database servers. The network communications protocol is important in the client/server model. DBMaker currently supports only TCP/IP (Transmission Control Protocol/Internet Protocol). When the client/server version of DBMaker is run on a system that does not normally support TCP/IP, it is necessary to install TCP/IP network software before running DBMaker. If the client application is run on UNIX or Windows, additional TCP/IP software is not required since these operating systems include built-in TCP/IP support. In Windows simply specify TCP/IP as one of the network protocols and install it on the system. Figure 3-3 shows the system architecture of DBMaker in the client/server mode.

On UNIX systems, when a client process connects to a database server, DBMaker's network server forks another server process to handle the subsequent queries. The original network server process continues waiting for connections from other clients. Windows NT is a multithreaded operating system. The NT version of the DBMaker network server (**dmserver.exe**) is a multithreaded program.

Therefore, when a client process connects to a server process running on an NT system, the DBMaker server process creates another thread in its process space to handle the subsequent queries. The DCCA is allocated from local memory, not shared memory. There is always only one DBMaker server process per database in Windows NT. As more operating systems add multithreading support, DBMaker will incorporate multithreading over process forking when possible. Current research indicates that multithreaded programs are more efficient than multi-process programs.

There are three components associated with DBMaker when used in the client/server mode. These components are: the server program, the client program, and the client library.

Server Program

The DBMaker server program is named **DmServer**. This program includes a network manager that handles network communication, and a database engine that handles data access. This program must be started first so that client programs can connect to the database server.

Client Program

The DBMaker SQL client program is named **dmsqlc**. This program is used to connect a client to a database and then issue SQL commands for data processing.

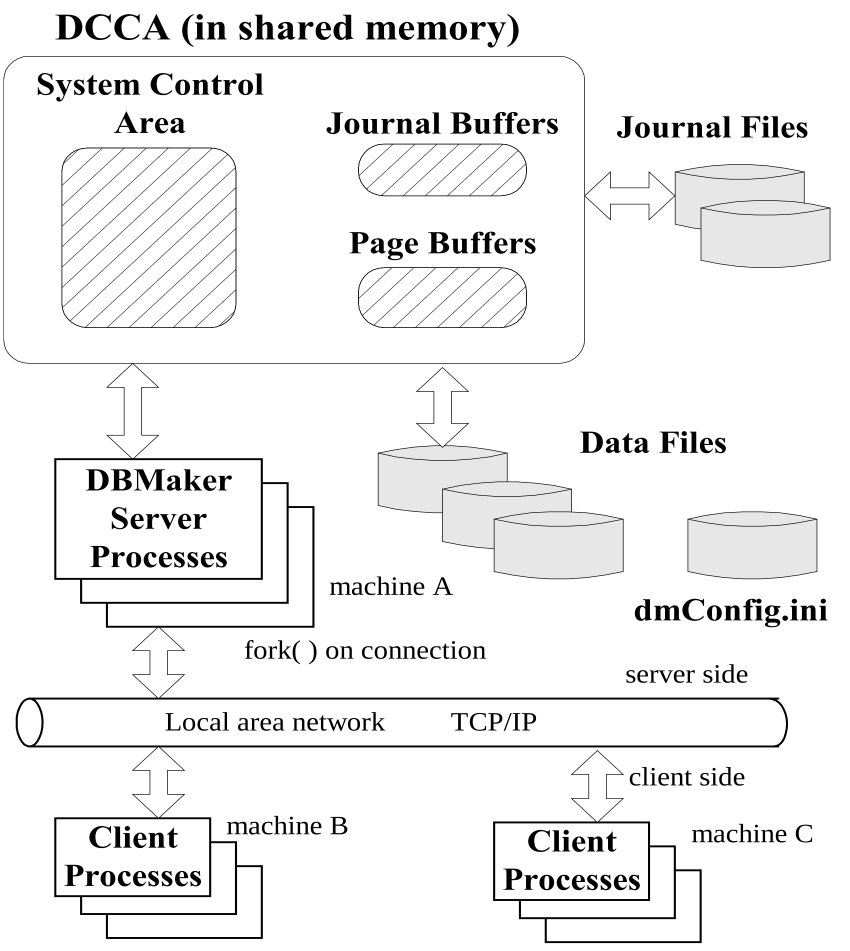


Figure 3-3: System architecture of the DBMaker client/server model

Client Library

The DBMaker client library is named **libdmapic.a** in UNIX, or **dmapi<*version number*>.lib** on Microsoft Windows systems. Users who plan to develop their own client programs must link these with the client library. For example, developers can use various development tools from many vendors to write their front-end applications. When building the applications, they must link those programs with the client library so that their custom applications can communicate with the server program.

4 Basic Database Administration

This chapter describes basic database administration, including creating a database, starting a database, connecting to a database, and shutting down a database. To perform the operations described in this chapter, database administrators can choose to use the command-line based dmSQL tool and edit the **dmconfig.ini** file, or use the JConfiguration Tool and JServer Manager utility.

The following sections describe configuration parameters and commands that are essential for basic database administration. The first section outlines the role and format of the configuration file. Subsequent sections describe the function of specific settings and how those settings affect a database's performance.

4.1 Configuration File - dmconfig.ini

The operation of DBMaker requires many configuration parameters. The DBMaker engine uses configuration parameters to specify how a database runs. File storage locations, runtime memory allocation, and network connections are just a few of the characteristics of a database that are set using configuration parameters. These parameters are stored as configuration variables in the **dmconfig.ini** file. A configuration variable is a keyword that accepts a value (please refer to *Format* later in this section). Users can customize the database by setting parameters in the **dmconfig.ini** file or using the JConfiguration Tool. The JConfiguration Tool's graphical user interface simplifies management of configuration parameters. More information about JConfiguration Tool may be found in the *JConfiguration Tool Reference*. Certain parameters (i.e., keywords) must be set before a database is created while others need only be set before the database is started. In addition, certain configuration parameters should not be changed after database creation or an error will be returned. The following sections describe how to manage settings by directly editing the keywords in the **dmconfig.ini** configuration file. See *Keywords in dmconfig.ini* for a complete reference of **dmconfig.ini** options.

dmconfig.ini Location

DBMaker searches for the **dmconfig.ini** in following three locations, in the order listed, when running on UNIX platforms:

- Current directory
- Directory specified by the DBMAKER environment variable
- DBMaker's installation directory: ***~DBMaker/Version***

If the relevant database section is not found in the **dmconfig.ini** file of one location, DBMaker searches in the next location.

However, for Microsoft Windows systems, the rule is different. DBMaker will only search for the **dmconfig.ini** file in following two locations:

- Directory specified by the DBMAKER environment variable
- DBMaker's installation directory. In a typical Windows installation, this is ***C:\DBMaker\Version***

When DBMaker requires the value of a configuration parameter for a particular database, it scans the above three directories (or the **installation** directory on Microsoft Windows systems) to find a **dmconfig.ini** which contains a section having the same section name as the database. Use any text editor to edit this file and add or modify the parameter values in **dmconfig.ini** so that DBMaker will use them when it is running.

When a corresponding section cannot be found in any **dmconfig.ini** files, DBMaker creates a new section for the database, using default values, in the first **dmconfig.ini** file found or in a new **dmconfig.ini** file in the local directory (or the **installation** directory on Microsoft Windows systems).

When a database is started DBMaker will return an error unless the corresponding section in **dmconfig.ini** is found. Although various sections may be put in different **dmconfig.ini** files and different **dmconfig.ini** files may be put in different directories, this is not recommended. A single global **dmconfig.ini** file will make maintenance easier.

JConfiguration Tool displays all database sections listed in the **dmconfig.ini** file. On UNIX systems, the JConfiguration tool displays all sections of all **dmconfig.ini** files shown in the locations listed earlier.

dmconfig.ini Format

The **dmconfig.ini** file is divided into sections. The first section lists the definitions of the most commonly used keywords. Subsequent sections begin with a header name corresponding to the name of a database. The keywords under each section define the configuration of that database. Any string following a semi-colon is considered a comment.

➲ **Example**

Generalized format of a **dmconfig.ini** file:

```
[Section_name1]
<key_word1> = <value1>
<key_word2> = <value2> <value3>    ; this is a comment;
...

[Section_name2]
<key_word3> = <value4> <value5>
<key_word4> = <value6>
...
```

FILE NAME AND SIZE

A database consists of operating system files, These files are defined in the **dmconfig.ini** file using keywords. The <*filename*> parameter is used in place of the

<*value*> parameter. The ***<filename>*** parameter can be a simple file name like **firstdb.sdb**, a relative path like ***mydb/firstdb.sdb***, or a full path like ***/disk1/mydb/firstdb.sdb*** ("/" for UNIX and "\" for Microsoft Windows).

The <*np*> parameter represents a number of pages. The default page size is 8 KB unless otherwise specified by the **DB_PgSiz** keyword.

In addition to using a number of pages, user can specify M (megabytes) or G (gigabytes) as the unit. If M or G is not used, the unit is page. When M or G are used, the actual size is one page less than the specified value. For example, if the page size is 16 KB and the file size is set to 8 MB, the size will be 8,176 KB rather than 8,192 KB.

➲ **Example**

Generalized format for indicating file names and sizes:

```
[Section_name1]
<key_word1> = <filename>
<key_word2> = <filename> <filename>
<key_word1> = <np>
```

File Locations

A database can be accessed by users who are running DBMaker from different directories. As a result, the current directory is different for each user. In this case, all of the file names in **dmconfig.ini** should be full paths.

Alternatively, the **DB_DbDir** configuration parameter can be utilized. This keyword indicates the home directory (i.e., database directory) of a database.

➲ **Example 1**

The following sets the name of the database directory to db, instead of the default DB1 as indicated by the section header. Furthermore, other database files are placed in alternative locations and on other disks.

```
[DB1]
DB_DbDir = /disk1/db
DB_DbFil = mydb1
DB_JnFil = /disk2/usr/DB1.JNL
```

The resulting physical file names are:

```
DB_DbFil -- /disk1/db/mydb1
DB_JnFil -- /disk2/usr/DB1.JNL
DB_BbFil -- /disk1/db/DB1.SBB (using default file name)
```

➲ **Example 2**

Using the **DB_DbFil** keyword:

```
[DB2]
DB_DbFil = mydb2
DB_JnFil = /disk2/usr/DB2.JNL
```

The resulting physical file names are:

```
DB_DbFil -- mydb2 (in current directory)
DB_JnFil -- /disk2/usr/DB2.JNL
DB_BbFil -- DB2.SBB (in current directory)
```

NOTE *The rule also applies to user-defined files.*

Some Important dmconfig.ini Keywords

The following list introduces some of the most important keywords. Keywords essential for database creation and startup are given in subsequent sections of this chapter. A complete list of keywords appears in chapter *Keywords in dmconfig.ini*. Examples of valid keywords that can appear in **dmconfig.ini**:

- **DB_DbDir** = <*filename*> — specifies the directory that the database files reside in
- **DB_DbFil** = <*filename*> — file name for the system database file as <*filename*>
- **DB_PgSiz** = <4, 8, 16, 32> — page size (4 KB, 8 KB, 16 KB or 32 KB)
- **DB_JnFil** = <*filename*> — file name for the system journal file as **<***filename***>**
- **DB_JnlSz** = <*np*> — size of the system journal file in <*np*> (number of pages)
- ***<logical_file>*** = **<***filename***>** <*np*> — specifies that the user-defined file with the name <***logical_file***> will be mapped to **<***filename***>** with <*np*> pages. In other words, **<***filename***>** is the physical file name for <*logical_file*>

- **DB_NBufs** = <*np*> — runtime data buffer size in <*np*> (number of pages)
- **DB_SvAdr** = <*IP address*> or <*host name*> — database server's IP address or its host name. In a client/server system, this option must be set on the client side.
- **DB_PtNum** = <*port number*> — TCP/IP port number used to communicate between the database client and database server
- **DB_MaxCo** = <*number*> — maximum number of connections that the database can handle.

NOTE *All pattern matching is case insensitive except for <logical_file>.*

Default Values

Some of the options have default values. Therefore, if a keyword does not appear in **dmconfig.ini**, its default value is used. See *Keywords in dmconfig.ini* for a more detailed description of the keywords and their default values.

Support Environment variables

For Read-Only Database is stored on CD-ROM, it's difficult for user to specify the path in **dmconfig.ini** file. It will be easier for user if DBMaker can support the default environment variable **$APP_HOME** and **$APP_DRIVE**.

- **$APP_HOME**: DBMaker home installation directory. It always gets DBMaker HOME information from register. Such as DBMaker which is installed in the *D:\dbmaker\5.4* directory, DBMaker will automatically replace **$APP_HOME** with *"D:\dbmaker\5.4"* when reading the **dmconfig.ini** file.
- **$APP_DRIVE**: This variable returns an empty string on the Linux operating system, and returns a home installation directory where the drive letter on the Windows operating system. Such as DBMaker installed in the *D:\dbmaker\5.4* directory, DBMaker will return the driver letter *"D:"* and automatically replace "*D:\dbmaker\5.4*"with *"D:"* when reading the **dmconfig.ini** file.

DBMaker also supports the system environment variables, such as ***$TEMP =*** *"C:\TEMP"* which defined in the operating system environment variables. DBMaker will automatically replace the ***$TEMP*** with *"C:\TEMP"* when reading the **dmconfig.ini** file.

If the default environment variable **$APP_HOME** or **$APP_DRIVE** is defined in the system environment variables, DBMaker will not find the defined value of the system environment variables when reading the **dmconfig.ini** file, but priority using the default environment variable value.

➲ **Example**

If user has a CD-ROM, user can put the DBMaker software and database on the CD-ROM with the following setting:

```
dmconfig.ini
[DBSAMPLE5]
DB_DbDir=$APP_DRIVE\database
DB_FoDir=$APP_DRIVE\database\fo
DB_TpFil=$TEMP\DBSAMPLE5.tmp
DB_SMode=6
```

Sample dmconfig.ini file

In the following example, two sections are defined in the **dmconfig.ini** file, one for the **Personnel** database and the other for the **LIBRARY** database.

➲ **Example**

A typical **dmconfig.ini** file:

```
[Personnel]
DB_DbFil = /disk1/bin/PERSONNEL.DB
DB_JnFil = /disk1/bin/PERSONNEL.JNL
f1.os = /disk1/bin/PERSONNEL.OS 100
f1.blob = /disk1/bin/PERSONNEL.BLOB 1000
DB_NBufs = 0              ; auto-configure number of data buffers
DB_NJnlB = 100           ; number of journal buffers
DB_MaxCo = 100            ; maximum number of connections
DB_JnlSz = 2000           ; size of journal file (pages)
DB_RTime = 0             ; restoration target time
DB_SvAdr = 192.72.116.130  ; server's IP address
DB_PtNum = 21000          ; and port number
```

```
[LIBRARY]
DB_DbFil=/disk3/usr/lib/library.db
DB_JnFil=/disk3/usr/lib/library.jnl
DB_SvAdr = 192.72.116.137
DB_PtNum = 26999
DB_JnlSz = 2000
```

4.2 Creating a Database

Creating a new database requires some planning. There are a number of configuration parameters that must be considered before creating a new database, some of these parameters cannot be changed after the database is created. Parameters that must be set during database creation are:

- Database name
- Database security (whether the database has different user authority levels)
- Case sensitivity (determines case sensitivity of certain schema objects within the database)
- BLOB frame size (the amount of disk space allocated for each BLOB frame)
- Language setting (determines the character set to be used- ASCII, Big5, etc.)
- Language code order (the pattern used to sort character type data)

All other configuration parameters can be changed after the database is created, however, it is important to consider other database parameters before database creation. These parameters include:

- Tablespace name, location, size, and extensibility
- Number of journal files
- Journal file name, size, and location
- System data and BLOB files names, sizes, and locations
- Default user data and BLOB files names, sizes, and locations

- System temporary file name and location
- User-defined file names, sizes, and locations
- DBMaker log file locations
- Backup directory location
- Table replication log directory location
- Allow for user file objects
- Enable use of raw devices (for UNIX platform only)
- Enable client/server database
- Database IP address and port number (for client/server databases)
- Default user ID and password
- Memory allocation

DBMaker provides an easy to use wizard for creating databases in the JServer Manager tool. A database administrator can easily create a database by editing the **dmconfig.ini** file and using dmSQL. The following subsections outline the process for creating a database. The subsections also approximately follow the sequence of steps used in the JServer Manager create database wizard.

Naming the Database

Before naming a database, be aware of the following naming rules:

- Database names can contain at most 128 characters
- Database names can contain any alphanumeric characters, including the underscore
- Character may be in any position
- Database names are not case-sensitive
- Database names must be unique among all computers that will connect to the database

Databases may be named from the create database wizard of the JServer Manager, or using dmSQL.

➲ **Example**

To create a database using dmSQL:

```
dmSQL> CREATE DB <database name>;
dmSQL> TERMINATE DB;
dmSQL> QUIT;
```

Schema Object Name Case Sensitivity

The case sensitivity of all identifiers in a database can be specified. Under the case insensitive mode, all identifiers appear in uppercase when defined. Once a database has been created, this setting cannot be changed. Setting the keyword equal to zero makes the database case sensitive. The keyword is set equal to one by default, so a database created without changing this setting will be case insensitive. The following **dmconfig.ini** variable specifies database case sensitivity:

DB_IDCap = <*value*> (default value = *1*)

Setting Storage Parameters

There are ten types of operating system files associated with a database: system data and BLOB files, default user data and BLOB files, system journal files, a system temporary file, user-defined files, DBMaker log files, backup files, and a table replication log file. When a database is first created, the user may assign names and locations for each file, or DBMaker will assign default values to them. It is important before creating a database to have a good understanding of the function these files serve within a database.

Many of the parameters discussed in this section may be modified from the Storage page of the JConfiguration Tool. To learn more about how to use JConfiguration Tool to change database parameters see the *JConfiguration Tool Reference.* More information about managing files is available in section 5.2 *File Types*.

When creating a database, DBMaker creates the system database file, the journal file, and the system BLOB file according to the related settings in the **dmconfig.ini** file. If no **DB_DbFil**, **DB_JnFil**, or **DB_BbFil** settings are defined, the default setting are used.

The default values are:

DB_DbFil -- database name + '.SDB'

DB_JnFil -- database name + '.JNL'

DB_BbFil -- database name + '.SBB'

SPECIFYING THE DATABASE DIRECTORY

The database directory is the default location where files associated with a database are created and stored. If the defined file is specified with a full path name, DBMaker uses that name to reference it. If a file name without a full path is used, DBMaker searches for the database directory. If it is not found, DBMaker uses the file name and assumes it is located in the current directory.

When creating a new database in Windows, DBMaker assigns a default database directory of (DBMaker Installation Directory)/bin. It is necessary to create a new directory for the database files to reside in. Multiple databases must not be created in the same database directory. The following **dmconfig.ini** keyword specifies the database directory:

DB_DbDir = *<pathname>* (default: *<DBMaker installation directory>\bin\<Database Name>)*

➲ **Example**

To set the database directory to */disk1/db*:

```
[DB1]
DB_DbDir = /disk1/db
```

CREATING THE SYSTEM TABLESPACE

DBMaker databases are composed of several logical divisions known as tablespaces. With tablespaces, the database can be divided into manageable areas. In the logical view, a tablespace contains one or more tables and indexes. In the physical view, a tablespace is the physical storage that consists of one or more files. A newly created database has two tablespaces, the system tablespace, and the default user tablespace.

The *system tablespace* consists of a system data file and a system BLOB file. The system tablespace records the system catalog for the entire database. The database administrator may specify the initial location of system data and BLOB files in the system tablespace.

The system tablespace cannot be dropped (i.e., deleted), although other user tablespaces can be. The initial size of the system database file is 200 pages (200 × **DB_PgSiz** KB). The following **dmconfig.ini** keywords define the system tablespace:

System data file: **DB_DbFil** = <*filename*> (default: <*Database Name*>.*SDB*)

System BLOB file: **DB_BbFil** = <*filename*> (default: <*Database Name*>.*SBB*)

The ***<filename>*** parameter can be a simple file name like **firstdb.sdb**, a relative path like ***mydb/firstdb.sdb***, or a full path like ***/disk1/mydb/firstdb.sdb*** ("/" for UNIX and "\" for Microsoft Windows).

➲ **Example**

Entering the following lines into the dmconfig.ini file results in the system tablespace files being stored in the ***/disk1/mydb/*** directory.

```
DB_DbFil = /disk1/mydb/firstdb.sdb
DB_BbFil = /disk1/mydb/firstdb.sbb
```

Creating the User Default Tablespace

The *default user tablespace* initially contains one data file and one BLOB file. User data is stored in these files. The database administrator may specify the initial size and location of data and BLOB files in the user default tablespace. Data file size is specified in pages (a page can be 4 KB, 8 KB, 16 KB or 32 KB). BLOB file size is specified in frames. Frame size can be defined by the user and is discussed in *Specifying the BLOB Frame Size* later in this chapter. By default, the user default tablespace is autoextend. This means that if the tablespace is full of data, DBMaker will enlarge the files (and therefore the tablespace) automatically. However, it is more flexible and efficient to create additional tablespaces to store user tables.

JDBA Tool can help to create new tablespaces and manage existing ones. If data or BLOB files are added to a tablespace without specifying a full path, the file is created in the database directory. The user default tablespace cannot be dropped (i.e., deleted), although other user tablespaces can be. The following **dmconfig.ini** keywords define the default user data and BLOB files:

User data file: **DB_UsrDb** = <*filename*> (default: *<Database Name>.DB*)

User BLOB file: **DB_UsrBb** = <*filename*> (default: *<Database Name>.BB*)

The ***<filename>*** parameter can be a simple file name like **firstdb.sdb**, a relative path like ***mydb/firstdb.sdb***, or a full path like ***/disk1/mydb/firstdb.sdb*** ("/" for UNIX and "\" for Microsoft Windows).

CREATING JOURNAL FILES

Journal files provide a real-time, historical record of all changes made to a database, and the status of each change. Up to eight journal files can be created. Each journal file has a fixed size. When all journal files are filled by active transactions (i.e. transactions are not committed, and their occupied journal blocks cannot be freed), the current transaction is aborted because no space is available; this is called journal full. Make sure that the longest transaction will not use all journal records in all the journal files. If journal files are created without specifying a full path, then they are created in the database directory. Journal files may not be modified after the database is started. To reduce the number of journal files, add more journal files, or change journal file size, restart the database in new journal mode. More information about new journal mode is available in section 4.3, *Starting a Database*. Also, refer to section 5.2, *File Types* for more information on journal files. The following **dmconfig.ini** keywords define journal file names, locations, and sizes:

Journal file name(s): **DB_JnFil** = <*filename*> (default: "*<Database Name>.JNL*")

Journal file size (pages): **DB_JnlSz** = <*size*> where *size* = 100 pages through 8G

➲ **Example**

The following lines in the **dmconfig.ini** file tell DBMaker to create two 500 page journal files on two separate disks in the ***/mydb*** directory

```
DB_JnFil = /disk1/mydb/firstdb1.jnl/disk2/mydb/firstdb2.jnl
DB_JnlSz = 500
```

Creating System Temporary Files

System temporary files are used by DBMaker to store information about the database, such as sorting results, while the database is active. These files are generated when necessary and deleted when the database is shut down. If temporary files are created without specifying a full path, then they are created in the database directory. Up to eight system temporary files may be specified. Each temporary file may hold up to 2 GB. Temporary files may be located on different disks for improved disk I/O performance. Users should reserve enough space on disk for an entire temporary file (maximum of 2 GB for a single file), or errors may result. System temporary files may be specified using JConfiguration Tool or editing **dmconfig.ini** before starting the database. The following **dmconfig.ini** keyword defines system temporary file names and locations:

DB_TpFil = <*filename*>[<*filename*>…] (default: <*Database Name*>.*TMP*)

Specifying the BLOB Frame Size

A BLOB frame is the smallest unit of storage used by BLOB files, which are used to store large object data such as LONG VARCHAR or LONG VARBINARY. BLOB frame size cannot be changed after creating the database. The minimum frame size is 8KB and the maximum frame size is 256 KB. Determining the frame size is a trade-off between disk utilization and performance. If entire BLOBs are frequently retrieved, adjusting the frame size to contain an entire BLOB provides better performance because only one disk access is required. However, there may be large variations in the size of the BLOB data. If the frame size is large enough to contain the largest BLOB, it could waste disk space, as other frames that contain smaller BLOBs will contain unused disk space. Alternatively, if frames are only large enough to contain the smallest BLOBs, performance degrades when fetching larger BLOBS stored in multiple frames. The following **dmconfig.ini** keyword specifies BLOB frame size:

DB_BFrSz = <*nk*>. The **<** *nk* **>** parameter is the frame size in kilobytes. The size of the system BLOB file is (page size + (number of frames – 1) × *nk*). Refer to Chapter 7, *Large Object Management* for more detailed information.

➲ **Example**

To set the BLOB frame size to 10 KB:

```
DB_BFrSz = 10
```

Setting the Number of Pages to Extend an Autoextend Tablespace

When all pages in the data file or BLOB file of an autoextend tablespace are full, DBMaker allows the tablespace to grow by automatically extending the number of pages or frames in the file. This setting tells DBMaker how many pages or frames to add to the full file in the event that it is filled. If the Database Administrator expects that the database will grow very quickly, then a higher number should be selected to lessen the frequency at which the file is appended. This number can be adjusted before starting a database by using JConfiguration Tool, or by editing the **dmconfig.ini** file. The following **dmconfig.ini** keyword specifies the number of pages/frames to extend an autoextend tablespace:

DB_ExtNp = <*np*>, where <*np*> is the number of pages to extend (default: 20 pages / frames)

Enabling User File Objects

FILE type data can be stored as user file objects or system file objects. User file objects are external files that are accessed through the machine where the database resides. In other words, a user file object is only a link to an external file outside the database. Enabling user file objects allows a FILE type column to link to the external files, which will be accessed by the database server. It may be disabled and enabled as needed. Inserted user file objects can be accessed even if the setting is turned off. User file objects may be enabled through the storage page of the JConfiguration Tool before starting the database. The keyword value may be modified before starting the database. Setting the keyword equal to 0 prevents file objects from being inserted. Setting the keyword equal to 1 allows file objects to be inserted. The following **dmconfig.ini** variable toggles file object capability:

DB_UsrFo = <*value*> (default: 0 / disabled)

CREATING A DIRECTORY FOR SYSTEM FILE OBJECTS

System file objects are created, deleted, and managed by DBMaker. All system file objects are placed in subdirectories of the system file object directory. Changing the system file object directory does not change the location where previously inserted system file objects reside. The system file object name and location may be set from the storage page of the JConfiguration Tool before starting the database, or during runtime with JServer Manager or JDBA Tool Run Time settings. The keyword value may be modified before starting the database. The following **dmconfig.ini** variable sets location of system file objects:

DB_FoDir = <*pathname*> (default: \<*database directory*>*fo*)

CREATING A DIRECTORY FOR USER-DEFINED FUNCTION DLL FILES

The database administrator may specify the directory where the dynamic link libraries (DLL) of user-defined functions (UDF) are placed. UDFs are compiled functions stored in a dynamic link library (.DLL for Windows operating system, or .so for UNIX operating systems) that the user wants to use in DBMaker. The DLLs stored in the Directory of User-defined Function DLL files are accessible to DBMaker and can be used in SQL statements or ODBC applications. UDFs should be loaded when the database starts. The following keyword specifies the location of UDF DLL files:

DB_LbDir = <*filename*> (default: current working directory)

Turning On the Log System

DBMaker provides a log system to record useful information including connections, users, execution times and SQL commands. The system can be used to record additional database information and is very useful for resolving errors generated by the run time environment.

The log format will be text *CSV* format, so user can use excel viewer to check it after rename the ***.log*** to ***.csv*** file. The following table lists all of the columns in log file, and a brief description of what is contained in each column.

COLUMN NAME	**DESCRIPTION**
LOG_TIME	The time for writing log
BEG_TIME	The command starting time
STATE	There are four states: _, O, X, S, according to " _, O, X, S" to judge it's unknown, ok, error, or slow, the check sequence is check rc first, and then check execution time.
RETCODE	Returned code: 0 or error code
EXE_TIME	Execution time
SV_FUNC	Execute which server function at present
CONNECT_ID	Connection ID
USERNAME	User's name
LOGIN_TIME	Login time
LOGIN_ADDR	Login IP address
STMT_ID	Statement ID
NUM_STMT	Number of statements
ERROR_ARG	Error argument
OTHER_INFO	If turn on other LGXXX setting (ex: LgPln), these information will be recorded in LOGNAME.TXT, user can mark [INFO_XXX] to .TXT to check
SQL_CMD	The most recently executed SQL command

The Log System can be activated by setting the keyword **DB_LgSvr** in the **dmconfig.ini** file before starting the database or by calling the stored procedure SETSYSTEMOPTION() at run time.

Log information is divided by level. What operations are logged and when can be specified individual for each level. When the log is on, DBMaker records the server's operation according to the logging options and stores the log in the directory which specified by **DB_LgDir**. DBMaker assigns the log name according to DBNAME and log index number. Since server log can include current date in the log filename, so the log filename would be unique and won't be overwritten. User can specify the number of days who wants to keep the log files available. The expired log file would be removed by the daemon service. This setting specified by **DB_LgDay** keyword in the **dmconfig.ini** file. But the number of log files might grow, packing/zipping the earlier closed log files would be necessary in order to save some storage. This setting corresponds to the **DB_LgZip** keyword in the dmconfig.ini file. Some system information is logged to DBNAME.LOG and the log information is logged to DBNAME_currentdate_1.LOG when the log is initially started. When the log's file size reaches the default 100 MB or the size specified by **DB_LgFSz**, subsequent information is stored in DBNAME_currentdate_2.LOG, Ex: DBNAME_20080706_2.LOG, DBNAME_20080708_3.LOG, …, DBNAME_currentdate _n.LOG; where n is 20 by default unless otherwise specified by **DB_LgDay** and **DB_LgFNo**.

Any additional log information generated when **DB_LgPln**, **DB_LgPar** or **DB_LgLck**, **DB_LgDay**, **DB_LgZip** are started or when DMERROR.LOG contains information is logged to DBNAME_currentdate_n.TXT. Regarding default file size and the rolling log feature, DBNAME_currentdate_n.TXT operates identically to DBNAME_currentdate_n.LOG just described. The additional information is recorded as INFO_connection_id_number in the OTHER_INFO field in the DBNAME_currentdate_n.LOG and is also recorded in the DBNAME_currentdate_n.TXT log file. The connection_id_number can be used to trace the source of the additional log information. Please note, the naming of the DBNAME_currentdate_n.LOG and DBNAME_currentdate_n.TXT are kept in lock-step so information is always logged to the respective log file having equal n values. This is handled by summing the file size of DBNAME_currentdate_n.LOG and DBNAME_currentdate_n.TXT. When this sum reaches the maximum log file size, both logs are considered full and subsequent information is immediately logged to DBNAME_currentdate_n+1.LOG and DBNAME_currentdate_n+1.TXT.Additional system information is logged when **DB_LgSys** is activated.

The following **dmconfig.ini** keywords affect the setting of the log system:

DB_LgDir = <*string*> (default: DB_DbDir\lgdir)

DB_LgSvr = <*value*> (default value = 0/ disabled)

DB_LgErr = <*value*> (default value = 3)

DB_LgSTm = <*value*> (default value = 5 seconds)

DB_LgFSz = <*value*> (default value =100 MB)

DB_LgFNo = <*value*> (default value = 20)

DB_LgPln = <*value*> (default value = 0 / disabled)

DB_LgSys = <*value*> (default value = 0)

DB_LgSql = <*value*> (default value = 0 / disabled)

DB_LgPar = <*value*> (default value = 0 / disabled)

DB_LgLck = <*value*> (default value = 0 / disabled)

DB_LgDay=<*value*> (default value = 30)

DB_LgZip= <*value*> (default value = 1 / enabled)

➲ **Example**

To log slow queries taking over 10 seconds having an error code > 10000, and to keep log files for five days and pack closed log files, set the **dmconfig.ini** as follows before starting the database:

```
[DBNAME]
.........
DB_LgSvr=3;
DB_LgErr=2;
DB_LgSTm=10;
DB_LgDay=5;
DB_LgZip=1;
```

Alternatively, the same result is achieved by calling SETSYSTEMOPTION:

```
dmSQL> CALL SETSYSTEMOPTION('LGSVR', '3');
dmSQL> CALL SETSYSTEMOPTION('LGERR', '2');
dmSQL> CALL SETSYSTEMOPTION('LGSTM', '10');
dmSQL> CALL SETSYSTEMOPTION('LGDAY', '5');
dmSQL> CALL SETSYSTEMOPTION('LGZIP', '1');
```

The log system is used on the server side, so the client or network error will not be recorded. Server performance is affected when logging is enabled. This is especially evident when the log level is high. There must be sufficient disk space to store the server log or log information will be lost.

Raw Devices

The DBMaker physical storage system is very flexible. In a UNIX system, DBMaker allows users to create a database with UNIX files only, with raw device files only, or with files from both file systems. In **dmconfig.ini**, if a file name begins with ***/dev/***, that file will be treated as a raw device.

I/O operations on raw devices will be faster than on regular UNIX files, so database administrators are encouraged to use raw devices as database files. To use raw devices as database files, the system manager must create raw devices before creating any databases. Please refer to the UNIX system manual for the procedure to create raw devices.

Multiple files can be put on a raw device without partitioning the raw device. To put multiple files on a raw device you must consider the following constraints:

- Multiple autoextend files cannot be set on a single raw device
- The file size cannot be changed after the initial set up when setting multiple files on a raw device
- The total size of all files in a single raw device is restricted to 8 TB or less
- If an autoextend file is placed on a raw device no other files can be put on the device. Other than the files you set as autoextend, the **DB_DbFil**, **DB_BbFil**, **DB_UsrDb**, **DB_UsrBb**, and **DB_TpFil** files are all autoextend files
- If **DB_DbFil**, **DB_BbFil**, **DB_UsrDb**, **DB_UsrBb**, and **DB_TpFil** are set to a raw device, they can have only one parameter; number of pages. You cannot set an offset to them. Because these files are autoexted files, they can only occupy a

raw device that can not shared with other files, and don't need an offset. For example:

```
DB_DbFil = /dev/sda 500; Creating a file with 500 pages
```

This is valid too, but it will create a file with 30 pages. The parameter 500 is ignored.

```
DB_BbFil = /dev/sdb 30 500;
```

NOTE *Microsoft Windows does not support raw devices.*

➲ **Example 1**

```
[MYDB]
f1 = /dev/sda 0 500
f2 = /dev/sda 500 200
f3 = /dev/sdb 300
```

To create a regular tablespace, **ts_raw**, containing the above raw device files:

```
dmSQL> CREATE TABLESPACE ts_raw DATAFILE f1, f2, f3 TYPE=DATA;
```

Then you will create three files in the raw devices. Suppose you had selected 4 K for the page size, then the first file has the size 500 × 4 K = 2000 K starting at address 0 in */dev/sda*, and the second has the size 200× 4 K = 800 K starting at address 500× 4 K = 2000 K in */dev/sda*. The third has the size 300× 4 K = 1200 K starting at address 0.

➲ **Example 2**

```
[MYDB2]
DB_JnlSz = 1000
DB_JnFil = dev/sda/j1.jnl 1000 /dev/sda/j2.inl 2000
```

Also suppose you had selected 4 K for the page size, then you will create two normal journal files J1.jnl, J2.jnl and two raw device journal files that one starts at the address 4,000 K of the */dev/sda* and the other at the address 8,000 K of the */dev/sda*.

Enabling Client/Server Database

Any database can be started as a single-user database or a multi-user database. Before creating the database, determine what the primary function of the database is and which user mode is more suitable. If the database is to be primarily a multi-user database, configure the database to use an IP address or DNS name that is appropriate for the network that the DBMaker server will be running on. Also, specify the TCP/IP port number that the database server will use. The client side database will also use this information to connect to the database. This setting can be changed any time before starting the database, but we highly recommend setting these parameters before database creation to ensure smooth operation. Clients will be unable to connect to an improperly configured server database. If both settings are disabled, the database will start in single-user mode. These parameters can be altered from the connections page of the JConfiguration Tool, or by editing the following **dmconfig.ini** keywords:

IP address/Server name: **DB_SvAdr** = <*IP_address*> or <*host name*> (default: local host name or 127.0.0.1)

Port number: **DB_PtNum** = <*port number*> (default: 2,300, 1,024 through 65,535)

Default User and Password

The default user name and password must already exist in the database. These two keywords are not examined when starting a database, but are checked when connecting to a database instead.

➲ **Example**

To specify a default user name and password to use when connecting to a database:

```
DB_UsrId = <user name>
DB_PasWd = <*****>
```

Changing Language Code Order

DBMaker provide various word sort order for data as defined by the keyword **DB_WsorT**. For example, **DB_WsorT** can set the case sensitive of the sort order. The default is binary order sort order.

DBMaker supports different character sets (language codes), such as US-ASCII for English, BIG5 for traditional Chinese, GBK for simplified Chinese, and JIS for Japanese. The keyword **DB_LCode** in **dmconfig.ini** file specifies which character set DBMaker will use. For each character set, there may be several sort orders.

In traditional Chinese for example, the sort order may be according to code sequence, stroke count, or phonetic equivalent. The default sort order for DBMaker is binary sequence. While creating a new database, the user-defined order definition file specified by the keyword **DB_Order** could change the behavior of the sort order. The language code parameter may also be set from the Create Database page of the JConfiguration Tool.

SETTING THE SORTING ORDER

A sort order is a set of rules that specifies how DBMaker presents data in response to database queries and DBMaker statements involving GROUP BY, ORDER BY, and DISTINCT clauses. The sort order also determines how certain queries are resolved, such as those involving WHERE and DISTINCT clauses.

You can specified the **dmconfig.ini** keyword **DB_WsorT** to set the word sort order. DBMaker considers character values that differ in case only as equal when a case-insensitive sort order is specified (e.g., 'John' = 'john'). it is often necessary to obtain query results with case-sensitivity considered when using a case insensitive sort order.

The following **dmconfig.ini** variable specifies the word sorting order case sensitivity:

DB_WsorT = *<value>* (default: 0 / binary sort order)
1 is case-insensitive sort order
2 is case-sensitive sort order

The following example shows how to set the local language and the sort order file before a new database is created.

➲ **Example**

To set the language type to traditional Chinese, use BIG5:

```
[MY_DB]
……..
DB_LCode =1               ; BIG5 for traditional Chinese
DB_Order = big5_stroke.ord ; order definition file
………
```

The keyword **DB_Order** indicates the user-defined order definition file named **big5_stroke.ord**, which should be placed in the ***shared/codeorder*** subdirectory of the DBMaker installation directory. The order definition file is a pure text file, which affects the sorting result in DBMaker. The keyword is used when the database is created and then it is recorded in the database and not used. Without this keyword, while creating the database, the sorting sequence would be in a binary sequence. Once a definition file has been specified, it must always exist or the database will fail to start.

User-defined Order Definition File

The order definition is a user-defined pure text file. The order definition file arranges the sequence of valid characters. An example of the naming scheme looks like *codename_ordertype.ord*, where *codename* is the name of language code and *ordertype* is the type of ordering e.g., *big5_stroke.ord.*

➲ **Example**

An order definition file:

```
Comment: Write information here.

[BEGIN]       // begin to arrange the character sequence

c             // ASCII 0x63
0x62          // Character 'b'
a             // ASCII 0x61

[SINGLE]      // Single-Byte Character Default Order

[DOUBLE]      // Double-Byte Character Default Order

0xA440        // one of Chinese characters
0xA441        // one of Chinese characters
0xA442        // one of Chinese characters
```

All lines before the **[BEGIN]** keyword are regarded as comments. All words after // or /* are also comments. After the **[BEGIN],** each line represents one character and should occupy the first position of the line followed by at least one space or a new line of characters. In the above example the character **c** is less than **b** and **b** is less than **a**.

If the text editor cannot be used to edit some characters, represent them with hexadecimal. For example, character **a** can be written as **a** or its code value **0x61**. It is also very useful for invisible characters.

The creator of the sort order may only be interested in some characters and let others be sorted by default, i.e., binary. The keywords **[SINGLE]** and **[DOUBLE]** can be used to represent the single character set and double character set, both of which are not specified in the definition file. If there is not a **[SINGLE]**, the absent single-byte characters will come before all characters in the definition file. If the **[DOUBLE**] is absent, the absent double-byte characters will come after characters in the definition file.

DBMaker ignores errors found in the definition file. For example, if **[BEGIN]** is lost, DBMaker uses the default sorting order for all characters. If the same character appears two or more times, only the first is processed; subsequent characters are ignored. After creating a database, database administrators should check the sort order behavior carefully to ensure it is correct.

In distributed database environments, all databases should use the same sort order definition file. When copying or moving a database to another machine, do not forget to copy any existing sort order definition files.

The Data Communications and Control Area

The *Data Communications and Control Area (DCCA)* is a memory block in which almost all information and data is placed. For multi-user databases, the DCCA is allocated from shared memory and is used to do inter-process communications. When a database starts, it will allocate a DCCA to hold all information about that database. The DCCA can be divided into three parts—page buffers, journal buffers, and the system control area.

There are several keywords in **dmconfig.ini** related to the usage of the DCCA:

- **DB_NBufs** = <*np*> — number of page buffers which DBMaker will use. The default value is 0 (automatically configure).
- **DB_NJnlB** = <*np*> — number of journal buffers which DBMaker will use. The default value is 64
- **DB_ScaSz** = <*np*> — number of pages in the system control area. The default value is 200

- **DB_MaxCo** = <*number*> — maximum number of concurrent transactions that the database can handle. **DB_MaxCo** is also used for formatting the journal file when the database is created or started with a new journal.

The size of the DCCA can be estimated by adding the size of the page buffers, the journal buffers, and the system control area. When the specified size of the DCCA is not large enough, DBMaker will automatically allocate the minimum necessary space to hold the information required to the DCCA instead of the default size used in the calculation above.

The size of the DCCA cannot exceed the allowable shared memory size of the system in a multi-user environment in UNIX, because in such a case the DCCA is allocated from shared memory. Users can refer to their UNIX manuals for instructions on how to increase the size of shared memory, which generally requires a rebuild of the kernel. DBMaker will run more smoothly with more buffers and a larger system control area.

The relationship between the DCCA, page buffers, journal buffers, and the system control area is explained in more detail in Chapter 18, *Performance Tuning*.

DCCA parameters may also be set from the Cache and Control page of the JConfiguration Tool. For details, refer to the *JConfiguration Tool Reference*.

4.3 Starting a Database

The purpose of starting a database is to allocate the required resources from the operating system, initialize them, and wait for users to connect. The settings of certain configuration parameters must be considered before starting a database. These parameters include:

- Database startup mode
- Enable client/server database
- Database IP address and port number (for client/server databases)
- Default user ID and password
- Memory allocation

- Method for reporting errors

A database may be started using dmSQL or JServer Manager. For more information on starting a database using dmSQL, refer to the following sections. To find out how to use JServer Manager to start a database, refer to the *JServer Manager User's Guide*.

Single-User

A user must start a single-user database every time they want to connect, and terminate the database when they finish using it.

➲ **Example**

To start a single-user database using dmSQL:

```
dmSQL> START DB <database name> <user name> <password>;
.
< do DML here >
.
dmSQL> TERMINATE DB;
```

NOTE *Only users with DBA privilege can start a database. For information about database privileges, refer to Chapter 8, Security Management. If a database is started in single-user mode, only one user can access the database at a time.*

Client/Server

The DBA must start the client/server database on the server machine so that all clients on remote machines (or on the same machine) can connect to the server database via the network. Two configuration variables must be set on the server before the database is started.

Starting a client/server database is a little more complicated than starting a single-user database. First, we need to know the server machine's IP address. The only ID to distinguish each machine on a network is the IP address. The **dmconfig.ini** keyword **DB_SvAdr** specifies the server's IP address.

The second item is the port number. The server program will bind to a given port number, specified by **DB_PtNum** in **dmconfig.ini**, to wait for connections. All client programs must connect to that port number in order to communicate with the database server.

➲ **Example 1**

To specify the Server IP address and the Server and Client port numbers:

```
DB_SvAdr = <server IP address> (on client side)
DB_PtNum = <port number> (on both server and client sides)
```

➲ **Example 2**

To start a client/server database on the server machine using DmServer:

```
UNIX> dmserver <database name>
```

➲ **Example 3**

Enter the user name and password. DmServer will start the database and wait for clients to connect:

```
UNIX> dmserver [-f] [-t port_number] [-u username [-p password]]
      database_name
```

Description of UNIX switches:

- **f** — run server program in foreground mode. DmServer normally runs in background mode.
- **t** — use this port rather than the port defined by **dmconfig.ini**.
- **u** — login user name
- **p** — password for the given login user name

If a username and password are not specified on the command line, DmServer will search for the **DB_UsrId** and **DB_PasWd** in **dmconfig.ini**. If not found, DmServer will prompt users to enter a username and password.

Start Mode

Specify the start mode of a database by using the **DB_SMode** keyword in **dmconfig.ini**. The **DB_SMode** keyword may have six values, corresponding to six start-up modes:

- 1 — Normal start starts up a system normally. If the database crashed in the last session, DBMaker will perform crash recovery automatically to bring the database to a consistent and stable state.
- 2 — New Journal. The database should be set to start in new journal mode if new journal file names and/or locations have been set in the **dmconfig.ini** file. New journal file names and locations may also be specified on the Storage page of the JConfiguration Tool. All old records will be overwritten if the previous journal file names are kept. This setting must be selected if the user wants to change the journal file size, add more journal files, or change the journal file name. We recommend performing a backup before selecting this option.
- 3 — Restore Backup Database uses the backed up database files (including the journal file) to start the database. DBMaker will use the incremental backup files to roll over the operations up to the point in time specified by **DB_RTime**. If no value is specified or the date specified is later than the time of the last incremental backup, **DB_RTime** will revert to its default value. Refer to Chapter 15, *Database Recovery, Backup, and Restoration* for more detailed information on rollover.
- 4 — Source for Target Database is used for database replication. Starting up a system in this mode makes it a primary (source) database. For more information on database replication, refer to Chapter 17, *Data Replication*.
- 5 — Target of Database Replication is used for database replication. Starting up a system in this mode makes it a slave database. For more information on database replication, refer to Chapter 17, *Data Replication*.
- 6 — Database is Read Only starts up a system normally, but the database is read-only or only provides read privilege to all users. Starting a primary database in read-only mode prohibits users from modifying it.

Start mode may also be specified on the Start Database page in JConfiguration Tool or the Start Database Advanced Settings window in JServer Manager.

Forced Startup

When attempting to start a damaged database, it is possible that an error message will always be returned. The only solution is to use "Forced Startup" provided by DBMaker. Set the configuration variable **DB_ForcS** to one and DBMaker will force the database to start up. Refer to Chapter 15, *Database Recovery, Backup and Restoration* for more detailed information.

Email Error Report System

Typically all error messages are stored in the dmerror.log file. Unless the database administrator consistently checks the dmerror.log file, certain database errors may pass unnoticed. DBMaker provides an email error report system to ensure that database administrators are made aware of errors in the system.

The error report system may be activated either by setting two configuration file keywords, with the JConfiguration Tool, or during database startup with the JServer Manager. The keywords that govern the behavior of the e-mail report system are **DB_ERMRv** and **DB_ERMSv**. Use **DB_ERMRv** to specify the recipient addresses for error report email, and use **DB_ERMSv** to set the address of an SMTP server to route email through. For more information on setting the email error report system with JConfiguration Tool or JServer Manager, refer to the *JConfiguration Tool Reference* and the *JServer Manger User's Guide*, respectively.

4.4 Connecting to a Database

This section discusses how to connect to a running client/server database. A user must first connect to a database before performing DML operations. After disconnecting, a client/server database is still active. Users can continue to make connections until the database is shut down.

Certain parameters exist for client/server connections, including port number, server address, connection time-out interval, and lock time-out interval. Connection parameters are set by changing keyword values in the **dmconfig.ini** file or by using the JConfiguration Tool.

A single-user database only allows a single user connection, every time a user is going to use the database they must start it, they do not need to make a connection. See Starting a Database for more information.

Client/Server Database

The **DB_SvAdr** and **DB_PtNum** keywords must be set in the **dmconfig.ini** file. If the **DB_UsrId** and **DB_PasWd** keywords are defined in **dmconfig.ini**, the **<*username*>** and **<*password*>** options in the CONNECT command can be ignored.

➲ **Example**

To connect to and disconnect from a client/server database with dmsqlc:

```
dmSQL> CONNECT TO <database name> <username> <password>;
.
< do DML here >
.
dmSQL> DISCONNECT;
dmSQL> QUIT;
```

Connection Time-Out

In a client/server model database, sometimes a client cannot connect to the server because the server machine is powered off or the IP address of the server machine is wrong. In these cases, users must wait for the connection to be established. To set the connection time-out value, users can set the **DB_CTimO** parameter (in seconds). The default value for this keyword is five seconds.

Lock Time-Out

Locks are required for concurrency control between multiple transactions on the same database objects. For more information on transactions and concurrency control, refer to Chapter 9, *Concurrency Control*. When connecting to a database a lock time-out keyword, **DB_LTimO**, should be defined in the **dmconfig.ini** file to indicate how long (in seconds) a user will wait for a lock that cannot be acquired.

For example, if **DB_LTimO** = 10, DBMaker will return a "lock time-out" error if the user waits for a lock for more than 10 seconds. **DB_LTimO** can be set to zero indicating that user does not want to wait at all. Setting **DB_LTimO** to -1 will turn off this feature. In this case, a user will wait for a lock until the lock is released. Each user can have a **DB_LTimO** value.

Compressing Data

Accessing database content (i.e., data) is the primary cause of network traffic. Compressing the data prior to network delivery reduces the amount that is actually transmitted resulting in a performance increase.

Set keyword **DB_NetZc** before connecting to a database to enable the network compression function. When active, this function compresses data transmitted from the server and decompresses the data when it is received by the client.

➲ **Example**

Set **dmconfig.ini** before connecting to a database to activate network compression:

```
[DBNAME]
DB_NetZc = 1;
```

4.5 Shutting Down a Database

A database should be shut down after all operations are finished. DBMaker will free all resources, such as the DCCA, for the operating system. If there are still active transactions in the database engine, DBMaker will abort them.

However, if there are still active connections to the database engine, DBMaker will shut down the database without killing the processes for those connections. In this case, the database administrator should manually kill the processes; otherwise, the error message "Cannot lock file transaction rollback" will occur when starting the database the next time.

Therefore, database administrators (DBA users) should ensure that all users are logged off before shutting down the database. To shut down a database, a DBA has to connect first and then issue the proper command. Only a DBA has the privilege to shut down a database.

➲ **Example**

To shut down single-user or client/server databases, use dmSQL:

```
dmSQL> CONNECT TO <database name> <DBA username> <password>;
dmSQL> TERMINATE DB;
dmSQL> QUIT;
```

5 Storage Architecture

This chapter introduces the storage architecture of DBMaker. The storage architecture of DBMaker includes the *logical level* and the *physical level*.

The logical level is the view that is presented to users, and organizes data in the database in a way which is easy to understand. The physical level consists of operating system files which correspond to information in the tablespaces, but which are managed by DBMaker and hidden from the user.

This chapter also explains how to control the storage allocation of a database by using tablespaces and files.

5.1 Architecture

A DBMaker database is composed of one or more logical divisions known as *tablespaces*. Tablespaces are the primary logical storage structure in DBMaker. In the logical view, a tablespace contains one or more tables and indexes as shown in Figure 5-1. In the physical view, a tablespace is the logical storage that consists of one or more operating system files as shown in Figure 5-2.

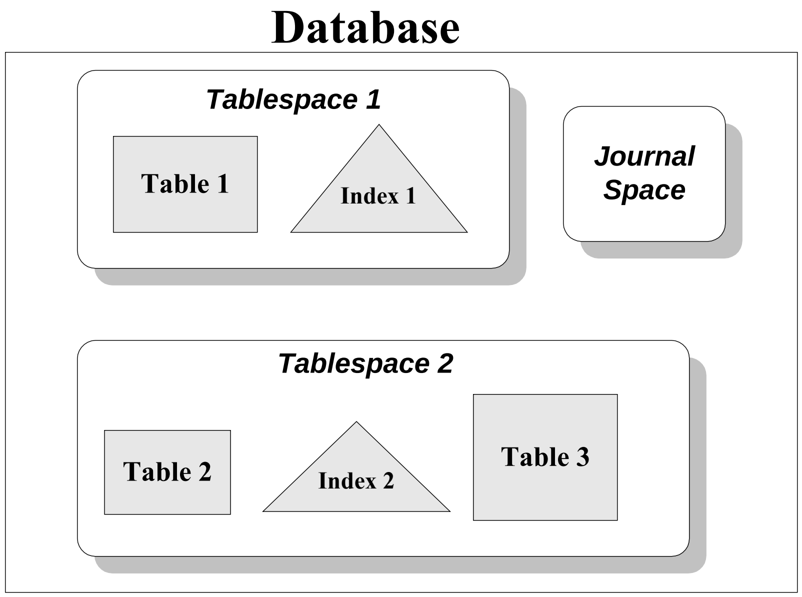


Figure 5-1: DBMaker database storage components in the logical view

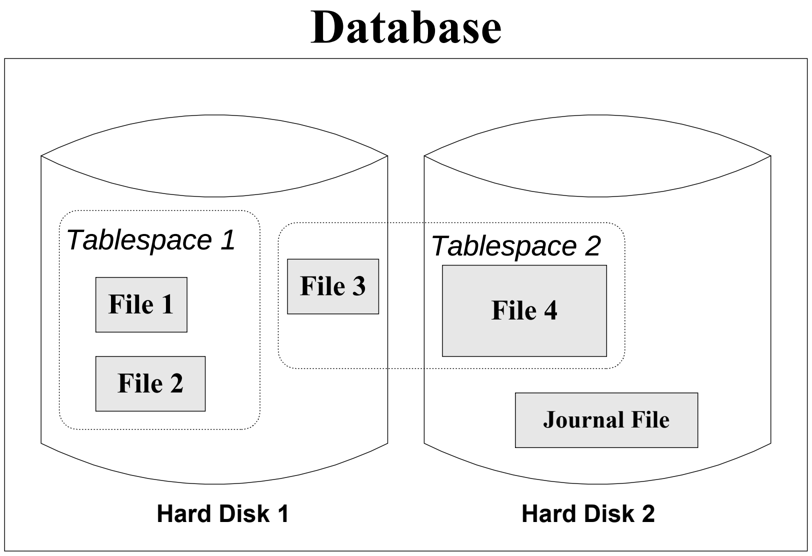


Figure 5-2: DBMaker database storage components in the physical view

5.2 File Types

Ten different operating system file types are used in DBMaker to store different aspects of a database: system data and system BLOB files, user data and user BLOB files, system journal files, a system temporary file, user-defined files, DBMaker log files, backup files, and a table replication log file. The system data file, the system BLOB file, user data files, and user BLOB files are of primary concern regarding database storage architecture and tablespaces. Journal files play an important role in storing records of transactions performed on the database, and are vital to database backup and recovery.

To increase database performance, DBMaker places data into two different types of files—data files and *Binary Large Object (BLOB)* files. BLOB data consists of large data objects in the form of image, voice, or large text, which cannot be packed into a page. DBMaker stores the BLOB data in BLOB files and stores the data rows and index keys in the data files. In order to achieve high performance, DBMaker manages these two file types in different ways.

User Data Files

Data files are comprised of pages, while BLOB files are comprised of frames. The maximum size of both data and BLOB files is 8 TB. However, there are two major differences between frames and pages:

- The size of a page can be 4 KB, 8 KB, 16 KB or 32 KB as defined by the **dmconfig.ini** keyword **DB_PgSiz** when creating a database
- A page can contain more than one tuple, but a frame only contains a single BLOB data item

A data page is the smallest unit of storage used by data files. The data page format is similar regardless of whether the data page stores table or index data. A data page contains four sections: the page header, row data, free space, and the row directory.

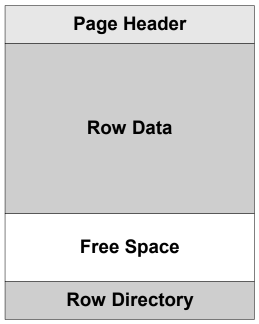


Figure 5-3: Format of a data page

The page header contains general page information for the DBMaker system. The row data area contains the actual table or index data that is displayed as rows and columns when looking in a table or index, and the row directory contains information about the rows in the page. Free space is the available space on that page that has not yet been used to store data.

User BLOB Files

A BLOB frame is the smallest unit of storage used by BLOB files. The size of the BLOB frame can only be set to a value other than the default before creating a database. The minimum frame size is 8 KB and the maximum frame size is 256 KB. A BLOB frame contains three sections: the frame header, BLOB data, and free space. For more information about BLOB files, refer to Chapter 7, *Large Object Management*.

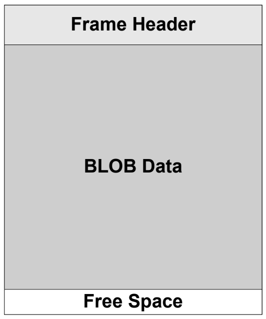


Figure 5-4: Format of a frame

Like the page header, the frame header contains general frame information for the DBMaker system. The BLOB data area contains the BLOB data, and each frame can only contain a single BLOB item. However, BLOB data that is larger than the frame size can be spread over several frames. Free space is the available space on that page that has not been used to store BLOB data.

Journal Files

DBMaker's journal is composed of one or several physical journal files. Internally, a journal file is composed of blocks, where each block is 512 bytes. Every action that causes a change in the database system is recorded by a journal record. Journal records are the logical elements in the journal, and several journal records may be packed into a journal block or a single journal record may span several blocks. A journal record owned by an active transaction cannot be reused.

All journal files form a ring of journal records; journal records are written to sequential journal blocks from the beginning of the file to the end. If the database has been configured to have more than one journal file, DBMaker automatically switches to a new journal file when the current file fills. Otherwise, journal records will be written over journal blocks at the beginning of the journal file. When all journal files are filled by active transactions, the current transaction will be aborted because no blocks are available; this is called *journal full*.

In addition to journal records, a journal file contains some blocks to record the journal status, called journal status blocks. These are used when recovering or restoring the database. Recovery and restoration will be described in later sections.

DBMaker maintains journal block buffers in memory to speed up file access. Before the actual modified data is written to disk, the journal record is written to disk using the *Write-Ahead-Log (WAL)* protocol. When the journal buffer is full or a transaction is committed, the buffer will be flushed to the journal files in accordance with the WAL protocol.

JOURNAL PARAMETERS IN DMCONFIG.INI

Several journal file parameters can be set to enhance database performance.

- **DB_JnFil** — Specifies the names of journal files. One to eight journal file names can be specified. A comma or a space separates every journal file name.

➲ **Example**

The database will have seven journal files specified on different drives to enhance performance:

```
DB_JnFil = myDb.jn1, myDb.jn2, myDb.jn3, /disk1/usr/myDb.jn4, myDb.jn5,
/disk2/usr/myDb.jn6, myDb.jn7
```

- **DB_JnlSz** — Specifies the size of a journal file. The unit is M, G and page, and the default unit is page (journal page size is set by **DB_PgSiz**). The journal file size is:

(**DB_JnlSz** * **DB_PgSiz**)KB

Decide on a reasonable size for journal files when creating a database. As the previous section stated, when all journal files are filled, the current transaction might be aborted because of a full journal. Therefore, a small journal file size may cause a long transaction to be aborted by the system. If database operations involve long transactions, choose a larger journal file size or more journal files.

- **DB_NJnlB** — Specifies the size of a journal buffer as a multiple of journal pages (journal page size is set by **DB_PgSiz**).

RESIZING JOURNAL SPACE

If journal full messages are frequently encountered when a database is running, enlarging the journal files will improve database performance. In DBMaker 3.0, previous backups cannot be used to restore a database to a specific point in time after re-sizing the journal files, however, in versions after 3.0 this is permitted. To protect a database from disk failure, perform a full backup immediately after resizing the journal files.

➲ **To resize a journal file, a DBA performs the following:**

1. Determine the number and size of journal files required by estimating disk space required to handle the largest transactions
2. Shut down the database
3. Update **dmconfig.ini** and re-specify these two parameters: **DB_JnFil**, **DB_JnlSz**

   NOTE *These settings may also be changed in the advanced settings – storage page of the JServer Manager start database wizard.*
4. Set the start mode to new journal mode in **dmconfig.ini**: **DB_SMode = 2**

   NOTE *This setting may also be changed in the advanced settings – start database page of the JServer Manager start database wizard.*
5. Restart the database
6. Reset the start mode back to normal in **dmconfig.ini**: **DB_SMode = 1**

   NOTE *This setting may also be changed in the start database page of the JConfiguration Tool.*
7. Perform an online full backup if a database is in *BACKUP-DATA* or *BACKUP-DATA-AND-BLOB* mode.

Tablespaces

A DBMaker database is partitioned into smaller logical areas of space known as *tablespaces*. Tablespaces are logical areas of storage that allow the database to be subdivided into manageable areas. Each tablespace contains one or more operating system files. Before starting to use tablespaces and files in DBMaker, be familiar with the terms below.

TABLESPACE TYPES

Tablespaces can be either fixed in size or automatically extensible. Tablespaces that are fixed in size are called *regular tablespaces*, and tablespaces that can have their size automatically extended are called *autoextend tablespaces*. DBMaker also has a special tablespace called the *system tablespace*.

THE SYSTEM TABLESPACE

All DBMaker databases have at least two tablespaces, one system tablespace (SYSTABLESPACE), and one default tablespace (DEFTABLESPACE). DBMaker generates a system tablespace to record the *system catalog table* whenever a database is created. The system catalog tables store information about the entire database.

THE DEFAULT TABLESPACE

The default tablespace stores user tables when users do not specify which tablespace to be allocated. However, creating additional tablespaces for user table storage is more flexible and efficient.

THE TEMPORARY TABLESPACE

The temporary tablespace（TMPTABLESPACE）is only used to store external temp tables(ETT). The temporary tablespace also is an auto-extend tablespace. It have exactly two types of files : data files and BLOB files. Data files' logical name is **DB_TMPDB**, and the physical name is **DB_TMPDir**/DBNAME.TDB; BLOB files' logical name is **DB_TMPBB**, and the physical name is **DB_TMPDir**/DBNAME.TBB.

When users call "create temporary table" or "select into" statement, ETTs will be generated and stored into "TMPTABLESPACE ". Users can create temp tables in TMPTABLESPACE(of course system will default store ETT in TMPTABLESPACE), but users can't create any permanent table in TMPTABLESPACE. Users can do "ALTER TABLESPACE TMPTABLESPACE

SET AUTOEXTEND OFF/ON" and "ALTER DATAFILE *DB_TMPDB/ DB_TMPBB* ADD *n* PAGES;", but users can not add files to TMPTABLESPACE or drop files from TMPTABLESPACE. TMPTABLESPACE will be created when a database is created, and the size will be reset to default size when the database is started up.

- Users can't create temporary tables in any other tablespace which is not TMPTABLESPACE.
- Users can't create any permanent tables in TMPTABLESPACE.
- Users can't add files to TMPTABLESPACE and drop files from TMPTABLESPACE.
- Users can't drop TMPTABLESPACE.

Regular Tablespaces

A regular tablespace has a fixed size and contains one or more data files. If a file in a regular tablespace is too small to hold all of the data intended for it, it can be enlarged manually. The maximum number of files that can be contained in a regular tablespace is 32,767. The total number of pages in all files in a tablespace must not exceed 8 TB.

Autoextend Tablespaces

Autoextend tablespaces automatically grow as required. Files in an autoextend tablespace will expand automatically; DBMaker expands them by the reverse order of insertion. That means the last data file added will be the first one to expand if normal data space is required.

Any autoextend tablespace can be changed to a regular tablespace to keep the tablespace from expanding, and vice versa, a regular tablespace can be changed to an autoextend tablespace if the space is exhausted. Alternatively, new files can be added or existing files enlarged to expand a regular tablespace. Raw device files can only be used with regular tablespaces, and cannot be used with autoextend tablespaces.

DBMaker automatically creates an autoextend tablespace called the system tablespace when creating a database. When creating any other tablespaces, regular tablespaces will be used by default. To prevent the default tablespace from growing without a limit, change it to a regular tablespace.

The **dmconfig.ini** file registers the number of pages for each data file. The number of pages in a data file is the initial size of a file belonging to an autoextend tablespace, and is the actual size of a file belonging to a regular tablespace.

5.3 Managing Tablespaces and Files

There are numerous things to consider when managing tablespaces and files for a database. For example, the size and type of new tablespaces must be determined at the time of database creation, additional tablespaces can later be created, autoextend tablespaces changed to regular tablespaces and vice-versa, data files added to tablespaces, the size of files in tablespaces set and altered, data files and tablespaces dropped when they are no longer required, and tables can be altered to other tablespace.

Either the JDBA Tool or a combination of dmSQL commands and modifications to the **dmconfig.ini** file can be used to manage tablespaces. The JDBA Tool provides an intuitive user interface for all tablespace management routines. For more information on how to use the JDBA Tool to manage tablespaces, refer to the *JDBA Tool User's Guide*.

Each DBMaker database has at least one tablespace called the system tablespace. When a database is created, DBMaker generates five files: a system data file, a user data file, a system BLOB file, a user BLOB file, and a journal file. The system data file, system BLOB file, and journal file are placed in the system tablespace. These three files record the system catalog tables for the entire database. The user data file and user BLOB file are placed in the default user tablespace.

User tables are stored in the default user tablespace unless additional tablespaces are created. Creating additional tablespaces to store user tables is more flexible and efficient.

Initial Setting of System Files and Tablespace

DBMaker generates the system tablespace and the three system files (the system data file, the system BLOB file, and the journal file) when creating a new database. These files are used to keep a record of the database schema and transactions. DBMaker concatenates the database name with the file extensions .SDB, .SBB, and .JNL to name the system data, BLOB, and journal files respectively. If the system data, BLOB, and journal file sizes are not specified, they will be created with default sizes of 200 × **DB_PgSiz** KB, 20 KB and 4,000 KB respectively. To use different names for the system files, specify them in the **dmconfig.ini** file, or through the storage page of the JConfiguration Tool.

➲ **Example**

To specify the names of the system files in the **dmconfig.ini** file:

```
[MY_DB]                                     ;database name
DB_DbDir = /disk1/usr                       ;database directory
DB_DbFil = datafile.sdb                     ;data file
DB_BbFil = blobfile.sbb                     ;BLOB file
DB_JnFil = jrnlfile.jnl                     ;journal file
```

If these values are in the **dmconfig.ini** file at the time the CREATE DB command is committed, then DBMaker will create the three system files as before, but this time it will use the names provided above instead of the default names. In this case, the system data file is named **datafile.sdb**, the system BLOB file is named **blobfile.sbb,** and the journal file is named **jrnlfile.jnl**.

The system tablespace is created as autoextend by default; therefore, size of the system tablespace is just an initial size, not a limitation. To limit the disk space used by the system tablespace, change the system tablespace to a regular tablespace by using the ALTER TABLESPACE command.

Once all of the space in a regular system tablespace is exhausted, the only way to enlarge it are to add files to the regular system tablespace, enlarge the system files by adding pages, or change the tablespace type to autoextend.

Initial Setting of Default User Files and Tablespace

DBMaker generates the default user tablespace and the two files (the user data file, the user BLOB file) when creating a new database. These files are used to store user data. DBMaker concatenates the database name with the file extensions .DB and .BB to name the user data and BLOB files respectively. Unless their size is specified in advance, the user data and user BLOB files will be created with default sizes of 200 × DB_PgSiz KB and 20 KB, respectively. To use different names for the default user files, specify them in the **dmconfig.ini** file, or in the storage page of the JConfiguration Tool.

➲ **Example**

In order to specify the names of the default user files in the **dmconfig.ini** file:

```
[MY_DB]                                      ;database name
DB_UsrDb = /disk1/usr/f1.db  200             ;data file
DB_UsrBb = /disk1/usr/f1.bb  20              ;blob file
```

If a database is created with these values in the **dmconfig.ini** file, then DBMaker will create the two files using the names provided above instead of the default names. In this case, the default data file will be named **f1.db** with a size of 200 pages and the default BLOB file will be named **f1.bb** with a size of 20 frames.

The default tablespace is initially created as an autoextend tablespace, so its initial size is not a limitation.

Creating Tablespaces

Additional tablespaces can be created to contain other data and BLOB files. A tablespace may be created using dmSQL or the JDBA Tool. Details on creating tablespaces with dmSQL can be found in the *SQL Command and Function Reference.* Details on how to create tablespaces with JDBA Tool can be found in the *JDBA Tool User's Guide.*

A tablespace must contain at least one data file, but additional files in the tablespace can be either data files or BLOB files. DBMaker creates a new file as a data file by default; the file type must be specified as BLOB to create a BLOB file.

Before creating a new tablespace, specify the size and filenames of the data files associated with the tablespace in the **dmconfig.ini** file.

➲ **Example 1**

The following entries are required in **dmconfig.ini** to specify three files named **f1**, **f2,** and **f3** with operating system filenames and page sizes:

```
[MY_DB]                          ;database name
f1 = /disk1/usr/f1.dat 1000      ;a data file with 1000 pages
f2 = /disk2/usr/f2.dat 500       ;a data file with 500 pages
f3 = /disk1/usr/f3.blb 1000      ;a blob file with 1000 frames
```

To create a regular tablespace **ts_reg** with two data files and one BLOB file, with the data files placed on different disks:

```
dmSQL> CREATE TABLESPACE ts_reg DATAFILE f1, f2, f3 TYPE=BLOB;
```

➲ **Example 2**

To create an autoextend tablespace with one data file and one BLOB file. The initial size of the data file is 500 pages, and the initial size of the BLOB file is 20 pages. If the data file or BLOB file is filled, it will expand automatically:

```
[MY_DB]                          ;database name
f4 = /usr/f4.dat 500             ;a data file with initial 500 pages
f5 = /usr/f5.blb 20              ;a blob file with initial 20 pages
```

To create a new tablespace that uses these files:

```
dmSQL> CREATE AUTOEXTEND TABLESPACE ts_aut DATAFILE f4 TYPE=DATA, f5 TYPE=BLOB;
```

RAW DEVICE FILES

On UNIX systems, if the prefix of the physical file name is **/dev/**, DBMaker will regard it as a *raw device file*. A raw device file supports faster access than a normal file. Thus, raw device files will improve database performance. Create a raw device file on a disk before associating this file with a tablespace. Only regular tablespaces may contain raw device files.

➲ **Example**

To specify a raw device file, **f2** with the operating system filename **/dev/rawf2** with 5000 pages**,** add the following to **dmconfig.ini**:

```
[MY_DB]                          ;database name
f2 = /dev/rawf2  5000            ;a raw device file with 5000 pages
```

To create a regular tablespace **ts_raw,** containing the above raw device file:

```
dmSQL> CREATE TABLESPACE ts_raw DATAFILE f2;
```

Expanding a Regular Tablespace

There are three ways to expand a regular tablespace:

- Add new files to a regular tablespace
- Add pages to existing files in a regular tablespace
- Set autoextend to ON

All of these functions may be performed with the JDBA Tool or a combination of SQL commands and modifications to the **dmconfig.ini** file. The following example shows how to expand a regular tablespace by editing the **dmconfig.ini** file and using SQL commands.

➲ **Example**

Before issuing a command, give DBMaker the name of the physical file that corresponds to the logical file named **file_blob** by adding a statement to the **dmconfig.ini** file in the section for that database. In this case, **file_blob** is the logical name that will be used in the database, and **file.blb** is the physical file name that is used by the operating system:

```
file_blob = file.blb 120
```

To add a new BLOB file named **file_blob** to a regular tablespace with 120 frames to the tablespace named **ts_app**:

```
dmSQL> ALTER TABLESPACE ts_app ADD DATAFILE file_blob TYPE = BLOB;
```

To add 100 pages to an existing data file named **file_data** in a regular tablespace named **ts_app**:

```
dmSQL> ALTER DATAFILE file_data ADD 100 PAGES;
```

After altering the size of the file by adding the extra pages, DBMaker will update the number of pages for the file in the **dmconfig.ini** file to reflect the new value.

Expanding an Autoextend Tablespace evenly

There are three ways to expand an autoextend tablespace:

- Expand from the first file always, it can obtain good performance, but all files in the tablespace are not be expanded evenly.
- Expand from the smallest file always, it can keep all files in a tablespace in even expansion, but performance may be bad because all rows of a table may be scattered into all files in turn.
- Expand from the smallest file firstly, after that, continue to expand this file and does not switch to the second smallest file until the current file's size is bigger than the sum of the second smallest file's size and the value of **DB_ExtHd**. In this way both performance and balance of file size can be taken into account.

For every ways above, DBMaker will go to next file to expand if the selected file can not be expanded because of full disk, file system limitation, or DBMaker's storage limitation. If the next file can not be expanded for the same reason, DBMaker will go to the next file to expand until all files are expanded. Please refer to **DB_ExtHd** for more information.

All of these functions may be performed with the JConfiguration Tool or a combination of SQL commands and modifications to the **dmconfig.ini** file. The following example shows how to expand an autoextend tablespace by editing the **dmconfig.ini** file and using SQL commands.

➲ **Example 1**

Use keyword **DB_ExtNp** and **DB_ExtHd** or *call SETSYSTEMOPTION('EXTHD','newValue')* to define the strategy of expanding an autoextend tablespace. DBMaker will automatically extend the autoextend tablespace according to the strategy users define.

In the configuration file **dmconfig.ini**, specify that the size of DBMaker to expand an autoextend tablespace is **30** pages, the threshold value of expanding a file repeatedly is **100** pages:

```
DB_ExtNp=30
DB_ExtHd=100
```

Add the following five files by modifying the **dmconfig.ini** file:

```
D1=/home/dbmaker/testdb/D1 10
B1=/home/dbmaker/testdb/B1 30
D2=/home/dbmaker/testdb/D2 50
B2=/home/dbmaker/testdb/B2 70
D3=/home/dbmaker/testdb/D3 100
```

Enter the following commands at the dmSQL prompt to create the autoextend tablespace **TS**.

```
dmSQL> CREATE AUTOEXTEND TABLESPACE TS DATAFILE D1 TYPE=DATA, B1 TYPE=BLOB, D2
TYPE=DATA, B2 TYPE=BLOB, D3 TYPE=DATA;
```

Create the table **tb_t1** with column **(c1 char(5000))** in tablespace **TS**, and insert into table **tb_t1** many rows, then the tablespace **TS** will be automatically expanded. According to the new rule, the files will be expanded as follows:

```
1st, extend the smallest file D1, add 30 pages. Now, D1=40, D2=50, D3=100.
2nd, extend D1, add 30 pages. Now, D1=70, D2=50, D3=100.
3rd, extend D1, add 30 pages. Now, D1=100, D2=50, D3=100.
4th, extend D1, add 30 pages. Now, D1=130, D2=50, D3=100.
5th, extend D1, add 30 pages. Now, D1=160, D2=50, D3=100.
6th, extend D2, because D1 > D2+EXTHD. Add 30 pages, D1=160, D2=80, D3=100.
7th, extend D2, until D2 > D3(the smallest)+EXTHD, then extend D3.
```

NOTE *The files B1 and B2 will not be expanded since there is no BLOB field in tb_t1.*

➲ **Example 2**

During runtime, users can call the system stored procedure **SETSYSTEMOPTION** to change the system option **EXTHD**.

```
dmSQL> CALL SETSYSTEMOPTION('EXTHD','1000'); // changing EXTHD to 10000 pages
```

During runtime, users can call the system stored procedure **GETSYSTEMOPTION** to show EXTHD's value.

```
dmSQL> CALL GETSYSTEMOPTION('EXTHD',?);  //reporting the current value of EXTHD
```

Adding Files to Tablespaces

Enlarge the size of a regular tablespace or autoextend tablespace, and consequently the database, by creating and adding new files to it. To increase the space available to insert or update data rows, add data files into a regular tablespace or autoextend tablespace. To increase the space available to store BLOB data, add BLOB files. Files may be added to a tablespace by using the JDBA Tool or by modifying the dmconfig.ini file and entering commands at the dmSQL prompt. The following is a guideline for adding files by modifying the dmconfig.ini file and entering commands at the dmSQL prompt.

Be sure to first add lines to the **dmconfig.ini** file that specify the size and filenames of new files when adding data files to a tablespace. Also, specify the file type as BLOB when adding BLOB files, otherwise DBMaker will create a data file by default.

➲ **Example 1**

To specify in the **dmconfig.ini** file a data file named **f7** with 3,000 pages, where the operating system filename is **/disk1/usr/f7.dat:**

```
[MY_DB]                           ;database name
f7 = /disk1/usr/f7.dat 3000       ;a data file with 3000 pages
```

To add the data file **f7** into the **ts_reg** tablespace:

```
dmSQL> ALTER TABLESPACE ts_reg ADD DATAFILE f7;
```

➲ **Example 2**

To specify in **dmconfig.ini** a BLOB file named **f8** with 5,000 pages; the operating system file name is **/disk1/usr/f8.blb**:

```
[MY_DB]                           ;database name
f8 = /disk1/usr/f8.blb 5000       ;a blob file with 5000 frames
```

To add this BLOB file to tablespace **ts_reg**:

```
dmSQL> ALTER TABLESPACE ts_reg ADD DATAFILE f8 TYPE=BLOB;
```

The file type must be stated or it will be added as a data file by default.

Adding Pages to Files in Tablespaces

In addition to adding files to a regular tablespace to enlarge a database, a database can be enlarged by increasing the size of existing files in a regular tablespace. File size can be increased in autoextend tablespaces by adding pages, which pre-allocates disk space for improved performance. When the size of a file is changed, DBMaker automatically updates the entry for the file in **dmconfig.ini** to reflect the increased number of pages.

File size may be altered using the JDBA Tool or by entering the ALTER DATAFILE command at the dmSQL prompt. The following is a guideline for altering file size by entering commands at the dmSQL prompt.

➲ **Example**

To alter the size and extend file **f1** by adding 100 pages (the file **f1** must already exist and be associated with a tablespace):

```
dmSQL> ALTER DATAFILE f1 ADD 100 PAGES;
```

Changing Regular to Autoextend Tablespaces

A database administrator may want to alter a tablespace from regular to autoextend when:

- Adding more data to a regular tablespace, but the tablespace has already grown to fill all files belonging to this tablespace and the disk still has space
- An unrestricted amount of space a tablespace will occupy is desired

After creating a regular tablespace, the database administrator can change it to an autoextend tablespace by using JDBA Tool or the ALTER TABLESPACE command at the dmSQL command prompt.

➲ **Example**

To change the regular tablespace **ts_reg** to an autoextend tablespace:

```
dmSQL> ALTER TABLESPACE ts_reg SET AUTOEXTEND ON;
```

Changing Autoextend Tablespaces to Regular Tablespaces

A database administrator may want to alter a tablespace from autoextend to regular when:

- Restricting the amount of space a tablespace will occupy is desired. An autoextend tablespace can grow to fill all available space on a disk.

After creating an autoextend tablespace, the database administrator can change it to a regular tablespace by using JDBA Tool or the ALTER TABLESPACE command at the dmSQL command prompt.

➲ **Example**

To change the autoextend tablespace, **ts_reg,** to a regular tablespace:

```
dmSQL> ALTER TABLESPACE ts_reg SET AUTOEXTEND OFF;
```

Shrinking Tablespaces and Files

Tablespaces may be reduced in size if there is a need to allocate disk space for other uses. Two dmSQL commands can be used to reduce tablespace size, the SHRINK DATAFILE command, and the SHRINK TABLESPACE command. The SHRINK DATAFILE command works on a user-specified file, while the SHRINK TABLESPACE command works on all files in the user-specified tablespace. These operations may be carried out by using the JDBA Tool. The following sections outline how to use commands at the dmSQL command prompt to reduce tablespace size.

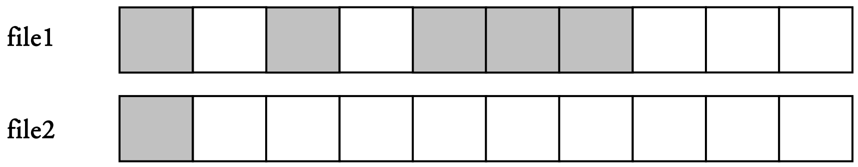
TRUNCATEONLY OPTION

The SHRINK command with the TRUNCATEONLY option removes contiguous free pages at the end of any data file that it is executed on. It does not compress the file; if there are free pages between used pages, they will remain in the file. The database administrator may choose to truncate all tailing free pages (without WITH *n* FREE PAGES option), or truncate free pages while still allowing a given number of free pages to remain (WITH *n* FREE PAGES option). Following are examples of both options.

Without WITH *n* FREE PAGES Option

The SHRINK command with the TRUNCATEONLY option (without WITH *n* FREE PAGES option) only truncates contiguous free pages at the end of a file.

For example, tablespace **ts_shrink** contains **file1** and **file2**. The following diagrams, where gray blocks represent used pages and white blocks represent free pages, represent the page status of file1 and file2.



The free pages at the end of both files may be removed by executing the SHRINK TABLESPACE command on the entire tablespace, or by executing the SHRINK DATAFILE command on both files. The TRUNCATEONLY option must be specified. The following examples demonstrate.

➲ **Example 1**

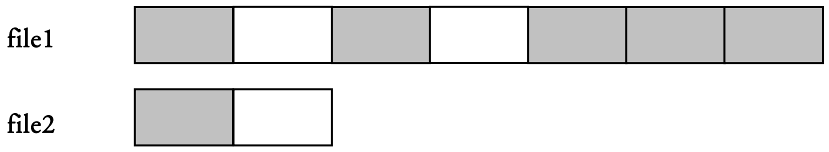
```
dmSQL> SHRINK TABLESPACE ts_shrink TRUNCATEONLY;
```

➲ **Example 2**

```
dmSQL> SHRINK DATAFILE file1 TRUNCATEONLY;
dmSQL> SHRINK DATAFILE file2 TRUNCATEONLY;
```

After truncating, the pages at the end of both files have been removed. A graphical representation of the page status of both files follows:

Result



Although all pages of **file2** are free, DBMaker reserves at least two pages (one is a PE page and one is a data page).

WITH *n* FREE PAGES Option

The WITH *n* FREE PAGES option specifies the total number of tailing free pages (not including the PE page) to remain in the file after it has been truncated.

Using the previous example of **file1** and **file2**, execute one of the following.

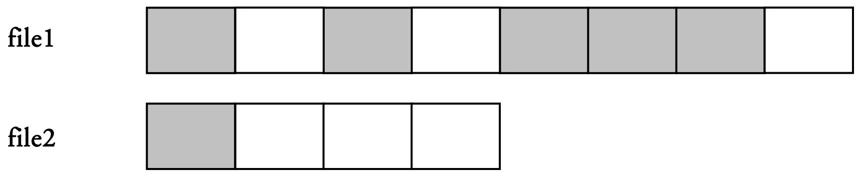
➲ **Example 1**

```
dmSQL> SHRINK TABLESPACE ts_shrink TRUNCATEONLY WITH 3 FREE PAGES;
```

➲ **Example 2**

```
dmSQL> SHRINK DATAFILE file1 TRUNCATEONLY WITH 3 FREE PAGES;
dmSQL> SHRINK DATAFILE file2 TRUNCATEONLY WITH 3 FREE PAGES;
```

Result:



The SHRINK TABLESPACE command and the WITH FREE PAGES option apply individually to each file in a tablespace. In the above case, there are three free pages reserved for each file in the same tablespace.

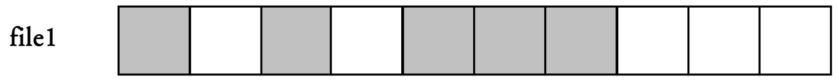
It is not possible to inadvertently add pages to a file by specifying more free tailing pages than the file currently has. For example, if there are 50 free pages in a file, specifying the option WITH 80 FREE PAGES option causes nothing to happen. After the SHRINK command, there are still 50 free pages and it does not enlarge the file size by adding 30 (80 - 50) free pages.

The SHRINK command should be executed with autocommit ON. The TRUNCATEONLY option cannot be rolled back. Users cannot roll back this command, even through crash-recovery.

COMPRESSONLY OPTION

Only the SHRINK TABLESPACE command supports the COMPRESSONLY option. It compresses each file in the tablespace. It does not compress records on the same page because the smallest unit used for compression is a page. It moves the used pages in tail of the file to free front pages. After using the command, all free pages are placed at the end of the file and all used pages at the front.

Result 1:



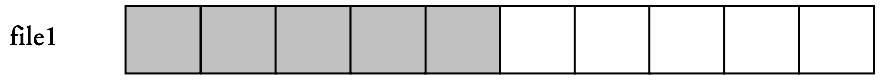
File1 has five used pages that are not adjacent.

➲ **Example**

To make it contiguous:

```
dmSQL> SHRINK TABLESPACE ts_shrink COMPRESSONLY;
```

Result 2:



The SHRINK command must be executed with autocommit ON. The COMPRESSONLY option can be rolled back. If the database crashes, the operation of COMPRESSONLY will be all done or all failure after crash-recovery.

There are some conflicts between the SHRINK command with the COMPRESSONLY option and using backup. DBMaker does not allow these two commands to be executed at the same time.

LIMITATIONS TO SHRINKING AND COMPRESSING TABLESPACES

The general limitations for these commands are:

- The SHRINK command can be used on data and BLOB files but not on a journal file
- Only a user with DBA authority can execute the SHRINK command

- The SHRINK command requires autocommit ON
- The SHRINK command was added in DBMaker 3.7; early versions of DBMaker do not recognize this command. Therefore, once a DBA executes the SHRINK command and then performs an incremental backup, earlier versions of DBMaker cannot restore the journal backup file
- The TRUNCATEONLY option cannot be rolled back
- The COMPRESSONLY option cannot compress the SYSTABLESPACE tablespace
- The COMPRESSONLY option does not check if user tables have an OID column or not. An OID column is used to reference a record elsewhere in the database. After using COMPRESSONLY, an OID column may no longer point to the correct record if the referenced record is in the compressed tablespace or file. It does not modify OID columns in user tables
- The COMPRESSONLY option and backup command cannot be executed at the same time

Dropping Tablespaces

If a tablespace is empty or contains information that is no longer required, a database administrator can drop it from the database. Any tablespace in a DBMaker database, except the system tablespace, can be dropped. To drop a tablespace, first drop all tables in the tablespace or ensure it is already empty of tables. For more information on how to drop tables from a tablespace, refer to Chapter 6, *Managing Schema and Schema Objects*.

Dropping a tablespace will automatically drop all the files associated with it, but will not remove them from the file system of the operating system. Those files will still exist in the file system and can only be removed using operating system commands to recover the disk space they occupy. The data stored in the physical files corresponding to a tablespace is not recoverable once the physical files have been removed from the file system. Be careful when removing files associated with tablespaces or valuable data may be lost.

Tablespaces may be dropped using JDBA Tool or by using the DROP TABLESPACE command at the dmSQL command prompt.

➲ **Example**

To drop the tablespace **ts_aut** and all files associated with it:

```
dmSQL> DROP TABLESPACE ts_aut;
```

Dropping Files From a Tablespace

Users are able to remove unwanted datafiles from a tablespace by using JDBA Tool or by using the ALTER TABLESPACE *tablespace-name* DROP DATAFILE *file-name* command at the dmSQL command prompt. But when use the latter, User need to drop the physical datafiles and remove the information in the dmconfig.ini manually after issuing and committing the ALTER TABLESPACE *tablespace-name* DROP DATAFILE command.

Unwanted datafiles can be removed from a tablespace with the following conditions:

- Users are not able to drop a datafile from a tablespace if it is the only datafile in that table space
- The datafile to removed must be empty
- Users cannot remove the system or default datafile from the system or default tablespace

➲ **Example**

To drop the datafile **f4** from a tablespace **ts_aut**:

```
dmSQL> ALTER TABLESPACE ts_aut DROP DATAFILE f4;
```

Read Only Tablespace

The read-only tablespace is a tablespace that does not allow any updates or creations of new objects in the tablespace.

Read-only tablespaces have many advantages:

- Eliminates the need to perform on going backups. A single backup after making it read-only is sufficient
- Recovery becomes easier
- Read-only tablespaces have less over head than updateable tablespace (no lock)
- Reduces I/O

➲ **Example**

To set tablespace **ts_reg** to read only tablespace:

```
dmSQL> ALTER TABLESPACE ts_reg SET READ ONLY;
```

To set tablespace **ts_reg** to read write tablespace:

```
dmSQL> ALTER TABLESPACE ts_reg SET READ WRITE;
```

Getting Information about Tablespaces and Files

Using JDBA Tool, it is straightforward to view the structure of tablespaces and files within a given tablespace. Tablespaces are displayed as part of the logical tree structure of all database objects. Selecting the tablespaces node on the tree will expand the tree to display all tablespaces in the database. Selecting a tablespace from the tree will display all files in the tablespace as well as details about the files, such as size, physical location, data type, or whether the tablespace is extensible.

Alternatively, use dmSQL to select all columns of the system table SYSTABLESPACE for information on tablespaces, or SYSFILE for information on user BLOB and data files.

➲ **Example 1**

To obtain information on tablespaces, such as tablespace names, whether they are regular or autoextend tablespaces, the number of files associated with tablespaces, and the number of total pages, browse the system table SYSTABLESPACE in the system catalog:

```
dmSQL> SELECT * from SYSTABLESPACE;
```

➲ **Example 2**

To obtain information about files in a similar manner by browsing the system table SYSFILE to get information about file names, file types, database internal file identification, which tablespace files are associated with and how many pages each file contains:

```
dmSQL> SELECT * from SYSFILE;
```

For more information about the system catalog tables SYSTABLESPACE and SYSFILE, refer to *System Catalog Reference*.

Checking File and Tablespace Consistency

DBMaker supports six commands to check the consistency of different parts of a database. These commands are time consuming when the database is large and they will take locks, and should only be used when necessary. File and tablespace consistency may be checked using one of these commands. The CHECK FILE command will check if a file is corrupted or if a tablespace contains the correct tables.

CHECKING FILES

DBMaker allows the contents of every page or frame in a data file to be checked. Any corruption found when checking files is usually caused by disk errors.

➲ **Example**

To check consistency for the **FILE1** data file:

```
dmSQL> CHECK FILE FILE1;
```

CHECKING TABLESPACES

DBMaker allows files and tables associated with a tablespace to be checked. When checking files and tables, DBMaker uses the same methods as the check file and check table commands, and returns the same results as if these commands were executed directly.

➲ **Example**

To check tablespace consistency for the **ts_reg** tablespace:

```
dmSQL> CHECK TABLESPACE ts_reg;
```

6 Managing Schema and Schema Objects

This chapter discusses the management of different types of schema objects in DBMaker, including tables, views, synonyms, indexes, serial numbers, data integrity, and domains.

The chapter includes topics on browsing the system catalogs to get information about schema objects, and how to estimate the disk storage space required for tables and indexes.

Schema object management may be carried out by using dmSQL commands or through the JDBA Tool. The JDBA Tool contains an intuitive graphical interface, provides easy-to-use wizards for most database management tasks, and displays the logical structure of the database in an unambiguous format. Using JDBA Tool will aid first time users of DBMaker in understanding the relationship between schema objects. Experienced users will find the logical display aids in the creation and management of database schema. The following sections show examples of how to manage database schema objects though dmSQL. For more information on using JDBA Tool to manage schema objects, refer to the *JDBA Tool User's Guide.* For more information about how to use the SQL language in DBMaker, refer to the *SQL Command and Function Reference User's Guide*.

6.1 Managing Schema

Schema are namespaces (logical grouping of database objects). Schema contain schema objects such as tables, views, indexes, commands, procedures and a domain and a synonym.

CREATE SCHEMA defines a new schema. After the schema is created, we can create objects within the schema. The schema owner is the grantor for any privileges granted.

The owner of the schema is determined as follows:

- If an AUTHORIZATION clause is specified, the specified user-name is the schema owner. And if schema-name is omitted, the specified user-name is used as the schema name.

For example:

```
dmSQL> CREATE SCHEMA AUTHORIZATION JEFFERY;
```

- If an AUTHORIZATION clause is not specified, the user that issued the CREATE SCHEMA statement is the schema owner.

➲ **Example 1**

As a user with RESOURCE authority, JEFFERY, creates a schema called **SCH_JEF**. JEFFERY is the default owner.

```
dmSQL> CREATE SCHEMA SCH_JEF;
```

➲ **Example 2**

As a user with DBA authority, creates a schema with the user JEFFERY as the owner, and the username JEFFERY is the default schema name.

```
dmSQL> CREATE SCHEMA AUTHORIZATION JEFFERY;
```

➲ **Example 3**

As a user with DBA authority, creates a schema called **SCH_ForJEF** with the user JEFFERY as the owner.

```
dmSQL> CREATE SCHEMA SCH_ForJEF AUTHORIZATION JEFFERY;
```

➲ **Example 4**

A user with DBA authority creates a schema, **inventory**. The user then creates a table and an index on that table. The user final grants authority on the table to the user JEFFERY.

```
dmSQL> CREATE SCHEMA inventory;
dmSQL> CREATE TABLE inventory.part (partNo smallint not null, quantity int);
dmSQL> CREATE INDEX partind ON inventory.part (partNo);
dmSQL> GRANT ALL ON inventory.part TO JEFFERY;
```

DROP SCHEMA removes schemas from the database. A schema can only be dropped by its owner or a DBA. Note that the owner can't drop the schema if it contains any objects.

➲ **Example 5**

Remove schema **SCH_JEF** from the database.

```
dmSQL> DROP SCHEMA SCH_JEF;
```

NOTE *User names and schema names cannot be the same.*

INFORMATION SCHEMA

Every database in DBMaker contains a schema called INFORMATION_SCHEMA. The schema contains a series of views that allow viewing, but not change the description of the objects belonging to the database.

DBMaker provides information schema views for obtaining metadata. These views provide an internal, system table-independent view of the DBMaker metadata. Information schema views allow applications to work properly even though significant changes have been made to the system tables. The information schema views included in DBMaker conform to the SQL-92 Standard definition for the INFORMATION_SCHEMA.

DBMaker supports a three-part naming convention when referring to the current server. The SQL-92 standard also supports a three-part naming convention. However, the names used in both naming conventions are different. These views are defined in a special schema named INFORMATION_SCHEMA, which is contained in each database. Each INFORMATION_SCHEMA view contains metadata for all data objects stored in that particular database. This table describes the relationships between the DBMaker names and the SQL-92-standard names.

DBMAKER NAME	EQUIVALENT SQL-92 NAME
Database	Catalog
Owner	Schema
Object	Object
user-defined data type	Domain

This naming convention mapping applies to these DBMaker SQL-92-compatible views. These views are defined in a special schema named INFORMATION_SCHEMA, which is contained in each database. Each INFORMATION_SCHEMA view contains metadata for all data objects stored in that particular database.

The INFORMATION_SCEHMA views are listed below.

- COLUMN_DOMAIN_USAGE
- COLUMN_PRIVILEGES
- COLUMNS
- DOMAINS
- SCHEMATA
- TABLE_PRIVILEGES
- TABLES
- VIEW_COLUMN_USAGE
- VIEW_TABLE_USAGE
- VIEWS

➲ **Example**

```
dmSQL> SELECT * FROM INFORMATION_SCHEMA.COLUMNS;
```

6.2 Managing Tables

Tables are the logical unit of storage used by DBMaker to store data. A table consists of several columns and rows. A column is sometimes referred to as a field or attribute, and a row can be referred to as a record or tuple.

In DBMaker, each table is identified by a unique schema name and table name.

For example, if two users called **Jeff** and **Kevin** each create a table named **friend** with the default schema name, then the table names **Jeff.friend** and **Kevin.friend** denote the two different tables.

In the JDBA Tool, all tables in a database can be viewed by expanding the tables node on the logical tree. Selecting a table displays that table's schema.

Creating Tables

Every table is defined with a table name and a set of columns. The number of columns in a table can range from 1 to 2000.

Each column has:

- A column name and a data type or a domain, which is described in Section 6.10, *Managing Domains*
- A length (the length might be predetermined by the data type, such as INTEGER), a precision and scale (for columns of the DECIMAL data type only) or a starting number (for columns of SERIAL data type only)

DBMaker supports a large number of data types that can be used to define columns. There are numerical types (SMALLINT, INTEGER, BIGINT, FLOAT, DOUBLE, DECIMAL, REAL, SERIAL and BIGSERIAL), binary types (BINARY, and VARBINARY), character types (CHAR, NCHAR, VARCHAR and NVARCHAR), BLOB types (LONG VARCHAR, LONG VARBINARY, FILE, media type and JSONCOLS), and time types (DATE, TIME and TIMESTAMP). See the *SQL Command and Function Reference* for more information about data types.

When creating a table, provide the table name, column definitions, and the name of the associated tablespace. A table will be placed in the system tablespace by default if it is not associated with another tablespace. Tables may be created using the JDBA Tool Create Table wizard or using the dmSQL command prompt. For information on creating a table with JDBA Tool, refer to the *JDBA Tool User's Guide*. The following is an example of how to create a table using dmSQL. Details on syntax and usage of the SQL command CREATE TABLE can be found in the *SQL Command and Function Reference*.

➲ **Example**

To create the **tb_staff** table in tablespace **ts_reg**:

```
dmSQL> CREATE TABLE tb_staff (nation CHAR(20),
                                  ID INTEGER,
                                name CHAR(30),
                            joinDate DATE,
                              height FLOAT,
                              degree VARCHAR(200),
                             picture LONG VARCHAR) IN ts_reg;
```

DBMaker provides many useful features that can be applied when creating tables:

- Defining a default value for a column
- Specifying that a column is not nullable
- Specifying the primary key or the foreign key for the table
- Specifying the LOCK MODE, FILLFACTOR, or NOCACHE options to improve database efficiency
- Specifying the table as temporary

DEFAULT VALUES FOR COLUMNS

A column in a table can be assigned a default value so that when a new row is inserted and a value for the column is omitted, the default value will be automatically supplied.

Default values for each column in a table may be specified. If a default value is not defined for a column, the default value for the column is set to NULL.

Legal default values can be constants , NULL or built-in functions. For more information about built-in functions, refer to the *SQL Command and Function Reference*.

Now DBMaker supports setting default column attribute for inserting and updating operations with the three keywords: USER, SYSTEM and ON UPDATE. The USER/SYSTEM keywords are optional. These keywords specify whether users can modify value of the column with a default value by using the INSERT/UPDATE statement. USER is used by default. The USER keyword specifies that users can modify its value, and the SYSTEM keyword specifies that users cannot modify its value. The ON UPDATE keyword also is optional. This keyword specifies that value of the column with a default value can be automatically updated when other columns' value is changed. The three keywords are mainly used in a table's definition, and for details of a table's definition, please refer to CREATE TABLE, ALTER TABLE ADD COLUMN and ALTER TABLE MODIFY COLUMN in chapter *SQL Command* of the *SQL Command and Function Reference*.

In addition, when update data, a user can use the connection option SYSTEM DEFAULT to specify whether the value of a column with SYSTEM DEFAULT attribute will be overridden to the default value. If the user sets this option to ON, the value will be updated to the default value; if the user sets this option to OFF, the original value will be updated to the value specified by the users. The default setting for this option is ON. Moreover, if having assigned value to the column by using the INSERT/UPDATE statement, the user can use the connection option LOAD SYSTEM DEFAULT to specify whether the value of a column with SYSTEM DEFAULT attribute will be overridden in the process of loading the tables of database. If the user sets this option to ON, the value will be updated to the default value; if the user sets this option to OFF, the original value will be updated to the value specified by the user. The default setting for this option is OFF.

➲ **Example 1**

To specify the default value of the column **nation,** in the table **tb_staff** as a constant — 'R.O.C.' and the default value of the column **joinDate** as the value of the built-in function **curdate()**:

```
dmSQL> CREATE TABLE tb_staff (nation CHAR(20) DEFAULT 'R.O.C',
                                     ID INTEGER,
                                   name CHAR(30),
                               joinDate DATE DEFAULT CURDATE(),
                                 height FLOAT,
                                 degree VARCHAR(200),
                                picture LONG VARCHAR) IN ts_reg;
```

Example 2a

```
dmSQL> CREATE TABLE computer(id INT, buy_time TIMESTAMP DEFAULT '2012-03-04
12:12:12', price int); //now attributes of buy_time is USER
dmSQL> INSERT INTO computer VALUES(1, '2012-10-10 10:10:20', 3400); //value of
buy_time will be replaced with '2012-10-10 10:10:20' which is specified by the
user
1 rows inserted
dmSQL> INSERT INTO computer VALUES(2, '2012-10-11 10:10:20', 5400);
1 rows inserted
dmSQL> SELECT * FROM computer;
    ID               BUY_TIME              PRICE
=========== =========================== ===========
          1 2012-10-10 10:10:20                3400
          2 2012-10-11 10:10:20                5400
2 rows selected
dmSQL> UPDATE computer SET price=3200 WHERE id=1; //value of buy_time will not be
updated
1 rows updated
dmSQL> SELECT * FROM computer;
    ID               BUY_TIME              PRICE
=========== =========================== ===========
          1 2012-10-10 10:10:20                3200
          2 2012-10-11 10:10:20                5400
2 rows selected
```

Example 2b

```
dmSQL> ALTER TABLE computer MODIFY (buy_time TO buy_time TIMESTAMP DEFAULT '2012-
03-04 12:12:12' ON UPDATE); //now attributes of buy_time is USER and ON UPDATE
dmSQL> UPDATE computer SET price=3000 WHERE id=1; //value of buy_time will be
replaced with the default value'2012-03-04 12:12:12'
1 rows updated
dmSQL> SELECT * FROM computer;
```

```
    ID               BUY_TIME              PRICE
=========== =========================== ===========
          1 2012-03-04 12:12:12                3000
          2 2012-10-11 10:10:20                5400
2 rows selected
dmSQL> UPDATE computer SET price=3000, buy_time='2012-10-10' WHERE id=1;//value
of buy_time will be replaced with '2012-10-10' which is specified by the user
1 rows updated
dmSQL> SELECT * FROM computer;
    ID               BUY_TIME              PRICE
=========== =========================== ===========
          1 2012-10-10 00:00:00                3000
          2 2012-10-11 10:10:20                5400
2 rows selected
```

➲ **Example 2c**

```
dmSQL> ALTER TABLE computer MODIFY (buy_time TO buy_time TIMESTAMP SYSTEM DEFAULT
'2012-03-04 12:12:12'); //now attributes of buy_time is SYSTEM
dmSQL> INSERT INTO computer VALUES(3, '2012-11-10 10:10:20', 4700); //value of
buy_time will not be replaced with '2012-11-10 10:10:20' which is specified by
the user
1 rows inserted
dmSQL> INSERT INTO computer VALUES(4, '2012-12-11 10:10:20', 2800);//value of
buy_time will not be replaced with '2012-12-11 10:10:20' which is specified by
the user
1 rows inserted
dmSQL> SELECT * FROM computer;
    ID               BUY_TIME              PRICE
=========== =========================== ===========
          1 2012-10-10 00:00:00                3000
          2 2012-10-11 10:10:20                5400
          3 2012-03-04 12:12:12                4700
          4 2012-03-04 12:12:12                2800
4 rows selected
dmSQL> UPDATE computer SET price=4500 WHERE id=3; //value of buy_time will not be
updated
1 rows updated
dmSQL> SELECT * FROM computer;
    ID               BUY_TIME              PRICE
```

```
=========== =========================== ===========
          1 2012-10-10 00:00:00                3000
          2 2012-10-11 10:10:20                5400
          3 2012-03-04 12:12:12                4500
          4 2012-03-04 12:12:12                2800
4 rows selected
```

➲ **Example 2d**

```
dmSQL> ALTER TABLE computer MODIFY (buy_time TO buy_time TIMESTAMP SYSTEM DEFAULT
'2012-03-04 12:12:12' ON UPDATE); //now attributes of buy_time is SYSTEM and ON
UPDATE
dmSQL> UPDATE computer SET price=4000, buy_time='2015-01-01' WHERE id=3; //value
of buy_time will be replaced with the default value'2012-03-04 12:12:12'
1 rows updated
dmSQL> SELECT * FROM computer;
    ID                BUY_TIME                PRICE
=========== =========================== ===========
          1 2012-10-10 00:00:00                3000
          2 2012-10-11 10:10:20                5400
          3 2012-03-04 12:12:12                4000
          4 2012-03-04 12:12:12                2800
4 rows selected
```

NOT NULL

Rules for columns or tables may be specified. These rules are called *integrity constraints*. One example is the NOT NULL integrity constraint defined on a column in a table. It enforces the rule that the column cannot contain a null value.

For example, the **tb_staff** table might always need an ID and a name for a new employee.

➲ **Example**

To create an ID and name for new employees on the **tb_staff** table:

```
dmSQL> CREATE TABLE tb_staff (nation CHAR(20) DEFAULT 'R.O.C',
                                  ID INTEGER NOT NULL,
                                name CHAR(30) NOT NULL,
                            joinDate DATE DEFAULT CURDATE(),
                              height FLOAT,
                              degree VARCHAR(200)) IN ts_reg;
```

PRIMARY KEY AND FOREIGN KEYS

The table owner can specify the primary key or foreign key with the CREATE TABLE command. Refer to Section 6.9, *Managing Data Integrity* for information on primary and foreign keys.

LOCK MODE

The *lock mode* of a table identifies the type of lock that DBMaker automatically places on objects when accessing the database. DBMaker supports three lock mode levels: TABLE, PAGE, and ROW. The ROW lock mode is used by default if the lock mode is not specified when a table is created. If the lock mode is set to a higher level (such as TABLE), the level of concurrency on database accesses will be lower, but the required lock resources (shared memory) will also be smaller. If the lock mode is set to a lower level (such as ROW), the level of concurrency on database accesses will be higher, but the required lock resources (shared memory) will be larger. In other words, if a user inserts or modifies rows in a table with the lock mode set to TABLE, no one else will be able to access the table. The reason for this is that an exclusive lock is taken on the entire table. For more information about lock modes, see Section 9.4, *Locks*.

➲ **Example**

To specify the lock mode on a table **tb_staff**:

```
dmSQL> CREATE TABLE tb_staff (nation CHAR(20) DEFAULT 'R.O.C',
                                  ID INTEGER NOT NULL,
                                name CHAR(30) NOT NULL,
                            joinDate DATE DEFAULT CURDATE(),
                              height FLOAT,
                              degree VARCHAR(200)) IN ts_reg
                    LOCK MODE ROW;
```

FILLFACTOR

The FILLFACTOR feature optimizes the utilization of space for data pages by reserving space for the expansion of existing records. It specifies the percentage of a page that can be filled before stopping new records from being inserted. Using this method records can be accessed more efficiently by avoiding the need to retrieve information for one record from multiple pages.

➲ **Example**

To set the FILLFACTOR of the **tb_staff** table to be 80%:

```
dmSQL> CREATE TABLE tb_staff (nation CHAR(20) DEFAULT 'R.O.C',
                                  ID INTEGER NOT NULL,
                                name CHAR(30) NOT NULL,
                            joinDate DATE DEFAULT CURDATE(),
                              height FLOAT,
                              degree VARCHAR(200)) IN ts_reg
                    LOCK MODE ROW
                    FILLFACTOR 80;
```

In this case, new rows cannot be inserted into the data page after the used space is larger than 80%. The legal values for the FILLFACTOR can be from 50% to 100%, and the default value is 100%.

NOCACHE

The NOCACHE feature is useful when accessing large tables with a table scan. Although DBMaker uses page buffers in shared memory to cache retrieved data and avoid frequent disk I/O, table scans on large tables can still cause frequent disk I/O activity. This happens during a table scan on a table with a larger number of data pages than the number of page buffers, which causes all page buffers to be exhausted.

Once the NOCACHE option is specified when creating a table, DBMaker only uses one page buffer to cache the data retrieved from a table during a table scan. This prevents the page buffers from being exhausted by only one large table scan.

➲ **Example**

To specify the NOCACHE option:

```
dmSQL> CREATE TABLE tb_staff (nation CHAR(20) DEFAULT 'R.O.C',
                                  ID INTEGER NOT NULL,
                                name CHAR(30) NOT NULL,
                            joinDate DATE DEFAULT CURDATE(),
                              height FLOAT,
                              degree VARCHAR(200)) IN ts_reg
                    LOCK MODE ROW
                    FILLFACTOR 80
                    NOCACHE;
```

TEMPORARY TABLES

A *temporary table* may be created for storing data. Temporary tables only exist during a single session and can only be used by their creator. DBMaker automatically drops temporary tables when the user that created it disconnects from the database. Temporary tables support fast data operations. Client users may also create a local temporary table using the CREATE LOCAL TEMPORARY TABLE syntax.

➲ **Example 1**

To create a temporary table named **tb_student**:

```
dmSQL> CREATE TEMPORARY TABLE tb_student (name CHAR(25) NOT NULL,
                                       birthday DATE,
                                       score INTEGER);
```

➲ **Example 2**

To create a local temporary table named **tb_student**:

```
dmSQL> CREATE LOCAL TEMPORARY TABLE tb_student (name CHAR(25) NOT NULL,
                                             birthday DATE,
                                             score INTEGER);
```

Browsing Table Schema

The schema of a table may be queried by using dmSQL or the JDBA Tool. JDBA Tool provides a graphical representation of table schema and allows table schema to be modified without entering any SQL commands. It is also possible to use the dmSQL command DEF TABLE to directly query a table's schema.

➲ **Example**

To view the schema for table **tb_staff**:

```
dmSQL> DEF TABLE tb_staff;
CREATE TABLE SYSADM.TB_STAFF (
 NATION  CHAR(20) default 'R.O.C' ,
 ID  INTEGER not null  ,
 NAME  CHAR(30) not null  ,
 JOINDATE  DATE default CURDATE() ,
 HEIGHT  FLOAT DEFAULT NULL ,
 DEGREE  VARCHAR(200) DEFAULT NULL )
 in TS_REG  LOCK MODE ROW  FILLFACTOR 80  NOCACHE;
```

Altering Tables

After a table is created in DBMaker, a user with modify permission can alter it by:

- Adding/dropping columns
- Modifying column definitions
- Changing the FILLFACTOR value
- Turning on/off the NOCACHE option
- Altering tables to Another Tablespaces

A table's schema may be altered using dmSQL commands or the JDBA Tool.

ADDING/DROPPING COLUMNS

A user with modify permission can add/drop one or multiple columns in a table whether the column is empty or not. Adding a new column to an empty table is the same as expanding the table schema and placing the new column in the last position. A user with modify permission can also add to the table a new column before or after any existing column.

When adding a new column to a table, DBMaker not only expands the table schema but also fills all rows in the new column with NULL values by default. If a user with modify permission wants to add a column with the NOT NULL integrity constraint to a table, give a specified value for the existing records on the column (a default value, as described in "Default Values for Columns"). For detailed SQL syntax, refer to the *SQL Command and Function Reference*.

➲ **Example 1**

To add a column named **photo** to the **tb_staff** table:

```
dmSQL> ALTER TABLE tb_staff ADD COLUMN photo LONG VARCHAR;
```

➲ **Example 2**

To add a column named **city** after the existing column name to the **tb_staff** table and set the default value to 'Taipei':

```
dmSQL> ALTER TABLE tb_staff ADD COLUMN city CHAR(20) default 'Taipei' AFTER name;
```

➲ **Example 3**

If the **tb_staff** table is not empty and a user wants to add a non-null column to it, the GIVE keyword can be used to specify a value for the existing records on the newly added column. To add a non-null column named **HireDate** to the **tb_staff** table:

```
dmSQL> ALTER TABLE tb_staff ADD (HireDate date NOT NULL give '2000-02-20');
```

➲ **Example 4**

To drop a column named **photo** from the **tb_staff** table:

```
dmSQL> ALTER TABLE tb_staff DROP COLUMN photo;
```

MODIFYING COLUMN DEFINITION

The definition of every existing column in a table can be altered, such as column name, data type, column order, default value, column constraint, etc. Before modifying the data type of one column, make sure that the new data type is compatible with the original one, or the modifying operation will fail due to data incapability. For example, a CHAR type data column cannot be modified to a DATE type data column.

➲ **Example 1**

To modify the column named **city** in the **tb_staff** table:

```
dmSQL> ALTER TABLE tb_staff MODIFY city NAME TO emp_photo;
```

➲ **Example 2**

To modify the data type for a column named **height** in the **tb_staff** table:

```
dmSQL> ALTER TABLE tb_staff MODIFY height TYPE TO decimal(10,2);
```

➲ **Example 3**

To modify the column order for a column named **height**, place it before the **HireDate** column:

```
dmSQL> ALTER TABLE tb_staff MODIFY height BEFORE HireDate;
```

➲ **Example 4**

To modify the default value for a column named **nation**:

```
dmSQL> ALTER TABLE tb_staff MODIFY nation DEFAULT TO 'Taiwan';
```

➲ **Example 5**

To modify the constraint for a column named **height**:

```
dmSQL> ALTER TABLE tb_staff MODIFY height CONSTRAINT TO CHECK value < 250;
```

CHANGING THE LOCK MODE

To gain a higher level of concurrency on simultaneous connections to a database, set the lock mode to a lower level (such as a **ROW** lock). However, doing this causes DBMaker to expend more resources; deciding which lock mode to use on a table always involves a trade-off. For more information about lock modes, see Section 9.4, *Locks*.

➲ **Example**

To change the lock mode for the **tb_staff** table:

```
dmSQL> ALTER TABLE tb_staff SET LOCK MODE PAGE;
```

CHANGING THE FILLFACTOR VALUE

FILLFACTOR may be specified during table creation or later modified. For more information on the FILLFACTOR option, refer to the subsection *FILLFACTOR* in *Creating Tables*.

➲ **Example**

To change the FILLFACTOR value for a table:

```
dmSQL> ALTER TABLE tb_staff SET FILLFACTOR 90;
```

TURNING NOCACHE ON/OFF

The ON/OFF option can be used at any time for NOCACHE. For more information on the NOCACHE option refer to the subsection *NOCACHE* in *Creating Tables*.

➲ **Example**

To turn the NOCACHE option for a the **tb_staff** table OFF:

```
dmSQL> ALTER TABLE tb_staff SET NOCACHE OFF;
```

ALTERING TABLES TO ANOTHER TABLESPACES

You can move a table to another tablespace, and at the same time move the index to another tablespace if the index and the table in the same tablespace. In addition, if the index and the table in different tablespace, the index will not be moved to another tablespace, so we can rebuild index in another tablespace.

Setting **FASTCOPY ON**, a user can improve execution speed of moving a table to another tablespace. When a table is moved, system will directly copy one data page to another data page, with log files operated only once in a copying and the buffer needless. Therefor the repeated operations of the log will be greatly reduced.

Move table to another tablespace can store the table to other disk, and avoid the table can't store data while disk full.

Altering table to another tablespace have some limitations:

- Users cannot alter a system table, temporary table or view to another tablespace.
- Users cannot move a permanent table to SYSTABLESPACE or TMPTABLESPACE.
- Users cannot rebuild index for permanent table in TMPTABLESPACE.
- Users cannot rebuild index for temporary table in NON-TMPTABLESPACE.
- Users cannot rebuild index for system table in other tablespace.
- Users cannot copy data from one table to the same table.
- Users cannot move table from one tablespace to the same one.

➲ **Example**

```
dmSQL> CREATE TABLE tb_staff (c1 int, c2 char(10)) in ts_reg; // create table
tb_staff in ts_reg
dmSQL> CREATE INDEX idx_desc ON tb_staff (c1); // defaultly store index in ts_reg
where the table tb_staff is stored
dmSQL> SET FASTCOPY OFF;
dmSQL> ALTER TABLE tb_staff MOVE TABLESPACE ts_app; // slowly move table tb_staff
and index idx_desc to ts_app
dmSQL> SET FASTCOPY ON;
dmSQL> ALTER TABLE tb_staff MOVE TABLESPACE ts_app; // quickly move table
tb_staff and index idx_desc to ts_app
dmSQL> REBUILD INDEX idx_desc FOR tb_staff IN ts_shrink; // rebuild index
idx_desc in ts_shrink
dmSQL> ALTER TABLE tb_staff MOVE TABLESPACE ts_aut; // only move table tb_staff
to ts_aut, index idx_desc no change
```

Using JSONCOLS Type

DBMaker supports JSONCOLS type. JSONCOLS type is a JSON representation that combines all the dynamic columns of a table into a structured output. JSONCOLS type is used to store all dynamic columns in the table, and can be used with dynamic columns. For details of dynamic columns, please refer to chapter *Using Dynamic Columns*. Users should consider using JSONCOLS type when the number of columns in a table is large and values of most columns are NULL, and operating on them individually is cumbersome.

JSONCOLS type can be defined with the CREATE TABLE or ALTER TABLE statements. For detailed SQL syntax, refer to chapter *SQL Command* in *SQL Command and Function Reference*. After a JSONCOLS column has been defined, users can use it like a normal column. A JSONCOLS column derives from LONG VARBINARY and, in DBMaker, users cannot create indexes on large object, so users cannot create an index on a JSONCOLS column that have been defined, but users can create a text index on it.

To define JSONCOLS type, use the <*JSONCOLS_type_name*> JSONCOLS keywords in the CREATE TABLE or ALTER TABLE statements.

In DBMaker, JSONCOLS type is represented as a JSONCOLS column. Define the JSON representation for JSONCOLS type in the following format:

```
{col1_name:col1_value,col2_name:col2_value,col3_name:col3_value....}
```

An example is as follows:

```
{"ID":1234,"NAME":"linda","PHONE":"1234567"}
```

NOTE *The dynamic columns that contain null values are omitted from the JSON representation for JSONCOLS type.*

If a user defines the JSON representation in wrong format when inserting or updating data, an error occurs and "Error (8077): [DBMaker] invalid JSON format"will be returned.

If the JSON representation contains a value of DATE, TIME or TIMESTAMP type, this value will be displayed as Epoch Time in the query result when users query this JSONCOLS column. In DBMaker, Epoch Time is defined as the number of milliseconds elapsed since midnight of Coordinated Universal Time (UTC) on January 1, 1970.

DBMaker stores the data of a JSONCOLS column in its internal way, therefore the display order of dynamic columns in query result may be different from insertion order.

If a user drops a JSONCOLS column or the table which contains this JSONCOLS column, the dynamic columns stored into this JSONCOLS column will be automatically dropped by system.

When using JSONCOLS type, users should consider the following guidelines:

- The JSONCOLS type cannot be changed. To change JSONCOLS type, users must delete and re-create the JSONCOLS type.
- Only one JSONCOLS type is allowed for per table.
- Constraints or default values cannot be defined on a JSONCOLS type.

The security model for JSONCOLS type is similar to that for normal columns, and that executing SELECT, INSERT, UPDATE, and DELETE statements on the

JSONCOLS column requires that the user has corresponding permissions on the JSONCOLS column.

Data manipulation of a JSONCOLS column can be performed by using the name of the individual dynamic columns, or by referencing the name of the JSONCOLS type and specifying the values of the JSONCOLS type by using the JSON representation of the JSONCOLS type. Dynamic columns can appear in any order in the JSONCOLS column.

➲ **Example**

Creating a table that has JSONCOLS type:

```
dmSQL> CREATE TABLE student(name CHAR(30), info JSONCOLS);
```

or

```
dmSQL> CREATE TABLE student(name CHAR(30));
dmSQL> ALTER TABLE student ADD COLUMN info JSONCOLS;
```

Inserting data into table **student** by using the name of the JSONCOLS type:

```
dmSQL> INSERT INTO student (name,info) VALUES
('jessia','{"desk_id":3,"birthday":"1986-09-19","score":90}');
1 rows inserted
dmSQL> INSERT INTO student (name,info) VALUES
('pine','{"desk_id":4,"birthday":"1987-03-03","score":95}');
1 rows inserted
```

Query table **student** by using "SELECT *":

```
dmSQL> SET blobwidth 80;
dmSQL> SELECT * FROM student;
       NAME                             INFO
================== ==================================================


jessia             {"score":90,"birthday":"1986-09-19","desk_id":3}
pine               {"score":95,"birthday":"1987-03-03","desk_id":4}
2 rows selected
```

Query table **student** by using the name of the JSONCOLS type:

```
dmSQL> SELECT name, info FROM student;
       NAME                            INFO
================== ==================================================
jessia            {"score":90,"birthday":"1986-09-19","desk_id":3}
pine              {"score":95,"birthday":"1987-03-03","desk_id":4}
2 rows selected
```

Updating data of table **student** by using the name of the JSONCOLS type:

```
dmSQL> UPDATE student SET info = '{"desk_id":7, "birthday":"1986-09-
19","score":88}' WHERE name='jessia';
1 rows updated
```

Modifying data type of the column named **birthday** to DATE:

```
dmSQL> ALTER TABLE student ADD DYNAMIC COLUMN birthday DATE;
dmSQL> SELECT info FROM student;
                               INFO
=====================================================================


{"score":88,"birthday":"1986-09-19","desk_id":7}
{"score":95,"birthday":"1987-03-03","desk_id":4}
2 rows selected
dmSQL> INSERT INTO student (name,desk_id,birthday,score) VALUES
('mike','8','1985-02-15','92');
dmSQL> SELECT info FROM student;
                               INFO
=====================================================================


{"score":88,"birthday":"1986-09-19","desk_id":7}
{"score":95,"birthday":"1987-03-03","desk_id":4}
{"BIRTHDAY":477244800000,"DESK_ID":"8","SCORE":"92"}
3 rows selected
```

Creating a text index on the JSONCOLS column named **info**:

```
dmSQL> CREATE TEXT INDEX idx_stu ON student(INFO);
```

Creating a view on the JSONCOLS column named **info**:

```
dmSQL> CREATE VIEW view1 AS SELECT info FROM student;
dmSQL> SELECT * FROM view1;
                                INFO
=====================================================================


{"score":88,"birthday":"1986-09-19","desk_id":7}
```

```
{"score":95,"birthday":"1987-03-03","desk_id":4}
{"BIRTHDAY":477244800000,"DESK_ID":"8","SCORE":"92"}
3 rows selected
```

Using Dynamic Columns

DBMaker supports dynamic columns. A dynamic column does not exist in the table definition, and it's the keys which can be derived from the JSON string and can be used only when a table has declared a column as JSONCOLS column. Dynamic columns are used to store semi-structured data, especially records with thousands of attributes or data whose data type frequently changes, and can be used with JSONCOLS type. For details of JSONCOLS type, please refer to chapter *Using JSONCOLS Type*. Users can consider using dynamic columns if the number of columns in a table is large and values of most columns are NULL.

Dynamic columns are stored in a JSONCOLS column, and description information on dynamic columns is stored in SYSDESCOL. For details of SYSDESCOL, please refer to chapter *DBMaker System Catalog Tables*. For details of a JSONCOLS column, please refer to chapter *Using JSONCOLS type*.

The security model for dynamic columns is similar to that for normal columns. Dynamic columns behave like any other columns with the following characters:

- An index can be created on a dynamic column.
- A dynamic columns only supports modifying data type.
- A dynamic column supports the following data types: SMALLINT, INT, FLOAT, DOUBLE, DATE, TIME, TIMESTAMP, CHAR, VARCHAR, NCHAR, NVARCHAR.
- A dynamic column must be nullable.
- A dynamic column cannot have a default value.
- A dynamic column cannot have a column constraint.
- A dynamic column cannot be used in a stored command.
- A dynamic column cannot be used in a stored procedure.

- A dynamic column cannot be used in a trigger.

After a JSONCOLS column has been created, dynamic columns can be directly used without defining. The default data type of dynamic columns is varchar(256), and users can change the default data type to another data type with ALTER TABLE ADD DYNAMIC COLUMN command. In addition, users also can declare data type of a dynamic column with this command when this dynamic column is inserted into a table. If users want to change the data type declared with this command to another data type, the ALTER TABLE MODIFY DYNAMIC COLUMN can be used. Moreover, if a user wants to drop description information of a dynamic column, the ALTER TABLE DROP DYNAMIC COLUMN command can be used. For detailed SQL syntax, refer to chapter *SQL Command* in the *SQL Command and Function Reference*. However, if a user first inserts data without executing ALTER TABLE ADD DYNAMIC COLUMN, but the inserted data cannot be converted to the data type that is later declared with this command by the user, the data will be display as NULL when a query statement is executed and no error occurs.

In addition, when a user inserts data into a dynamic column or update data of a dynamic column with parameters, DBMaker only supports insering data with VARCHAR type, at this moment, a user can insert data with other data types by using implicit data conversion function.

Data manipulation of dynamic columns can be performed by using the name of the individual dynamic columns, or by referencing the name of the JSONCOLS type and specifying the values of the JSONCOLS type by using the JSON representation of the JSONCOLS type. Dynamic columns can appear in any order in the JSONCOLS column.

➲ **Example**

The following operations base on table **student**. For details of table **student**, please refer to the example in *Using JSONCOLS Type*.

Inserting data into table **student** by using the names of the dynamic columns:

```
/* implicit data conversion is closed by default */
dmSQL> INSERT INTO student (name,score) VALUES(?,?);
dmSQL/Val> 'demi','85';     /* it is ok */
```

```
1 rows inserted
dmSQL/Val> 'finly',82;     /* INT cannot be converted to CHAR */
ERROR (9629): value list syntax error
dmSQL/Val> END;
dmSQL> SET itcmd ON;
dmSQL> INSERT INTO student (name,score) VALUES(?,?);
dmSQL/Val> 'finly',82;     /* using implicit data conversion */
1 rows inserted
dmSQL/Val> END;
dmSQL> SET itcmd OFF;
dmSQL> INSERT INTO student (name,desk_id,birthday,score)
VALUES('linda','1','1982-01-01','91');
1 rows inserted
dmSQL> INSERT INTO student (name,desk_id,birthday,score) VALUES('glow','2','1984-
03-25','93');
1 rows inserted
dmSQL> INSERT INTO student (name,desk_id,birthday,score)
VALUES('kitty','abc','1980-02-27','97');
1 rows inserted
```

Query table **student** by using "SELECT *":

```
dmSQL> SELECT * FROM student;
     NAME                           INFO
============== ================================================


jessia         {"score":88,"birthday":"1986-09-19","desk_id":7}
pine           {"score":95,"birthday":"1987-03-03","desk_id":4}
mike           {"BIRTHDAY":477244800000,"DESK_ID":"8","SCORE":"92"}
demi           {"SCORE":"85"}
finly          {"SCORE":"82"}
linda          {"BIRTHDAY":378662400000,"DESK_ID":"1","SCORE":"91"}
glow           {"BIRTHDAY":448992000000,"DESK_ID":"2","SCORE":"93"}
kitty          {"BIRTHDAY":320428800000,"DESK_ID":"abc","SCORE":"97"}
8 rows selected
```

Query table **student** by using the names of the dynamic columns:

```
dmSQL> SELECT name, desk_id, birthday, score FROM student;
     NAME         DESK_ID      BIRTHDAY              SCORE
============== =========== ============ ======================
jessia          7            19*          88
```

```
pine           4            19*           95
mike           8            19*           92
demi               NULL     NU*           85
finly              NULL     NU*           82
linda          1            19*           91
glow           2            19*           93
kitty         abc           19*           97
8 rows selected
```

Updating/deleting data of table **student** by using the names of the dynamic columns:

```
dmSQL> UPDATE student SET score='88' WHERE name='linda';
1 rows updated
dmSQL> DELETE FROM student WHERE desk_id='2';
1 rows deleted
```

Adding description of dynamic columns to this table:

```
dmSQL> ALTER TABLE student ADD DYNAMIC COLUMN desk_id INT;
dmSQL> ALTER TABLE student ADD DYNAMIC COLUMN score DOUBLE;
```

Inserting data into table **student**:

```
dmSQL> INSERT INTO student (name, desk_id, age, score) VALUES('jane','12','1982-
05-07',96);
ERROR (6150): [DBMaker] the insert/update value type is incompatible with column
data type or compare/operand value is incompatible with column data type in
expression/predicate
dmSQL> INSERT INTO student (name, desk_id, age, score) VALUES('jim',8,'1984-09-
26',98);
1 rows inserted
dmSQL> SELECT name, desk_id, birthday, score FROM student;
     NAME         DESK_ID       BIRTHDAY           SCORE
==============  ============  ============  ======================
jessia                     7  1986-09-19    8.80000000000000e+001
pine                       4  1987-03-03    9.50000000000000e+001
mike                       8  1985-02-15    9.20000000000000e+001
demi                    NULL      NULL      8.50000000000000e+001
finly                   NULL      NULL      8.20000000000000e+001
linda                      1  1982-01-01    8.80000000000000e+001
kitty                   NULL  1980-02-27    9.70000000000000e+001
jim                        8      NULL      9.80000000000000e+001
8 rows selected
```

Modifying the data type of the dynamic column named **score**:

```
dmSQL> ALTER TABLE student MODIFY DYNAMIC COLUMN score TYPE TO INT;
```

Creating an index on the dynamic column named **desk_id**:

```
dmSQL> CREATE INDEX idx1 ON student(desk_id);
```

Dropping description information of the dynamic column named **birthday**:

```
dmSQL> ALTER TABLE student DROP DYNAMIC COLUMN birthday;
```

Locking Tables

Although DBMaker automatically handles the lock mechanism whenever a database is accessed, a table may be manually locked for subsequent SELECT or UPDATE statements. Locking a table while a user is viewing or modifying it will prevent updates by other people.

DBMaker supports some options for locking tables, such as *shared locks* for viewing data or *exclusive locks* for modifying data, and the WAIT or NO WAIT option which is used when obtaining a lock. For more information about these features, see the *SQL Command and Function Reference*. To learn about table locks, concurrency control, and transaction handling, refer to Chapter 9, *Concurrency Control*.

➲ **Example**

To lock the **tb_staff** table for later selections and not wait if it cannot get the table lock right away:

```
dmSQL> LOCK TABLE tb_staff IN SHARE MODE NO WAIT;
```

Dropping Tables

A user can drop a table when the table is not being used any more. When a table is dropped, all data and indexes for this table are dropped, and pages allocated by the dropped table are released.

➲ **Example 1**

To drop the **tb_staff** table use the DROP TABLE command:

```
dmSQL> DROP TABLE tb_staff;
```

➲ **Example 2**

To drop the **tb_staff** table use the DROP TABLE IF EXISTS command:

```
dmSQL> DROP TABLE IF EXISTS tb_staff;
```

6.3 Managing Views

DBMaker provides an ability to define a *virtual table* called a *view*, which is based on existing tables and is stored as a definition with a user-defined view name. The view definition is stored persistently in the database, but the actual data is not physically stored there. Rather, the data is stored in the original base tables the views were derived from. A view is defined by a query that references one or more tables or other views.

Views are a very helpful mechanism in a database. For example, complex queries can be defined once and used repeatedly without having to be rewritten. Furthermore, views can be used to enhance the security of a database by restricting access to a predetermined set of rows or columns in a table.

A user cannot determine from a view which rows of tables to update, since a view is derived from queries on tables. Due to this limitation, views can only be queried unless the view is derived from a single table.

Creating Views

Views may be created with dmSQL or the JDBA Tool. A view is defined by a name together with a query that references tables or other views.

Users can specify a list of column names for a view. If column names are not specified, the view will inherit column names from the underlying tables.

Use CREATE VIEW syntax. For example, to allow other users to see only two columns from the **tb_staff** table, create a view with the SQL command shown below. Users can then view only two columns, (**name** and **ID)**, from the **tb_staff** table through the view **vi_staff**.

➲ **Example 1**

```
dmSQL> CREATE VIEW vi_staff (empName, empId) AS SELECT name, ID FROM tb_staff;
```

Use CREATE OR REPLACE VIEW syntax. For example, a view named **vi_staff** already exists, it will allow other users to see only two columns, (**name** and **ID)**, from the **tb_staff** table, but we need to change the view definition to allow the same users to see only three columns from the **tb_staff** table, but not change the privileges on the view. Replace the view with the SQL command shown below. Users can view three columns, (**name**, **ID** and **age**), from the **tb_staff** table through the view **vi_staff**.

➲ **Example 2**

```
dmSQL> CREATE OR REPLACE VIEW vi_staff (empName, empId, empAge) AS SELECT name,
ID, age FROM tb_staff;
```

Browsing View Schema

The construction of a view may be queried by using dmSQL or the JDBA Tool. Use the dmSQL command DEF VIEW to directly query a table's schema.

➲ **Example**

To view the construction for view **vi_staff**:

```
dmSQL> DEF VIEW vi_staff;
dmSQL> CREATE VIEW SYSADM.VI_STAFF(empname,empid) AS SELECT name,id from
SYSADM.TB_STAFF;
```

Dropping Views

A view can be dropped when it is no longer required. When a view is dropped, only the definition stored in the system catalog is removed. The base tables that the view was derived from are unaffected.

➲ **Example 1**

To drop the view **vi_staff** use the DROP VIEW command:

```
dmSQL> DROP VIEW vi_staff;
```

➲ **Example 2**

To drop the view **vi_staff** use the DROP VIEW IF EXISTS command:

```
dmSQL> DROP VIEW IF EXISTS vi_staff;
```

6.4 Managing Synonyms

A *synonym* is an alias for any table or view. Since a synonym is simply an alias, it requires no storage other than a definition in the system catalog.

Synonyms are useful for simplifying a fully qualified table or view name. DBMaker normally identifies tables and views with fully qualified names that are composites of the owner and object names. By using a synonym, anyone can access a table or view using the corresponding synonym without having to make use of the fully qualified name. Because a synonym has no owner name, each synonym in the database must be unique so DBMaker can identify them. Synonyms may be created or dropped with dmSQL or the JDBA Tool.

Creating Synonyms

➲ **Example 1**

Use **CREATE SYNONYM** command:

```
dmSQL> CREATE SYNONYM staff FOR SYSADM.tb_staff;
```

Assume that the owner of the table **tb_staff** is **SYSADM**. This command creates the alias **staff** for the table **SYSAMD.tb_staff**. All database users can directly reference the table **SYSAMD.tb_staff** through the synonym **staff**.

➲ **Example 2**

Use **CREATE OR REPLACE SYNONYM** command**:**

```
dmSQL> CREATE OR REPLACE SYNONYM staff FOR SYSADM.tb_staff;
```

Assume that an alias **staff** for the table **SYSAMD.tb_staff** is already exists, you can replace it without drop it.

Dropping Synonyms

A synonym that is no longer required can be dropped. When a synonym is dropped, only its definition is removed from the system catalog.

➲ **Example 1**

To drop the **staff** synonym with the DROP SYNONYM command:

```
dmSQL> DROP SYNONYM staff;
```

➲ **Example 2**

To drop the **staff** synonym with the DROP SYNONYM IF EXISTS command:

```
dmSQL> DROP SYNONYM IF EXISTS staff;
```

6.5 Managing Indexes

An index provides support for fast random access to a row. Building indexes for a table speeds up searching. For example, when a user executes the query SELECT NAME FROM tb_staff WHERE id = 306004, it is possible to retrieve the data in a much shorter time if there is an index created for the **ID** column.

An index can be composed of more than one column, up to a maximum of 32. All the table's columns can be used in an index.

An index can be *unique* or *non-unique*. In a unique index, no more than one row can have the same key value, with the exception that any number of rows may have NULL values. If a user creates a unique index on a table, DBMaker will check whether all existing keys are distinct or not. If there are duplicate keys, DBMaker will return an error message. After creating a unique index on a table, if a user inserts a row in the table, DBMaker ensures there are no existing rows with the same key as the new row.

When creating an index, the sort order of each index column can be specified as ascending or descending. For example, suppose there are five keys in a table with the values 1, 3, 9, 2 and 6. In ascending order, the sequence of keys in the index is 1, 2, 3, 6 and 9, and in descending order, the sequence of keys in the index is 9, 6, 3, 2 and 1.

When a user implements a query, the index order occasionally affects the order of data output.

➲ **Example**

If the following query is executed:

```
dmSQL> SELECT name, age FROM friend_table WHERE age > 20;
```

Using an index with a descending order on the column **age**, the output appears as follows:

```
      name                       age
----------------          ----------------
Jeff                                    49
Kevin                                   40
Jerry                                   38
Hughes                                  30
Cathy                                   22
```

A user can specify the *fillfactor* for tables when creating an index. The fillfactor denotes how dense the keys will be in the index pages. The legal fillfactor values are in the range from 1% to 100%, and the default is 100%. If a user updates data often after creating the index, the user can set a loose fillfactor in the index, for example 60%. If the user never updates the data in this table, then the fillfactor can be left at the default value of 100%.

A user may also specify to create an index on a separate tablespace. This can result in improved disk I/O for searches that use the index if multiple disks are used.

Before creating indexes on a table, it is recommended to load all data first, especially if there is a large amount of data for that table. If a user creates an index before loading the data into a table, the indexes will be updated each time the user loads a new row. It is far more efficient to create an index after loading a large amount of data than to create an index before loading the data.

Creating Indexes

Indexes may be created using the Create Index wizard of the JDBA Tool, or the dmSQL CREATE INDEX command. To create an index on a table, specify the index name and index columns. Specify the sort order of each column as ascending or descending. The default sort order is ascending.

➲ **Example 1**

To create an index, **idx_desc,** on the column **ID** for the table **tb_staff** in descending order use the DESC option:

```
dmSQL> CREATE INDEX idx_desc ON tb_staff (ID DESC);
```

➲ **Example 2**

To create a unique index, **idx_uniq,** on the column **ID** for the table **tb_staff** use the UNIQUE option:

```
dmSQL> CREATE UNIQUE INDEX idx_uniq ON tb_staff (ID);
```

➲ **Example 3**

To create an index with a specified fillfactor use the FILLFACTOR option:

```
dmSQL> CREATE INDEX idx_fill ON tb_staff (name, id DESC) FILLFACTOR 60;
```

➲ **Example 4**

To create an index on tablespace **ts_reg**:

```
dmSQL> CREATE INDEX idx_reg ON tb_staff (name, id DESC) IN ts_reg FILLFACTOR 60;
```

Creating Expression Indexes

Indexes can be created not only on simple columns, but also on Expression columns or User Defined Function (UDF) columns.

➲ **Example 1**

To create an index, **idx_expr**, on the expression **basepay+bonus** for table **tb_salary**:

```
dmSQL> CREATE INDEX idx_expr ON tb_salary (basepay+bonus);
```

➲ **Example 2**

To create an index, **idx_substr**, on the UDF substring **(nation,1,3)** for table **tb_ salary**:

```
dmSQL> CREATE INDEX idx_substr ON tb_salary (substring(nation,1,3) desc);
```

➲ **Example 3**

To create an index, **idx_udf**, on the expression and UDF **abs(bonus)** for table **tb_salary**:

```
dmSQL> CREATE INDEX idx_udf ON tb_salary(basepay+abs(bonus)-tax desc);
```

Creating Indexes on XML column

To improve XML querie performance, we can create special XML index on XML columns. The XML index supports XML UDF: **extract()** and **extractvalue()**.The following example shows how to create an index on an XML column using dmSQL. Please see the *SQL Command and Function Reference* for additional details on the syntax and usage of the SQL command CREATE INDEX.

➲ **Example 1**

To create an index use the **extract** XML UDF:

```
dmSQL> CREATE INDEX idx_extr ON tb_extract (extract(id,
'/order/items/item/@product', NULL));
```

➲ **Example 2**

To create an index use the **extractValue** XML UDF:

```
dmSQL> CREATE INDEX idx_extrV ON tb_extract (extractValue(id,
'/order/items/item/@product', NULL));
```

The primary difference between extract() and extractvalue() are:

extract()

- allows a multi-value, a single value or zero value result.
- asc / desc are not allowed
- unique index is not allowed
- extractValue()
- allows a single value or a zero value of the UDF results (when the UDF result is a multi-value result then the create index fails for existing tuple and the insert data fails for newly inserted tuple)

- allows asc / desc
- allows unique index

Creating Filtered Indexes

Filtered Indexes (Conditional Index) is an index with the WHERE clause. A filtered index is an optimized index especially suited to cover queries that select from a well-defined subset of data. That is to say, Filtered Index is inserted into index page before filter, filtered index's data not include all rows, it can be partial of rows defined by filter condition (WHERE clause). It uses a filter predicate to select a portion of rows in the table. A well-designed filtered index can improve query performance as well as reduce index maintenance and storage costs compared with full-table indexes.

A well-designed filtered index improves query performance and execution plan quality because it is smaller than a full-table index and has filtered statistics. The filtered statistics are more accurate than full-table statistics because they cover only the rows in the filtered index.

An index is maintained only when data manipulation language (DML) statements affect the data in the index. A filtered index reduces index maintenance costs compared with a full-table index because it is smaller and is only maintained when the data in the index is changed. It is possible to have a large number of filtered indexes, especially when they contain data that is changed infrequently. Similarly, if a filtered index contains only the frequently modified data, the smaller size of the index reduces the cost of updating the statistics.

Creating a filtered index can reduce disk storage. You can replace a full-table index with multiple filtered indexes without significantly increasing the storage requirements.

The CREATE INDEX statement with the WHERE clause defines an index on of an existing table. The filtered index type only allows non-unique index and unique index, but not allows primary key. The maximum number of columns that can be contained in the WHERE clause is 32.

DBMaker allows one of the following users to create a filtered index.

- Users with DBA authority or higher
- The table owner
- Users granted to CREATE INDEX privilege

The WHERE clause can be any combination of the following predicate, includes:

- any columns of the table
- constant values
- comparison: =, >, >=, <, <=, !=.   ex: c1>=3
- like, ex: c3 like 'abc'
- is null, is not null. ex: c4 is null
- in list, ex: c5 in (1,3,5)
- operator: +, -, *, /. ex: c1+c2>5
- UDF, ex: abs(c6)>5
- blob operator: match, contain
- combination of AND, ex: c1=3 and c2=5 and c3=7
- combination of OR, ex: c1=3 or c2=5 or c3=7

The WHERE clause cannot allow the following statements:

- sub-query
- host variable
- mix of AND and OR, ex: c1=3 or c2=5 and c3=7

NOTE *User cannot specify a filtered condition (WHERE clause) on a text index.*

➲ **Example 1**

To create a filtered index, **filidx_basepay**, using the **where** clause **like** predicate for table **tb_salary**:

```
dmSQL> CREATE INDEX filidx_basepay ON tb_salary (id, basepay) where name like
'abc%';
```

➲ **Example 2**

To create a filtered index, **filidx_income**, using the **where** clause for table **tb_salary**:

```
dmSQL> CREATE INDEX filidx_income ON tb_salary(basepay+bonus,tax)where id>30;
```

Dropping Indexes

Indexes may be dropped using the JDBA Tool, or the dmSQL DROP INDEX statement. If the index is a primary key and is referred to by other tables it cannot be dropped. For information on primary keys, refer to the section *Managing Data Integrity*.

➲ **Example**

To drop the index **idx_desc** from the table **tb_staff**:

```
dmSQL> DROP INDEX idx_desc FROM tb_staff;
```

Rebuilding Indexes

Indexes may be rebuilt using JDBA Tool, or the dmSQL REBUILD INDEX statement. In general, the index will need to be rebuilt when it becomes fragment, which reduces its efficiency. Rebuilding an index will drop the old index and then create a new one.

User can move a table to another tablespace, if the index and the table in the same tablespace, then the index will be moved to another tablespace. If the index and the table in different tablespaces, then the index will not be moved to another tablespace, so we can rebuild index in another tablespace.

➲ **Example 1**

To rebuild the index **idx_fill** for the table **tb_staff**:

```
dmSQL> REBUILD INDEX idx_fill FOR tb_staff;
```

➲ **Example 2**

```
dmSQL> REBUILD INDEX idx FOR tb_staff IN ts_reg;
```

6.6 Managing Auto Indexes

With the development of the cloud database, indexes are managed automatically rather than manually. According to the query statements executed by users, the auto index daemon can analyze users' requirements to manage indexes more intelligently.

DBMaker supports auto indexes, its behavior is similar to non-unique index, but it can be automatically created or dropped by the auto index daemon. If set AUTOCOMMIT ON, DBMaker only requires *Update(U)* lock when creating an auto index, which means DBMaker allows other users to query the table simultaneously.

DBMaker supports auto index daemon to operate an auto index, it includes two main parts: **Collection Mechanisms** and **Handling Mechanisms**. Only if the auto index daemon startup, the Collection Mechanisms will analyze execution plan of the query statements executed by users, and then record the analytic result (zero or multiple logs a time) into the file *DMSCAN.LOG*. The Handing Mechanism will be awoken according to the setting of both **DB_IdxTm** and **DB_IdxTv**, by using the dmSQL command *SYNC AUTO INDEX*, or with the Sync Auto Index wizard of JDBA Tool. The main jobs of the Handing Mechanism are read *DMSCAN.LOG*, analyze all logs to decide whether there is a need to create auto indexes, update indexes' use information, and drop the auto indexes not in use for a period specified by **DB_IdxDp** (only auto indexes will be dropped; other types of indexes will not be dropped). The indexes created or dropped in this process will be recored into the file *DMAUTOIDX.LOG*, which is a convenience for users who want to know the working condition of the auto index daemon.

In fact, when a user on client performs a query statement, the log information will not be recorded immediately into the file *DMSCAN.LOG*, the collection mechanisms will first store the log information in the buffer (fixed size is 2560 bytes) of the client, and will not write the log information into the file *DMSCAN.LOG* untill the buffer is filled. In addition, if the user disconnects the database or executes the *SYNC AUTO INDEX* command, the data in the buffer will also be written into the file *DMSCAN.LOG*. In this way, not only can the information disorder be avoided because of multiple users' simultaneous writing to *DMSCAN.LOG*, but also it helps focus the I\O operation to improve the performance of auto index's collection mechanisms.

To automatically create and drop an auto index, users need to set these keywords **DB_IdxSv**, **DB_IdxLg**, **DB_IdxTm**, **DB_IdxTv**, **DB_IdxDp** and **DB_IdxLn** in **dmconfig.ini** file to control the Auto Index Daemon. With the database running, only a user with DBA or SYSDBA or SYSADM authority can call **setSystemOption()** to set these options except **IDXLG**.

The difference between auto indexes and other type of indexes is reflected in the following four aspects:

- Auto indexes can be created automatically by the auto index daemon.
- Auto indexes can be dropped by the auto index daemon automatically if the index can merged with other indexes or it is not be used exceed the number of specified dropped days.
- The maximum column number of an index created by user is 32, but the maximum column number of an auto index created by auto index daemon is 16.
- Creating an auto index requires **U** lock when set *COMMIT ON*, but creating other indexes requires **X** lock.

DBMaker also supports "*SET LOADAUTOINDEX ON|OFF*" syntax and ODBC function *SQLSetConnectOption* for user to decide whether load auto index or not. For more information on ODBC functions, please refer to the *ODBC Programmer's Guide*.

➲ **Example 1**

To turn on the auto index server with the system stored procedure **SetSystemOption**.

```
dmSQL> CALL SETSYSTEMOPTION('IDXSV','1');  //activate auto index server
dmSQL> SELECT * FROM tb_staff WHERE joinDate='1986-07-20';   //create auto index
by daemon
dmSQL> sync auto index;    //wake up auto index daemon
dmSQL> SELECT * FROM sysindex;
dmSQL> SELECT * FROM sysindexref;
```

➲ **Example 2**

To reset the number of days to **60** to drop an auto index with the system stored procedure **SetSystemOption**.

```
dmSQL> CALL SETSYSTEMOPTION ('IDXDP','60');
```

Creating Auto Indexes

Auto index can be auto created by auto index daemon or manual created by users' using the dmSQL command *CREATE AUTO INDEX*. About how to manually create auto indexes, please refer to the syntax CREATE INDEX in the *SQL Command and Function Reference*.

When auto index daemon creates an auto index on a table, specify the auto index type and auto index columns id, specify the sort order of each column as ascending or descending. The default sort order is ascending.

The name of the auto index created by auto index daemon is made up of *AUTO, underline, column id* and *column order* (*column order is D* or *null* with *D* meaning desc and *null* meaning asc). If the two indexes with a same name can merge, there is no need to create an index; if they cannot merge, users need to create a new index and name it index_name_Rxxx. Index_name is the name of the two indexes and the three-digit number **xxx** is random.

➲ **Example 1**

To create the auto index **AUTO_ID_2** on the columns **ID** and **NAME** for the table **tb_staff** in descending order, use the option DESC by dmsql commands:

```
dmSQL> CREATE AUTO INDEX AUTO_ID_2 ON tb_staff (ID DESC, NAME);
```

➲ **Example 2**

If the new index has a same name with the old index and the two indexes cannot merge, users can extend the new index's name to **AUTO_ID_2__R321**. The three-digit number **321** is random.

```
dmSQL> CREATE AUTO INDEX AUTO_ID_2__R321 ON tb_staff (ID DESC, NAME);
```

Creating Expression Auto Indexes

Auto indexes can be auto created not only on simple columns, but also on Expression columns or User Defined Function (UDF) columns.

➲ **Example 1**

To create an auto index, **auto_idx_expr**, on the expression **basepay+bonus** for table **tb_salary**:

```
dmSQL> CREATE AUTO INDEX auto_idx_expr ON tb_salary (basepay+bonus);
```

➲ **Example 2**

To create an auto index, **auto_idx_substr**, on the UDF substring **(nation,1,3)** for table **tb_ salary**:

```
dmSQL> CREATE AUTO INDEX auto_idx_substr ON tb_salary (substring(nation,1,3)
DESC);
```

Dropping Auto Indexes

The auto index daemon can automatically drop the auto indexes not in use for a period specified by **DB_IdxDp**, additionally, users also can drop auto indexes with the dmSQL command *DROP AUTO INDEX*. If the index is created on a column as a primary key or referred by other tables, it cannot be dropped. For information on primary keys, please refer to the section *Managing Data Integrity*.

➲ **Example**

To drop the index **AUTO_1D_2** from the table **tb_staff**:

```
dmSQL> DROP INDEX AUTO_1D_2 FROM tb_staff;
```

6.7 Managing Text Indexes

A *text index* is a mechanism that provides fast access to rows in a table that contains one or more words or phrases in columns. Text indexes contain a representation of all the text found in the columns they are based on, but the data is encoded and structured to make retrieval much faster than directly from the table. Once a user creates a text index for a table, its operation is transparent. The DBMS uses the index to improve full-text query performance whenever possible.

DBMaker provides two text indexing methods: signature and inverted file (IVF).

Text indexes can be built on all character type columns, including CHAR, VARCHAR, LONG VARCHAR, LONG VARBINARY, and FILE data types. A table can have many text indexes and a text index can be built using multiple columns. A user may create text indexes by using either the JDBA Tool or the CREATE [SIGNATURE | IVF] TEXT INDEX dmSQL command.

➲ **Example**

To use a text index in the **data** column automatically (without specifying it):

```
dmSQL> SELECT id FROM tb_book WHERE data MATCH 'compute';
```

The string operators for DBMaker include MATCH, CONTAIN, CONTAINS, and LIKE. Only the MATCH and CONTAINS operators can be applied to a text index search.

DBMaker provides two different types of text index: signature and inverted-file. Signature text index is more efficient for small amount of data. Inverted-file text index usually consumes more storage space but its response for queries is faster for large amount of data.

Creating Signature Text Indexes

DBMaker creates signature text indexes if no text index method is specified by the command. A user may create text indexes using either the JDBA Tool, the CREATE TEXT INDEX dmSQL command or the CREATE SIGNATURE TEXT INDEX command.

➲ **Example**

Creating a signature text index, **tidx_name**, on the name column for the table **tb_staff**:

```
dmSQL> CREATE SIGNATURE TEXT INDEX tidx_name ON tb_staff(name);
```

SIGNATURE TEXT INDEX PARAMETERS

DBMaker provides two parameters for conveniently configuring performance and storage size of signature text indexes.

- **Total text size (MB)** — the estimated total size of all source documents, in megabytes (MB). The range is 1 through 200 and the default is 32. Please note, the real total text size is not limited to 200 MB; if the size is larger than 200, set to 200. However, we strongly recommend using IVF text index to index very large amounts of data for significantly better query performance.
- **Scale** — the expected index size-to-total text size ratio. If a user sets **total text size** to 20 (MB) and expects the text index to use 10 MB of storage, then he should set **scale** to 50 (in percent). The larger **scale**, the better search performance. The range is 10 through 200 and the default value is 40 (in percent).

➲ **Example**

To create a text index, **tidx_scale** on the column **name** of the table **tb_staff** that contains about 40 megabytes of data, and we wish the text index uses about 20 megabytes of storage space:

```
dmSQL> CREATE SIGNATURE TEXT INDEX tidx_scale ON tb_staff(name)
       TOTAL TEXT SIZE 40 MB
       SCALE 50;
```

Users can use the default setting as text index parameters. To get higher text index performance or to reduce the text index size, change the text index parameters. Set the parameters and monitor the text index performance, and then re-adjust the parameters.

Creating Inverted-File (IVF) Text Indexes

A user can create inverted-file text indexes by using the CREATE IVF TEXT INDEX command.

➲ **Example**

To create an inverted-file text index, **ivfidx_name**, on the name column for the table **tb_staff**:

```
dmSQL> CREATE IVF TEXT INDEX ivfidx_name ON tb_staff(name);
```

INVERTED-FILE TEXT INDEX PARAMETERS

There are two parameters for use in the creating IVF text index command:

Storage path — the logical working directory where the inverted-files will reside in. Users should define the logical directory in the **dmconfig.ini** file. The default is the value of **DB_DbDir**, the database's home directory. The detail storage management and naming convention of inverted-file index will be described in the next section.

Total text size (MB) — the approximate total size of documents will be indexed in the future. The unit of size is mega-byte (MB). Based on the size, DBMaker decides how many partitions will be made. It may range between 1 MB to 10,000 MB , and the default value is 500 MB.

➲ **Example**

To create an inverted-file text index, **ivfidx_name** in the path *\IVFDIR* on the column **name** of the table **tb_staff** that contains about 400 MB of data:

First, add a logical path in the database's **dmconfig.ini** section.

```
MYPATH1 = \IVFDIR
```

Use the following command.

```
dmSQL> CREATE IVF TEXT INDEX ivfidx_name ON tb_staff(name)
    2> STORAGE PATH MYPATH1
    3> TOTAL TEXT SIZE 400 MB;
```

While creating inverted-file text indexes, it requires large amount of memory resource. DBMaker will take a simple rule to decide the maximum memory usage for creating text indexes. If DBMaker cannot detect free memory or free memory resource less than 128 MB, then the maximum memory usage will be 64 MB, otherwise will be half of free memory resource. Users can specify the approximate upper bound of memory usage manually through **dmconfig.ini** by adding a keyword entry **DB_IFMem** in megabyte (MB).

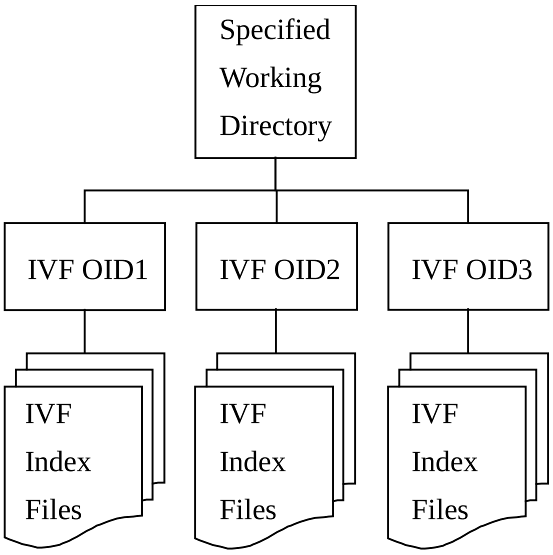
➲ **Example**

To specify 100 MB memory usage for creating inverted-file text-index in **dmconfig.ini**:

```
DB_IFMem = 100
```

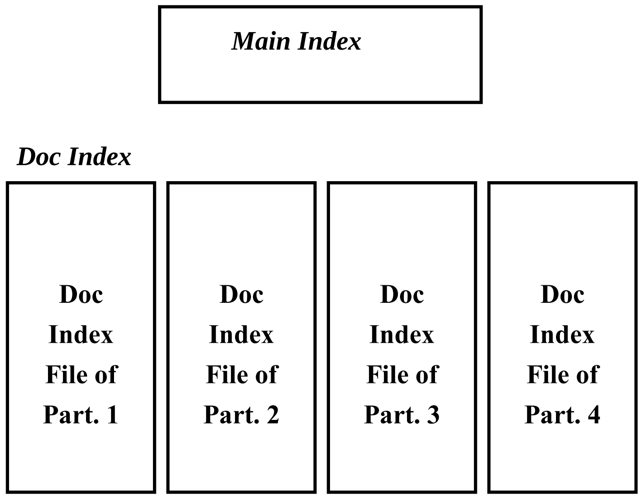
STORAGE OVERVIEW

In addition to the working directory specified by the **Storage path** parameter, DBMaker will generate sub-directories in this directory to manage different inverted-file indexes. Each inverted-file index has a unique time version, so DBMaker can use this property to generate a unique sub-directory to store index files. Naming the sub-directory is described later. Sub-directories and inverted-files cannot rollback when an inverted-file index is dropped.



For example, *\DBMaker5.4* is a specified working directory, and we create an IVF under this working directory with index name IVF1, time version 1024476670, then a sub-directory IVF1.1024476670 is created. All inverted-files are resided in the sub-directory. There are three kinds of inverted-file with different term type, Single-Byte term, Uni-Gram term and Bi-Gram term, and each inverted-file has several partitions decided by text size.

The following is the conceptual structure of IVF index files with four partitions.


Main Index
Doc Index
Doc Index File of Part. 1
Doc Index File of Part. 2
Doc Index File of Part. 3
Doc Index File of Part. 4

Inverted-File Structure with Four Partitions

Choosing between signature and inverted-file will depend on the following factors:

1. Index size – the size of signature index will not exceed the ratio set by the **Scale** parameter, which is 40% of total data size by default. The average size of inverted-file indexes is about 1.5 times of the data size, but could grow to two or even three times, depending on the property of data.

2. Response time for queries – on a modern personal computer with sufficient memory and processing power, users can expect sub-second response time from inverted-file indexes even the data size is gigabytes. Signature indexes will take longer to respond, especially when the data size is getting large.

3. Integration with database – unlike signature indexes which are stored as BLOB objects, inverted-file indexes are stored as external files, so, for example, users cannot rollback a dropping inverted-file index operation.

Try both types of text index to find which one suits the data's characteristics best. As a rule of thumb, for data size smaller than 100 MB, signature indexes respond to queries reasonably fast and usually take less storage space.

Creating Text Indexes on Multiple Columns

A text index can be built using multiple columns. Use CONTAINS and the concatenation operator (||) to perform multi-column text queries. Users can query on all columns of the index or just part of them. That is, the column list in a match query must be contained in the column list of a text index to use the text index. Users are also allowed to use the multi-column query syntax even if no text index is created on the column list, but no text index will be used.

Searching on multiple columns is logically equivalent to merging all columns' data then searching.

➲ **Example 1**

To create an inverted-file text index **ivfidx_multiple** on columns **author**, **subject** and **content** of the table **tb_document**:

```
dmSQL> CREATE IVF TEXT INDEX ivfidx_multiple ON
tb_document(author,subject,content);
dmSQL> SELECT author FROM tb_document WHERE
    2> CONTAINS(author || subject || content, 'reagan');
```

➲ **Example 2**

To query on partial column lists:

```
dmSQL> SELECT author FROM tb_document WHERE
    2> CONTAINS(author || content, 'reagan');
dmSQL> SELECT author FROM tb_document WHERE CONTAINS(subject, 'reagan');
dmSQL> SELECT author FROM tb_document WHERE subject MATCH 'reagan';
```

➲ **Example 3**

In this example, the column **subject** is included in the text index **ivfidx_multiple** but **abstract** is not, so this query will not use any text index.

```
dmSQL> SELECT author FROM tb_document WHERE
    2> CONTAINS(subject || abstract, 'reagan');  // no text index used
```

➲ **Example 4**

This example illustrates the behavior of query on multiple columns.

```
dmSQL> CREATE TABLE tb_example (c1 char(20), c2 char (20), c3 serial);
dmSQL> INSERT INTO tb_example VALUES('apple orange', 'banana grape');
dmSQL> INSERT INTO tb_example VALUES('grape orange', null);
dmSQL> CREATE TEXT INDEX ivfidx_example ON tb_example (c1, c2);
dmSQL> SELECT c3 FROM tb_example WHERE CONTAINS (c1 || c2, 'apple');
     C3
===========
     1
 1 rows selected
dmSQL> SELECT c3 FROM tb_example WHERE CONTAINS (c1 || c2, 'orange & grape');
     C3
===========
     1
     2
 2 rows selected
```

Creating Text Indexes on Media Types

DBMaker's large object columns can register media types. For example, a LONG VARBINARY column knows its content is a Microsoft Word file. This allows DBMaker to invoke the proper functions to perform full-text searches on Microsoft Word documents. DBMaker also provides media UDF for converting some media formats to pure text and a UDF (CHECKMEDIAFORMAT) to query the media format. Table 6-1 summarizes the media types available and their associated SQL commands.

MEDIA TYPE	DATA TYPE	FILE TYPE
Microsoft Word	MsWordType	MsWordFileType
HTML	HtmlType	HtmlFileType
XML	XmlType	XmlFileType
Microsoft PowerPoint	MsPPTType	MsPPTFileType
Microsoft Excel	MsExcelType	MsExcelFileType
PDF	PDFType	PDFFileType

Table 6-1: Media types and corresponding SQL commands

Internally, MsWordType MsPPTType, MsExcelType and PDFType are treated as a LONG VARBINARY object; HtmlType and XmlType are LONG VARCHAR objects and MsWordFileType, HtmlFileType , XmlFileType, MsPPTFileType, MsExcelFileType and PDFFileType are FILE objects.

FULL-TEXT SEARCH ON MEDIA-TYPE COLUMNS

Users can create a text index and perform a full-text search on the media type, but first the media format must be converted into pure text. DBMaker will not understand new media formats so conversion of the media format into pure text will not be possible nor will full-text search on the media format

DBMaker provides the following media UDFs for converting some media format to pure text:

DOC, XLS, PPT, HTM, PDF.

- DOCTOTXT(BLOB) RETURNS NCLOB;
- XLSTOTXT(BLOB) RETURNS NCLOB;
- PPTTOTXT(BLOB) RETURNS NCLOB;
- HTMTOTXT(CLOB) RETURNS CLOB;
- PDFTOTXT(BLOB) RETURNS NCLOB;

MATCH and CONTAINS can also be used for performing full-text search on media-type columns just as on regular text columns.

➲ **Example 1**

Converts a PowerPoint document to a temporary BLOB containing the pure text of blob as Unicode.

```
dmSQL> CREATE TABLE tb_ppt(pptfile LONG VARBINARY);
dmSQL> INSERT INTO tb_ppt VALUES(?);
dmSQL/Val> &e:\udf\pptfile\pfile.ppt;
dmSQL/Val> END;
dmSQL> SELECT PPTTOTXT(pptfile) FROM tb_ppt;
```

➲ **Example 2**

Create a table with a MS Word type column, insert some data, and search.

```
dmSQL> CREATE TABLE tb_minutes(id INT, doc MsWordFileType);
dmSQL> INSERT INTO tb_minutes VALUES(1, 'c:\meeting\20020403-1.doc');
dmSQL> SELECT id FROM tb_minutes WHERE doc MATCH 'Jeff';
    id
===========
         1
 1 rows selected
```

➲ **Example 3**

Create a signature text index on the column **doc** of the table **tb_minutes** and search.

```
dmSQL> CREATE TEXT INDEX tidx_doc ON tb_minutes(doc);
dmSQL> SELECT id FROM tb_minutes WHERE doc MATCH 'Jeff';
    id
===========
         1
 1 rows selected
```

CHECK COLUMN DATA'S MEDIA TYPE

It is possible that a media-type column contains data of different types. DBMaker can verify the content during inserting or updating data to media-type columns. DBMaker provides a built-in function CHECKMEDIAFORMAT to check whether the column's data match the specified media format. If types match, the function returns 1, otherwise returns 0. These media type formats can be supported to office 2007-2010 version.

DBMaker support the following media type formats: DOC, XLS, PPT, HTM and PDF.

- **DOC**: Microsoft Words Document
- **XLS**: Microsoft Excel Document
- **PPT**: Microsoft Power Point Document
- **HTM**: Hypertext Markup Language
- **PDF**: Portable Document Format

NOTE *The format of PDF supported by DBMaker are 1.2 to 1.7.*

➲ **Example**

To check weather the media type format is correct:

```
dmSQL> CREATE TABLE tb_checkmedia(note LONG VARBINARY);
dmSQL> INSERT INTO tb_checkmedia VALUES(?);
dmSQL/Val> &E:\DOCS\Media.doc;
dmSQL/Val> &E:\DOCS\Media2007.docx;
dmSQL/Val> END;
dmSQL> SELECT checkmediaformat(note,'doc') FROM tb_checkmedia;
```

- It will return 0, 1 or NULL.
- Returns 1 when the BLOB's content matches the specified media format
- Returns 0 when the BLOB's content does not match the specified media format
- Returns NULL when the blob is NULL.
- When the table column defined with original media type (system domain) and the media format is not the correct format, the migration may fail. To resolve this problem, either remove the invalid media data or change the data type to the CLOB or the BLOB data type to avoid the step of checking for the correct media format.

Dropping Text Indexes

Text indexes may be dropped using the JDBA Tool or the dmSQL DROP TEXT INDEX statement.

➲ **Example**

To drop the **tidx_name** text index from the table **tb_staff**:

```
dmSQL> DROP TEXT INDEX tidx_name FROM tb_staff;
```

Rebuilding Text Indexes

Unlike indexes, the text index will not simultaneously reflect table content if new records are inserted or old records are updated. Therefore, they need to be rebuilt manually. Data updated after the most recent rebuild will not be found during a text index search.

➲ **Example**

```
dmSQL> CREATE TABLE tb_song (id INT, name VARCHAR(20));
dmSQL> INSERT INTO tb_song VALUES(1,'Endless Love');
1 rows inserted
dmSQL> CREATE TEXT INDEX tidx_name ON tb_song(name);
dmSQL> INSERT INTO tb_song VALUES(2,'Love Story');
1 rows inserted
dmSQL> SELECT * FROM tb_song WHERE name MATCH 'love';
    id              name
=========== ====================
         1  Endless Love
 1 rows selected
```

There should be two records to match the search pattern, but only one is retrieved.

INCREMENTALLY REBUILD TEXT INDEXES

The REBUILD TEXT INDEX command rebuilds the updated data incrementally by collecting all new and updated records, building new signature vectors, and appending the new vectors to the tail of the text index. When only a few records have changed, the REBUILD TEXT INDEX <*index_name*> INCREMENTAL command is the fastest method for rebuilding.

➲ **Example**

To rebuild a text index incrementally and display the results:

```
dmSQL> REBUILD TEXT INDEX tidx_name FOR tb_song;
dmSQL> SELECT * FROM tb_song WHERE name MATCH 'love';
     id              name
=========== ====================
          1 Endless Love
          2 Love Story
 2 rows selected
```

FULLY REBUILD TEXT INDEXES

If a large number of documents are deleted or updated, use the REBUILD TEXT INDEX <*index_name*> FULL command to fully rebuild a text index with its original type (signature or inverted file) and parameters.

➲ **Example 1**

To fully rebuild the **tidx_name** text index for the table **tb_song**:

```
dmSQL> REBUILD TEXT INDEX tidx_name FOR tb_song;
```

To reset the creating text index parameters of a text index or use a different type of text index, it must be dropped and re-created.

➲ **Example 2**

To rebuild the tidx_name signature text index for the table tb_song as an inverted-file index:

```
dmSQL> DROP TEXT INDEX tidx_name FROM tb_song;
dmSQL> CREATE IVF TEXT INDEX tidx_name ON tb_song(name);
```

➲ **Example 3**

To fully rebuild the **tidx_name** signature text index for the table **tb_song** with a new total text size.

```
dmSQL> DROP TEXT INDEX tidx_name FROM tb_song;
dmSQL> CREATE TEXT INDEX tidx_name FROM tb_song(name) TOTAL TEXT SIZE 60 MB;
```

Boolean Text Search

Not only can the MATCH operator search a simple text pattern, but also complex Boolean operations.

A user can specify the following Boolean characters in a search pattern:

'&' – AND

'|' – OR

'-' – EXCLUDE

'(' – Left bracket

')' – Right bracket

The precedence of Boolean characters is: bracket > EXCLUDE = AND > OR. When a MATCH pattern contains Boolean characters, all the other characters between Boolean characters are processed as simple search patterns. For example, if the MATCH pattern is "coffee | tea | apple juice", then the search pattern includes "coffee", "tea" and "apple juice".

➲ **Example 1**

To search for documents that contain 'love' and 'friend':

```
dmSQL> SELECT * FROM tb_song WHERE name MATCH 'love & friend';
```

➲ **Example 2**

The following searches the documents that contain 'love' or 'friend' but do not include 'endless love'.

```
dmSQL> SELECT * FROM tb_song WHERE name MATCH '(love | friend) - endless love';
```

➲ **Example 3**

SQL syntax Boolean operators, such as AND and OR, can be used to get the same results as the MATCH pattern's Boolean operators. However, it has lower performance since only the last part of the search pattern uses the text index. For example, the following SQL command will only apply the text index scan when searching for the string 'friend', and will use a standard non-indexed search for the string 'love':

```
dmSQL> SELECT * FROM tb_song WHERE name MATCH 'love' AND name MATCH 'friend';
```

Fuzzy Search

Sometimes users would like to search imprecise patterns. If only exact phrases are allowed, the query 'William Clinton' will not find 'William Jefferson Clinton' and vice versa. A Boolean expression like 'William & Clinton' may return many irrelevant results. DBMaker provides a fuzzy search feature allowing users to perform imprecise queries without receiving too many irrelevant results.

A phrase led by a '?' (Question mark) will be evaluated as a fuzzy expression, e.g., '?intel pentium'. Words used for a search in a fuzzy expression can be separated by up to four words in the target text. For example, '?intel pentium' will find 'Intel will release its 1GHz Pentium III processor', and '?amd k7 athlon''AMD has renamed its K7 processor as Athlon'.

A number of words in the query may be missing from the result set. For example '?William Jefferson Clinton' can find 'William Clinton' and 'William J Clinton', but the first word of the query must appear; the query '?William Clinton' will not find 'Bill Clinton'.

Fuzzy expressions can be combined with other text Boolean operations.

➲ **Example**

```
dmSQL> SELECT content FROM tb_document WHERE content MATCH '?intel pentium & ?amd
k6';
dmSQL> SELECT title FROM tb_document WHERE title MATCH 'al gore | ?george bush';
```

The phrase in a fuzzy expression cannot contain any other operators. Thus the expression '?intel pentium & ?amd k6' is evaluated as '(?intel pentium) & (?amd k6)', and '?(intel & pentium)' generates an error.

Near logic full-text search

A fuzzy match allows users to perform inexact queries without receiving many irrelevant returns. A near match search is similar to a fuzzy search, but more exact. It ensures that all words in the query string appear in the text. A phrase led by a '~'(tilde mark) will be evaluated as a near expression. For example, '?amd sales 1ghz athlon' will find 'AMD announced quarterly sales of its 1ghz Athlon chip', but not, 'AMD announced quarterly sales of its Athlon chip'.

Near match search expressions can be combined with other text Boolean operations.

➲ **Example**

```
dmSQL> SELECT content FROM tb_document WHERE content MATCH '~intel pentium & ~amd
k6';
dmSQL> SELECT title FROM tb_document WHERE title MATCH 'al gore | ~george bush';
```

Fuzzy/Near Logic Matching Rules

The following four rules apply to the matching of a query string to the result string.

1. The first word of the query must appear, e.g., the query '?William Clinton' does not find 'Bill Clinton'.

2. Words can be separated by a preset number of words ("proximity"), e.g., '?intel pentium' will find 'Intel will release its 1 GHz Pentium III processor', and '?amd k7 athlon''AMD has renamed its K7 processor as Athlon'. Currently the number of additional words in the matched result set between words in the query string can be no greater than 4.

3. A number of words in the query may be missing from the result set, e.g., '?William Jefferson Clinton' can find 'William Clinton' and 'William J Clinton'. The maximum allowed number of missing words is determined by the formula:

max_miss = num_words - round(num_words × threshold).

The current threshold is 0.75.

4. All words in the query must appear in the original order, e.g., '?amd 1ghz k7 athlon' will find 'AMD will announce 1 GHz Athlon', but not 'AMD Athlon, formerly known as K7'.

A phrase led by a '~' (tilde mark) will be evaluated as a near expression, e.g., '~intel pentium'. Near search is a special case of fuzzy search that meets the rules 1, 2 and 4 above but does not allow for missed words.

User-Defined Stopword

As opposed to a keyword-based system, full-text retrieval software indexes every word in a document, with the exception of stopwords. Stopwords are terms that full-text retrieval software is programmed to ignore during the indexing and retrieval processes, to prevent the retrieval of extraneous records. Generally, a stopword list includes articles, pronouns, adjectives, adverbs and prepositions (e.g., the, they, very, not, of) that are common in the English language. You can also apply this rule to the Chinese language or any double-byte-encoded text, for example: 的、呢、啊 and 哈.

Search Path for Stopword List

DB_StpWd = <string>

This keyword indicates the name of the stopword list definition file that is put in the *shared/stopword* subdirectory of DBMaker's installation directory. The stopword list definition file is a pure text file, which would affect the text index result in DBMaker. This keyword is used when the database creating and retrieving text index. Without this keyword, database search pre-defined stopword list definition based on LCode.

default value:

DB_LCode	Stopword List Definition
0 English (ASCII)	en.tab
1 Traditional Chinese (BIG5)	tw.tab
2 Japanese (Shift JIS + Half Corner)	jp.tab

3 Simplified Chinese (GB)	cn.tab
4 Latin1 code (ISO-8859-1)	en.tab
5 Latin2 code (ISO-8859-2)	en.tab
6 Cyrillic code (ISO-8859-5)	en.tab
7 Greek code (ISO-8859-7)	en.tab
8 Japanese code（EUC-JP）	jp.tab
9 Simplified Chinese (GB18030)	cn.tab
10 Unicode (UTF-8)	en.tab

valid range: file name of the user-defined stopword list definition file

see also: **DB_LCode**

where to use: server side (only for creating and searching text index)

Default Stopword List

- If you do not specify any configuration, DBMaker should load default stopwords that are in a pre-defined file and based on LCODE. This feature can be used by users of previous versions of DBMaker.
- DBMaker searches the pre-defined file in local directory, and installs the directory.

User Defined Stopword List

- You can specify a stopword list through the configuration file **DB_StpWd**. DBMaker loads the file when you create a text index or retrieve objects from the text index.
- DBMaker searches the file in local or user specified directory, and installs the directory.

Disable Stopword List

There are two ways to disable a stopword list:

- Rename or remove the pre-defined file
- Define a non-existing stopword list in the configuration file

6.8 Managing Memory Tables

Memory tables, for almost all intents and purposes, function in the same manner as a permanent table in DBMaker. The differences lie in the fact that memory tables are temporary tables, their life cycle being connection based. This means that a memory table exists only as long as long as there is a connection to the database. When a user severs their connection to the database, when logging out of the system to clock out for the day for example, the memory table and it's contents will be lost to the user. Memory tables are only visible during their connection to the database. Once connection to the database is lost, they are no longer visible. Unlike a permanent table, memory tables are only stored in the memory of the machine that created them. They can not be used by a group and they can only have data selected or inserted, they do not support updating or delete data functions. Memory tables do support the transaction controls: commit, rollback, define savepoint, rollback to savepoint, and internal savepoint.

To create a memory table use the dmSQL syntax CREATE MEMORY TABLE. Details on syntax and usage of the SQL command CREATE MEMORY TABLE can be found in the *SQL Command and Function Reference*.

➲ **Example**

To create memory table **tb_memory**:

```
dmSQL> CREATE MEMORY TABLE tb_memory (id INT, name CHAR(10), brithday DATE);
```

Hash Index Management

Memory tables are stored in the memory of the machine that created it, for this reason memory tables do not support the B-tree structure of other table types. To help users when using memory tables users are able to create hash indexes on memory tables. Hash indexes can only be created on memory tables. The benefit of a hash index is that users have very quick access to data stored in the hash index. Hash indexes also improve equal expression and equal join performance. To create a hash index on a table users can use the CREATE HASH INDEX *index_name* ON *table_name* (*column_name*, …) [*bucket n*]; where index name is the name of the hash index being created, table name is the name of the memory table, column name is the name of the column in the memory table being effected (this value cannot specify asc/desc columns) and bucket n sets the array size for the hash table being created. For example, with the memory table created, a hash index **hidx**, can be made on memory table **tb_memory**, using columns **id** and **name** with an array size of 31.

➲ **Example**

To create hash index **hidx** on memory table **tb_memory** from the previous example:

```
dmSQL> CREATE HASH INDEX hidx ON tb_memory (id, name) bucket 31;
```

6.9 Managing Data Integrity

Applying constraints, or rules, to ensure the data meets certain criteria, can ensure the integrity of data. For example, verifying that an input value for a particular data item is within the correct range of values, e.g., a new employee's age must be between 16 and 90, is an example of data integrity.

In general, the different types of data integrity applicable to tables include those described in the following subsections.

Not Null

By default, all columns in a table allow NULL values. NOT NULL indicates that NULL values are not permissible in a column defined with the NOT NULL keyword.

Unique Indexes

Unique indexes, mentioned in section 6.5, *Managing Indexes*, can be used to ensure no two rows of a table have duplicate values, except NULL values, in a specified column or set of columns.

Unique Constraints

A UNIQUE constraint may be set on a column, a set of columns, or an entire table. The UNIQUE constraint ensures that every row in a column has a different value. No row may have the same value in the column or columns that a UNIQUE constraint is placed on.

➲ **Example**

To create a table with the UNIQUE constraint on column **Name**:

```
dmSQL> CREATE TABLE tb_student (Name CHAR(50) CONSTRAINT u UNIQUE, mathematics
SMALLINT);
```

Check Constraints

A CHECK constraint on a column or set of columns requires that a specified condition be true for every row of the table. If an INSERT or UPDATE statement is issued and the condition of the CHECK constraint is evaluated as false, the statement will fail.

In general, a CHECK constraint can be defined on a column (column constraint) or set of columns (table constraint).

Column Constraints

A *column constraint* is defined on a specific column and does not affect the other columns of the same table. When inserting a new row or updating an existing row, each column constraint is evaluated.

Table Constraints

A *table constraint* is defined on a set of columns. When inserting a new row or updating an existing row, the table constraint is evaluated after, all column constraints are evaluated as true. Only after the table constraint is also evaluated as true will the statement be processed.

➲ **Example 1**

To create a table with column and table constraints:

```
dmSQL> CREATE TABLE tb_student (mathematics SMALLINT
                                   CHECK VALUE >= 0 AND VALUE <= 100,
                               chemistry SMALLINT
                                   CHECK VALUE >= 0 AND VALUE <= 100,
                                   CHECK mathematics + chemistry <= 200);
```

➲ **Example 2**

To create a table with column and table constraints using standard SQL99 syntax:

```
dmSQL> CREATE TABLE tb_student (mathematics SMALLINT
               CONSTRAINT con_math CHECK VALUE >= 0 AND VALUE <= 100,
                               chemistry SMALLINT
               CONSTRAINT con_chem CHECK VALUE >= 0 AND VALUE <= 100,
               CONSTRAINT con_sum CHECK mathematics + chemistry <= 200);
```

The keyword VALUE is used to represent the value of the column in column constraints, but the columns names are used to represent the values of the columns in a table constraint.

Primary Keys

A table can have one *primary key*, which includes a column or a group of columns with unique values to identify each row. A primary key is similar to a unique index except that its columns cannot contain NULL values. When a user creates a primary key, DBMaker will create a unique index called **PrimaryKey** on the table. After creating a table, primary keys may modified or added as long as all columns to be in the primary key contain unique, non-null values. A primary key may be added to a table or modified by using the ALTER TABLE statement. Furthermore, a primary key added to a table or modified in this fashion may be stored on a different tablespace from the table.

Creating Primary Keys

➲ **Example 1**

To create a table with its primary key on the ID column:

```
dmSQL> CREATE TABLE tb_student (
          ID      INTEGER  PRIMARY KEY,
          name    CHAR(30),
          nation CHAR(20)
        );
```

Example 2

To create a table with a compound primary key on the **ID** and **name** columns on tablespace **ts_reg**:

```
dmSQL> CREATE TABLE tb_student (
          ID      INTEGER,
          name    CHAR(30),
          nation CHAR(20),
          PRIMARY KEY (ID, name)
        ) in ts_reg;
```

Example 3

To add a primary key to the **tb_student** table:

```
dmSQL> ALTER TABLE tb_student PRIMARY KEY (ID , name);
```

Example 4

To add a primary key PK1 to the **tb_student** table in tablespace **ts_reg** using SQL99 standard syntax:

```
dmSQL> ALTER TABLE tb_student ADD CONSTRAINT PK1 PRIMARY KEY (ID , name) IN
ts_reg;
```

Example 5

To create a table with its primary key on the **ID** column using SQL99 standard syntax:

```
dmSQL> CREATE TABLE tb_student (
          ID      INTEGER  CONSTRAINT pk1 PRIMARY KEY,
          name    CHAR(30),
          nation CHAR(20)
        );
```

DROPPING PRIMARY KEYS

A user can drop a primary key when it is no longer necessary. Before dropping the primary key, all foreign keys that refer to that primary key should be dropped.

➲ **Example**

To drop the primary key for the **tb_student** table:

```
dmSQL> ALTER TABLE tb_student DROP PRIMARY KEY;
```

Foreign Keys (Referential Integrity)

A column in a table containing the same values as the primary key from another table is known as a *foreign key*. A foreign key denotes the relationship between the two tables. A user can create a foreign key on a column or a group of columns in a table, and use it to reference a column or group of columns from another table. The referenced columns should be a primary key or a unique index, and cannot contain NULL values.

Referenced columns must already contain the key values being inserted into a new row for the foreign key table. If they are not present, the user will not be allowed to insert the row. In addition, all key values in the foreign key table must be deleted before deleting the key values in the referenced table.

A user can create or drop a primary key or foreign key whenever it is necessary. DBMaker will check the uniqueness of a primary key when it is created. DBMaker will also check whether all the key values already exist in the referenced table when a foreign key is created.

CREATING FOREIGN KEYS

A foreign key is used to refer to another table by specifying the referencing and referenced columns. Both the referencing and referenced columns should be mapped to each other; their schema should be the same. The mapping columns should be the same type and length. The referenced columns (specified by the primary key or unique index) should be NOT NULL, but the referencing columns (specified by the foreign key) can be NOT NULL or NULL. If the referenced column(s) are not specified, the primary key on the referenced table is regarded as the referenced column(s). Foreign keys may be created using the JDBA Tool Create foreign key wizard or the dmSQL FOREIGN KEY option.

➲ **Example 1**

To create a foreign key **f1** for the **tb_salary** table, referencing the **tb_staff** table with a compound primary key for its **ID** and **name** columns:

```
dmSQL> ALTER TABLE tb_salary FOREIGN KEY f1(ID, name) REFERENCES tb_staff;
```

➲ **Example 2**

Alternatively, specify the foreign key while creating the **tb_salary** table*:*

```
dmSQL> CREATE TABLE tb_salary (
         ID    INT,
         name  CHAR(30),
         basepay INT,
         bonus   INT,
         tax INT,
       FOREIGN KEY f1 (ID, name) REFERENCES tb_staff);
```

➲ **Example 3**

Using SQL99 standard syntax, specify foreign key **f1** while creating table **tb_example**:

```
dmSQL> CREATE TABLE tb_example (
       c1 int,
       c2 int CONSTRAINT f1 REFERENCES tb_other (c1) ON DELETE SET NULL);
```

If a primary key exists for the **tb_staff** table, a foreign key can be made for another table to refer to it without specifying the referenced columns.

DROPPING FOREIGN KEYS

If the relationship defined by a foreign key is not necessary, drop it using the JDBA Tool or the dmSQL DROP FOREIGN KEY command.

➲ **Example**

To drop a foreign key from the **tb_salary** table:

```
dmSQL> ALTER TABLE tb_salary DROP FOREIGN KEY f1;
```

6.10 Managing Serial Numbers

DBMaker provides a feature to automatically generate serial numbers. This feature is especially useful in multi-user environments for generating and returning unique sequential numbers without the overhead of disk I/O or transaction locking.

Serial numbers are signed 32-bit integers in DBMaker. A table can only have one column containing the SERIAL data type for generating serial numbers.

A user can specify the starting number for the SERIAL type column in any table when creating a table. If the starting number for a SERIAL type column is not specified, it is set to 1.

To trigger DBMaker to generate a serial number, insert a new row and supply a NULL value for the serial column. If a user inserts a new row and supplies an integer value instead of a NULL value, DBMaker will not generate a serial number. If the supplied integer value is greater than the last serial number generated, DBMaker will reset the sequence of generated serial numbers to start with the supplied integer value. SERIAL type columns cannot be defined with default values or constraints.

Creating Serial Columns

A SERIAL type column may be created using the JDBA Tool or dmSQL. A serial column must be defined with the SERIAL, BIGSERIAL type keyword and an optional starting number.

➲ **Example**

To create a SERIAL type data column **ID** for the **tb_staff** table and specify its starting number as 1001:

```
dmSQL> CREATE TABLE tb_staff (nation CHAR(20) DEFAULT 'R.O.C',
                                  ID SERIAL(1001),
                                name CHAR(30) NOT NULL,
                            joinDate DATE DEFAULT CURDATE(),
                              height FLOAT,
                              degree VARCHAR(200)) IN ts_reg;
```

Generating Serial Numbers

DBMaker automatically generates serial numbers when a NULL is inserted into the SERIAL type column.

➲ **Example**

To insert a new row into the **tb_staff** table and generate a serial number for the column **ID** :

```
dmSQL> INSERT INTO tb_staff VALUES
              ('U.S.A', NULL, 'Jeff', 6.6, 'Director', NULL);
```

Retrieving Serial Numbers

DBMaker keeps the last generated serial number in the LAST_SERIAL column of the system table SYSCONINFO for each connection. After inserting a record containing a serial number, the serial number can be retrieved from LAST_SERIAL.

➲ **Example**

To get the serial number that has just been generated for the inserted record:

```
dmSQL> SELECT LAST_SERIAL FROM SYSCONINFO;
LAST_SERIAL
===========
        200
 1 rows selected
```

Resetting Serial Numbers

A user can reset the counter for a serial column. This allows a new sequence to be started in a serial column without having to modify the table.

➲ **Example**

To alter the serial counter value for the **tb_staff** table from its current value to 3000:

```
dmSQL> ALTER TABLE tb_staff SET SERIAL 3000;
```

6.11 Managing Domains

A *domain* is a type of integrity constraint used when defining a column. Domains specify the data type for the column, and may specify a default value or a value constraint. When a column is defined using a domain, it inherits the properties of the domain, (data type, default value, and value constraint), without requiring the user to specify them.

Specifying the default value and value constraint using domains achieves the same result as specifying them in a standard column definition. If a user specifies a default value for a column, it will override the default value specified in a domain.

Any value constraints specified in the column definition will be used in addition to the value constraints specified in the domain. If a user defines a column using a domain and specifies additional value constraints, the additional value constraints must not conflict with those defined in the domain.

DBMaker does not check for conflicting value constraints, so it may be possible to define value constraints that would not allow the user to enter any values. All data types supported by DBMaker except the SERIAL type can be used in domains.

Creating Domains

A domain is defined by a domain name, an optional default value, and an optional constraint. For example, a user might want to ensure that all columns dealing with some form of titles, (e.g., movie, CD, or videotape), have a data type of VARCHAR, are no more than 35 characters in length, and do not permit insertion of NULL values. Domains may be created using the JDBA Tool or the dmSQL CREATE DOMAIN statement.

➲ **Example 1**

The keyword VALUE is used to represent the value of the column defined on the domain. To create a specific domain that is used in the subsequent CREATE TABLE statements:

```
dmSQL> CREATE DOMAIN title_type VARCHAR(35) CHECK VALUE IS NOT NULL;
```

➲ **Example 2**

To define columns as in the CREATE TABLE statement:

```
dmSQL> CREATE TABLE movie_titles (title title_type, ..., ...);
```

CREATING DOMAIN WITH TEXT CONVERTER

A media type is a domain having specific characteristic with the media format. User can create domain using the TEXT CONVERTER syntax in CREATE DOMAIN clause. When user have specified the TEXT CONVERTER syntax on the domain, DBMaker uses the TEXT CONVERTER expression to convert the CLOB, NCLOB, BLOB or FILE data to pure text for creating a text index and PURETEXT() UDF. The TEXT CONVERTER function-name should contain only BLOB related argument type. The return type must be the CLOB or the NCLOB data type or DBMaker returns an error. Not more than **32,767** domains can be created with the TEXT CONVERTER syntax.

NOTE *User cannot specify expression, function without argument or function with more argument in the TEXT CONVERTER clause.*

➲ **Example**

To define a domain **MSWORDTYPE**:

```
dmSQL> CREATE DOMAIN MSWORDTYPE AS BLOB
       TEXT CONVERTER DOCTOTXT
       CHECK VALUE IS NULL OR CHECKMEDIAFORMAT(VALUE,'DOC') = 1;
```

To create a table **tb_MT** with domain **MSWORDTYPE**:

```
dmSQL> CREATE TABLE tb_MT (C1 MSWORDTYPE);
```

Dropping Domains

A domain can be dropped only when there are no columns referenced on it. Drop domains using the JDBA Tool or the dmSQL DROP DOMAIN statement.

➲ **Example**

To use the DROP DOMAIN statement:

```
dmSQL> DROP DOMAIN title_type;
```

6.12 Unloading and Loading Objects

Sometimes the user may need to save database data to an external text file. DBMaker provides the UNLOAD and LOAD commands just for this purpose. Objects that are unloaded from the database are not removed from the database; they are simply saved as one or more external text files. When an object is loaded onto a database, the schema of that object is also recreated.

Unloading Objects

Unload is a tool provided by dmSQL used to transfer the contents of a database to an external text file. After the unload procedure succeeds, dmSQL will produce two text files. One stores the script, with extension name **.s0**, to establish the database object and the other stores the BLOB data, with the extension name **.bn**.

There are eight options for the unload command: unload database, unload table, unload schema, unload data, unload project, unload module, unload procedure, and unload procedure definition. Unloading an object requires that the user have SELECT privilege on the object in question. For instance, if a user has SELECT privilege on a table, then only that user can unload the content of this table. Only a user with DBA, SYSDBA or SYSADM authority may unload the database.

UNLOAD [DB | DATABASE]

A user with DBA, SYSDBA or SYSADM authority may unload the content of a database to an external text file. This file includes information about security, tablespaces, definitions, indices, synonyms and data. For each database, dmSQL will generate at least two external files, one script, and one BLOB data.

➲ **Example 1**

```
dmSQL> UNLOAD DB TO empdb;
```

The name of the external text file is empdb. By default, dmSQL will create these files in the current working directory. In the above statement, there are at least two text files created, **empdb.s0** and **empdb.b0**. If the unloaded BLOB file **empdb.b0** exceeds the maximum size allowed by the operating system, dmSQL will sequentially generate files **empdb.b1**, **empdb.b2**, …, **empdb.bn**, up to a maximum number of 99. dmSQL will always generate one script file **emodb.s0**, with a maximum size limited by the operating system.

However, if a user use the command SET UNLOAD EXTERNAL '*connection_string*'(the format of *connection_string* is "DSN=<*db_name*>;UID=<*user_name*>;PWD=<*password*>;") before using the command UNLOAD DB TO *file_name*, dmSQL will not unload data into the scrip file namely empdb.s0, Instead, dmSQL will print "set external db '*connection_string*'" in empdb.s0, and unloading tables' data will be printed as "load external db from 'select * from *external_table_name*' into *local_table_name*". Please refer to the following example:

➲ **Example 2**

```
dmSQL> SET UNLOAD EXTERNAL 'DSN=DBSAMPLE5;UID=SYSADM;PWD=;';
dmSQL> UNLOAD DB TO empdb;
```

Here the scrip file empdb.s0 is as follows:

```
…
set external db 'DSN= DBSAMPLE5;UID=SYSADM;PWD=;';
create table Lauser1.Latb3 (
 c1  SMALLINT default null ,
 c2  FLOAT default null ,
 c3  DOUBLE default null ,
 c4  DECIMAL(10, 3) default null ,
 c5  CHAR(10) default null ,
 c6  BINARY(12) default null )
 in DEFTABLESPACE  lock mode page  fillfactor 100 ;
load external database from 'select * from Lauser1.Latb3' into Lauser1.Latb3;
create  index idx31 on Lauser1.Latb3 ( c1 asc ) in DEFTABLESPACE;
create  index idx32 on Lauser1.Latb3 ( c3 desc ) in DEFTABLESPACE;
create  index idx33 on Lauser1.Latb3 ( c5 asc ) in DEFTABLESPACE;
…
```

UNLOAD TABLE

The UNLOAD TABLE command unloads tables to an external file and will record the definition, synonyms, indices, primary key, foreign keys, and data of the table. Use the wild cards "_" and "%", which correspond to "?" and "*" in DOS in the owner and table name. The wild card "_" represents a character, and "%" represents a set of characters.

UNLOAD SCHEMA

The usage of this option is very similar to unload table. It can only unload the definition of a table; it cannot unload the data in a table. It uses the same wild cards as the UNLOAD TABLE option.

UNLOAD DATA

This option will unload all data from a table. It will not unload the definition of the table. UNLOAD DATA uses the same wildcards as the previous two options. Only users with the SELECT privilege on the unloaded table may execute the UNLOAD DATA command.

DBMaker 3.6 and later versions support an additional syntax for unloading data:

```
UNLOAD DATA FROM (select statement) TO file_name
```

If the select statement is a join, the projection columns must be from the same table, the following statement is executable. DDL commands, delete, insert, or updates are not permitted.

➲ **Example 1**

Valid syntax:

```
dmSQL> UNLOAD DATA FROM (SELECT t1.c1, t1.c2 FROM t1, t2 WHERE t1.c1= t2.c1) TO
f1;
```

➲ **Example 2**

Illegal syntax:

```
dmSQL> UNLOAD DATA FROM (SELECT t1.c1, t2.c1 FROM t1, t2 WHERE t1.c1 = t2.c1) TO
f1;
```

No aggregate or built-in functions are permitted in the projection columns.

➲ **Example 3**

Illegal syntax:

```
dmSQL> UNLOAD DATA FROM (SELECT avg(c1) FROM t1) TO f1;
dmSQL> UNLOAD DATA FROM (SELECT now()FROM t1) TO f1;
```

Views and synonyms are permitted.

➲ **Example 4**

Valid syntax:

```
dmSQL> UNLOAD DATA FROM (SELECT * FROM s1 WHERE c1 > 10) TO f1;
dmSQL> UNLOAD DATA FROM (SELECT * FROM v1 WHERE c1 < 10) TO f1;
```

UNLOAD PROJECT

This option allows a user to unload an ESQL/C project to an external text file.

UNLOAD MODULE

This option allows a user to unload a module to an external file.

UNLOAD [PROC | PROCEDURE]

This option allows a user to unload the stored procedures to an external file.

UNLOAD [PROC DEFINITION | PROCEDURE DEFINITION]

This option allows a user to unload the definition of the stored procedure to an external text file.

➲ **Example 1**

The following will unload the table **e tab** for the current user; if there are any blanks in the table name add double quotes:

```
dmSQL> UNLOAD TABLE FROM "e tab" TO empfile;
```

➲ **Example 2**

The following will unload all tables with the names starting with **emp** for the SYSADM owner, for example, **emptab**, **empname**, … :

```
dmSQL> UNLOAD TABLE FROM SYSADM.emp% TO empfile;
```

➲ **Example 3**

The following will unload the schema of all tables with the name **ktab**:

```
dmSQL> UNLOAD SCHEMA FROM %.ktab TO kfile;
```

Unload the table with names containing wild cards. Use the escape character "\", or double quotes on the name.

➲ **Example 4**

The following commands will unload data from a table named **abc%**:

```
dmSQL> UNLOAD DATA FROM abc\% TO abcfile;
dmSQL> UNLOAD DATA FROM "abc%" TO abcfile;
```

Loading Objects

The LOAD command is a tool provided by dmSQL, and is used to transfer a database object that has already been unloaded to a text file, into the database. There are seven options: load database, load table, load schema, load data, load project, load module, and load procedure. A file must be unloaded and loaded with the same option. For example, load a database from a text file that was unloaded with the database option. When loading a text file, set the number of commands <*n*> to automatically commit the transaction. The default is 1000. The size of <*n*> will affect whether the transaction succeeds or not and the loading speed. The journal will fill easily with a large <*n*> value and could cause the transaction to fail. A small <*n*> value will increase the number of transactions committed and slow down the loading speed. If there are errors occurring during the loading procedure, an error messages will be recorded in a log file, which the system will use to undo executed commands. The log file is stored in the same directory as the external text file being loaded and does not stop the loading procedure.

LOAD [DB | DATABASE]

Use the LOAD [DB | DATABASE] command to transfer the contents of a database to a new database. First, unload the database to transfer to an external text file. Next, use the LOAD DB command to load the contents of the database from the text file. Before loading a database, create a new one. The name of the new database can be different from the old one. Only a DBA, SYSDBA or a SYSADM may execute this command.

However, if a user use the command SET UNLOAD EXTERNAL '*connection_string*'(the format of *connection_string* is "DSN=<*db_name*>;UID=<*user_name*>;PWD=<*password*>;") before using the command UNLOAD DB TO *file_name*, dmSQL will not unload data into the scrip file. Therefore, when a user loads the database with this scrip file, dmSQL will connect to ODBC driver manager's data source, reads data from it and then save data into the local database directly. dmSQL uses "set external [database|db] '*connection_string*'" in the scrip file to connect external database, if fails, an error will be returned. dmSQL only keeps the last external database connection, and therefore close previous external database connections if a new one is set. In addition, because there is no disconnect command, the external database will be disconnected only when dmSQL tool is closed.

The database runs in normal mode if LOAD DB is set to SAFE. The load utility rollbacks to the last committed command if an error occurs during loading, the error messages are displayed in the screen, and writes to the load utility's log file. When using the set LOAD DB in fast mode, the rule for loading the utility in DBMaker versions earlier than 3.6, will make the whole load procedure work under the no journal mode. Setting LOAD DB in fast mode will speed up the load utility, but it will make the database shut down in no journal mode if any error occurs. For example, suppose that the load file has tablespace creation but it is not specified in the dmconfig.ini file. If LOAD DB is set to use the safe option, the following error message, "ERROR(8002): [DBMaker] keyword entry is required for configuration file", will be reported and then the load command will rollback. If LOAD DB is set to use the fast option, then the following error message occurs, "ERROR(30017), [DBMaker] errors occurred in no-journal mode, shut down database". The default option is SET LOADDB SAFE.

➲ **Example**

The following set option for LOAD DB has been added to versions above DBMaker 3.6:

```
SET LOADDB [safe | fast]
```

LOAD TABLE

This option permits loading the contents of a table, including schema and data, from a text file. When loading a table from a text file, make sure that the table name is unique.

LOAD SCHEMA

This option allows users to load the schema, not including the data, from a table contained in a text file. When loading a table schema from a text file, ensure that the table name is unique.

LOAD DATA

A corresponding table must exist when loading data from an external text file. In versions earlier than 3.6 when the errors occur during the LOAD DATA procedure, it will rollback to the last committed command.

➲ **Example**

DBMaker 3.6 and later versions support the following options:

```
SET LOADDATA SKIP [error] | STOP [on error]
```

If LOAD DATA SKIP ERROR is set then the following error messages will be skipped during the loading of data:

ERROR(401)	unique key violation
ERROR(410)	referential constraint violation: value does not exist in parent key
ERROR(6521)	table or view does not exist

ERROR(6002)	syntax error
ERROR(6015)	incomplete SQL statement input

The errors will be skipped and the load utility will resume execution of subsequent commands. The above errors are the most common errors to occur during loading of data. When LOAD DATA STOP or STOP ON ERROR is set, the whole LOAD command will be rolled back if an error occurs. The default value for this option is LOAD DATA SKIP ERROR. All the error messages that occur during the loading of data will be written into the log file.

LOAD MODULE

This option allows a user to load a module from an external text file.

LOAD PROJECT

This option allows a user to load an ESQL/C project from an external text file.

LOAD [PROC | PROCEDURE]

This option allows a user to load a stored procedure from an external text file.

➲ **Example 1**

The following command loads the database from a file named **empdb**, and commits it automatically every 100 commands during loading. The system will generate a log file named **empdb.log** in the same directory:

```
dmSQL> LOAD DB FROM empdb 100;
```

➲ **Example 2**

The following command will load a table from a file named **empfile**, and it will commit automatically every 50 commands during loading:

```
dmSQL> LOAD TABLE FROM empfile 50;
```

➲ **Example 3**

The following command will permit the loading of data from an external data file named **datafile** and will commit automatically every 1,000 commands using the default setting:

```
dmSQL> LOAD DATA FROM datafile;
```

6.13 Browsing System Catalogs

DBMaker keeps detailed information on all schema objects in the *system catalog* tables. For more information on system catalog tables, see *System Catalog Reference.*

SCHEMA OBJECT INFORMATION	SYSTEM CATALOG TABLE NAME
Tables	SYSTABLE
Columns	SYSCOLUMN
Views	SYSVIEWDATA
Synonyms	SYSSYNONYM
Indexes	SYSINDEX
Domains	SYSDOMAIN
Serial numbers	SYSCONINFO
Table constraints	SYSTABLE
Column constraints	SYSCOLUMN
Domain constraints	SYSDOMAIN

Table 6-2: Schema information in the System Catalog tables

6.14 Calculating the Space Required

As stated in previous sections, only tables and indexes occupy physical disk space. To manage disk space, estimate the size of each object and decide which tablespace each object will belong to before creating them. In the estimation phase, the database administrator must have a clear picture of how to construct tablespaces using tables and how much hardware will be required to support the database in the future.

Generally, tables that are split between several tablespaces will get higher performance than tables in a single large tablespace. Conversely, many small tablespaces are harder to manage.

How to Estimate the Size of a Table

The following formulas show how to estimate the size of a table and the size of an index:

table size = row size × number of rows × 1.05

index size = key size × number of rows × 1.20

These two formulas are used to estimate the size needed for a tablespace by adding the size of all tables and indexes in it. In the above formulas, 1.05 and 1.20 are estimates of the resource overhead used to calculate the system resources required. The row size and key size contain the internal record header size. The following subsections show how to calculate the size of a row and a key.

Row Size

The storage size of a row, excluding BLOB data, cannot exceed 3996 bytes and consists of the space required for data storage and an internal record header.

The size of an internal record header is equal to:
internal record header size = (number of columns + 1) × 4

Each data type has space requirements:

TYPE	COLUMN LENGTH
BIGINT	8
BINARY(n)	N
BIGSERIAL	8
CHAR(n)	N
SMALLINT	2
INTEGER	4
FLOAT	4
SERIAL	4
DOUBLE	8
DECIMAL(p,s)	[(p + 1) / 2] + 2
TIME	4
DATE	4
TIMESTAMP	11
OID	16
VARCHAR(n)	1-3992n
FILE	20
LONG VARBINARY	48 + X
LONG VARCHAR	48 + X

Table 6-3: Data Types and sizes

NOTE *VARCHAR is a variable-length data type that takes any character that can be entered from the keyboard. A BLOB type column (LONG VARCHAR or LONG VARBINARY) occupies at least 48 bytes in the data file and the actual data is stored in the BLOB file or a data file. For more detailed information, refer to Chapter 7, Large Object Management. If the value in a column is NULL, it does not occupy any space.*

➲ **Example**

To create a table with five columns defined:

```
dmSQL> CREATE TABLE tb_staff(ID INTEGER NOT NULL,
                            name CHAR(30) NOT NULL,
                          height FLOAT,
                          degree VARCHAR(200),
                         picture LONG VARCHAR);
```

After issuing this command, insert rows into the table and calculate the size of the record:

(3001, "Jeff Yang", 175.5, "Stanford PhD.", [pic1]) where **pic1** is an image.

DATA ITEM	**TYPE**	**SIZE**
ID	integer	4 bytes
name	char	30 bytes
height	float	4 bytes
degree	varchar	13 bytes
picture	long varchar	48 bytes
row header	—	24 bytes
	Total	**123 bytes**

= (4 + 30 + 4 + 13 + 48) + (5 + 1) × 4 = 123 bytes

(3002, "George Wang", 180.0, "NCTU Ms.", NULL)

DATA ITEM	**TYPE**	**SIZE**
ID	integer	4 bytes
name	char	30 bytes
height	float	4 bytes
degree	varchar	8 bytes

DATA ITEM	TYPE	SIZE
picture	long varchar	0 bytes
row header	—	24 bytes
	Total	**70 bytes**

= (4 + 30 + 4 + 8 + 0) + (5 + 1) × 4 = 70 bytes

DBMaker will verify that the row size does not exceed MAXTUPLEN[1] bytes when inserting or updating rows. When creating a table, DBMaker also verifies that the smallest possible row size does not exceed MAXTUPLEN bytes.

The smallest row size in the above example can be calculated as follows:

minimum row size = (4 + 30 + 0 + 0 + 0) + (5 + 1) × 4 = 58 bytes

The minimum row size does not exceed MAXTUPLEN bytes, so DBMaker will allow this table to be created.

KEY SIZE

Key storage size is made up of the space required for data storage in the index columns and an internal record header. It also requires an extra 16 bytes for an internal row identifier. The internal row identifier also requires 4 bytes in the record header.

The size of the internal record header is equal to:

Internal record header size = (no. of columns + 1 + 1) × 4

For example, if an index is created on a single column with the SMALLINT type, the size of each key will be:

key size = 2 + 16 + (1 + 1 + 1) × 4 = 30 bytes

[1] MAXTUPLEN : The value is 3968 in 4KB page size, 8064 in 8KB page size, 16256 in 16KB page size and 32640 in 32KB page size, respectively.

In this case, two bytes are used by the data in the key column, 16 bytes are used for the internal row ID for each key, and 12 bytes are used for the record header.

Estimating the Size of Tablespaces and Tables

The following example demonstrates how to estimate the size of a tablespace and its tables. Assume there is a tablespace that contains three tables **A**, **B**, and **C**, and one index **D** created for table **A**. Columns in table **A** are defined as INTEGER, and CHAR(10). Columns in table **B** are defined as SMALLINT, CHAR(10), FLOAT, and VARCHAR(200). Columns in table **C** are defined as SMALLINT, INTEGER, and LONG VARCHAR. Index **D** is created on the first column in table **A**. Table **A**, table **B** and table **C** consist of 1500, 3000, and 250 rows respectively.

The row and key sizes for this database can be calculated as shown below. Suppose the average length of the VARCHAR column in table **B** is 80 bytes, and the size of each BLOB column in the data file in table **C** is 48 bytes:

In table **A**:	row size = (4 + 10) + 3 × 4 = 26 bytes
In table **B**:	row size = (2 + 10 + 4 + 80) + 5 × 4 = 116 bytes
In table **C**:	row size = (2 + 4 + 48) + 4 × 4 = 70 bytes

If the average size of each BLOB item in table **C** is 9000 bytes, then specify the BLOB frame size to be 11KB. See Chapter 7, *Large Object Management* for more information about BLOB data.

Index **D**:	keysize = 4 + 16 + 3 × 4 = 32 bytes

The table sizes for this database can be calculated as shown below. Note that the size of table **A** also includes the size of index **D**.

Table **A**:	table size = (26 × 1500 × 1.05) + (32 × 1500 × 1.2) = 98550 bytes
Table **B**:	table size = 116 × 3000 × 1.05 = 365400 bytes
Table **C**:	table size = 70 × 250 × 1.05 = 18375 bytes

In the BLOB file, the size of table C is 250 frames (every row needs a frame).

After examining the figures above, create a tablespace with at least one data file (482325 bytes) and one BLOB file (250 frames with an 11 KB frame size) to store the tables and index shown above.

Estimate the size of a tablespace when creating it to avoid needing to add or enlarge files later.

6.15 Checking Database Consistency

DBMaker includes several commands that a user with DBA privilege can use to check the consistency of a database. Examples of database consistency include an index that has a key but does not exist in the table, or a key that exists in a foreign table but does not exist in the parent table. DBMaker supports six commands to check different levels of consistency. These commands are time consuming when the database is large and they will take locks, administrators should only use them when necessary.

Checking Indexes

DBMaker allows a user with privilege on the index in question to check an index and its relationship to a table. It checks if the index structure (i.e., B-tree) is correct, if the data is in order, and if the index keys exactly match the data records.

If an index seems to have a problem, use this command to verify that a problem exists. If DBMaker finds inconsistencies in the index, drop and rebuild the index to fix it.

➲ **Example**

To check the index consistency for the index **idx_desc** in the **tb_staff** table:

```
dmSQL> CHECK INDEX tb_staff.idx_desc;
```

Checking Tables

DBMaker allows a user to check all records, indexes, and BLOB data associated with a table and the relationship between foreign and parent tables, given that the user has privilege on those objects. If there is any inconsistency in a table, unload all records in the table, drop the table, recreate it, and then reinsert all records.

➲ **Example**

To check consistency for the **tb_staff** table:

```
dmSQL> CHECK TABLE tb_staff;
```

Checking Catalogs

DBMaker allows a user with DBA privilege to check the consistency of system tables. If the system catalogs have errors, the database may be seriously corrupted.

➲ **Example**

To check the consistency of the system catalogs:

```
dmSQL> CHECK CATALOG;
```

Checking Databases

DBMaker also allows a user with DBA privilege to check the whole database including the system catalogs and all tablespaces.

➲ **Example**

To check the consistency of an entire database:

```
dmSQL> CHECK DB;
```

If corruption exists and the database has been backed up, use the most recent backup to restore it. For more information, refer to Chapter 15, *Database Recovery, Backup, and Restoration*.

When the database has no backup and an index is corrupted, drop and recreate the affected indexes. If any other type of corruption has occurred, immediately backup the database including all data and journal files. Then try to shut down and restart the database, then run the CHECK commands again. After DBMaker automatically recovers from a crash, some types of corruption may be fixed. If any inconsistency remains, contact CASEMaker technical support to help fix the remaining problems with the database.

Checking Users' Files

DBMaker allows users to check their files when the database starts in the way of warm start. If users enable this function, DmServer checks all users' files to make sure they are still on the disk when the database starts in the way of warm start, if not, DBMaker gives a warning message to notice a user with DBA authority or higher to avoid operating the files which already have been moved. This warning message is recorded into the log file *DMEVENT.LOG*. Users can enable this function by setting **DB_ChkFl**.

NOTE *This function does not support single-user mode.*

6.16 Updating Statistics for Schema Objects

Outdated statistics values for schema objects (tables, indexes, columns) may cause the DBMaker optimizer to use an inefficient plan for a SQL statement. If users have inserted large amounts of data into the database after the last time the database administrator updated the statistics values, update the values again.

Statistics are only updated after the database has been started. Updating statistics also requires processor resources and will affect database performance. Selecting an interval and a time that does not interfere with peak table usage will prevent degradation of performance while still providing updated statistics.

When the database is running, a user can change the specified statistics value with the system stored procedure **SetSystemOption**.

➲ **Example 1**

To update the statistics values for all schema objects:

```
dmSQL> UPDATE STATISTICS;
```

➲ **Example 2**

To forcibly update the statistics values for all schema objects:

```
dmSQL> UPDATE STATISTICS ALL;
```

If a database is extremely large, it will take a lot of time to update statistical values for all of the schema objects. An alternative method is to update statistics on specific schema objects that have been modified since the last update, and set the sampling rate.

➲ **Example 3**

To update statistics for tables:

```
dmSQL> UPDATE STATISTICS table1, table2, user1.table3;
```

➲ **Example 4**

To update statistics for index **idx_desc** on **tb_staff**:

```
dmSQL> UPDATE STATISTICS tb_staff (index idx_desc);
```

➲ **Example 5**

To update statistics for tablespace **ts_reg**:

```
dmSQL> UPDATE TABLESPACE STATISTICS ts_reg;
```

➲ **Example 6**

To update statistics for indexes **idx_desc** and **idx_fill** on **tb_staff**:

```
dmSQL> UPDATE STATISTICS tb_staff (index idx_desc, idx_fill);
```

When the database is running, a user can change the specified statistics value with the system stored procedure **SetSystemOption**.

➲ **Example 7**

The following syntax is used to set update statistics sample to **60** when the database is running:

```
dmSQL> CALL SETSYSTEMOPTION('STSSP', '60');
```

Please refer to Chapter 18, *Performance Tuning* for more information about updating statistics and the SQL optimizer.

7 Large Object Management

A *Large Object* (LO) is any variable length data object, such as document text, images, sound, or video. DBMaker has a great deal of flexibility when dealing with large objects and provides an excellent mechanism for unstructured data.

DBMaker does not limit the number of LOs that can be in a table, and there is no aggregate size limit on LO columns. This means the capacity of each LO column can be up to 2 GB. DBMaker can use extensions to the SQL language to directly access Large Objects, eliminating the need for users to learn any special syntax. All access to LO columns is transparent in SQL statements, which makes using large objects easy to learn. Furthermore, users can input or output LO data to and from a file using SQL commands or the ODBC interface. A LO is always written to disk as a single unit. However, users can read all or part of a LO. The SELECT, UPDATE, INSERT and DELETE statements are permitted with LOs. LO items can only be used in Boolean expressions if users would like to test them for NULL values. DBMaker also provides the MATCH function for use with LOs to perform searches with pattern matching. The MATCH function is similar to the LIKE function except it only works on LO columns and does not permit the use of wildcard characters.

DBMaker does not permit the operation of arithmetic or string expressions on LO items, nor can the LO items be used in any of the following ways:

- With aggregate functions
- With the IN, ANY, EXIST or LIKE predicates

- With the GROUP BY clause
- With the ORDER BY clause

There are two kinds of LOs: Binary Large Objects (BLOBs), which are stored in database files, and File Objects (FOs), which are stored as external files on a host file system.

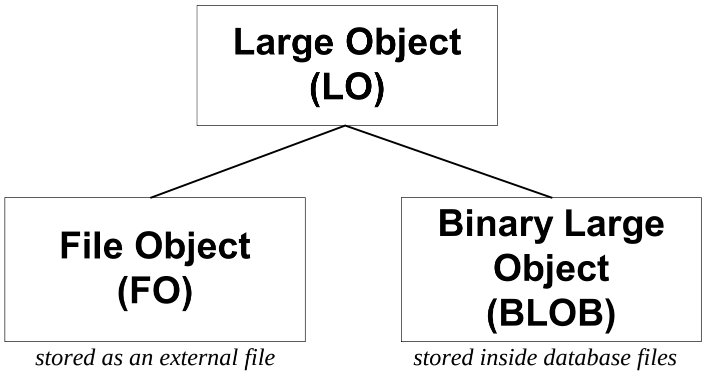

Large Object (LO)
File Object (FO)
Binary Large Object (BLOB)
stored as an external file
stored inside database files

Figure 7-1: Large Objects supported by DBMaker

A BLOB, stored in database files, can only be accessed through DBMaker and insists on the data integrity provided by DBMaker, such as transaction controls, logging and recovery. A BLOB can only be shared among tuples in the same table while updating records. However, a FO can be shared between tables in a database. In addition, when the data needs to be shared by the other non-database applications, using FOs will be more flexible.

7.1 Managing BLOBs

There are two types of BLOB items, LONG VARCHAR (or CLOB) and LONG VARBINARY. Data of the LONG VARCHAR type can consist of any kind of text data such as memos, long text, HTML source files, or program source files. The LONG VARBINARY data type can hold any kind of binary data including images, sound, spreadsheets and program modules.

A BLOB may be stored in a data file or a BLOB file, depending on its size. Although the format of a data file is fixed, the format of BLOB files in the database should be customized to obtain better performance and disk utilization.

The choice of BLOB logging is optional because it occupies a large amount of the journal space and can pull down performance. To save logging space and improve performance, the BLOB journal may be turned off. However, if BLOB logging is turned off, DBMaker will not ensure that the BLOB contents will be correct after the database has been restored from a backup. If the BLOB journal is turned on, make sure the journal file has enough space to accommodate the BLOB data.

Customizing BLOB Space

DBMaker automatically decides where to store BLOB data. If the size of a LONG VARCHAR or LONG VARBINARY column is small and the total length of a tuple does not exceed the limitation for the maximum tuple size, DBMaker will put the BLOB data in a column together with the other data in the database. This increases efficiency because the BLOB data is also fetched when DBMaker fetches a tuple.

When the total length of a data tuple exceeds the limitation of the maximum tuple size, DBMaker will store the BLOB data separately. In this situation, getting the BLOB data (called an indirect BLOB) requires two disk operations, one to fetch the data tuple, and one to fetch the BLOB data.

According to its size, an indirect BLOB may be stored in a DATA file or in a BLOB file in the same tablespace as the table. The data in an indirect BLOB column is stored in a data file when its size is equal to or less than 16,240[2] (in 16 KB page size) bytes, otherwise, it is stored in a BLOB file.

[2] This value is 3952 bytes in 4 K page size, 8048 bytes in 8 K page size, 16240 bytes in 16 K page size and 32624 bytes in 32 K page size, respectively.

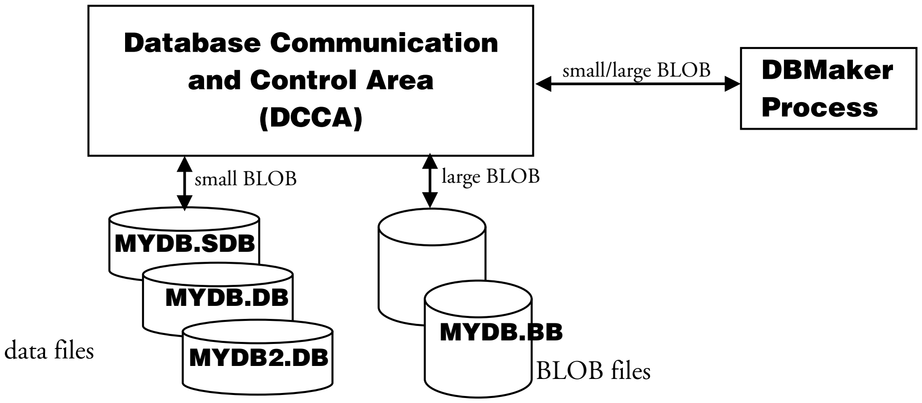


Figure 7-2: DBMaker accesses BLOB data through DCCA

Data files contain pages, and BLOB files contain frames. There are two major differences between pages and frames:

- There are four values to choose for page size: 4 KB, 8 KB, 16 KB or 32 KB. Page size is defined using the keyword **DB_PgSiz** when creating databases, while the size of a frame can be customized.
- A page can contain more than one tuple, but a frame can only contain a single BLOB.

The frame size of a BLOB file can be customized before database creation to increase performance and disk utilization. To customize the frame size, specify the value in kilobytes of the **DB_BfrSz** configuration keyword in **dmconfig.ini**. The default value of **DB_BfrSz** is *32*. Refer to Section 4.2, *Creating a Database* for more information on configuration parameters that must be set before database creation.

➲ **Example1**

To specify the BLOB frame size by adding a line to the **dmconfig.ini** file:

```
DB_BfrSz = 16                         ; BLOB frame size = 16K bytes
```

The valid range of **DB_BfrSz** is from 8 to 256.

The frame size of all BLOB files in a database is the same. Once a database is created, the BLOB frame size cannot be changed from its initial setting. DBMaker will keep this value in the database system information table. When the database is restarted, DBMaker will get the original value from the system information page and ignore the **DB_BfrSz** keyword in **dmconfig.ini**.

➲ **Example2**

To query the SYSINFO system table for the frame size:

```
dmSQL> SELECT INFO, VALUE FROM SYSINFO WHERE INFO = 'FRAME_SIZE';

              INFO                            VALUE
================================ ================================
FRAME_SIZE                       16384

1 rows selected
```

Determining the frame size is a trade-off between disk utilization and performance. If entire BLOBs are frequently retrieved, adjusting the frame size to contain the entire BLOB will result in better performance because only one disk access is required. However, there may be large variations in the size of the BLOB data. If the frame size is set large enough to contain the largest BLOB, it may waste disk space, as other frames that contain smaller BLOBs will contain unused disk space.

Alternatively, the frame size is only large enough to contain the smallest BLOBs, performance will be degraded when fetching larger BLOBs that are stored in multiple frames.

Each frame contains a header to record frame information. If the frame size is 8 KB, for example, the space occupied by the BLOB will be less than 8192 bytes. About 1.8 KB is reserved to store information (such as where other frames are) for each BLOB item, so the usable space in the first frame of a BLOB is much less than the size of the entire frame. Thus, if the actual size of a BLOB is 8192 bytes, it will occupy two frames: the first 6.2 KB of the BLOB are stored in the first frame and the remaining bytes of the BLOB are stored in the second frame.

A group contains an BE page and NBE[3] PE blocks, the BE page is a PAGE_SIZE KB data page. Then each PE blocks contains NPE[4] + 1 frames. The first frame is a PAGE_SIZE[5] -KB PE page. The remaining NPE frames are for data, and their size is determined by **DB_BfrSz**.

[3] NBE specifies the number of PE blocks is controlled by a BE page. The value is 2004 in 4 KB page size, 2026 in 8 KB page size, 2716 in 16 KB page size and 2723 in 32 KB page size, respectively.

[4] NPE specifies the number of pages is controlled by a PE page. The value is 165 in 4 KB page size, 333 in 8 KB page size, 671 in 16 KB page size and 1347 in 32 KB page size, respectively.

[5] PAGE_SIZE is defined by the keyword DB_PgSiz in dmconfig.ini

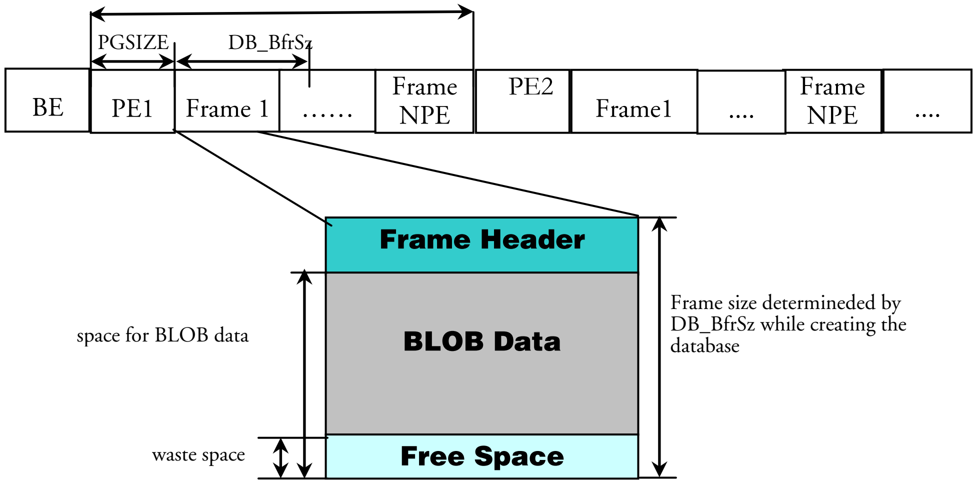


Figure 7-3: the structure of a BLOB file

Users can calculate the size of a BLOB file according to the total number of frames, PE pages, BE pages and data frames.

The following formula shows how to approximately calculate the size of a BLOB in KB:

Number of BE pages = [total frames / 676685[6]], ([] means gain the nearest bigger integer)

Number of PE pages = [(total frames – number of BE pages) / NPE + 1]

BLOB file size = (Number of BE pages + Number of PE pages) × PAGE SIZE KB + (total frames - Number of BE pages - Number of PE pages) × **DB_BfrSz** KB

For example, if page size is 8 KB and the BLOB frame size is 32 KB, the size of a BLOB file with 3 frames is:

Number of BE pages = [3 / 676685] = 1

Number of PE pages = [(3 - 1) / 333 + 1] = 1

[6] this value depends on the Page size, this is 332665 in 4 KB page size, 676685 in 8 KB page size, 1825153 in 16 KB page size and 3670605 in 32 KB page size.

BLOB file size = (1 + 1) × 8 + (3 - 1 - 1) × 32 = 48 KB

DB_BbFil specifies the system BLOB file name in the system tablespace, SYSTABLESPACE. Users cannot specify the size of the system BLOB file. The default file name for the system BLOB file is the database name concatenated with '.SBB'.

DB_UsrBb specifies the default user BLOB file name in the default tablespace, DEFTABLESPACE, and its size.

For more details on adding new BLOB files to an existing user tablespace, refer to the subsection *Adding Files to Tablespaces* in Section 5.3.

Generating BLOBs

A BLOB column is the same as other columns except that its data type is LONG VARCHAR or LONG VARBINARY.

➲ **Example**

To create two BLOB columns named **note** and **photo**:

```
dmSQL> CREATE TABLE tb_staff (id INTEGER, note LONG VARCHAR, photo LONG
VARBINARY);
```

Insert BLOB data from the **ab.txt** file and image file **img001.gif** using host variables:

```
dmSQL> INSERT INTO tb_staff VALUES(2,?,?);
dmSQL/Val> &ab.txt, &img001.gif(2,4);
dmSQL/Val> END;
```

The resulting LONG VARBINARY column is represented in hexadecimal format. The following results will be returned when browsing the table:

```
dmSQL> SELECT * FROM tb_staff;
  id       note            photo
======  ===========  =================
     2  <script lan  ffd8ffe000104a464
```

DBMaker also supports fetching BLOB data into a user-specified file. For more information on how to insert and fetch BLOB data, refer to the *JDBA Tool User's Guide* and the *ODBC Programmer's Guide*.

Updating BLOBs

A BLOB item is always written to disk as a whole. Thus, when updating a BLOB column, DBMaker will drop the original BLOB item and then insert the new data as a new BLOB item.

➲ **Example**

To update contents for a BLOB column, using the UPDATE command:

```
dmSQL> UPDATE tb_staff SET note = 'Hello !' WHERE id > 0;
dmSQL> SELECT * FROM tb_staff;
 id       note          photo
===== =========== =================
    1 Hello !     31323334353637
    2 Hello !     33343536
```

From the user's viewpoint, there must be a BLOB for each tuple. However, to save disk space, DBMaker creates only a single copy of the BLOB data shared by all tuples with an ID greater than zero. DBMaker maintains an internal counter to record how many tuples reference a BLOB. When updating a BLOB column for a tuple that links to the shared BLOB, DBMaker generates a new BLOB item and decreases the counter of the shared BLOB by a value of one. This prevents any changes made to the BLOB column from influencing other tuples. In DBMaker*,* this is known as *loose coupling*. This makes disk utilization more efficient, but a BLOB item can only be shared by tuples that are in the same table. If a BLOB is not linked with tuples DBMaker automatically drops it.

Predicate Operations on BLOB Columns

BLOB objects can only be used in CONTAIN, MATCH or Boolean expressions when testing for NULL values.

➲ **Example 1**

To fetch all data from the **tb_staff** table from the **NOT NULL note** column:

```
dmSQL> SELECT * FROM tb_staff WHERE note IS NOT NULL;
```

DBMaker provides pattern matching for BLOBs. The CONTAIN and MATCH function is similar to the LIKE function except that wildcard characters are not supported. The difference between CONTAIN and MATCH is that the former is a partial word match and the latter is a full word match. For example, 'This is a character.' CONTAIN 'char' and 'This is a character.' MATCH 'character', but 'This is a character.' NOT MATCH 'char'.

➲ **Example 2**

To find all staff from the **note** column containing 'Database Administrator':

```
dmSQL> SELECT * FROM tb_staff WHERE note MATCH 'Database Administrator';
```

7.2 Managing File Objects

Each *file object* (FO) column references external files. Using FOs is beneficial when the data is also used by other applications, since the file can be accessed directly. Most current multimedia tools can only process multimedia data when it is stored as a complete file of the required type. Multimedia data that is stored in BLOB or data files must be fetched from DBMaker by the user and redirected to a file that can be processed by the appropriate tool. However, if BLOB data is stored as an FO a user can simply get the file name from DBMaker and pass the name to the appropriate multimedia tool.

There are two kinds of FOs: *system FO* and *user FO*. System file objects are created when a user inserts data on the client side and DBMaker passes it through and stores it in an external file specified by the configuration parameter **DB_FoDir**. System FOs are created by DBMaker and can be recognized by their client's source file name extension User file objects are external files that are simply linked to a column. A user file object may be a file on any device that is accessible by DBMaker through the server's operating system.

The major difference between system and user file objects is that DBMaker generates a system FO automatically. This means a system FO will be deleted when no column references it. Therefore, with system FOs, users can leave storage management to DBMaker. Another advantage to using system FOs is that data is duplicated to the server side, so users can manage data from the server. DBMaker's backup and restoration features also protect system FOs.

A user FO will not be deleted when there are no more references to it. The major advantage to user FOs is that DBMaker can link a column to an existing file directly. It does not need to duplicate data, such as a file on a CD-ROM. This conserves disk space and makes it easier to share a file among several records. However, if a file is deleted outside of DBMaker, all columns linking to this file may become invalid. A file linked as a user FO must open its read permission.

User FO files must be accessible from the database. They can be scattered in many directories on the server side. Instead of specifying a USER FO directory, users need to set the **DB_UsrFO** keyword to 1 in **dmconfig.ini** to enable the use of USER file objects. USER FOs are disabled by default.

Users can get the file name and file size of a FO by using the built-in functions, filename() and filelen().

Customizing the System File Object Path

DBMaker generates a series of file object subdirectories for storing system file objects. These subdirectories are located in the file object directory, which is specified by the **DB_FoDir** keyword in the **dmconfig.ini** file. When a file object subdirectory is filled to a threshold value by file objects, a new subdirectory is created. The threshold value is specified by the **DB_FoSub** keyword in the **dmconfig.ini** file, and may have a value from 100 to 10,000.

File object names take the form ZZ*xxxxxx.ext* where *xxxxxx* is a six digit base 36 serial number, and *ext* is the file extension of the object. The file extension of the object depends on the SET EXTNAME command. Refer to *System File Object Extension Names* for more information.

Subdirectories follow a naming convention based on the name of the first file object in the subdirectory. It takes the form SUB*xxxxxx* where *xxxxxx* is the six-digit base 36 number of the first file object to be inserted into the directory.

Although an FO directory can be shared by more than one database to simplify FO management, it is not recommended because it becomes inconvenient when backing up a database. The file object path may be changed before database startup by modifying the configuration parameter, or during runtime.

SETTING THE FO PATH OFFLINE

Users should specify where to put system FOs by setting the value of **DB_FoDir** in **dmconfig.ini**. The value of **DB_FoDir** is the full path of an existing directory. DBMaker must own the write-permission on that directory.

➲ **Example 1**

To create system FOs in the */disk1/usr/fo* directory, add the following line to the **dmconfig.ini** file:

```
DB_FoDir = /disk1/usr/fo
```

➲ **Example 2**

To set **DB_UsrFo** = 1 and enable user objects:

```
DB_UsrFo = 1           ; enable USER file objects
```

SETTING THE FO PATH ONLINE

DBMaker provides a system procedure for users to modify the system file object directory while the database is running. This operation changes the setting of the following 3 items to the new value:

- **Run-time FO directory** — after the change is made, all new system file objects will be stored in the new FO directory.
- **DB_FoDir** — the next time the database is restarted, it will use the new FO directory.
- **$DB_FoDir alias.** — the default FO alias, which corresponds to the setting of the **DB_FoDir** keyword in **dmconfig.ini**.

➲ **Example 1**

To change the FO directory to a new directory, e.g. */home/DBMaker/mydb/fo*, execute the following command:

```
dmSQL> CALL SETSYSTEMOPTION('fodir', '/home/DBMaker/mydb/fo');
```

In addition to being able to change the file object directory, users may query the system to return the current settings for the FO directory path.

➲ **Example 2**

The following command returns the current FO directory setting:

```
dmSQL> CALL GETSYSTEMOPTION('fodir', ?);
OPTION_VALUE: /home/DBMaker/mydb/fo
```

Generating File Objects

Several steps are required to generate file objects within DBMaker. First a FILE type column must be created on a table. Either system or user file objects may be inserted into a FO type column. To create a FO column, set the column type to FILE when creating the table.

➲ **Example 1**

To create a table named **tb_person** with a file object column called **photo**:

```
dmSQL> CREATE TABLE tb_person (name CHAR(10), photo FILE);
```

➲ **Example 2**

If the FO to be input is on the on the server, DBMaker will link the FO column to the existing file and generate a user FO. If the FO is on the client side, DBMaker will create a system FO by copying the file from the client side to the FO directory on the server side. To insert FO data:

```
dmSQL> INSERT INTO tb_person VALUES ('cathy','/disk1/image/cathy.bmp')
    2>;                                     // stored as a USER FO
dmSQL> INSERT INTO tb_person VALUES ('jeff',?);
dmSQL/Val> &jeff.gif;                       // stored as a SYSTEM FO
dmSQL/Val> END;
```

➲ **Example 3**

There are three varieties of fetching methods for a FO column: content, file name, and file size. To fetch an FO file named **cathy.bmp**:

```
dmSQL> SELECT photo, FILENAME(photo), FILELEN(photo) FROM tb_person ;
  photo          filename(photo)        filelen(photo)
========= =========================== ================
012034451 /disk1/image/cathy.bmp      21100
349045821 /disk1/usr/fo/ZZ000000.hmp  12034
```

For more information about manipulation on FO columns, refer to the *JDBA Tool User's Guide* and the *ODBC Programmer's Guide.*

System File Object Extension Names

Users can set the system file object extension name using the SET EXTNAME command.

➲ **Example 1**

To set the system file object <*extension_name*>:

```
SET EXTNAME TO <extension_name>
```

There are two types of <*extension_name*>:

- A character string less than seven characters, such as 'bmp', 'avi' and 'jpg'
- Using the SOURCE option, the extension name is set equal that of the client's source file

➲ **Example 2**

To use the SET EXTNAME command:

```
dmSQL> CREATE TABLE tb_example (c1 INT, f1 FILE);
dmSQL> INSERT INTO tb_example (c1, f1) VALUES (?, ?);

dmSQL/Val> SET EXTNAME TO FOB;
dmSQL/Val> 1, &readme.txt;             //extension name : '.FOB'
1 rows inserted
dmSQL/Val> SET EXTNAME TO doc;
dmSQL/Val> 2, &readme.txt;             //extension name : '.doc'
```

```
1 rows inserted
dmSQL/Val> SET EXTNAME TO SOURCE;
dmSQL/Val> 3, &readme.txt;            //extension name : '.txt'
dmSQL/Val> END;
dmSQL> SELECT FILENAME(f1) FROM tb_example;
    c1               FILENAME (f1)
==========  ==============================
         1  /usr1/fo/ZZ000001.FOB
         2  /usr1/fo/ZZ000002.doc
         3  /usr1/fo/ZZ000003.txt
3 rows selected
```

Updating File Objects

To update the contents of a FO column, use the SQL UPDATE command. DBMaker replaces the FO column with a new file.

As with inserting FOs, a FO column may be updated for a new SYSTEM FO or linked to a USER FO.

➲ **Example**

To link the **photo** column to **/disk2/image/common.bmp**:

```
dmSQL> UPDATE tb_person SET photo = '/disk2/image/common.bmp' WHERE name =
'cathy';
```

Alternatively, users can input new data from a file on the client side. For more information, refer to the *JDBA Tool User's Guide* and *ODBC Programmer's Guide*.

If the results of the UPDATE operation contains more than one tuple, only one file is created. This file is shared among the tuples to save disk space. DBMaker maintains an internal counter to record how many tuples reference the file. In addition, if a user modifies the contents of the file through an external application program all tuples will recognize the modification.

When no tuples retain links to a system FO after UPDATE or DELETE operations, DBMaker automatically deletes the file after that transaction is committed. However, DBMaker never removes any USER FO, even when no tuples are referencing it, because DBMaker did not generate this file.

Renaming File Objects

Sometimes full disks or a disk layout reorganization make it is necessary to change the positions or names of FOs. DBMaker permits users to use the MOVE FILE OBJECT statement to change the name or path of the FO. Before using the MOVE FILE OBJECT command, use the operating system to move the files to the new location; DBMaker will make sure the new files exist before allowing the move.

➲ **Example 1**

To get the names of the files that will be moved using filename()**:**

```
dmSQL> MOVE FILE OBJECT '/disk1/usr/fo/ZZ000000.FOB' TO '/disk3/pub/photo1.bmp';
```

DBMaker also permits users to move FOs from one directory to another. Note that DBMaker permits using only one * character for the specified file name in the source directory, but does not allow using any * character in the destination directory. DBMaker does not support recursively-moving files. To move all of the files, not including subdirectories, from one directory to another, specify the former directory by adding the '/' or '/*' characters at the end of the directory.

➲ **Example 2**

Let there be four files in ***/disk1/usr/fo*** named **ABC1.FOB**, **ABC2.FOB**, **ABC3.FOB**, **ABC4.FOB.** To move **ABC1.FOB**, **ABC2.FOB**, **ABC3.FOB**, **ABC4.FOB** from ***/disk1/usr/fo*** to ***/disk3/pub*** file objects use:

```
dmSQL> MOVE FILE OBJECT '/disk1/usr/fo/ ' TO '/disk3/pub/ ';
dmSQL> MOVE FILE OBJECT '/disk1/usr/fo/* ' TO '/disk3/pub/ ';
dmSQL> MOVE FILE OBJECT '/disk1/usr/fo/*.FOB ' TO '/disk3/pub/ ';
dmSQL> MOVE FILE OBJECT '/disk1/usr/fo/A* ' TO '/disk3/pub/ ';
```

To move **ABC1.FOB** from ***/disk1/usr/fo*** to ***/disk3/pub***:

```
dmSQL> MOVE FILE OBJECT '/disk1/usr/fo/*1.FOB ' TO '/disk3/pub/ ';
dmSQL> MOVE FILE OBJECT '/disk1/usr/fo/A*1.FOB ' TO '/disk3/pub/ ';
```

Unloading System File Objects

DBMaker lets the database administrator decide whether unload system file object using UNLOAD FILEOBJ ON/OFF command or not. The default setting for this option is *ON*. All of file objects on the server will be unloaded to current working directory when users do not specify this option. If the users don't want to unload all the FO files to the working directory for poor disk space, use the SET UNLOAD FILEOBJ OFF command to copy file objects by manually into an unload's working directory.

When loading these unloaded files to a new database, all of system file objects would be reordered and the names would be changed. If the users had quoted these FO names in their applications and don't want to reorder these system file objects when migrating to a new database, they can use the SET UNLOAD FILEOBJ NAME option setting before UNLOAD DB, dmSQL will unload script as client file object with full path and filename. User must make sure the **DB_UsrFo =1** before loading database, otherwise, it will fail to load, and then all the file objects must exist on the server with same path and filename.

Retrieving the Length of File Objects

A FO's length can be retrieved using the built-in functions FILELENEX and FILELEN. For more information about these functions, please refer to the *SQL Command and Function Reference* .

Predicate Operations on File Objects

As with BLOBs, users can test FOs for NULL values and use the CONTAIN and MATCH functions to perform pattern searches. Furthermore, the FO item can be used in arithmetic expressions with the FILELEN() built-in function, in Boolean expressions with the FILEEXIST () built-in function, and in string expressions with the FILENAME() built-in function.

In the event that a file has been removed or renamed from within the operating system, use the FILEEXIST() built-in-function to test which files exist. A value of 0 indicates that the referenced FO file does not exist, 1 indicates that it still exists, and NULL indicates that the tuple is a NULL value.

➲ **Example 1**

To select tuples with the .gif extension from the column **photo**:

```
dmSQL> SELECT * FROM tb_person WHERE FILENAME(photo) LIKE '%.gif';
```

➲ **Example 2**

To fetch tuples with a size greater than 100KB from the column **photo**:

```
dmSQL> SELECT * FROM tb_person WHERE FILELEN(photo) > 102400;
```

➲ **Example 3**

To fetch all tuples for files that exist:

```
dmSQL> SELECT * FROM tb_person WHERE FILEEXIST(photo)=1;
```

File Object UNC Names

Universal Naming Convention (UNC) filenames can be used for the file object path and directory names in Microsoft Windows environments. This makes it easy to specify the path and directory names when a DBMaker server is running on a Microsoft Windows platform. Directories on machines other than the machine hosting the server can also be specified.

➲ **Example 1**

To retrieve all system FOs created in the *\\NTMACHINE\E\FO* directory when used in the **dmconfig.ini** file:

```
DB_FoDir = \\NTMACHINE\E\FO
```

➲ **Example 2**

To show how file objects work with UNC names:

```
dmSQL> CREATE TABLE tb_example (c1 INT, c2 FILE);
dmSQL> INSERT INTO tb_example VALUES (?, ?);
dmSQL/Val> 1, '\\NTMACHINE\D\DB\memo1.txt';
1 rows inserted
dmSQL/Val> 2, &c:\temp\memo2.txt;
1 rows inserted
dmSQL/Val> END;
```

```
dmSQL> SELECT c1, FILENAME(c2) FROM tb_example;
   c1                     FILENAME(c2)
========= ===================================================
       1  \\NTMACHINE\D\DB\memo1.txt
       2  \\NTMACHINE\E\FO\ZZ000001.txt

2 rows selected
```

File Object Path Default Aliases

DBMaker supports two alias names for the file object path: $**DB_DbDir** and $**DB_FoDir**. The file object path alias provides an alias name to represent the real file object path. Users may insert, update and delete file objects using the alias path name. File objects can be moved more easily to another directory path.

The two alias names are set using the keywords **DB_DbDir** and **DB_FoDir**. in the **dmconfig.ini** file for $**DB_DbDir** and $**DB_FoDir**, respectively.

➲ **Example 1**

The file object path alias is set to the path specified by the **DB_FoDir** keyword in **dmconfig.ini**:

```
…
DB_FoDir = "/usr1/tmp/employeedata/FO"
```

➲ **Example 2**

To use the file object path alias to insert values into the FILE type column **photo**:

```
dmSQL> CREATE TABLE tb_example (c1 INT, photo FILE);
dmSQL> INSERT INTO tb_example VALUES (2, '$DB_FoDir/photo471.jpg');
```

In the above example, file **photo471.jpg** could also have been inserted using the full file object path '***/usr1/tmp/employeedata/FO/*****photo471.jpg**'.

FO and Applications

FILE type data is only supported by DBMaker, and is not defined by ODBC. Development tools, such as Inprise/Borland Delphi or Microsoft Visual Basic do not recognize FILE type data as valid. The configuration parameter **DB_FoTyp** may be

```
1 rows inserted
dmSQL/Val> SET EXTNAME TO SOURCE;
dmSQL/Val> 3, &readme.txt;            //extension name : '.txt'
dmSQL/Val> END;
dmSQL> SELECT FILENAME(f1) FROM tb_example;
    c1                FILENAME (f1)
==========  ==============================
         1  /usr1/fo/ZZ000001.FOB
         2  /usr1/fo/ZZ000002.doc
         3  /usr1/fo/ZZ000003.txt
3 rows selected
```

Updating File Objects

To update the contents of a FO column, use the SQL UPDATE command. DBMaker replaces the FO column with a new file.

As with inserting FOs, a FO column may be updated for a new SYSTEM FO or linked to a USER FO.

➲ **Example**

To link the **photo** column to **/disk2/image/common.bmp**:

```
dmSQL> UPDATE tb_person SET photo = '/disk2/image/common.bmp' WHERE name =
'cathy';
```

Alternatively, users can input new data from a file on the client side. For more information, refer to the *JDBA Tool User's Guide* and *ODBC Programmer's Guide*.

If the results of the UPDATE operation contains more than one tuple, only one file is created. This file is shared among the tuples to save disk space. DBMaker maintains an internal counter to record how many tuples reference the file. In addition, if a user modifies the contents of the file through an external application program all tuples will recognize the modification.

When no tuples retain links to a system FO after UPDATE or DELETE operations, DBMaker automatically deletes the file after that transaction is committed. However, DBMaker never removes any USER FO, even when no tuples are referencing it, because DBMaker did not generate this file.

SETTING THE DB_BMODE VALUE

The keyword **DB_BMode** specifies the backup mode of a database. Setting the value to 0 enables NON-BACKUP mode, 1 enables BACKUP-DATA mode, and 2 enables BACKUP-DATA-AND-BLOB mode.

- NON-BACKUP (0) mode — does not support incremental backup for tablespaces, including system or user-defined
- BACKUP-DATA (1) mode — supports incremental backup for system tablespace, data in user-defined tablespaces, but not BLOBs in user-defined tablespaces
- BACKUP-DATA (2) mode — supports incremental backup for system tablespace, data in user-defined tablespaces, BLOBs in user-defined tablespaces created with the BACKUP BLOB ON option, but not BLOBs in user-defined tablespaces created with the BACKUP BLOB OFF option

To turn on BLOB journal logging, add a line to the **dmconfig.ini** file:

```
DB_BMode = 2    ;log all data including BLOB
```

For details on database backup mode, refer to Chapter 15, *Database Backup, Recovery, and Restoration.*

SETTING THE CREATE TABLESPACE BACKUP OPTION

The backup mode for an individual tablespace is set when it is being created. The syntax for the CREATE TABLESPACE command follows:

```
CREATE [AUTOEXTEND] TABLESPACE tablespace_name [backup_mode]
DATAFILE [tsfile , tsfile, ...];
```

where:

```
backup_mode ::= BACKUP BLOB OFF | BACKUP BLOB ON
tsfile ::= file_name TYPE = DATA | file_name TYPE = BLOB
```

Users can place important BLOBs in tablespaces with the BACKUP BLOB ON flag. It is a good idea to place BLOBs that do not need to be backed up in tablespaces with the BACKUP BLOB OFF setting in order to improve the system performance. Tablespace creators determine the trade-off.

Data and BLOB files must be specified in the dmconfig.ini file before the tablespace is created. This may be accomplished through the "user files" page in the JConfiguration Tool. Refer to the *JConfiguration Tool Reference* for detailed instructions on creating data and BLOB files.

➲ **Example 1**

To create tablespace **ts_reg** with BACKUP BLOB OFF and **ts_aut** with BACKUP BLOB ON:

```
dmSQL> CREATE TABLESPACE ts_reg BACKUP BLOB OFF
    2> DATAFILE f1 TYPE = DATA, f2 TYPE = BLOB;
dmSQL> CREATE TABLESPACE ts_aut BACKUP BLOB ON
    2> DATAFILE f3 TYPE = DATA, f4 TYPE = BLOB;
```

➲ **Example 2**

Query the **BK_MODE** column from the **SYSTABLESPACE** table to know the backup mode for each tablespace. The value 1 means that BACKUP BLOB is OFF, while 2 means that it is ON. Querying the backup mode of a tablespace will yield the following result:

```
dmSQL> SELECT TS_NAME, BK_MODE FROM SYSTEM.SYSTABLESPACE;
   TS_NAME        BK_MODE
==============  =============
SYSTABLESPACE               2
DEFTABLESPACE               2
ts_reg                      1
ts_aut                      2
4 rows selected
```

A summary of the interaction of the backup modes between a database and its tablespaces follows. 'Yes' indicates that the type of tablespace in question is backed up 'No' indicates that it is not.

DATABASE BACKUP MODE	TABLESPACE BACKUP MODE	USER-DEFINED TABLESPACE (DATA)	USER-DEFINED TABLESPACE (BLOB)	SYSTEM TABLESPACE (DATA AND BLOB)
NON BACKUP (**DB_BMode** = 0)	---	No	No	No
BACKUP DATA (**DB_BMode** = 1)	---	Yes	No	Yes
BACKUP DATA AND BLOB (**DB_BMode** = 2)	BACKUP BLOB OFF	Yes	No	Yes
	BACKUP BLOB ON	Yes	Yes	Yes

Before setting the backup mode, ensure that the journal file is large enough to record all BLOB data; otherwise, a journal full message may be returned. For information on how to adjust journal file size, refer to the subsection *Resizing Journal Space* in Chapter 5.

For concepts on data files, the BLOB file and the tablespace, refer to Chapter 5, *Storage Architecture*.

For information on the CREATE TABLESPACE command, refer to the *SQL Command and Function Reference.*

For information on the SYSTABLESPACE table, refer to *System Catalog Reference*.

File Object Journal Logging

DBMaker does not support journal logging of FOs. When backing up the database, back up all FOs belonging to the database as well by manually copying them into a backup directory. Alternatively, Backup server may be used to automatically back up file objects to a backup directory. For more information on backing up file objects, refer to section 15.6, *Backup Server*. To determine what files belong to a database, query the SYSFILEOBJ table.

➲ **Example**

To retrieve the filenames of all FOs by querying the SYSFILEOBJ table:

```
dmSQL> SELECT FILE_NAME FROM SYSFILEOBJ;
```

Copy all FOs to the backup storage location. When restoring the database from a backup, copy all FOs as well. If the file paths or file names have changed, use the MOVE FILE OBJECT command to update the file names in the SYSFILEOBJ table.

7.4 Large Objects and SELECT INTO Command

The SELECT INTO command takes selected data and inserts it into a specified table. File objects and BLOBs can be moved from one table to another using this command. The SELECT INTO command can be used in a distributed database (DDB) environment.

In a local-to-local SELECT INTO statement, DBMaker needs to duplicate the BLOB data or increase the *shared counter* of the system file object or a user file object.

In a DDB environment, DBMaker copies the BLOB data from one site to another, but there are many considerations for file objects. DBMaker provides the distributed file object duplication mode (SET DFO DUPMODE command) to take care of processing file objects in a DDB environment.

SET DFO DUPMODE

The DFO DUPMODE tells a database whether file objects are to be copied to the target database or not. There are two modes for DFO DUPMODE: NULL and COPY mode.

➲ **Example**

Syntax for DFO DUPMODE: NULL and COPY mode:

```
dmSQL> SET DFO DUPMODE NULL;
dmSQL> SET DFO DUPMODE COPY;
```

SET DFO DUPMODE NULL

There are two cases in DDB mode to consider:

- The source and target databases are the same, including the local database or both remote databases. Since they are the same database, DBMaker only increases the shared counters of the file objects.
- The source and the target database are not the same. The target FILE column is set to NULL. Thus, the file objects in the source database are not sent out.

SET DFO DUPMODE COPY

There are three situations to consider for file objects:

- For user file objects, DBMaker passes only the source file name to the target database. The user needs to copy the files to a place where the target database can access them. Sometimes the UPDATE command or the MOVE FILE OBJECT command must be used to change the file names on the target database if the new directories are not the same on the source database.
- For system file objects between two different databases, DBMaker creates a new system file object on the target database and copy the contents of the source database to it.
- For system file objects on the same database, local-to-local or remote-to-remote, DBMaker increments only the shared counters.

Limitations

DFO DUPMODE mode does not affect a SELECT INTO command used on a BLOB (LONG VARCHAR and LONG VARBINARY) column. BLOB data can be copied using the SELECT INTO command regardless of the DFO DUPMODE.

In a DDB environment, if a SELECT INTO command is used on a user file object and the option of DFO DUPMODE is set to COPY, then the user should be aware of the location of the linked file on the target database. The linked file object should exist in the same relative path on the target database. If it is not, the user should use the operating system to copy the file from the source to the target database and use the UPDATE or the MOVE FILE OBJECT commands for these columns if the file paths of the source and target databases are different.

If the user has not performed the above operations, an error message will be returned when querying the file object, because the file does not exist in the full path or the path of the file is incorrect.

When selecting a system file object from a remote database into the local database, DBMaker has to keep a record of the shared information. The information is kept within one SELECT INTO command. Therefore, there is still a duplicate file problem, which wastes space. Additionally, selecting system file objects into a remote database creates duplicate files.

The SET EXTNAME option does not affect the result of the SELECT INTO command. The extension names of the file objects on the source and target databases are the same. For example, the file name of the source database is 'ZZ000001.BMP', and then the target name of the file object on the target may be 'ZZXXXXXX.BMP'.

➲ **Example**

DBMaker assumes the data of CHAR, VARCHAR or BINARY as the file name, so users must make sure **db2** can access the ***/etc/hosts*** file from the view of **db1**. Select the CHAR column into the FILE column, where column **c2** in table **t2** on database **db2** is FILE type:

```
dmSQL> SELECT c1, '/etc/hosts' FROM db1:t1 INTO db2:t2(c1, c2);
```

Considering the FILE type column on the target database, the table below summarizes the effect of the SELECT INTO command with the different source data types:

TYPE ON THE SOURCE DATABASE	ENVIRONMENT	SET DFO DUPMODE	RESULT
string expression CHAR VARCHAR BINARY	non-DDB or DDB Environment.	…	Source: passes the file name. Target: inserts new user file objects.
FILE	The source and the target are the same database.	…	Increases the shared counter of the file objects.
	The source and the target database are not the same.	NULL	Target: inserts the NULL value.
		COPY	The source is the user file object: Source: passes the file name. Target: inserts the new user file object. The source is the system file object: Source: passes the content of the file object. Target: inserts the new system file object.
LONG VARCHAR LONG VARBINARY Other	…	…	Not supported.

8 Security Management

This chapter provides guidelines on setting up the security policies for a database, and includes information on security, authority levels, and table privileges.

8.1 Security Policies

DBMaker provides two kinds of security:

- **Database authority** — determines who can log on to DBMaker and the actions they can perform
- **Object privileges** — controls access rights for DBMaker objects. DBMaker objects include tables, columns, views, domains, and synonyms

8.2 Database Authority

Database authority is used to determine access for a database. DBMaker controls database access with *user names* and *passwords* and has five classes of users as shown in Figure8-1.

The SYSADM is the most powerful authority level in DBMaker. There can be only one SYSADM for every database. A user with SYSADM authority can grant SYSDBA, DBA, RESOURCE or CONNECT authority to other users, set ACL (Access Control List) for other users, and has all the privileges of the SYSDBA and DBA authority level on the database.

Users with SYSDBA authority can both grant or revoke CONNECT, RESOURCE and DBA to other users, change other users' passwords, except users with SYSADM or SYSDBA authority, and set ACL (Access Control List) for users with lower auathority. Users with SYSDBA authority have all the privileges of DBA authority and only users with SYSADM authority can grant or revoke SYSDBA authority from users. For users with SYSDBA authority, if the SYSADM revokes SYSDBA, the users still has DBA authority, but if the SYSADM revokes DBA, the users has neither SYSDBA nor DBA authority.

Users with DBA authority level have all privileges for all objects in the database and can grant, change, or revoke object privileges for any user except users with SYSADM, SYSDBA or DBA authority. They can also create new resources like tablespaces and files, and perform database administrative operations like starting/terminating and backing up databases.

Users with RESOURCE authority are allowed to create new tables or views, and to grant privileges on their own tables to other users.

Users with only CONNECT authority can access objects that they have been granted privileges for, but cannot create new tables or views. They may also select information from the system tables.

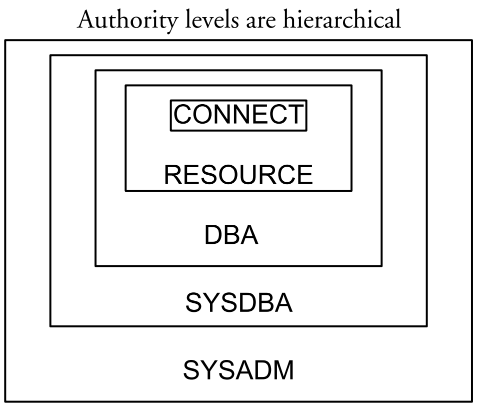


Figure 8-1: DBMaker database authority level hierarchy

LEVEL	PRIVILEGES
SYSADM	Can grant and revoke security authority levels (CONNECT/RESOURCE/DBA/SYSDBA) to all users except users with the SYSADM authority. Can change the passwords of all users. Has all privileges of the SYSDBA authority level.
SYSDBA	Can grant and revoke security authority levels (CONNECT/RESOURCE/DBA) to all users except users with the SYSADM and SYSDBA authority. Can change the passwords of all users except users with the SYSADM and SYSDBA authority. Has all privileges of the DBA authority level.
DBA	Has all privileges on tables except SYSTEM tables. Can grant/change/revoke object privileges of all users and groups. Can add/remove users from groups. Has privileges on database administration commands such as starting or terminating a database, creating/dropping/ altering a tablespace, and backing up a database. Has all the privileges of the CONNECT and RESOURCE authority levels.
RESOURCE	Can create and drop tables, views, domains, and synonyms. Can only drop tables, views, domains, and synonyms created by the user. Can grant/revoke owned table/view privileges to other users. Has any table privileges granted to the user. Has all the privileges of the CONNECT authority level.
CONNECT	Can log on to the database. Can select the SYSTEM tables. Has any table privileges granted to the user. This authority level must be granted before the other authority levels.

Table 8-1: DBMaker database authority levels

Managing Users

DBMaker provides several SQL commands for managing users. These commands allow new users to be added, existing users to be removed from a database, user passwords to be set or changed, and user authority levels to be granted.

ADDING A USER

The SYSADM must assign each user a user name and a password by using the GRANT (database authority) command before a user can log on.

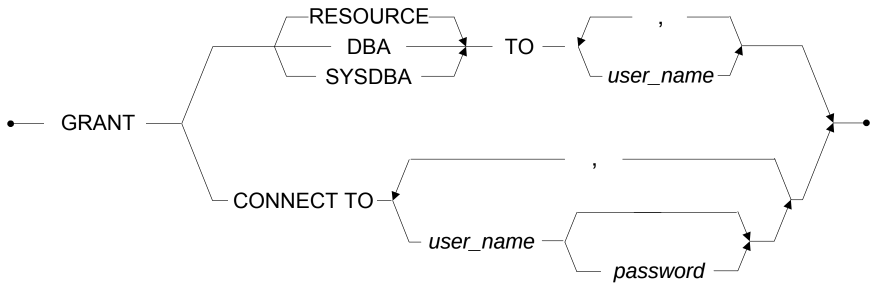


Figure 8-2 Syntax for the GRANT command

The GRANT command grants authority levels to users. Only the SYSADM can grant authority levels to other users. The SYSADM authority level cannot be granted to other users. As a result, for each database there is only one user with the SYSADM user name and SYSADM authority level. The SYSADM is also the default user who creates the database. Only the password can be changed for the SYSADM user name.

The SYSADM can grant CONNECT, RESOURCE, DBA , SYSDBA and ACL authority to other users. If the GRANT command is used to grant RESOURCE, DBA or SYSDBA authority to a user, it will not take effect until the next time the user connects to the database.

Users with SYSADM or SYSDBA authority can grant a password to users with CONNECT authority. If the SYSADM does not specify the password, it means that user does not need a password to log on to database. A password can be any valid SQL identifier, which is not longer than sixteen bytes.

➲ **Example 1**

To grant CONNECT authority level and the password *jeff123* to user Jeff:

```
dmSQL> GRANT CONNECT TO Jeff jeff123;
```

➲ **Example 2**

To increase the authority level for user Jeff to RESOURCE:

```
dmSQL> GRANT RESOURCE TO Jeff;
```

➲ **Example 3**

To increase the authority level for user Jeff to DBA:

```
dmSQL> GRANT DBA TO Jeff;
```

➲ **Example 4**

To increase the authority level for user Jeff to SYSDBA:

```
dmSQL> GRANT SYSDBA TO Jeff;
```

CHANGING A PASSWORD

The ALTER PASSWORD command can be used to change a user's password.

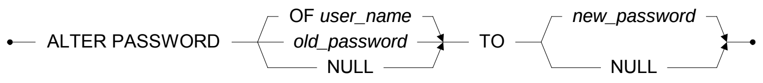


Figure 8-3 Syntax for the ALTER PASSWORD command

There are two ways to use the command:

- A user can change their own password with the ALTER PASSWORD <*old_password*> TO <*new_password*> command. The <*old_password*> must match the original password stored in the database.
- The SYSADM can change any user's password with the ALTER PASSWORD OF <*user_name*> TO <*new_password*> command. It is not necessary for the SYSADM to know the old password of other users.

➲ **Example 1**

The user Jeff changes his password from no password to ***xyz@#***:

```
dmSQL> ALTER PASSWORD NULL TO "xyz@#";
```

➲ **Example 2**

The SYSDBA changes the password for user Jeff to ***abc@#***:

```
dmSQL> ALTER PASSWORD OF Jeff TO "abc@#";
```

➲ **Example 3**

The SYSADM changes the password for user Jeff to ***xyz@#***:

```
dmSQL> ALTER PASSWORD OF Jeff TO "xyz@#";
```

Removing a User or Changing a User's Authority Level

The SQL REVOKE command removes a database authority level.

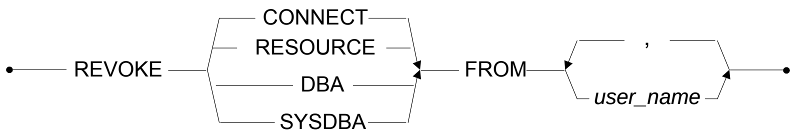


Figure 8-4 Syntax for REVOKE command

Revoking a user's RESOURCE, DBA or SYSDBA authority does not take effect until the next time the user connects to the database.

Users with the SYSADM authority can revoke CONNECT, RESOURCE, DBA, SYSDBA and ACL (Access Control List) from other users, but cannot revoke privileges from users with SYSADM authority.

The users with SYSDBA authority can also revoke CONNECT, RESOURCE, DBA and ACL (Access Control List) from users with lower authority, but can not revoke privileges from the user with SYSDBA authority.

➲ **Example 1**

To revoke SYSDBA authority from user Jeff:

```
dmSQL> REVOKE SYSDBA FROM Jeff;
```

After executing the command, Jeff will no longer have SYSDBA authority but will still have DBA authority.

➲ **Example 2**

To revoke DBA authority from user Jeff:

```
dmSQL> REVOKE DBA FROM Jeff;
```

After executing the command, Jeff will no longer have DBA authority but will still have CONNECT authority.

➲ **Example 3**

To remove Jeff's CONNECT authority and take away his ability to log on:

```
dmSQL> REVOKE CONNECT FROM Jeff;
```

Revoked Privilege	**Description**
SYSDBA	Revoking SYSDBA authority for a user means the user can no longer grant or revoke security authority levels (CONNECT/RESOURCE/DBA) to other users, and can no longer change the passwords of other users. The user will retain the DBA authority. All tables, views, domains, and synonyms created by this user remain in the database.
DBA	Revoking DBA authority for a user means the user can no longer create or drop tables and grant or revoke privileges from other users. The user will retain only the CONNECT authority unless granted the RESOURCE privilege. All tables, views, domains, and synonyms created by this user remain in the database.

REVOKED PRIVILEGE	DESCRIPTION
RESOURCE	Revoking RESOURCE authority means the user can no longer create or drop tables. The user will retain only the CONNECT authority unless granted the DBA privilege. All tables, views, domains, and synonyms created by this user remain in the database.
CONNECT	Revoking this authority means the user can no longer log on to the database. All privileges owned by this user on tables and views will be revoked. All tables, views, domains, and synonyms created by this user remain in database.

Table 8-2: Description of revoking DBMaker database authority levels

Managing Groups

To simplify management of authority levels, use a group to collect several users or other groups. Database privileges can then be granted to all members in a group at the same time with one command. Though a group is different from a user, it can be treated as a user. Object privileges granted to a group apply to all members in the group.

Only users with SYSADM, SYSDBA or DBA authority levels can:

- Create groups
- Add members to groups
- Remove members from groups
- Drop groups

CREATING GROUPS

The CREATE GROUP statement is used to create a new group.

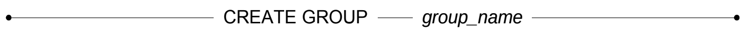


Figure 8-5 Syntax for the CREATE GROUP command

The *group identification* (group name) uniquely identifies the name of a group in DBMaker. The group name cannot be SYSTEM, PUBLIC, GROUP or any existing user or group names.

➲ **Example**

To create a new group named COMMITTEE:

```
dmSQL> CREATE GROUP COMMITEE;
```

ADDING MEMBERS TO GROUPS

After creating a new group, users can be added using the ADD <*user name or group name*> TO GROUP command.

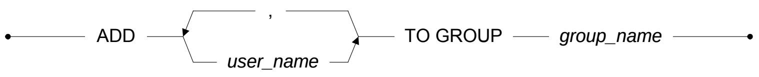


Figure 8-6 Syntax for the ADD … TO GROUP command

A group cannot be added as a new member of itself. Members of a group can include any existing user or group name.

➲ **Example**

To add user **Jeff** and group **RD** to the **COMMITEE** group and grant SELECT privilege to the **CASEMaker.TB_STAFF** table:

```
dmSQL> ADD Jeff, RD TO GROUP COMMITEE;
dmSQL> GRANT SELECT ON CASEMaker.TB_STAFF TO COMMITEE;
```

All members in **COMMITEE** will have the SELECT privilege for the **CASEMaker. TB_STAFF** table.

Removing Members from Groups

The REMOVE <*user name or group name*> FROM GROUP command can be used to remove users from a specified group.

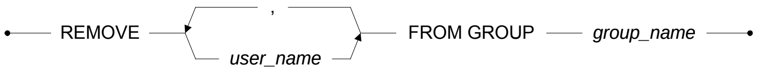


Figure 8-7 Syntax for the REMOVE … FROM GROUP command

The members removed from the group will lose all privileges granted to the specified group, but will retain privileges granted to them directly.

➲ **Example**

To remove user **Jeff** from the **COMMITEE** group:

```
dmSQL> REMOVE Jeff FROM GROUP COMMITEE;
```

After this command is executed, user **Jeff** will be removed from the group **COMMITEE** and lose SELECT privilege on the table **CASEMaker.TB_STAFF**.

Dropping Groups

The DROP GROUP command will drop a specified group from a database; all members in the group will lose the privileges granted for the group.

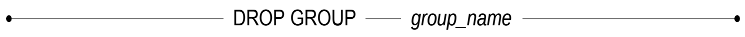


Figure 8-8 Syntax for the DROP GROUP command

➲ **Example**

To drop the **COMMITTEE** group from the database:

```
dmSQL> DROP GROUP COMMITEE;
```

Checking IP Addresses

You may want clients to connect to your database using only specific IP addresses, for example 192.72.112.*, or want certain IP addresses are forbidden to connect to the database, for example 192.168.0.*. Enabling IP checking allows you to control the IP addresses clients can use to access your database and the forbidden IP addresses. All users' settings are stored in the system catalog SYSACL.

The catalog contains three columns USER_NAME, ADDRESS and PRIVILEGE.

- USER_NAME: The name and settings of the user trying to connect
- ADDRESS: The IP address allowed to connect to the database
- PRIVILEGE: The specified IP address is allowed or blocked (ALLOW or BLOCK privilege) to connect to the database

The user name **PUBLIC** is reserved. If you use the user name **PUBLIC** all users must satisfy specified settings to connect to the database.

When a database is created, the view SYSORDERACL is automatically created and displays information on ip addresses of all users, therefore users can select these information to check whether an ip address is allowed to connect the database.

ENABLE IP CHECKING

You can use the keyword **DB_StACL** in the **dmconfig.ini** file to enable IP checking. You must configure IP checking setting before starting the database.

- **DB_StACL = 1:** Enables IP checking
- **DB_StACL = 0:** Disables IP checking (default)

CREATE A RULE

There are two kinds of IP checking rules: whitelist-based and blacklist-based. For the same ip address, constraints' result under the two rules follows the following table:

Order / Match	Whitelist-based	Blacklist-based
Match only the allowed ip addresses list	Allowed	Allowed
Match only the blocked ip addresses list	Blocked	Blocked
Match neither	Blocked	Allowed

Match both	Blocked	Allowed

Whitelist-based set, column ACLORDER of SYSAUTHUSER is marked with 0; blacklist-based set, column ACLORDER of SYSAUTHUSER is marked with 1.

To allow most ip addresses, and meanwhile forbid certain specific ip addresses, whitelist-based is more appropriate; to forbid most ip addresses, and meanwhile allow certain specific ip addresses, blacklist-based is more appropriate. The default rule is whitelist-based.

Example 1a

Glow selects blacklist-based as his IP checking rule, and forbid clients to connect to the database using all ip addresses of IP segment 127.0.0.* except 127.0.0.1:

```
dmSQL> GRANT BLOCK TO Glow '127.0.0.*';
dmSQL> GRANT ALLOW TO Glow '127.0.0.1';
```

Example 1b

Jeff selects whitelist-based as his IP checking rule, and allow clients to connect to the database using all ip addresses of IP segment 192.168.0.* except 192.168.0.3:

```
dmSQL> GRANT ALLOW TO Jeff '192.168.0.*';
dmSQL> GRANT BLOCK TO Jeff '192.168.0.3';
```

A user can select only one kind of IP checking rules. To alter the IP checking rule for a user, use the **ALTER ACL ORDER** statement. Please note that, before altering the rule, it is recommended that users should revoke all granted constraints. About details of constraints, please refer to the two sections: *Create a constraint*, *remove a constraint*.

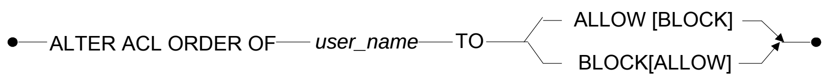


Figure 8-9 ALTER ACL ORDER

Example 2

```
dmSQL> REVOKE BLOCK FROM Vivian ALL;
dmSQL> REVOKE ALLOW FROM Vivian ALL;
dmSQL> ALTER ACL ORDER OF Vivian TO ALLOW BLOCK;
```

CREATE A CONSTRAINT

After IP checking rules' setting, users can create a constraint for the specified IP checking rule with the GRANT ALLOW statement and the GRANT BLOCK statement.

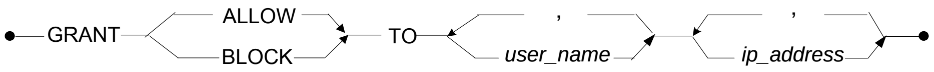


Figure 8-10 GRANT ALLOW/BLOCK TO USERLIST IPLIST

The GRANT ALLOW statement is same as the GRANT ACCESS statement.

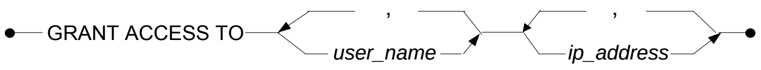


Figure 8-11 GRANT ACCESS TO USERLIST IPLIST

➲ **Example**

```
dmSQL> GRANT ACCESS TO vivian,joe '192.72.5.23','140.21.55.*';
dmSQL> GRANT ALLOW TO jane,jetty '192.72.12.20','140.15.45.*';
dmSQL> GRANT BLOCK TO pine,jim '192.70.16.20','139.15.45.*';
```

Please note that only ALLOW privilege on ip addresses can be granted to group PUBLIC, and if users grant BLOCK privilege on ip addresses to group PUBLIC, ERROR (6890) will be returned. In addition, on the basis that ALLOW privilege on a number of ip addresses has been granted to group PUBLIC, a users belonging to group PUBLIC also can add whitelist-based or blacklist-based to add new constraints for himself.

REMOVE A CONSTRAINT

To revoke a constraint for the specified IP checking rule, use the REVOKE ALLOW statement and the REVOKE BLOCK statement.

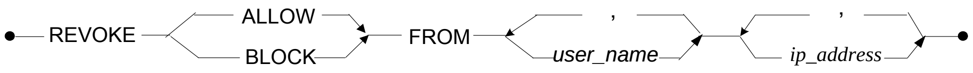


Figure 8-12 REVOKE ALLOW/BLOCK FROM USERLIST IPLIST

The REVOKE ALLOW statement is same as the REVOKE ACCESS statement.

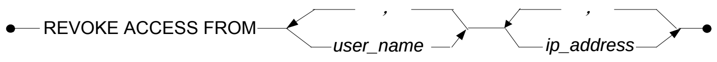


Figure 8-13 REVOKE ACCESS FROM USERLIST IPLIST

To revoke all constraints of a user for the specified IP checking rule at a time, use the "REVOKE ALLOW/BLOCK FROM user_name ALL" statement. ALL indicates all IP addresses.

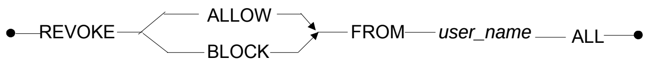


Figure 8-14 REVOKE ALLOW/BLOCK FROM user_name ALL

➲ **Example**

```
dmSQL> REVOKE ACCESS FROM vivian,joe '192.72.77.*','140.44.88.23';
dmSQL> REVOKE ALLOW FROM jane,jetty '192.72.12.20','140.15.45.*';
dmSQL> REVOKE BLOCK FROM pine,jim '192.70.16.20','139.15.45.*';
dmSQL> REVOKE BLOCK FROM glow ALL;
```

8.3 Object Privileges

An *object* in a database includes the following items: tables, views, and columns in tables/views, domains, or synonyms. DBMaker provides security management for objects, which enables users to GRANT or REVOKE object privileges for other users.

All users can reference a domain by default, but only the creator can drop the domain. The privileges for a synonym are based on a base table. Refer to Chapter 6, *Managing Schema and* Schema Objects for detailed definitions of views, domains, and synonyms.

Granting Object Privileges

The user that creates an object becomes the owner of the object and has all privileges for it. An owner can also grant privileges on the object to other users by using the SQL GRANT <*object privilege*> command.

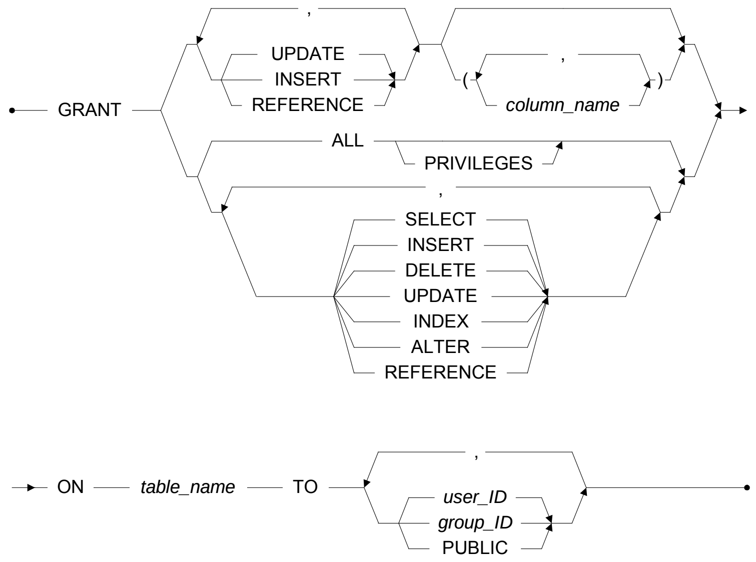


Figure 8-15 Syntax for the GRANT command

A user with DBA authority can grant privileges for any table or view in a database. A user with the RESOURCE authority can grant privileges only on tables or views created them. All privileges supported by DBMaker are described in Table 12-3.

INSERT, UPDATE, and DELETE privileges should be controlled to prevent corruption of information in a database. ALTER and INDEX privileges should be restricted to developers.

UPDATE, INSERT, and REFERENCE privileges can be restricted to some specific columns. Each column name must be qualified and be in *every* table identified in the ON clause.

PRIVILEGE	**DESCRIPTION**
SELECT	Allows users to select data from a table or view.
INSERT	Allows users to insert rows into a table or view and optionally insert into specified columns.

DELETE	Allows users to delete rows from a table or view.
UPDATE	Allows users to update a table or view and optionally update specified columns.
INDEX	Allows users to create or drop indexes for a table.
ALTER	Allows users to alter the definition of a table.
REFERENCE	Allows users to create a foreign key on a source table that references a primary key for a destination table or view.
ALL [PRIVILEGES]	Allows users to exercise all the above privileges for a table or view. PRIVILEGES is an optional keyword.

Table 8-3: Description for granting DBMaker table level privileges

The user in a GRANT command must have at least CONNECT authority. The group name is created using the CREATE GROUP command. The keyword PUBLIC includes all current and future users.

➲ **Example 1**

Jeff executes the GRANT command to give **Cathy** the read privilege to data in the **TB_INFO** table, created by him:

```
dmSQL> GRANT SELECT ON TB_INFO TO Cathy;
```

➲ **Example 2**

A DBA executes the GRANT command to give **Cathy** the read privilege to data in the **TB_INFO** table created by **Jeff**:

```
dmSQL> GRANT SELECT ON Jeff.TB_INFO TO Cathy;
```

➲ **Example 3**

A DBA gives INSERT and UPDATE privileges for the **PHONENO** column of **TB_INFO** table to **Cathy**:

```
dmSQL> GRANT INSERT (PHONENO) ON Jeff.TB_INFO TO Cathy;
dmSQL> GRANT UPDATE (PHONENO) ON Jeff.TB_INFO TO Cathy;
```

Cathy will have no privileges for deleting information from the column.

➲ **Example 4**

Use of the PUBLIC keyword to permit all users to read data in the **Jeff.TB_INFO** table:

```
dmSQL> GRANT SELECT ON Jeff.TB_INFO TO PUBLIC;
```

Revoking Object Privileges

The REVOKE <*object privileges*> command revokes privileges granted to a user. The syntax for this command is shown in Figure 8-12.

The privileges in the REVOKE (object privileges) command are the same as those for the GRANT (object privileges) command. In the diagram, the user name represents an authorized user in the database, the group name represents a group of users, and the PUBLIC keyword represents all users in the database

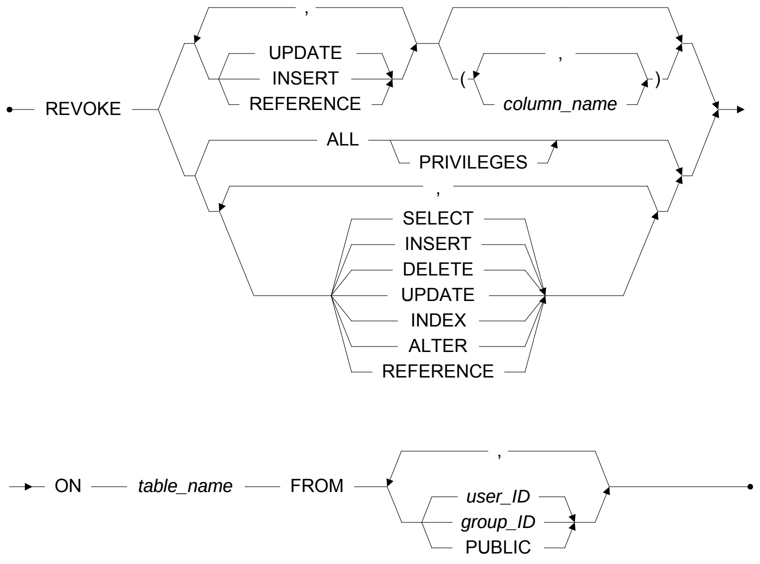


Figure 8-16 The REVOKE (object privileges) command

➲ **Example 1**

The following command revokes the SELECT privilege for the **TB_INFO** table from **Cathy**:

```
dmSQL> REVOKE SELECT ON TB_INFO FROM Cathy;
```

➲ **Example 2**

The following command revokes the SELECT privilege for table **Jeff.TB_INFO** from **Cathy**:

```
dmSQL> REVOKE SELECT on Jeff.TB_INFO FROM Cathy;
```

➲ **Example 3**

The following command revokes the UPDATE privileges on the column **PHONENO** in table **Jeff.TB_INFO** from **group1**:

```
dmSQL> REVOKE UPDATE (PHONENO) on Jeff.TB_INFO FROM group1;
```

➲ **Example 4**

The following command revokes all privileges granted to **PUBLIC** on the **TB_INFO** table:

```
dmSQL> REVOKE ALL ON TB_INFO FROM PUBLIC;
```

➲ **Example 5**

The following command revokes INSERT, UPDATE, and SELECT privileges for the **TB_INFO** table from user **Cathy** and all users in **group2**:

```
dmSQL> REVOKE INSERT, UPDATE, SELECT ON TB_INFO FROM Cathy, group2;
```

8.4 Security System Catalog

All information on authority levels, privileges, and groups is recorded in the following system catalogs:

- SYSAUTHUSER — authority level of each user
- SYSAUTHTABLE — privileges on tables

- **SYSAUTHCOL** — columns of a table to which a user has been restricted for INSERT, UPDATE and REFERENCE privileges
- **SYSAUTH** — group name, group creator, and number of group members
- **SYSACL** — user IP checking rules

The security system catalogs are owned by SYSTEM. No user including SYSADM can modify system catalogs. See *System Catalog Reference* for more details on the DBMaker system catalogs.

9 Concurrency Control

Transactions and concurrency control are described in this chapter. How DBMaker maintains concurrent access and data accuracy in a multi-user environment with the lock mechanism is also described. The Transaction Section presents the transaction concept and the functions used in managing a transaction. Section Transaction Isolation Levels describes the four transaction levels. Section Multi-User environment describes the necessity of concurrency control in a database system. Finally, section Locks explains concurrency control techniques used by DBMaker.

9.1 Transactions

In a database, a *transaction* is a work unit that is composed of one or more SQL statements. It is an *atomic* operation. That means it should either complete a series of statements entirely or do nothing at all. Serial, atomic, permanent, consistent, and isolated are the properties of a transaction.

Transaction States

A transaction must be in one of the following states:

- **Active** — When a transaction starts to execute, it immediately goes into an active state. In the active state, a transaction can perform various database operations.
- **Partially Committed** — When a transaction reaches its last statement in DBMaker (such as COMMIT WORK), it enters into the partially committed state. The transaction has completed its execution and can still be aborted if an

error occurs during the actual output. The result cannot be written to disk and a hardware failure may preclude its successful completion.

- **Committed** — When a transaction has completed its execution successfully it enters into the committed state.
- **Failed** — When a transaction cannot proceed to a normal conclusion, it enters into the failed state. This may be caused by hardware or logic errors, or a user abort of the transaction during an active state.
- **Aborted** — When a transaction has ended unsuccessfully, it enters into the aborted state. In this situation, any change or effect that a transaction has applied to the database must be rolled back.

The state diagram corresponding to a transaction is shown in Figure 9-1.

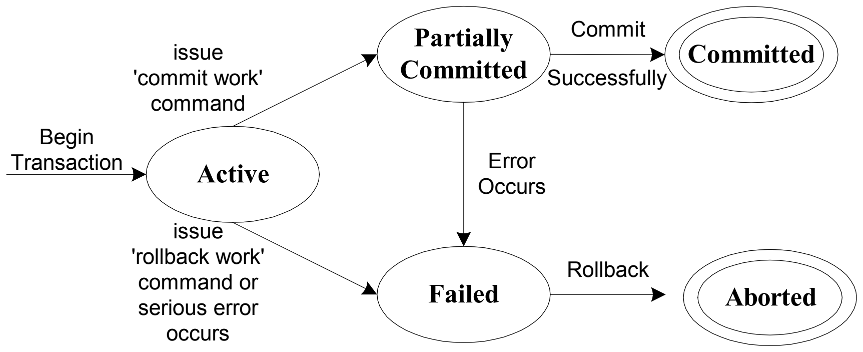


Figure 9-1 The transaction states

Managing a Transaction

When connecting to DBMaker, a transaction starts automatically and enters the active state. DBMaker will automatically begin a new transaction after the preceding transaction has been terminated.

Every time a statement executes a transaction is committed automatically by DBMaker. This is known as *autocommit mode*. In this mode, the lifetime of a transaction equals the lifetime of a single SQL statement. That means when one transaction is terminated at the end of a SQL statement, another begins with the next SQL statement. Each SQL statement is an independent transaction.

To force a transaction to remain uncommitted until several SQL statements have been executed, change to *manual commit mode* by issuing a SET AUTOCOMMIT OFF command. In this mode, a transaction can only be committed by using the SQL command COMMIT WORK. As many SQL statements as necessary can be executed before ending the transaction. To end the transaction, issue a COMMIT WORK command to commit changes, or issue a ROLLBACK WORK command to abort any changes made and terminate the transaction.

To return to autocommit mode, issue a SET AUTOCOMMIT ON command. The default transaction mode is AUTOCOMMIT ON.

NOTE *After a transaction is terminated, all resources allocated are released.*

Using a Savepoint

A *savepoint* is an intermediate point that can be arbitrarily declared within the context of a transaction. A savepoint is used to rollback the work performed after a savepoint has been declared within a transaction.

For example, a transaction with a series of statements is executed, and an error occurs while executing the twentieth statement. If a savepoint is marked between the fifteenth and sixteenth statements, the first fifteen statements can be preserved. A user can roll back to the savepoint and begin issuing commands from the sixteenth SQL statement after correcting the error. Figure 9-2 shows an example of how the user does not need to abort the transaction and resubmit all the statements.

However, if the user does not mark a savepoint between the fifteenth and sixteenth statements, the transaction must be aborted and the first fifteen statements resubmitted. This is inconvenient and wastes time. A savepoint solves this problem.

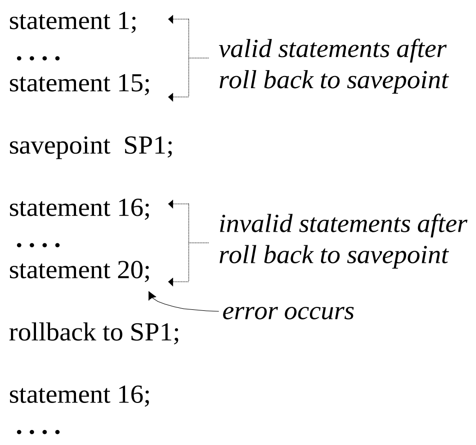


Figure 9-2 Using Savepoints

The SAVEPOINT and ROLLBACK TO … commands mark a savepoint and rollback to a specific savepoint.

➲ **Example 1**

The `SAVEPOINT` command:

```
dmSQL> SAVEPOINT <savepoint_name>;
```

➲ **Example 2**

The `ROLLBACK TO` ... command:

```
dmSQL> ROLLBACK TO <savepoint_name>;
```

The user specifies the <*savepoint_name*>. After rolling back to a savepoint, the system resources that were allocated after the savepoint, like locks, are released.

9.2 Transaction Isolation Levels

Transactions Concurrency Issues

Concurrently executing transactions in the same database, may exhibit certain unwanted phenomenon. These are commons called **dirty read**, **non-repeatable read** and **phantom read**.

The following discussion is based on the following data (unless otherwise noted).

A table table1 with column c1 having values 1, 3 and 5. Two transactions T1 and T2 execute concurrently in this table.

DIRTY READ

Definition: A transaction reads data written by a concurrent **uncommitted** transaction.

➲ **Example**

```
     T1          T2
------------  ------------
             Insert 4
Select c1<5
……            ……
             Commit or rollback
```

When T1 executes "select c1 <5", this result is returned: c1 = 1, 3, 4. However, this result may be incorrect because T2 can rollback later.

NON-REPEATABLE READ

Definition: A transaction re-reads data that it had previously read and finds that the data has been modified by another transaction.

➲ **Example**

```
    T1              T2
------------  -----------------
```

```
Select c1<5
             Update c1=2 where c1=1
             Commit
Select c1<5
```

In the first select, the result c1 = 1, 3 is returned to T1.Next, T2 updates the 1 to a value of 2 and **commits** this update. Later, T1 executes the same query and the the result 2, 3 is returned. T1 executeed the **same statement** twice but different values were returned each time**.**

PHANTOM READ

Definition: Two reads of **same predicate** return different sets of items. The second read returns at least one item not in the original set.

➲ **Example**

```
        T1                     T2
------------------          ------------
Select c1<5
                           Insert 4
                           Commit
Select c1<5
```

First, the T1 select statement executes and the result c1 = 1, 3 are returned. Next T2 insert the value 4 and **commits** it. Later T1 executes the same query statement , the result 1, 3, 4 are returned. T1 executed the **same statement** twice but different results were returned each time. Some items in the second result are not in the first result. In this example, the value 4 in the second result is the phantom.

The Four Transaction Isolation Levels

Irrespective of the three concurrency problems, the ANSI/ISO SQL defines four transaction concurrent levels:

ISOLATION LEVEL	DIRTY READ	NON-REPEATABLE READ	PHANTOM READ
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

Set Transaction Isolation levels in DBMaker

DBMaker provides three methods for setting the transaction isolation levels using the dmconfig keyword, the ODBC function, and the SQL syntax in dmsql.

DMCONFIG KEYWORD

The related keyword is **DB_IsoLv**.

```
DB_IsoLv {1,2,3,4}
1 : READ UNCOMMITTED
2 : READ COMMITTED
3 : REPEATABLE READ
4 : SERIALIZABLE
```

The default value is *1*.

The **DB_IsoLv** is set to indicate that each transaction's default isolation level. For example, if **DB_IsoLv** = 3, each transaction's default isolation level is Repeatable Read.

ODBC FUNCTION

SQLSetConnectionOption is used for setting and SQLGetConnectionOption is used for getting the current transaction isolation level.

```
SQLSetConnectOption( HDBC, SQL_ATTR_TXN_ISOLATION, level)
Level : {
SQL_TXN_READ_UNCOMMITTED,
SQL_TXN_READ_COMMITTED,
SQL_TXN_REPEATABLE_READ,
SQL_TXN_SERIALIZABLE
}
```

```
SQLGetConnectOption( HDBC, SQL_ATTR_TXN_ISOLATION, &level)
```

SQL SYNTAX

```
Type "SET TRANSACTION ISOLATION LEVEL [level]" in the command line.
      [level] :
{ READ COMMITTED
           | READ UNCOMMITTED
           | REPEATABLE READ
           | SERIALIZABLE
             }
Type "CALL GETSYSTEMOPTION('isolv',?);" in the command line in the dmsql will get
information about isolation level.
```

➲ **Example**

The get information about the isolation level:

```
dmSQL> SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
dmSQL> CALL GETSYSTEMOPTION('isolv',?);
OPTION_VALUE :SQL_TRANSACTION_READ_UNCOMMITTED
```

9.3 Multi-User Environment

When more than one user is accessing a database, consider what can happen when they try to access data simultaneously.

Sessions

A *connection* is a communication pathway between a user and DBMaker. A communication pathway is established using shared memory or a network.

Before using the database resources, establish a connection to DBMaker using the following SQL statement.

➲ **Example**

To connect a user to a DBMaker database:

```
dmSQL> CONNECT TO database_name user_name password;
```

When a user connects to a DBMaker database, the specific connection is called a *session*. A session lasts from the time a user connects to a DBMaker database until the time the user disconnects from it. A session can only have one active transaction at a time.

The Necessity of Concurrency Control

In a multi-user database system environment, more than one user can connect to a database at the same time. This could possibly result in many transactions updating the same database simultaneously.

If no concurrency control mechanism is used, several situations could result in data inconsistency:

- The lost update problem
- The temporary update problem
- The incorrect summary problem

LOST UPDATE PROBLEM

A lost update problem occurs when two transactions update a data item at approximately the same time.

➲ **Example**

Transactions T1 and T2 read and modify the value of X but use different calculations to modify the value. This results in the transactions each containing a different value for X. T1 writes the value it holds for X to the database after it is read but before it is written by T2. T2 then writes the value it holds for X to the database, overwriting the value written by T1. The value written by T1 is lost:

```
      T1                      T2
--------------          --------------
read(X);
                        read(X);
X = X - N;
                        X = X + M;
write(X);
```

```
                         write(X);
```

Temporary Update Problem

A temporary update problem occurs when a transaction updates a value, but is rolled back after another transaction updates the same value.

➲ **Example**

Transaction T1 reads and modifies the value of X, writes it back to the database, and then continues with other commands. While transaction T1 continues executing, transaction T2 reads the value of X, modifies it to a new value, and writes it back to the database. Transaction T1 then fails before completion, and must roll back all values to restore the database to its original status. The database management system restores the original value of X, overwriting the value written by transaction T2. The value of X calculated by transaction T2 exists only temporarily:

```
      T1                        T2
--------------  --------------
read(X);        read(X);
X = X - N;
write(X);       X = X + M;
                write(X);
rollback;
```

Incorrect Summary Problem

An incorrect summary problem occurs when a transaction is calculating the aggregate sum of a number of records while other transactions are updating those records.

➲ **Example**

Transaction T1 calculates the aggregate sum using the values of X and Y at the same time transaction T2 is modifying those values. Transaction T2 updates the value of X before transaction T1 uses it to calculate the sum, and updates the value of Y after transaction T1 uses it to calculate the sum. This results in transaction T1 using some values to calculate the sum before they are updated, and using others after they are updated. When both transactions complete, the value of the sum is incorrect with respect to the values in the database:

```
      T1               T2
----------------  --------------
sum = 0;
                  read(X);
                  X = X - N;
                  write(X);
read(X);
sum = sum + X;
read(Y);
sum = sum + Y;
                  read(Y);
                  Y = Y + N;
                  write(Y);
```

There are various techniques to solve concurrency problems, such as locks and time stamps. The next section shows how the locking technique is applied in DBMaker to control concurrent execution of transactions.

9.4 Locks

In this section, the lock concept is first presented. Then, the DBMaker lock mechanism is introduced, including lock granularity and lock modes. Finally, dealing with deadlock is demonstrated.

Lock Concept

In general, a multi-user database system uses several forms of locking to synchronize the access of concurrent transactions. Before accessing the data objects, such as tables and tuples, a transaction must lock those data objects.

DBMaker locking is fully automatic and does not require any user action. Implicit locking occurs in all SQL statements; the users do not need to explicitly lock any data objects in the database.

Shared and Exclusive Locks

In general, three types of locking are used to allow multiple-read with single-write operations in a multi-user database.

- **Share Locks (S)** — A transaction involving a read operation on a data object. To support a higher degree of data concurrency, several transactions can acquire share locks on the same data object at the same time.
- **Update Locks (U)** — A transaction involving an intended to update operation or update operation on a data object. This lock is compatible with Share locks but is not compatible with Exclusive locks. An object can have just one Update lock at a time
- **Exclusive Locks (X)** — A transaction involving an update operation on a data object. This transaction is the only one that can access the object until the exclusive lock is released

Two-Phase Locking

The two-phase locking protocol is used to ensure the transactions are serialized. In the two-phase locking protocol, each transaction must issue all lock requests before it can issue any unlock requests.

The protocol can be divided into two phases:

- **Expanding (growing) phase** — This phase allows the transaction to issue any new lock requests that are required. Unlock requests are not permitted in this phase.
- **Shrinking phase** — This phase allows the transaction to release locks acquired in the expanding phase. New lock requests are not permitted in this phase.

The two-phase locking protocol is currently used by DBMaker to provide concurrency control by serializing transactions.

Deadlock

When two or more transactions are waiting for the release of data locked by other transactions before it can proceed, a deadlock occurs.

➲ **Example**

T1 is waiting for T2 to release the share lock of X, while T2 is waiting for T1 to release the share lock of Y. Therefore, deadlock occurs and the system will wait indefinitely:

```
        T1                  T2
-------------------  --------------
share_lock(Y);
read(Y);
                     share_lock(X);
                     read(X);
exclusive_lock(X);
(T1 waits for T2)    exclusive_lock(Y);
                     (T2 waits for T1)
```

Lock Granularity

There are three granularity levels for data locks in DBMaker: relation (table), page, and tuple (row). A relation contains several pages, and a page contains several tuples.

A lock applied on a higher level carries through to lower levels. For example, if a user gets an exclusive lock (X lock) on a relation, all pages and tuples that are included in this relation will have the X lock applied to them. Therefore, no user can access any tuple or page from this relation. However, if a user gets an X lock on a tuple, another user can get an X lock on another tuple simultaneously. There is no interference between two objects at the same level when using the X lock. Figure 9-3 shows the lock granularity (levels) in DBMaker.

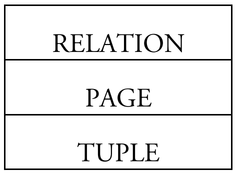


Figure 9-3: Lock granularity

Using a higher lock granularity results in a lower degree of data concurrency, in contrast, the higher lock granularity uses fewer system resources (such as shared memory). Selecting the lock granularity level is a trade-off between concurrency and resources. In DBMaker, the default lock granularity level is row, but if a different lock granularity is required, it can be specified when creating a table. Refer to Chapter 5, *Storage Architecture* for more information.

Lock Types

The main lock modes (types) supported in DBMaker are shared (S) , update (U) and exclusive (X) locks. More than one user can have an S lock on a data object simultaneously, but only one user can have an X lock or U lock on a data object. In addition to S, U, and X locks, another lock mode called an *intention lock* is supported.

When a data object is locked, the system will automatically assign an intention lock to the next higher granularity object. For example, an S lock specified on a tuple will generate an intention S (IS) lock on the page which includes this tuple, and an IS lock on the relation which the tuple belongs to.

The supported intention lock modes are:

- IS—Indicates that the S lock is specified at a lower granularity
- IU—Indicates that the U lock is specified at a lower granularity
- IX—Indicates that the X lock is specified at a lower granularity
- SIX—Indicates that an S lock is specified at the current granularity and an X lock is specified at a lower granularity. This is a combination of S and IX locks
- SIU—Indicates that an S lock is specified at the current granularity and an U lock is specified at a lower granularity. This is a combination of S and IU locks
- UIX—Indicates that an U lock is specified at the current granularity and an X lock is specified at a lower granularity. This is a combination of U and IX locks

The result from the compatibility of each of the lock modes is listed in Table 9-1. T represents true, which means the matrix for each of the two lock modes are compatible and can exist on a data object simultaneously. F represents false, which means the matrix for each of the two lock modes are not compatible and cannot exist simultaneously.

If lock requests on a data object conflicts with an existing lock on that object, this request will not execute until the existing lock is released, or until the waiting time for the lock request times out. If the error message 'Lock timeout' is returned to the user, the waiting time for the lock has expired. The default waiting time is 5 seconds. However, users can specify a different waiting time by setting the value of the **DB_LTimO** keyword in the **dmconfig.ini** file to another value according to their individual requirements.

➲ **Example**

The following shows how to set the waiting time to 8 seconds:

```
DB_LTimO = 8;
```

	N	IS	S	IU	SIU	IX	U	SIX	UIX	X
N	T	T	T	T	T	T	T	T	T	T
IS	T	T	T	T	T	T	T	T	T	F
S	T	T	T	T	T	F	T	F	F	F
IU	T	T	T	T	T	T	F	T	F	F
SIU	T	T	T	T	T	F	F	F	F	F
IX	T	T	F	T	F	T	F	F	F	F
U	T	T	T	F	F	F	F	F	F	F
SIX	T	T	F	T	F	F	F	F	F	F
UIX	T	T	F	F	F	F	F	F	F	F
X	T	F	F	F	F	F	F	F	F	F

Table 9-1: Compatibility matrix for lock modes

Dealing with Deadlock

By analyzing the "wait for" graph, DBMaker automatically detects a deadlock situation. If a deadlock is detected, a victim transaction is aborted to solve the deadlock problem.

➲ **Example**

DBMaker detects a deadlock when transaction T2 issues an X lock on Y. Transaction T2 will be aborted to resolve the deadlock problem and the user executing transaction T2 will receive the error message, "`transaction aborted due to deadlock`":

```
        T1                T2
------------------  ------------
share_lock(Y);
read(Y);
                    share_lock(X);
                    read(X);
exclusive_lock(X);
(T1 waits for T2)   exclusive_lock(Y);
                    (T2 waits for T1)

                    T2 aborted by DBMaker
```

10 Triggers

Triggers are a very useful and powerful feature of the DBMaker database server. Triggers automatically execute predefined commands in response to specific events, regardless of which user or application program generated them.

Triggers allow a database to be customized in ways that may not be possible with standard SQL commands. The database can consistently control complex or unconventional database operations without requiring any action on the part of users or application programs.

Use triggers to:

- Implement business rules
- Create an audit trail for database activities
- Derive additional values from existing data
- Replicate data across multiple tables
- Perform security authorization procedures
- Control data integrity
- Define unconventional integrity constraints

Exercise restraint when using triggers to avoid forming complex interdependencies within the database that may be difficult to follow and change. Use triggers only when the desired functionality cannot be implemented using standard SQL commands and integrity constraints.

10.1 Trigger Components

DBMaker stores trigger definitions in the system catalog.

Every DBMaker trigger has six main components:

- **Trigger Name** — a name that uniquely identifies the trigger
- **Trigger Action Time** — the time relative to when a trigger will be fired
- **Trigger Event** — a specific situation that occurs in the database in response to some user action, such as inserting data into a table
- **Trigger Table** — the name of the table the trigger executes on
- **Trigger Action** — a SQL statement or stored procedure that is executed when the trigger event occurs
- **Trigger Type** — the type of trigger

Each of these components must be present in all triggers. In addition, there is an optional component, the REFERENCING clause.

Trigger Name

The trigger name uniquely identifies a trigger. Trigger names have a maximum length of 128 characters and may contain letters, numbers, the underscore character, and the symbols # and $. The first character cannot contain a number, and the name cannot contain spaces.

Trigger Action Time

The trigger action time specifies whether it should fire before or after the SQL statement that activates it. The trigger action time is specified by the BEFORE and AFTER time keywords. The BEFORE keyword instructs the trigger to fire before the trigger statement. The AFTER keyword instructs the trigger to fire after the trigger statement. Only one trigger time can be specified for each trigger.

Trigger Event

The trigger event is the database operation that causes a trigger to operate, or *fire*. The trigger event may be an INSERT, UPDATE or DELETE statement that operates on the trigger table. There can be only one trigger event for each trigger statement. However, multiple trigger events can be used to activate multiple triggers.

Trigger Table

The trigger event operates on the associated trigger table. The trigger table must be a base table; it cannot be a temporary table, view or synonym. A trigger may only have one trigger table.

Trigger Action

A trigger action is the command that a trigger executes when it fires. The trigger action may be an INSERT, UPDATE, DELETE, EXECUTE PROCEDURE or SQL block statement. A trigger can only have a single trigger action.

Trigger Type

The trigger type specifies how many times the trigger will fire for each trigger event. There are two types of triggers: row triggers and statement triggers. The FOR EACH ROW option specifies a row trigger, which fires a trigger action once for each row modified by the trigger event. The FOR EACH STATEMENT option specifies a statement trigger, which fires a trigger action once for each trigger event.

REFERENCING Clause

The REFERENCING clause defines correlating names for the old and new values in a column. This is primarily used when the default OLD and NEW names cannot be used because of a name conflict with a table.

10.2 Trigger Operation

DBMaker checks to see if a trigger should be fired and will execute the defined triggers each time a user or an application program causes a trigger event. Firing triggers from within a database ensures that DBMaker handles data consistently across all applications. This guarantees that when a specific event occurs, a related action is also performed.

Users can create triggers to implement domain, column, referential, and unconventional integrity constraints. However, these can also be done by declarative integrity control. Triggers do not have an owner, but are associated with a table.

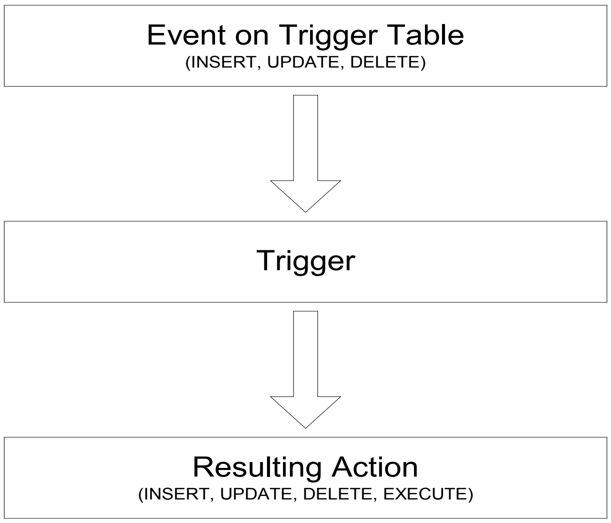


Figure 10-1 Trigger event and action

10.3 Creating Triggers

The CREATE TRIGGER command creates a new trigger associated with a specific table. Only a user with privilege on the trigger table can execute the command. The user must also have the necessary object privileges for all objects referenced in the trigger definition in order to successfully create a trigger.

Basic Requirements

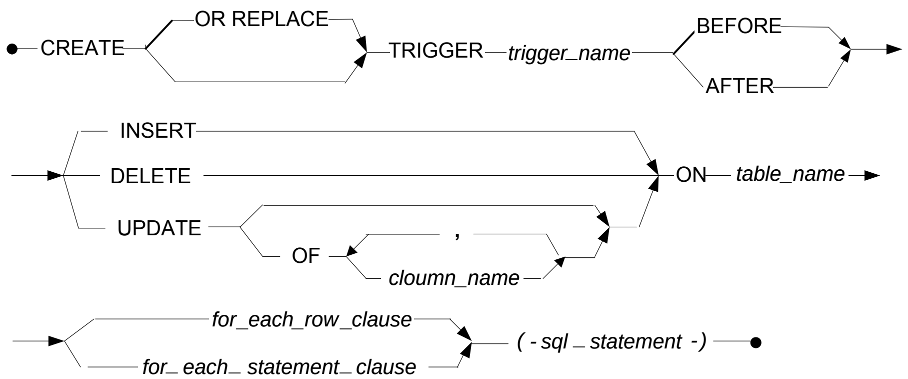
All of the CREATE TRIGGER statements must contain at least the following:

- A trigger name
- The trigger action time (before or after)
- The trigger event
- The trigger table
- The trigger type (row or statement)
- The trigger action

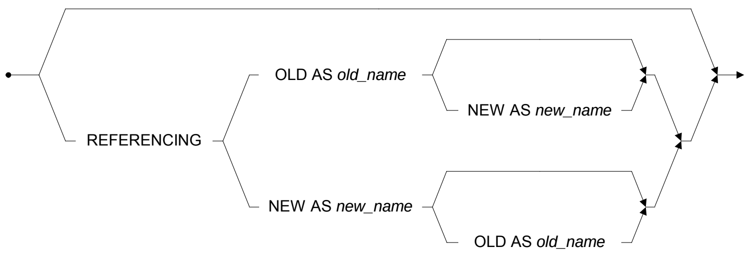
Security Privileges

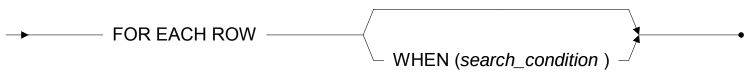
All SQL statements in the trigger action operate with the same privileges as the owner of the trigger table, and not with the privileges of the user executing the trigger event. If the trigger exists, any user executing the trigger event will result in the trigger firing.

CREATE TRIGGER Syntax



FOR EACH ROW clause





FOR EACH STATEMENT clause

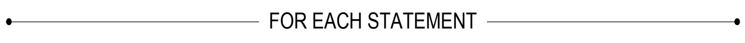


Figure 10-2 Syntax for the CREATE TRIGGER Statement

OR REPLACE is used to re-create the trigger that already exists, that is to say, users can use this clause to change the definition of an existing trigger without dropping it.

➲ **Example**

The following statement create or replace a trigger on table **tb_staff**:

```
dmSQL> CREATE OR REPLACE TRIGGER tr_staff_insert AFTER INSERT ON tb_staff
       FOR EACH ROW WHEN (new.ID > 0)
       (INSERT INTO tb_salary(new.ID, new.Name,NULL, NULL, NULL));
```

Specifying the Trigger Action Time

You can use the trigger time and trigger type in combination to create four triggers for each table for the same event (INSERT, DELETE, or UPDATE). For each event the BEFORE/FOR EACH ROW, AFTER/FOR EACH ROW, BEFORE/FOR EACH STATEMENT and AFTER/FOR EACH STATEMENT combinations are possible.

A BEFORE/FOR EACH STATEMENT trigger executes once and only once before the triggering statement is performed. That is before the occurrence of the trigger event. An AFTER/FOR EACH STATEMENT trigger executes once and only once after the triggering statement is complete. Note that BEFORE and AFTER statement triggers are executed even if the triggering statement does not process any rows.

BEFORE OR AFTER INSERT OR DELETE TRIGGER EVENTS

The following examples show how to create triggers that fire before or after INSERT or DELETE trigger events. The trigger action is represented by <*sql_statement*>.

➲ **Example 1**

To define four triggers for an INSERT event on table **tb**:

```
dmSQL> CREATE TRIGGER tr1 BEFORE INSERT ON tb FOR EACH STATEMENT <sql_statement>;
dmSQL> CREATE TRIGGER tr2 BEFORE INSERT ON tb FOR EACH ROW <sql_statement>;
dmSQL> CREATE TRIGGER tr3 AFTER INSERT ON tb FOR EACH ROW <sql_statement>;
dmSQL> CREATE TRIGGER tr4 AFTER INSERT ON tb FOR EACH STATEMENT <sql_statement>;
```

➲ **Example 2**

To define four triggers for a DELETE event on table **tb**:

```
dmSQL> CREATE TRIGGER tr1 BEFORE DELETE ON tb FOR EACH STATEMENT <sql_statement>;
dmSQL> CREATE TRIGGER tr2 BEFORE DELETE ON tb FOR EACH ROW <sql_statement>;
dmSQL> CREATE TRIGGER tr3 AFTER DELETE ON tb FOR EACH ROW <sql_statement>;
dmSQL> CREATE TRIGGER tr4 AFTER DELETE ON tb FOR EACH STATEMENT <sql_statement>;
```

BEFORE OR AFTER THE UPDATE TRIGGER EVENT

The situation is different for UPDATE events. Two types of UPDATE triggers can be created: UPDATE <*table*> triggers, or UPDATE OF <*column*> triggers. An UPDATE <*table*> trigger fires whenever the table is updated. An UPDATE OF <*column*> trigger fires when specific columns are updated. Either one UPDATE <*table*> trigger or multiple UPDATE OF <*column*> triggers can be created on a single table. UPDATE OF <*column*> triggers may contain multiple columns, but columns in all UPDATE OF <*column*> triggers in a table must be mutually exclusive.

➲ **Example**

To create a column trigger **tr_UpdateColumn** on table **tb_salary** of columns **basepay**, **bonus** that has four columns, **id**, **name**, **basepay**, and **bonus**:

```
dmSQL> CREATE TRIGGER Tr_UpdateColumn AFTER UPDATE OF basepay,bonus ON tb_salary
       FOR EACH ROW
       (INSERT INTO tb_OldSalary VALUES (old.basepay, old.bonus));
```

If a second UPDATE column trigger **tr_UpdateBonus** that specifies column **bonus** is created, the command will fail because **bonus** already appears in trigger **tr_UpdateColumn**:

```
dmSQL> CREATE TRIGGER tr_UpdateBonus AFTER UPDATE OF bonus,tax ON tb_salary
                           FOR EACH ROW
                           (INSERT INTO tb_oldTax VALUES (old.bonus, old.tax));
ERROR (6150): [DBMaker] the insert/update value type is incompatible with column
data type or compare/operand value is incompatible with column data type in
expression/predicate
```

If there are four columns in a table, you can create at most four UPDATE column triggers or one UPDATE table trigger, for triggers of the same type (for instance, a BEFORE/FOR EACH ROW trigger).

FOR EACH ROW / FOR EACH STATEMENT Clause

The FOR EACH STATEMENT clause specifies that a trigger will fire once and only once for each trigger event. The trigger fires even if the trigger event statement does not process any rows.

The FOR EACH ROW clause specifies that a trigger will fire once for each row that the trigger event modifies. If the trigger event does not modify any rows, the trigger will not fire. The OLD and NEW keywords are used to identify which values from the trigger table are to be used in the trigger action. The OLD keyword indicates that trigger table values from before the trigger event are used in the trigger action. The NEW keyword indicates that trigger table values from after the trigger event are used in the trigger action.

➲ **Example 1**

The following statement shows how to create an UPDATE column trigger on table **tb_Sales**. The **totSales** field is a calculated field derived from the two fields: **unitPrice** and **unitSale**. Both **unitPrice** and **unitSale** are triggering columns.

```
dmSQL> CREATE TRIGGER tr_TotalSale AFTER UPDATE OF unitPrice, unitSale
                ON tb_Sales FOR EACH ROW
                        (UPDATE tb_Sales
                                  SET totSales = new.unitPrice * new.unitSale);
```

➲ **Example 2**

In this example, there are four triggers.

```
dmSQL> CREATE TRIGGER tr_BeforeUpdatePro BEFORE UPDATE ON tb_Orders
                     FOR EACH STATEMENT
                    (EXECUTE PROCEDURE checkPrivilege);

dmSQL> CREATE TRIGGER tr_BeforeUpdate BEFORE UPDATE ON tb_Orders
                     FOR EACH ROW
                    (INSERT INTO tb_Old_Value (old.customer, old.amount));

dmSQL> CREATE TRIGGER tr_AfterUpdate AFTER UPDATE ON tb_Orders
                     FOR EACH ROW
                    (INSERT INTO tb_New_Value (new.customer, new.amount));
```

```
dmSQL> CREATE TRIGGER tr_AfterUpdatePro AFTER UPDATE ON Orders
                       FOR EACH STATEMENT
                      (EXECUTE PROCEDURE Log_Time);
```

If a user executes an UPDATE statement that changes two rows of the **tb_Orders** table, the effect and order of the execution is as follows:

1. Procedure checkPrivilege is called
2. Insert one row to tb_Old_Value table
3. Update one row
4. Insert one row to tb_New_Value table
5. Insert one row to tb_Old_Value table
6. Update one row
7. Insert one row to tb_New_Value table
8. Procedure Log_Time is called

Stored procedures cannot contain COMMIT, ROLLBACK, or SAVEPOINT transaction control statements. Triggers can specify only a single triggered action, which must be enclosed in parentheses.

Using the Referencing Clause

In row triggers, the <*sql_statement*> (or action body) should indicate whether the column values used are from before or after the trigger event. For example, to log the old price and new price when updating the price of a sale item, use the keywords OLD and NEW as shown in example 2 in the section *FOR EACH ROW / FOR EACH STATEMENT Clause*.

However, in some rare cases the tables may contain columns with the names NEW or OLD. If this is the case, use the referencing clause to define correlation names. The reference clause allows for the creation of two prefixes that can be used with a column name: one to reference the old value of the column, and one to reference the new value. These prefixes are called correlation names. Use the keywords OLD and NEW to indicate the correlation names.

➲ **Example**

```
dmSQL> CREATE TRIGGER tr_log_price AFTER UPDATE OF price ON New
```

```
                                    REFERENCING OLD as pre NEW as post
                                    FOR EACH ROW
                                    (INSERT INTO logTbl
                                     VALUES (item_no, today(), pre.price,
                                     post.price));
```

In this example, the triggering table name is NEW, so the correlation names **pre** and **post** are used in the action body. Referencing clauses are only valid for row triggers, and are not allowed in statement triggers.

If a trigger event is INSERT, there is no old value for the newly inserted record, so the old value is not available. Similarly, if the trigger event is DELETE, there is no new value for the deleted record, so the new value is not available. For an UPDATE event trigger, both old and new values are available.

Using the WHEN Condition

A WHEN condition clause may precede a FOR EACH ROW triggered action to make the action execution dependent on the result of a Boolean expression. The WHEN clause consists of the keyword WHEN followed by a parenthetical conditional statement. The WHEN clause follows an action time but precedes the triggered action body. The WHEN clause is not allowed in the definition of a statement trigger, it is only allowed in row trigger.

➲ **Example 1**

The following trigger logs a customer complaint into the **tb_logComplain** table when a customer calls to complain about something. Assume the call code '**c**' means it is a complaint call.

```
dmSQL> CREATE TRIGGER tr_log_complain AFTER INSERT ON tb_Customer_Call
                                      FOR EACH ROW
                                      WHEN (new.call_code = 'c')
                                      (INSERT INTO tb_logComplain
                                       VALUES (CURRENT DATE(), Cus_Name));
```

The WHEN clause is evaluated for each row when the WHEN condition is included in a trigger definition. If the WHEN condition evaluates to TRUE for a row, the triggered action is fired for that row. If the WHEN condition evaluates to FALSE or unknown for a row, the triggered action is not fired for that row.

The result of WHEN condition only affects the execution of the triggered action, it has no effect on the triggering statement.

➲ **Example 2**

To create three triggers to record all INSERT, UPDATE and DELETE operations on table **tb_staff**:

```
dmSQL> CREATE TRIGGER tr_staff_insert AFTER INSERT ON tb_staff
                                      FOR EACH ROW
                                      (INSERT INTO tb_salary
                                    VALUES (new.Id, new.Name, NULL, NULL, NULL));

dmSQL> CREATE TRIGGER tr_staff_update AFTER UPDATE ON tb_staff
                                      FOR EACH ROW
                                      (INSERT INTO tb_staff_bak
                                     VALUES (old.Id, old.Name,new.Id, new.Name));

dmSQL> CREATE TRIGGER tr_staff_upd AFTER DELETE ON tb_staff
                                      FOR EACH ROW
                                      (INSERT INTO tb_staff_bak
                                       VALUES (old.Id, old.Name,
                                       NULL, NULL));
```

➲ **Example 3**

If a primary key is changed, all the foreign keys can be changed in cascade. Suppose **deptNo** is the primary key on table **tb_dept**, **id** is foreign key on table **tb_staff**.

```
dmSQL> CREATE TRIGGER tr_dept_update BEFORE UPDATE OF deptNo ON tb_dept
                                    FOR EACH ROW
                                    WHEN (NEW.deptNo <> OLD.deptNo)
                               (UPDATE tb_staff SET tb_staff.ID = NEW.deptNo
                                     WHERE tb_staff.ID = OLD.deptNo);
```

➲ **Example 4**

If the primary key is deleted, all the foreign keys can be deleted in cascade.

```
dmSQL> CREATE TRIGGER tr_dept_delete BEFORE DELETE ON tb_dept
```

```
                                    FOR EACH ROW
                                    (DELETE FROM tb_staff
                                     WHERE tb_staff.ID = OLD.deptNo);
```

➲ **Example 5**

If a primary key is updated, all the foreign keys can be set to NULL.

```
dmSQL> CREATE TRIGGER tr_dept_delete BEFORE UPDATE ON tb_dept
                                    FOR EACH ROW
                                    (UPDATE tb_staff set ID = NULL
                                     WHERE tb_staff.ID= OLD.deptNo);
```

➲ **Example 6**

If the number of parts in stock is lower than a given level, the parts should be reordered. The part number and quantity will be recorded to a table called **tb_pending_orders** for further action.

tb_Inventory: part_no int, parts_on_hand int, reorder_level int, reorder_qty int

tb_pending_orders: part_no int, qty int, order_date date

```
dmSQL> CREATE TRIGGER tr_reorder AFTER UPDATE OF parts_on_hand ON tb_Inventory
                                 FOR EACH ROW
                                 WHEN (new.parts_on_hand < new.reorder_level)
                                 (INSERT INTO tb_pending_orders
                                  VALUES (new.part_no, new.reorder_qty,
                                  CURRENT DATE()));
```

Specifying the Trigger Action

The trigger action is the SQL statement that is performed when the trigger event occurs. The trigger action can be an INSERT, DELETE, UPDATE, EXECUTE PROCEDURE or SQL block statement. No other statements are allowed. Stored procedures cannot contain COMMIT, ROLLBACK or SAVEPOINT transaction control statements. Triggers can specify only a single trigger action, which must be enclosed in parentheses.

➲ **Example**

The following statement creates a trigger on table **tb_staff**.

```
dmSQL> CREATE TRIGGER tr_staff_insert AFTER INSERT ON tb_staff
                                  FOR EACH ROW WHEN (new.ID > 0)
                                  (INSERT INTO tb_salary(new.ID, new.Name,NULL,
                                  NULL, NULL));
```

In this example, the trigger name is **tr_staff_insert**. The AFTER option is specified, which means this trigger will be fired after the INSERT statement executes on table **tb_staff**. The triggering event is INSERT, the triggering table is **tb_salary**. The trigger type is FOR EACH ROW. The SQL action that is triggered is INSERT.

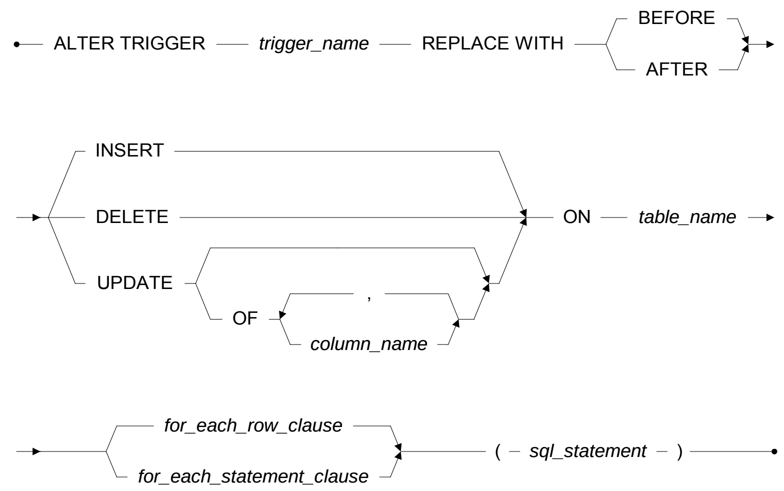
```
dmSQL> CREATE TRIGGER tr_salary_Del AFTER DELETE ON tb_salary
                                FOR EACH ROW
                                (INSERT INTO tb_old_salry
                                 VALUES (Old.name));
```

In the above example, the trigger **tr_salry_Del** will add the deleted customer name into the **tb_old_salry** table when one deletes a record from the **tb_salary** table. You cannot create a trigger on a temporary table, view or system table.

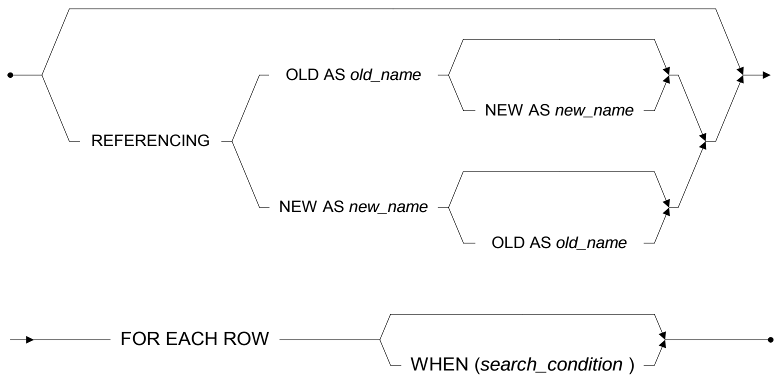
10.4 Modifying a Trigger

A trigger cannot be modified, but its definition can be replaced. When you want to modify a trigger definition, use the ALTER TRIGGER statement.

ALTER TRIGGER Syntax



FOR EACH ROW clause



FOR EACH STATEMENT clause

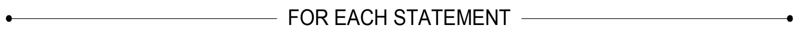


Figure 10-3 Syntax for the ALTER TRIGGER command

Replacing a Trigger Action

To replace a trigger action, use the statement ALTER TRIGGER *tr_name* REPLACE WITH...

➲ **Example 1**

If a manager quits then their data needs to be deleted from the **tb_manager** table. To create a trigger on the **tb_staff** table:

```
dmSQL> CREATE TRIGGER tr_staff_del AFTER DELETE ON tb_staff
                             FOR EACH ROW
                             ( DELETE FROM tb_manager WHERE Id = old.Id );
```

➲ **Example 2**

It is possible to add another condition to the triggered action, such as "delete the data from **tb_manager** table only when the employee is a project manager." To replace a trigger action on the **tb_staff** table and add a condition:

```
dmSQL> ALTER TRIGGER tr_staff_del REPLACE WITH AFTER DELETE ON tb_staff
                            FOR EACH ROW
                            ( DELETE FROM tb_manager
                              WHERE Id = old.Id
                              AND title = 'Project Mananger');
```

Alternatively, the trigger can be dropped and recreated.

10.5 Dropping a Trigger

The DROP TRIGGER statement can be used to delete a trigger from the database.

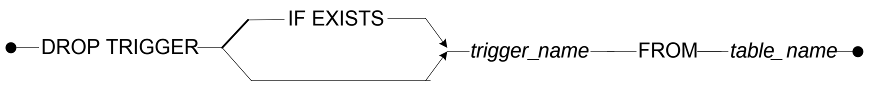


Figure 10-4 Syntax for the DROP TRIGGER statement

Dropping the Trigger

Deleting a table will cause triggers referencing the table to be deleted. When a table schema is altered, DBMaker tries to execute the trigger according to the new table definition the next time the trigger is executed. If the specified column in a triggering event or action is dropped, the trigger execution and statement will fail. The only solution is to drop the trigger or modify the trigger definition according to the new table schema. Drop a trigger by specifying the name of the trigger to delete and the associated table.

➲ **Example 1**

To drop the **myTrigger** trigger from **myTable** table with DROP TRIGGER command:

```
dmSQL> DROP TRIGGER myTrigger FROM myTable;
```

➲ **Example 2**

To drop the **myTrigger** trigger from **myTable** table with DROP TRIGGER IF EXISTS command:

```
dmSQL> DROP TRIGGER IF EXISTS myTrigger FROM myTable;
```

➲ **Example 3**

To create a trigger named **tr_staff_upd** for table **tb_staff**:

```
dmSQL> CREATE TRIGGER tr_staff_upd AFTER UPDATE ON tb_staff
                          FOR EACH ROW
                          ( DELETE FROM tb_salary WHERE id = old.id );
```

If the column **id** in table **salary** is dropped or the type is changed, an execution error will occur when the triggering statement (update on **tb_staff**) is performed causing the DBMS to attempt to fire trigger **tr_staff_upd**.

10.6 Using Triggers

There are several ways to use triggers.

Stored Procedures in Action Body

One of the most powerful trigger features is the ability to use a stored procedure as a trigger action. The EXECUTE PROCEDURE statement calls a stored procedure, enabling data to pass from the triggering table to the stored procedure and then execute the procedure.

➲ **Example**

To create a trigger and use the EXECUTE PROCEDURE statement:

```
dmSQL> CREATE TRIGGER tr_sales_update AFTER UPDATE OF price ON tb_Sales
                                 FOR EACH ROW
                                 (EXECUTE PROCEDURE
                                 logPrice(item_no, new.price, old.price));
```

Users can pass values to a stored procedure in the argument list. If the stored procedure call is part of the action for a row trigger, users can use the OLD and NEW correlation values to pass the column values it. If the stored procedure is part of an action statement trigger, only constants can pass to the stored procedure.

Within a trigger action, you can update non-triggering columns in the triggering table, with or without a stored procedure. A stored procedure fired by a trigger cannot contain transaction control statements, like BEGIN WORK, COMMIT WORK, ROLLBACK WORK, SAVEPOINT or DDL statements.

The stored procedure as a trigger action cannot be a cursory procedure that returns more than one row.

SQL Block in Action Body

Triggers also can use a trigger SQL block as a trigger action. The trigger SQL block contains a set of SQL statements that can be temporarily created and executed by a database.

The action and syntax of a trigger SQL block are similar to that of an anonymous SQL Block. A trigger SQL block allows users to execute a set of SQL statements (batch of SQL) when trigger action is executed, and supports all SQL syntax blocks including variables, grammar logic and cursors etc. At the same time, the SQL statements in the SQL block can still use the *OLD/NEW* parameters and *REFERENCING* clause.

A trigger SQL block contains more than one SQL statement, and each statement is end with ';'. So dmSQL must support block delimiter. Block delimiter can be a serial of a-z, A-Z, @, %, ^, and contains two characters at least and seven characters at most. In block delimiter, ';' doesn't denote end of input. Users must set block delimiter before writting the trigger SQL block in dmSQL. Otherwise, an error will be returned. For more information on SQL block's variables and syntax logic, please refer to Chapter 12, *Anonymous Stored Procedures.*

NOTE *A trigger SQL block' compound statements are bounded by the keywords "**BEGIN**" and "**END**".*

➲ **Example 1**

To create a SQL block in a trigger action:

```
dmSQL> set block delimiter @@;
dmSQL> create table tab_t1(c1 int,c2 int);
dmSQL> create table tab_t2(c1 int,c2 int);
dmSQL> insert into  tab_t1 values(111,222);
dmSQL> @@
     2> Create trigger uptrg before update on tab_t1 for each row
     3> begin
     4> insert into tab_t2 values(new.c1,new.c2);
     5> insert into tab_t2 values(old.c1,old.c2);
     6> end;
     7> @@
dmSQL> update tab_t1 set c1= 1 where c1 = 111;
dmSQL> select * from tab_t1;
dmSQL> select * from tab_t2;
```

➲ **Example 2**

If a table contains column named NEW or OLD, the trigger SQL block supports REFERENCING clause.

```
dmSQL> Create table tk_tb1(c1 int,new int,old int);
dmSQL> Create table tk_tb2(c1 int,new int,old int);
dmSQL> set block delimiter @@;
dmSQL> @@
    2> Create trigger uptrg before update on tk_tbl
    3> REFERENCING OLD as pre NEW as post
    4> for each row
    5> begin
    6> insert into tk_tb2 values(pre.c1, pre.new, pre.old);
    7> insert into tk_tb2 values(post.c1, post.new, post.old);
    8> end;
    9> @@
```

Trigger Execution Order

The column numbers in the triggering columns determine the order of trigger execution. The trigger execution begins with the trigger with the smallest triggering column number and proceeds in order to the highest number. In the following example a = column1, b = column 2, c = column 3 and d = column 4.

➲ **Example**

The operation UPDATE **t1** SET b = b + 1, c = c + 1 will fire both triggers. Trigger **trig1**, having a lower triggering column number than **trig2**, will be executed first. The following assumes four columns named **a**, **b**, **c** and **d** from table **t1**.

```
dmSQL> CREATE TRIGGER trig1 AFTER UPDATE OF a,c ON t1
                            FOR EACH STATEMENT (UPDATE t2 set c1=c1+1);

dmSQL> CREATE TRIGGER trig2 AFTER UPDATE OF b,d ON t1
                            FOR EACH STATEMENT (UPDATE t2 set c2=c2+1);
```

Security and Triggers

First, the user must have permission to run the trigger event; otherwise, the user cannot trigger the event. However, the user does not have to have permission to run the triggered action because the SQL statements in the triggered action operate under the domain privilege of the trigger owner. Once a trigger is created successfully, the trigger creator has privilege to execute the triggered action. Any one else who can issue the triggering statement can also fire the trigger.

➲ **Example**

User **B** can update on both tables **T1** and **T2**, and user **A** can update **T1**, but not **T2**. Now user **B** creates a trigger on update **T1**, and the action updates **T2**. When user **A** updates **T1**, the triggered action (**update T2**) is executed successfully since the triggered action is running under the domain privilege of user **B**. This security rule simplifies execution and eliminates the requirement for the user to have more privileges to execute the triggered action.

Cursors and Triggers

UPDATE or DELETE statements within a cursor act differently than a single update or delete statement. The entire trigger will be executed with each update or delete with the WHERE CURRENT OF clause.

For example, if four rows are changed with a cursor, the BEFORE/FOR EACH STATEMENT, BEFORE/FOR EACH ROW, AFTER/FOR EACH STATEMENT and AFTER/FOR EACH ROW triggers will be executed once for each row for a total of four times.

Cascading Triggers

Executing one trigger may cause another trigger to also be executed. You can use cascading triggers to enforce referential integrity. DBMaker supports a maximum of 64 cascading triggers.

➲ **Example**

To first delete a customer from the **tb_customer** table, trigger the action to delete customer related records in the **tb_order** table, which in turn will trigger the action to delete order related records in the **tb_item** table:

```
dmSQL> CREATE TRIGGER tr_cas1 AFTER DELETE ON tb_customer
                           FOR EACH ROW
                           (DELETE FROM tb_orders WHERE cust_num = old.cust_num);

dmSQL> CREATE TRIGGER tr_cas2 AFTER DELETE ON tb_orders
                           FOR EACH ROW
```

```
                    (DELETE FROM tb_items WHERE order_num = old.order_num);
```

In DBMaker, if users create recursive triggers, it will not return an error at trigger creation time. However, users will get an error when the recursive triggers execute and meet the maximum limit of cascading trigger levels.

10.7 Enabling and Disabling Triggers

When a trigger is created, the trigger is in **enabled** mode, which means the triggered action executes when the trigger event occurs.

Sometimes users may need to disable a trigger:

- When users have to load a large amount of data, disabling the triggers temporarily will speed up the loading operation
- When the objects referenced in a trigger are unavailable

➲ **Example 1**

To disable trigger **Mytrigger** for table **Mytable**:

```
dmSQL> ALTER TRIGGER Mytrigger ON Mytable DISABLE;
```

➲ **Example 2**

To enable trigger **Mytrigger** on table **Mytable**:

```
dmSQL> ALTER TRIGGER mytrigger ON mytable ENABLE;
```

In summary, a trigger has two possible modes:

- **Enabled** — Trigger is enabled when created and triggered action fires when the event occurs.
- **Disabled** — Disabled trigger does not execute, even if the event occurs.

10.8 Create Trigger Privileges

To create a trigger for a table, a user must be the table owner or DBA. The trigger creator must have privileges to all objects referenced in the CREATE TRIGGER statement to be successful.

In DBMaker, a trigger has no owner; it is associated with a table. The table owner and DBA have all privileges associated with a trigger. They can create, drop, or alter the triggers.

The SQL statements in the trigger action operate under the domain privileges of the trigger owner, not the domain privileges of the user executing the trigger event.

11 Stored Commands

A stored command is a compiled SQL DML statement stored in the database. A stored command is precompiled in an executable format. The same command can be executed without compiling and optimizing. It is possible to create a stored command for any frequently used SQL statement, achieving better performance. Stored commands are considered a subset of stored procedures that only contain one SQL statement without program logic.

11.1 Creating Stored Commands

Use the CREATE COMMAND statement to create a stored command.

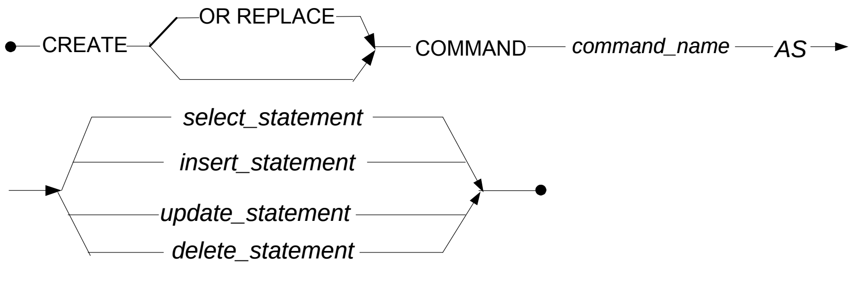


Figure 11-1 Syntax for the CREATE COMMAND statement

OR REPLACE is used to re-create the stored command that already exists, that is to say, users can use this clause to change the definition of an existing stored command.

Input parameters in the SQL statement can be used when creating a stored command. The actual value of the input parameters for a stored command can be assigned at the time of execution.

➲ **Example 1**

To create a stored command named **sc_student_insert** for the SQL DML statement using a table with the definition **tb_student** (**id** INT, **score** INT, **name** CHAR(32)):

```
dmSQL> INSERT INTO tb_student VALUES (1, ?, ?);
```

➲ **Example 2**

Alternatively use CREATE COMMAND:

```
dmSQL> CREATE COMMAND sc_student_insert AS INSERT INTO tb_student VALUES
(1, ?, ?);
```

➲ **Example 3**

Alternatively use CREATE OR REPLACE COMMAND:

```
dmSQL> CREATE OR REPLACE COMMAND sc_student_insert AS INSERT INTO tb_student
VALUES (1, ?, ?);
```

➲ **Example 4**

To create stored commands for other DML statements:

```
dmSQL> CREATE COMMAND sc_student_select AS SELECT id,name FROM tb_student;
dmSQL> CREATE COMMAND sc_student_update AS UPDATE tb_student SET id = id+1 WHERE
score > ?;
dmSQL> CREATE COMMAND sc_student_delete AS DELETE FROM tb_student WHERE score
> ?;
```

After creating a stored command, a user with permission can execute it directly using dmSQL or in an application program. If the stored command has input parameters, its value can be determined using parameter marks, constants, NULL, DEFAULT, or built-in functions (built-in functions can't have arguments), when executing the stored command. When executing stored commands, the number of stored command input parameters should equal the number of input parameters.

11.2 Executing a Stored Command

Use the EXECUTE COMMAND statement to execute a stored command.

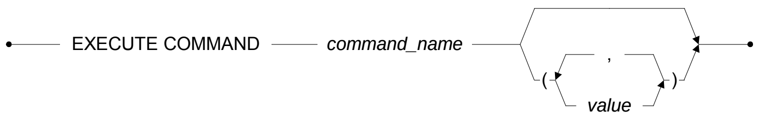


Figure 11-2 Syntax for the EXECUTE COMMAND statement

➲ **Example 1**

```
dmSQL> EXECUTE COMMAND sc_student_insert (200, 'john');
```

➲ **Example 2**

```
dmSQL> EXECUTE COMMAND sc_student_insert (DEFAULT, ?);
```

➲ **Example 3**

```
dmSQL> EXECUTE COMMAND sc_student_insert (?, NULL);
```

➲ **Example 4**

```
dmSQL> EXECUTE COMMAND sc_student_insert (?, ?);
```

A stored command may be dropped when it is no longer wanted.

11.3 Rebuilding a Stored Command

Use the REBUILD COMMAND statement to rebuild a stored command.

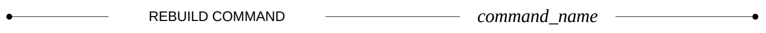


Figure 11-3 Syntax for the REBUILD COMMAND statement

➲ **Example**

```
dmSQL> REBUILD COMMAND sc_student_insert;
```

11.4 Dropping a Stored Command

Use the DROP COMMAND statement to drop a stored command.

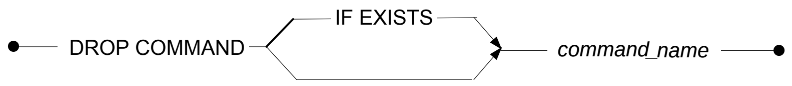


Figure 11-4 The syntax of DROP COMMAND statement

- **Example1**

```
dmSQL> DROP COMMAND sc_student_insert;
```

- **Example 2**

```
dmSQL> DROP COMMAND IF EXISTS sc_student_insert;
```

11.5 Stored Command Security

Stored commands are treated like database schema objects. So users must consider security and object privileges when creating or using them.

Only the creator or users that have the RESOURCE privilege can create a stored command. A user can only create a stored command from a SQL DML statement if the user has privileges to execute the SQL DML statement.

- **Example**

User **joe** with resource privilege wants to create a stored command **sc_CheckDate** with the following syntax:

```
dmSQL> CREATE COMMAND sc_CheckDate AS SELECT FirstName, LastName, Hiredate FROM
SYSADM.tb_staff WHERE HireDate > '1995-01-01';
```

The **tb_staff** table is owned by SYSADM, so the system administrator must first grant select permission to user **joe** on the table **SYSADM.tb_staff** before user **joe** can create the stored command.

A user must have the execute privilege for a stored command to execute it. In order to allow a stored command to be used by others, the user can grant the execute privilege on a stored command. However, only users with the necessary privileges (DBA, SYSDBA, SYSADM, the creator of the stored command, or others granted the privilege) may grant or revoke execute privileges for stored commands.

The DBA has execute privilege for all stored commands in a database. The owner of a particular stored command has execute, grant and revoke privileges for that stored command. Only the owner of stored command can drop it.

Granting Execute Privilege

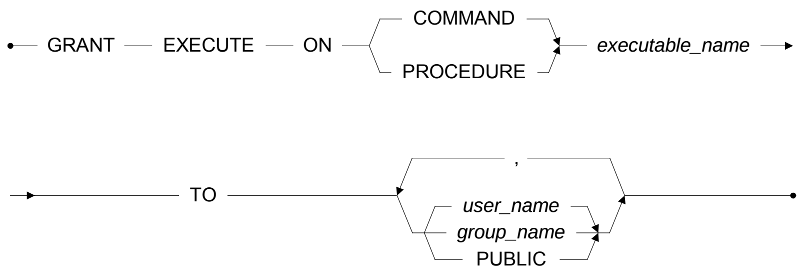


Figure 11-5 Syntax for the GRANT EXECUTE privilege

➲ **Example**

To grant **John** the EXECUTE privileges for commands on **sc_student_insert:**

```
dmSQL> GRANT EXECUTE ON COMMAND sc_student_insert TO John;
```

Revoking Execute Privileges

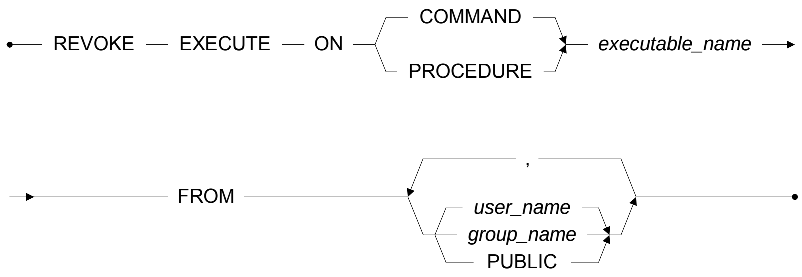


Figure 11-6 Syntax for the REVOKE EXECUTE privilege

➲ **Example**

To revoke the EXECUTE privileges from **John** for **sc_student_insert** :

```
dmSQL> REVOKE EXECUTE ON COMMAND sc_student_insert FROM John;
```

11.6 Lifecycle of a Stored Command

A stored command will be invalid if one of the related tables in the stored command is dropped or altered. If any programs were written previously using old column information, it may cause unpredictable results at time of execution.

The benefit of stored command is improved performance when repeatedly executing a SQL command. DBMaker also considers when the execution plan should be refreshed, such as UPDATE STATISTICS. When the UPDATE STATISTICS command is issued, all execution plans for the stored command will be updated to achieve better performance.

11.7 Getting Information for Stored Commands

Information about stored commands is found in the system table SYSCMDINFO. Table 11-1 lists the columns of the SYSCMDINFO table and their values.

COLUMN NAME	VALUE	COMMENT
MODULENAME	Module name that the stored command belongs to	This column is used by an ESQL application or stored procedure. It can be ignored if it is a pure stored command.
CMDNAME	Stored command name	none
CMDOWNER	Stored command owner	none
STATEMENT	Original SQL string	none
NUM_PARM	Number of parameters	none
STATUS	0, 1 or 2	0 - invalid stored command. It cannot be executed. 1 – Valid stored command. It can be executed. 2 – The stored command needs to be rebound. It can be executed after internal rebinding.
REBTIME	Time of rebuild stored command	none
CMDPLAN	Dump stored command execution plan string	none

Table 11-1: Details of the SYSCMDINFO table

Retrieve stored command information using the following statement in dmSQL:

```
dmSQL> SELECT * FROM SYSCMDINFO;
```

12 Stored Procedures

A stored procedure is a special kind of user-defined function. DBMaker supports stored procedures written in three languages: ESQL/C, Java and SQL. Once a stored procedure is created, it is stored in the database as an executable object. This allows the database engine to bypass repeated SQL compilation and optimization, increasing the performance of frequently repeated tasks. The stored procedure is executed either as a command in interactive SQL, or invoked in application programs, trigger actions or other stored procedures.

Stored procedures accomplish a wide range of objectives including improving database performance, simplifying application writing and limiting or monitoring access to a database.

Because stored procedures are stored in the database as executable objects, they are available to every application running on the database. Several applications can use the same stored procedure. This reduces application development time.

12.1 ESQL Stored Procedures

An ESQL stored procedure is an ESQL/C program. Stored procedures can perform any function a C application can, including calling other C functions and system calls. Therefore, a C compiler is required for writing stored procedures.

An ESQL/C program for a stored procedure consists of a CREATE PROCEDURE statement, a declare section if needed, and the code section. If your program does not use any host variables, the declare section can be omitted.

➲ Example

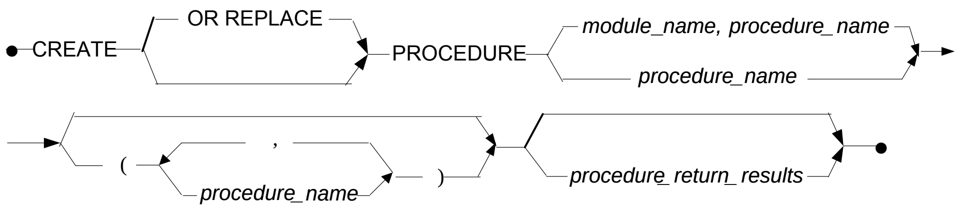
To create a stored procedure called **sp_Aphone** with one input parameter, one output parameter, and a return value (**status**):

```
EXEC SQL CREATE PROCEDURE sp_Aphone (CHAR(13) name, CHAR(13) phone OUTPUT)
         RETURNS STATUS;
{
    EXEC SQL BEGIN CODE SECTION;

        EXEC SQL SELECT PHONE FROM TBL WHERE NAME = :name INTO :phone;

        EXEC SQL RETURNS STATUS SQLCODE;

    EXEC SQL END CODE SECTION;
}
```

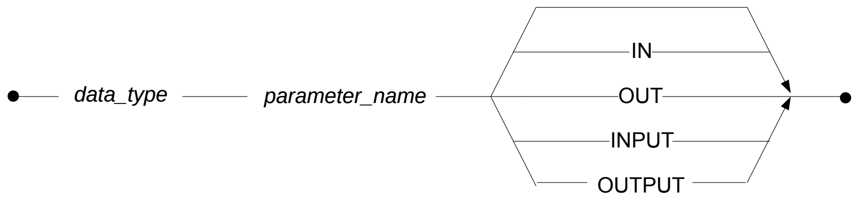
The structure of this program will be explained in the following sections.

Create Procedure Syntax

In the head of a procedure definition is a CREATE PROCEDURE statement. The syntax for the CREATE PROCEDURE statement is:



<*procedure_parameters*> clause



<*procedure_return_result* > clause

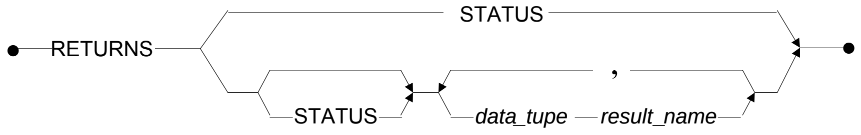


Figure 12-1 Syntax for the CREATE PROCEDURE statement

OR REPLACE is used to re-create the stored procedure if it already exists, that is to say, users can use this clause to change the definition of an existing stored procedure

NOTE *Please note that if replace procedure aborted, the originally stored procedure already be dropped.*

➲ **Example**

These examples show the syntax of the CREATE PROCEDURE statement.

```
dmSQL> CREATE PROCEDURE sp_example1 (INTEGER n IN) RETURNS STATUS;
dmSQL> CREATE PROCEDURE sp_example2 (INTEGER n1 IN, INTEGER n2 OUTPUT) RETURNS
       CHAR(12) nm;
dmSQL> CREATE PROCEDURE sp_example3(CHAR(10) par1 OUTPUT, SMALLINT par2)
       RETURNS STATUS, TIMESTAMP ret1, FLOAT ret2;
dmSQL> CREATE OR REPLACE PROCEDURE sp_example4(CHAR(10) par1 OUTPUT, SMALLINT
       par2) RETURNS STATUS, TIMESTAMP ret1, FLOAT ret2;
```

In a CREATE PROCEDURE statement the procedure name and the name and type of any I/O parameters must be provided.

Using Parameters

If parameters are required, a list of type-name pairs for the parameters must be given in parentheses. IN/OUT (or INPUT/OUTPUT) parameter attributes must be put after each type-name pairs. If there is no parameter attribute, IN will be used by default. Input parameters are used to pass a value to a procedure. In example 1, there is one input parameter **name**. Every time the procedure is executed, a value for the input parameter must be provided.

Output parameters are used to get a single result, not a result set, after the procedure is executed. In example 1, procedure **sp_Aphone** has an output parameter, **phone**. The output parameter must have a buffer assigned to it to receive the result. After the procedure executes, the phone number for the name inputted can be retrieved from the buffer.

The result list is required for a stored procedure to retrieve a result set of tuples from the database. If the procedure does not return selected results then there is no need for the result list. The keyword RETURNS is used to start the result list. It is a list of name type pairs.

The STATUS keyword indicates an integer value be returned after the procedure executes.

➲ **Example**

To execute a procedure with one input parameter and a value:

```
EXEC SQL CREATE PROCEDURE sp_Select (FLOAT ifl) RETURNS STATUS,
                                                  FLOAT fl,
                                                 DOUBLE db;
{
    EXEC SQL BEGIN CODE SECTION;
    EXEC SQL RETURNS STATUS SQLCODE;
    EXEC SQL RETURNS SELECT fl, db FROM t8 WHERE fl < :ifl into :fl, :db;
    EXEC SQL END CODE SECTION;
}
```

DBMaker now supports the following data types for input and output parameters: INTEGER, SMALLINT, CHAR(), DATE, TIME, TIMESTAMP, FLOAT, DOUBLE and REAL.

Return Select Statement

A procedure can return a result set using the host variable mechanism to pass information to the user executing the stored procedure. In the code of the stored procedure use the RETURNS keyword to instruct the preprocessor to generate a host variable related to C code. The RETURNS keyword precedes the select statement that produces the result set.

➲ **Example**

There are two RETURNS in this example, one in the create procedure statement and the other in the select statement forming a pair. If there is to be a result set returned, declare output parameters with RETURNS in the create procedure statement and put the RETURNS keyword in the select statement:

```
EXEC SQL CREATE PROCEDURE  sp_Allphone RETURNS CHAR(12) name, CHAR(12) phone;
{
    EXEC SQL BEGIN CODE SECTION;
        EXEC SQL RETURNS SELECT NAME, PHONE FROM TBL INTO :name, :phone;
    EXEC SQL END CODE SECTION;
}
```

Module Names

When a user creates a stored procedure DBMaker will use the owner name and procedure name as the default dynamic link library name. The user can call or drop his or her stored procedures using only the procedure name. Any user can call another user's procedure using the full procedure name: *owner.procedure_name*.

A user can also specify a module name in the CREATE PROCEDURE syntax to change the default dynamic link library name. If a module name is specified in the CREATE PROCEDURE syntax, users will need to call or drop the procedure with the full procedure name; *module_name.owner.procedure_name*, even if the user created it.

Variable Declaration

The host variables in stored procedures are declared in the same way as in ESQL/C. The declare section in a stored procedure must be put before the code section, not in ESQL/C programs. C variables can be placed before or after the declare section, but need to be before the code section.

Code Section

All statements should be in the code section except the variable declaration. Any non-declaration statement before the code section may cause problems resulting in compile errors or wrong results being returned. Statements after the CODE SECTION will not be executed.

Configuration Settings for Stored Procedures

When a stored procedure is created, a corresponding dynamic link library is built and stored on the server. By default, the library file is placed in the DBMaker server's working directory. The database administrator can set a preferred path to place the library files for stored procedures using the configuration keyword **DB_SPDir**.

The keyword **DB_SPLog** is used by client users to set the directory they prefer to receive error message files and trace log files, transmitted from the database server while creating or executing stored procedures.

➲ **Example 1**

To set the default path of dynamic link library files for stored procedures to */usr1/dbmaker/data/SP* add the following line in the **dmconfig.ini** file:

```
DB_SPDir=/usr1/dbmaker/data/SP
```

➲ **Example 2**

To set the stored procedure log file directory to *c:\usr\jerry\data\SP* add the following line in the **dmconfig.ini** file:

```
DB_SPLog=c:\usr\jerry\data\SP
```

Creating a New Stored Procedure from File

First, write the stored procedure and save it to a file, then use DBMaker tools like dmSQL or JDBATool to insert this new stored procedure into the database.

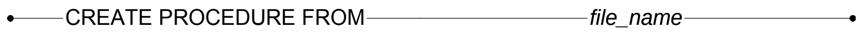


Figure 12-2 Syntax for the CREATE PROCEDURE FROM <file_name> statement

➲ **Example**

To create a procedure using multiple files:

```
dmSQL> CREATE PROCEDURE FROM 'proc1.ec';
dmSQL> CREATE PROCEDURE FROM '.\esql\sp\proc2.ec';
dmSQL> CREATE PROCEDURE FROM 'c:\users\jerry\sp\proc3.ec';
```

The previous examples show how to create stored procedures using dmSQL.

➲ **Alternatively, use JDBATool**

1. Click the object **Stored Procedure** in the Tree.
2. Click the **Create** button. The **Introduction** window of the **Create Stored Procedure** wizard is displayed.
3. Import a stored procedure by selecting the **Import** button.
4. Selecting **Import** opens the **Open** window. Files can be imported from any source, including the SPDIR directory of other databases on the server or network drives. Select the desired file by typing in the path in the **File name** field, or browse through the directory tree until the correct path is found.

   NOTE *Imported files must be ASCII format files that contain C++ code.*

5. Select **Open** to open the file.
6. The **Create Stored Procedure** window will reappear if the imported file contains properly formatted (ASCII) text. Select **Save As** to store the stored procedure to another location, or select **OK** to compile and store the stored procedure in the database.

   NOTE *If there are any errors when creating a stored procedure, they will be shown in the lower part of the window.*

Executing Stored Procedures

You can invoke a stored procedure in dmSQL, a C program (ODBC or ESQL), another stored procedure, or using a trigger action.

DMSQL

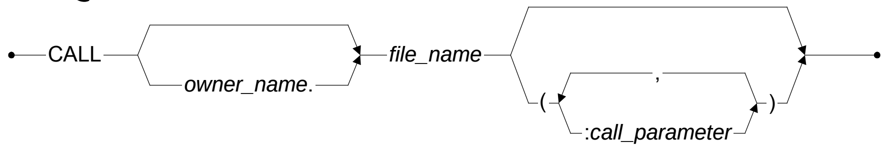


Figure 12-3 Syntax for the CALL statement within dmSQL

➲ **Example 1**

To execute a stored procedure in dmSQL:

```
dmSQL> CALL sp_example1 (3);                 // execute procedure sp_example1
dmSQL> CALL SYSADM.sp_example2 (5, ?);       // execute SYSADM's procedure
sp_example2
dmSQL/Val> 100;                              // input the value of parameter
dmSQL> ? = CALL SYSADM.sp_example2 (5, 100); // execute procedure p2 and get
                                             // returned status
```

➲ **Example 2**

If the procedure returns a result set, dmSQL automatically handles the output parameters and displays the result set on the screen. The result set appears on the screen as if you had typed a SELECT statement using dmSQL:

```
dmSQL> CALL sp_Aphone('jeff');
 STATUS        PHONE
======== =================
       0 408-255-2689

dmSQL> CALL sp_Allphone;
  NAME         PHONE
======== =================
Jerry    02-775-8615
Jeff     408-255-2689
```

ESQL

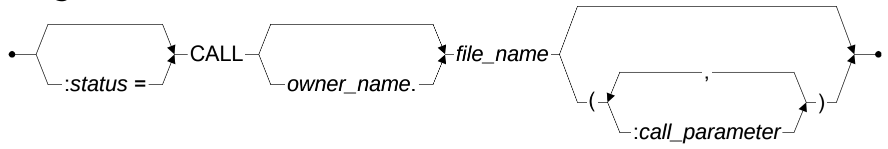


Figure 12-4 Syntax for the CALL statement within ESQL

➲ **Example**

To execute a stored procedure in ESQL:

```
EXEC SQL :s = CALL sp_example1 (3);
EXEC SQL CALL SYSADM.sp_example2 (5, :n2) INTO :nm;
EXEC SQL :s = CALL jack.sp_example3 (:par1, 7) INTO :ret1, :ret2;
```

The syntax used in an ESQL program is similar to dmSQL. Use host variables to receive the status, output parameter, and result set values.

EXECUTING NESTED STORED PROCEDURES

Invoked a nested ESQL/C stored procedure happens in exactly the same way as in any ESQL/C program. There is one exception: regular ESQL programs cannot use the RETURNS keyword, but stored procedures can when invoking another stored procedure.

Assume stored procedure **sp_Allphone** returns a multiple tuple result set. A regular ESQL program must use a cursor to fetch the tuples when invoking this procedure as shown in the last section. Another stored procedure **sp_another** can use the same method to fetch tuples, examine data and return the entire result set of the called stored procedure to the caller directly from within the current stored procedure.

➲ **Example**

To call a statement from within the **sp_another** stored procedure:

```
EXEC SQL RETURNS CALL sp_Allphone INTO :oName, :oPhone;
```

When a stored procedure returns another stored procedure's result set, the caller should have exactly the same result set list, or the first n result columns from the called procedure.

EXECUTING STORED PROCEDURES IN ODBC PROGRAMS

You can also call a stored procedure in an ODBC program by binding parameters for a procedure and using columns to return the result set. In an ODBC program, you can bind partial columns of the result set. After the procedure executes, output parameters are returned in the host variables. Use a fetch, like a SELECT command, to get the result set.

➲ **Example 1**

Procedure **sp_proc1** declaration:

```
dmSQL> CREATE PROCEDURE sp_proc1(CHAR(12) p1, CHAR(12) p2 OUTPUT) RETURNS INTEGER
r1;
{
EXEC SQL BEGIN CODE SECTION;
EXEC SQL SELECT c2 FROM t1 WHERE c1 = :p1 INTO :p2;
EXEC SQL RETURNS SELECT c1 FROM t2 INTO :r1;
EXEC SQL END CODE SECTION;
}
```

➲ **Example 2**

ODBC program that calls **sp_proc1**:

```
SQLPrepare(cmdp,(UCHAR*)"call sp_proc1(?, ?)", SQL_NTS);

strcpy(bpname, "12345");

SQLBindParameter(cmdp, 1, SQL_PARAM_INPUT_OUTPUT, SQL_C_CHAR, SQL_CHAR,
                      20, 0, &p1, 20, NULL);
SQLBindParameter(cmdp, 2, SQL_PARAM_INPUT_OUTPUT, SQL_C_CHAR, SQL_CHAR,
                      20, 0, &p2, 20, NULL);

SQLBindCol(cmdp, 1, SQL_C_LONG, &i, sizeof(long), NULL);
SQLExecute(cmdp);                                 /* get p2          */

while ((rc=SQLFetch(cmdp))!=SQL_NO_DATA_FOUND)      /* fetch result set */
```

TRACING STORED PROCEDURE EXECUTION

DBMaker provides trace functionality to help users trace the execution of stored procedures for debugging.

➲ **Example**

Using the TRACE command:

```
EXEC SQL TRACE ON;                        // Start TRACE
EXEC SQL SELECT c1 FROM t1 INTO :var1;
EXEC SQL TRACE ("var1 = %d\n", var1);     // TRACE the value of var1
EXEC SQL TRACE OFF;                       // END OF TRACE
```

Turn on and use the TRACE function to place variables for tracing and print messages. After the stored procedure executes, all trace information will be written to a file named **_spusr.log** in the **DB_SPLog** keyword directory found in the **dmconfig.ini** file on the client machine.

12.2 JAVA Stored Procedures

There are a number of scenarios where it makes sense to use Java stored procedures. Given Java's popularity today, it is certainly possible that members of a development team are more proficient in Java than ESQL. DBMaker supports Java stored procedures to give Java programmers the ability to code in their preferred language. For experienced ESQL developers, using Java allows you to take advantage of the Java language to extend the functionality of database applications. Using Java also allows you to reuse existing code and dramatically increase productivity.

DBMaker supports a java method as a java stored procedure. DBMaker replaces the URL argument of DriverManager.getConnection(url, …) with **jdbc:default:connection**. You can use all java classes to implement a Java method as a Java stored procedure, including all JDBC classes. DBMaker has new syntax to register related .jar files, and create, execute, and drop a Java stored procedure.

The syntax for the CREATE JAVA PROCEDURE statement is:

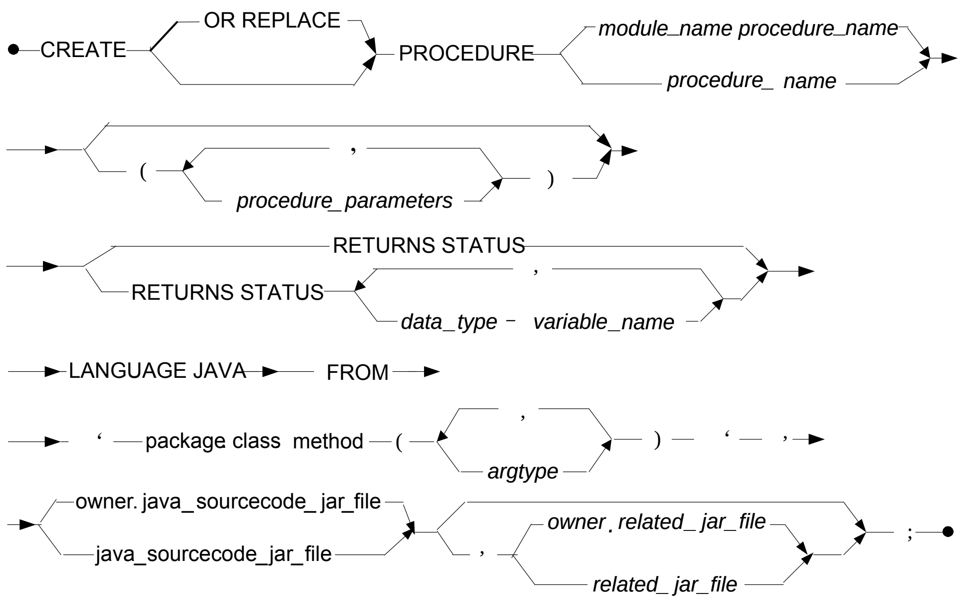


Figure 12-5 Syntax for the CREATE JAVA STORED PROCEDURE statement

➲ **Example 1**

DBMaker supports java method xx.yy.AA(String) in */home/usr/mary/sp/aa.jar*, but */home/usr/john/sp/bb.jar* is still needed to run method AA(String):

For example: To register aa.jar and bb.jar files into database, first, you must put user sysadm's */home/usr/mary/sp/aa.jar* into */home/sysadm/spdir/jar/SYSADM/*.

DB_SPDir is defined in dmconfig.ini as "*/home/sysadm/spdir*", so the user must move the jar file into */DB_SPDir/jar/uppercase_username/* directory before adding the jar file in to the database.

➲ **Example 2**

You have a java method xx.yy.AA(String) in aa.jar, but you still need bb.jar to run method AA(String):

To register aa.jar and bb.jar files into database

```
dmSQL> ADD JARFILE xaa aa.jar;
dmSQL> ADD JARFILE xbb bb.jar;
```

NOTE *Users must move the physical jar files, i.e. aa.jar and bb.jar, into directory* ***DB_SPDir****/jar/uppercase_username/ before executing the "add jarfile" syntax.* ***DB_SPDir*** *is the keyword to define the DBMaker stored procedure directory in* ***dmconfig.ini****.*

To create a java stored procedure "JSP_AA(char(10) par1)" by the java method AA(String):

```
dmSQL> CREATE PROCEDURE JSP_AA (char(10) par1) RETURNS STATUS LANGUAGE JAVA FROM
'xx.yy.AA(String)', xaa, xbb;
```

To execute a java stored procedure "JSP_AA":

```
dmSQL> EXECUTE PROCEDURE JSP_AA ('aaaaaa');
dmSQL> CALL JSP_AA ('bbb');
```

To drop a java stored procedure "JSP_AA":

```
dmSQL> DROP PROCEDURE JSP_AA;
```

To deregister aa.jar and bb.jar files from database:

```
dmSQL> REMOVE JARFILE xaa;
dmSQL> REMOVE JARFILE xbb;
```

➲ **The related Create Procedure syntax in JAVA**

To register a .jar file into the database:

```
ADD JARFILE logical_file_name physical_jarfile_name
```

To deregister a .jar file into the database:

```
REMOVE JARFILE logical_file_name
```

To create a java stored procedure with CREATE PROCEDURE command:

```
CREATE  [OR REPLACE] PROCEDURE procedure-name
      [(procedure-parameter [, procedure-parameter ...])]
      {
        [RETURNS STATUS]
        | [RETURNS [STATUS,] procedure-result [,procedure-result ...]]
         }
```

```
 LANGUAGE JAVA FROM  'package.class.method([ ' argtype[,argtype…] ' ])',
[owner.]java-sourcecode-jar-file [, owner.related-jar-file]
```

OR REPLACE is used to re-create the procedure if it already exists, that is to say, users can use this clause to change the definition of an existing procedure.

To execute a java stored procedure:

```
EXECUTE PROCEDURE [owner.]procedure-name
EXECUTE PROC [owner.]procedure-name
[? =] CALL  [owner.]procedure-name  [(procedure-parameter-value [, procedure-
parameter-value ...])]
```

To drop a java stored procedure:

```
DROP PROCEDURE [owner.]procedure-name
```

To load/unload procedure:

```
UNLOAD PROCEDURE/PORC FROM [owner_patt.]proc_patt TO unload_filename
LOAD PROCEDURE/PORC FROM unload_filename
```

To load/unload jar file:

```
UNLOAD JARFILE FROM [owner_patt.]jarfile_patt TO unload_filename
LOAD JARFILE FROM unload_filename
```

NOTE *Users must move the physical jar files into the new DB_SPDir/jar/uppercase_username/ directory before loading jar files.*

Executing Java Stored Procedures

Explanations using the Java stored procedures are delivered using the following examples.

➲ **Example 1 (INPUT parameter)**

Insert one tuple into the table **tb_staff** using a Java stored procedure.

1. Write a java method addEmployee(int,String) to insert one tuple into the table **tb_staff**. Then compile the Java method and zip the class into an AA.jar file.

```
public static void addEmployee(int id, String name)
{
        Connection conn = DriverManager.getConnection("jdbc:default:connection");
        PreparedStatement pstmt = conn.prepareStatement("insert into tb_staff
values(?,?)");
        pstmt.setInt(1, empid);
        pstmt.setString(2, name);
pstmt.execute();
}
```

2. Create a Java stored procedure for the Java method: addEmployee(int,String).

To execute the SQL statement to add the NEW jar file:

```
dmSQL> ADD JARFILE logical_AA AA.jar;
```

To execute one of the following SQL statements to create the Java stored procedure:

```
dmSQL> CREATE PROCEDURE JSP_addEmp (int id, char(10) name) RETURNS STATUS
LANGUAGE JAVA FROM 'xx.yy.addEmployee(int,String)', logical_AA;
```

or:

```
dmSQL> CREATE OR REPLACE PROCEDURE JSP_addEmp (int id, char(10) name) RETURNS
STATUS LANGUAGE JAVA FROM 'xx.yy.addEmployee(int,String)', logical_AA;
```

3. Run the Java stored procedure.

To execute the SQL statement to run the java SP:

```
dmSQL> EXECUTE PROCEDURE JSP_addEmp(1234, 'jeff');
```

➲ **Example 2 (OUTPUT parameter)**

Select one employee name from the **tb_staff** with the predicator (empid) using a Java stored procedure.

1. Write a Java method oneEmployee(int,byte[]) to get one employee name from the table **tb_staff** with the predicator (emp id). Then compile the Java method and zip the class into the BB.jar file.

```
public static void oneEmployee(int id, byte[] name)
{
        Connection conn = DriverManager.getConnection("jdbc:default:connection");
        PreparedStatement pstmt = conn.prepareStatement("select name from
tb_staff where id = ?");
        pstmt.setInt(1, id);
        ResultSet rs = pstmt.executeQuery();
        Rs.next();
        String empName = rs.getString(1);
        Name = empName.getBytes();
}
```

2. Create a Java stored procedure for the Java method: oneEmployee(int,String).

To execute the SQL statement to add the NEW jar file:

```
dmSQL> ADD JARFILE logical_BB BB.jar;
```

To execute the SQL statement to create the java SP:

```
dmSQL> CREATE PROCEDURE JSP_oneEmp (int id, char(10) name OUTPUT) RETURNS STATUS
LANGUAGE JAVA FROM 'xx.yy.oneEmployee(int,byte[])', logical_BB;
```

3. Run the Java stored procedure.

To execute the SQL statement to run the java SP:

```
dmSQL> EXECUTE PROCEDURE JSP_oneEmp(1234, ?);
```

➲ **Example 3 (Resultset)**

Select one resultset from the table **tb_staff** using a Java stored procedure.

1. Write a Java method rsEmployee() to get one employee name from the table **tb_staff**. Then compile the Java method and zip the class into a CC.jar file.

```
public static ResultSet rsEmployee()
{
        Connection conn = DriverManager.getConnection("jdbc:default:connection");
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("select id, name from tb_staff");
        Return rs;
}
```

2. Create a Java stored procedure for the Java method: rsEmployee(int,String).

To execute the SQL statement to add the NEW jar file:

```
dmSQL> ADD JARFILE logical_CC CC.jar;
```

To execute the SQL statement to create the java SP:

```
dmSQL> CREATE PROCEDURE JSP_rsEmp RETURNS STATUS, int outId, char(10) outName
LANGUAGE JAVA FROM 'xx.yy.rsEmployee()', logical_CC;
```

3. Run the Java stored procedure.

To execute the SQL statement to run the Java stored procedure:

```
dmSQL> EXECUTE PROCEDURE JSP_rsEmp();
```

4. Fetch the result set using a general fetch (or extended fetch) method.

Input/Output Argument

The Java stored procedure input/output argument for DBMaker supports the following data types:

BINARY	VARCHAR
CHAR	FLOAT
REAL	DOUBLE
SMALLINT	INTEGER
TIMESTAMP	DATE
TIME	DECIMAL

They are seven basic Java types and arrays that DBMaker supports:

JAVA TYPE	ARRAY
byte	byte [ ]
short	short [ ]
int	int [ ]
long	long [ ]
float	float [ ]
double	double [ ]
char	char [ ]

The following classes are also supported:

JAVA CLASS	ARRAY

Byte	Byte[ ]
Short	Short[ ]
Integer	Integer[ ]
Long	Long[ ]
Float	Float[ ]
Double	Double[ ]
Character	Character[ ]
String	String[ ]

12.3 SQL Stored Procedures

Using SQL statements to create stored procedures rather than ESQL and Java is sometimes a better approach.

A SQL stored procedure is a stored procedure with logic implemented using only SQL statements. The SQL stored procedure contains a set of SQL statements that can be stored on a server. Once on the server, clients can avoid executing many individual statements by making use of the SQL stored procedure. SQL stored procedures contain permanent stored procedures, temp stored procedures and anonymous stored procedures. For more information on SQL stored procedures please refer to *DBMaker SQL Stored Procedure User's Guide*.

Architecture

SQL stored procedures contain compound statements. These are bounded by the keywords BEGIN and END. The following example shows a SQL stored procedure statement.

```
BEGIN                            #block header
Variable declarations
Condition declarations
Cursor declarations
Condition handler declarations
```

```
Assignment, flow of control, SQL statements and other compound statements
END;                        #block end
```

This example shows how SQL stored procedures consist of one or more component declarations and statements that make a block. Blocks support nesting within a single SQL stored procedure. Some components declarations are options: variable, condition, and handler, however, when present they must precede assignment, flow control, SQL and other compound statements. Note that cursor declarations may appear anywhere block.

Create SQL Stored Procedure Syntax

CREATE SQL STORED PROCEDURE FROM FILE

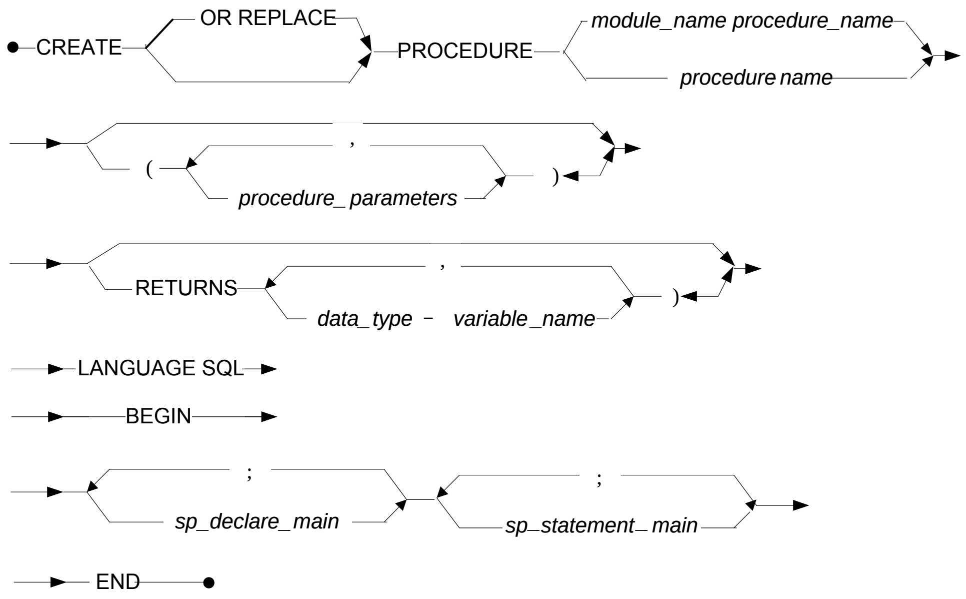


Figure 12-6 Syntax for the CREATE SQL Stored Procedure statement

OR REPLACE is used to re-create the procedure if it already exists, that is to say, you can use this clause to change the definition of an existing procedure.

NOTE *Not support executing CREATE OR REPLACE COMMAND syntax and in stored procedure.*

NOTE *Not support CREATE OR REPLACE PROCEDURE syntax while setting AUTOCOMMIT OFF.*

SQL stored procedure can be created using an external file (*.sp). Use the dmSQL command line tool to create SQL stored procedures by referencing external files as shown in the following example.

➲ **Example 1**

To create a SQL stored procedure by file reference:

```
dmSQL> CREATE PROCEDURE FROM 'CRETB.SP';
dmSQL> CREATE PROCEDURE FROM '.\SPDIR\CRETB.SP';
dmSQL> CREATE PROCEDURE FROM 'D:\DATABASE\SPDIR\CRETB.SP';
```

➲ **Example 2**

To create or replace a SQL stored procedure by file reference:

```
dmSQL> CREATE OR REPLACE PROCEDURE FROM 'CRETB.SP';
dmSQL> CREATE OR REPLACE PROCEDURE FROM '.\SPDIR\CRETB.SP';
dmSQL> CREATE OR REPLACE PROCEDURE FROM 'D:\DATABASE\SPDIR\CRETB.SP';
```

CREATE SQL STORED PROCEDURE IN SCRIPT

Users can create SQL stored procedures not only from files, but also in dmSQL. The user can call and drop the SQL stored procedure of his own, and can execute SQL stored procedures on which the privilege has been granted to him. For more information, please refer to the information in chapter 12.3, *SQL stored procedures*.

SQLSP contains more than one SQL statements, and each statement is end of ';'. So dmSQL must support block delimiter. Block delimiter can be a serial of a-z, A-Z, @, %, ^, and contains two characters at least and seven characters at most. In block delimiter, ';' doesn't denote end of the input. Users must set block delimiter before write SQL stored procedure in dmSQL, otherwise, it will return error.

➲ **Example**

To create stored procedure syntax in script:

```
dmSQL> SET BLOCK DELIMITER @@;
dmSQL> @@
    2> CREATE PROCEDURE sp_in_script2
    3> LANGUAGE SQL
    4> BEGIN
    5> INSERT INTO t1 VALUES(1);
    6> END;
    7> @@
dmSQL> SET BLOCK DELIMITER;
```

Using Parameters

SQL values are passed to and from SQL stored procedures via parameters.

Parameters can be useful in SQL stored procedures when implementing logic that is conditional on a particular input or set of input scalar values or when you need to return one or more output scalar values but do not want to return a result set.

Variable Declaration

Local variable support in SQL stored procedures allows you to assign and retrieve SQL values in support of SQL stored procedure logic.

Variables in SQL Stored procedures are defined with the DECLARE statement. Use DECLARE to define items local to a routine, in other words, local variables, conditions, handles and cursors. DECLARE must directly follow a BEGIN as part of a BEGIN **…** END compound statement. No other statements may precede a DECLARE statement. Declarations must follow this order: variables and conditions must be declared first, next declare cursors and finally declare handlers.

DBMaker supports two kinds of SQL variables: **Connection Varilable (CV)** and **Statement Variable (SV)**. The two variables both are used to enhance the dmsql command extensibility and portability. The following sections give illustrations and examples of the **CV** and **SV**.

CONNECTION VARIABLE (CV)

CV is a connection variable that only can be defined in local connections. Connection variables in a connection are independent of those in other connections. That is to say, the connection variables only can be used by the connection that owned them and cannot be got or used by other connections.

For users, a connection variable is a global variable of sql command in the local connection, and the connection variables can be used in the dmsql command line tool and SQLSP. User can assign a value to a connection variable in one statement, and then refer it to another statement. It enables you pass values from one statement to another statement. Once connection is disconnected from the database, all connection variables will be automatically freed. CV is a predefined data type. You can set the variable to define its type before using it. If you refer to a variable that has not been initialized, "Error (6344): [DBMaker] invalid variable name" will be returned.

➲ **Example 1**

CV can be declared and used with dmSQL command line tool. In the following example, **@a**, **@aa** and **@b** are connection variables, they are not only can do insert, update, delete operations but also can be used in **WHERE** clause or udf function call etc.

```
dmSQL> DECLARE SET INT @a = 1;
dmSQL> SELECT @b;
ERROR (6344): [DBMaker] invalid variable name : B name
dmSQL> DECLARE SET INT @aa = NULL;
dmSQL> SELECT @aa;
          @AA
========================
         NULL
dmSQL> DECLARE SET INT @b = 2;
dmSQL> SELECT @a, @b;
    @A          @B
=========== ===========
         1           2

dmSQL> SELECT * FROM t1 WHERE c1=@a;
dmSQL> INSERT INTO t1 VALUES(@b+100);
dmSQL> UPDATE t2 SET c2 = @b+2 WHERE c1 = @a+1;
dmSQL> DELETE FROM t1 WHERE c1=@b;
dmSQL> SELECT SUM(c1) INTO @b FROM t1;
```

➲ **Example 2**

In SQLSP, CV can be declared and used like normal variable. In this example, **@val2** and **@val3** are connection variables.

```
dmSQL> SET BLOCK DELIMITER @@;
dmSQL> @@ CREATE PROCEDURE tsp2
    2> LANGUAGE SQL
    3> BEGIN
    4> DECLARE SET INT @val2 =100+100;
    5> DECLARE SET INT @val3 =LENGTH('test');
    6> SET @val3 =LENGTH('test');
    7> END; @@
```

STATEMENT VARIABLE (SV)

SV is a statement variable. Statement variable is a declared variable in one statement and only can be used in the specified statement. Statement variable in different commands is independent with each other and only can be used in SQLSP.

The definition of statement variables is same as SQLSP's local variables. Once a SQL statement is terminated, the statement variables will be automatically freed.

➲ **Example**

To distinguish the difference between the definition of **CV** and **SV**:

```
dmSQL> SET BLOCK DELIMITER @@;
dmSQL> DECLARE SET INT @aa = 100;
dmSQL> CREATE TABLE tab(c1 INT);
dmSQL> @@ CREATE PROCEDURE tsp(OUT c1 INT)
    2> LANGUAGE SQL
    3> BEGIN
    4> DECLARE vals INT DEFAULT 100;  --vals is sv
    5> INSERT INTO tab VALUES(@aa);   --aa is cv
    6> INSERT INTO tab VALUES(vals);  --vals is sv
    7> SET c1 = @aa;                  --c1 is sv
    8> SET @aa = 200;                 --error not declare cv in sqlsp
    9> END; @@
```

Cursors

Used in SQL stored procedures, cursors allow defining of result sets and then performing complex logic on each row within the set. Note that a result set is simply a set of data rows. Using the same method, SQL Stored procedures can also define result sets and return them directly to the caller or a client application.

Think of a cursor as a pointer to one row within a set of rows. The cursor can point any row in the result set, however it may only reference a single row at any given time.

The DECLARE CURSOR statement first defines a cursor then the following SQL statements are used to manipulate the cursor: OPEN, FETCH and CLOSE.

Assignment Statements

Assignment statements are used to assign values to SQL variables and parameters. Values can be assigned to variables using a SET statement or a CURSOR FOR SELECT FROM statement. Additionally, a variable may have a default value that was set when the variable was declared. Literals, expressions, query results, and special register values can all be assigned to variables. Variable values can further be assigned to SQL stored procedure parameters, other variables in SQL stored procedures, and can be referenced as parameters within SQL stored procedure statements that executed within the routine.

The SET Variable statement assigns values to local variables, output parameters, and new transition variables. The SET Variable statement is under transaction control. SET assignment statements accept simple and complex expressions.

NOTE *String data type variable assignments must be less than 1024 bytes.*

SIMPLE EXPRESSIONS

Simple expressions are classified by numeric, character, timestamp and binary data types. The numeric data types are: INTEGER, BIGINT, SMALLINT, DOUBLE, FLOAT, REAL and DECIMAL. The character data types are: CHAR, NCHAR, VARCHAR and NVARCHAR. Timestamp data types are: DATE, TIME and TIMESTAMP.

A simple expression includes operators (+, -, *, /) variables, constants, values and strings. Simple expressions have much greater implementation efficiency than complex expressions. In particular, multi-loop statements greatly improve execution speed.

COMPLEX EXPRESSIONS

Complex expressions include all the same assignment values found in simple expressions, besides, SQL functions, such as built-in SQL functions and user-defined SQL functions, also is included.

Control Flow Statements

SQL control statements support variables and flow of control statements for controlling the sequence of statement execution. Statements such as IF and CASE are used to conditionally execute blocks of SQL control statements. Statements such as WHILE and REPEAT are used to repetitively execute a set of statements until a task is completed.

SQL control statements fall into the following categories: variable related statements, conditional statements (CASE and IF), loop statements (FOR, LOOP, WHILE and REPEAT), goto statements, return statements, transfer of control statements (ITERATE and LEAVE), labels and SQL stored procedure compound statements.

Returning Result Sets

Cursors can be used to do more than iterate through rows of a result set. In SQL stored procedures, cursors can also return result sets to the calling program.

Return Status of SQL Stored Procedure

Status code reflects whether an stored procedure is successfully executed. User cannot define status code in an stored procedure.

Statues code:

-1: the SP execute error

0: the SP execute OK

1: the SP excute have warning

If you want to return status of SP, you should add 'RETURN STATUS' before 'LANGUAGE SQL'.

➲ **Example**

Call another SQL stored procedures:

```
CREATE PROCEDURE call_test
RETURNS STATUS
LANGUAGE SQL
BEGIN
                  DECLARE cur CURSOR WITH RETURN FOR select * from call_tb;
                  OPEN cur;
END;
CREATE PROCEDURE CASE_TEST_1(IN inval INT, OUT outval1 INT, OUT outval2 INT)
LANGUAGE SQL
BEGIN
         SET outval1 = 1;
         SET outval2 = 2;
END;
CREATE PROCEDURE call1(IN inval INT, OUT outval1 INT, OUT outval2 INT)
LANGUAGE SQL
BEGIN
         CALL CASE_TEST_1(inval, outval1, outval2);
END;
```

Executing SQL Stored Procedures

SQL stored procedures are executed using the CALL statement. CALL statement can be executed using graphical user interface tools like the JDBA Tool or directly from DBMaker's Command Line Tool, dmSQL.

The executable CALL statement calls a procedure. This statement can be embedded in an application program, issued using dynamic SQL statements, or dynamically prepared.

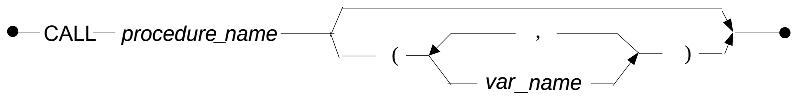


Figure 12-7 Syntax for the CALL statement within dmSQL

Anonymous Stored Procedures

The anonymous stored procedure contains a set of SQL statement that can be temporarily created and executed by a database. There is no need for permanently storage the SQL statements in DBMaker as a database object. The anonymous stored procedure just temporarily exists in a single SQL block, which only used once by the creator.

An anonymous stored procedure is a special kind of SQL stored procedure similar to the Anonymous SQL Block. It can let user execute more than one SQL statement (batch of SQL) once in the client side, and supports all SQL syntax blocks including variables, grammar logic and cursors etc,.The anonymous stored procedure cannot using the stored procedure parameters and specific commands belong to dmSQL command line tool (such as: set etc.). Users must set block delimiter before writing the anonymous SQL block in dmSQL, otherwise, an error will be returned. For more information on anonymous stored procedure's variables and syntax logic, please refer to Chapter 12.3, *SQL stored procedures*.

NOTE *An anonymous stored procedure' compound statements are bounded by the keywords "**BEGIN**" and "**END**"*

Compared with the SQL stored procedure, there is no name in the anonymous stored procedure, and cannot be referenced by other database objects. That is to say, the execution of an anonymous stored procedure is instant. When an anonymous stored procedure has completed its creation successfully, it enters into the execution state. Once it's executed successfully, DBMaker will delete them immediately. The information of anonymous stored procedure will not be saved to the system table SYSPROCINFO, also cannot be permanently stored in DBMaker for reuse. Anonymous stored procedure reduces the time interval between the code updates and program execution, thus improves the efficiency of problem diagnosis, prototyping and testing code execution, provides the convenient for the multiple tasks updates and execution.

➲ **Example**

To create an anonymous stored procedure in a SQL block:

```
dmSQL> set block delimiter @@;
dmSQL> @@
    2> BEGIN
    3> CREATE TABLE tab(c1 INT, c2 INT);
    4> INSERT INTO tab values(123,456);
    5> END;
    6> @@
dmSQL >SELECT * FROM tab;
    C1          C2
=========== ===========
        123         456
1 rows selected
dmSQL> @@
    2> BEGIN
    3> DECLARE c1 INT;
    4> DECLARE SET INT @a2 = 100;
    5> SET c1 = 200;
    6> INSERT INTO tab VALUES(@a2,c1);
    7> END;
    8> @@
dmSQL> SELECT * FROM tab;
    C1          C2
=========== ===========
        123         456
        100         200
2 rows selected
```

12.4 Dropping a Stored Procedure

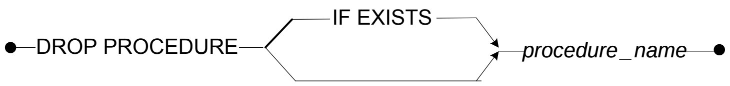


Figure 12-8 Syntax for the DROP PROCEDURE statement:

➲ **Example 1**

The first statement drops the stored procedure **sp_proc1** the second statement drops stored procedure **user1.sp_proc2**.

```
dmSQL> DROP PROCEDURE sp_proc1;
dmSQL> DROP PROCEDURE user1.sp_proc2;
```

➲ **Example 2**

The first statement drops the stored procedure **sp_proc1** if exists, the second statement drops stored procedure **user1.sp_proc2** if exists.

```
dmSQL> DROP PROCEDURE IF EXISTS sp_proc1;
dmSQL> DROP PROCEDURE IF EXISTS user1.sp_proc2;
```

12.5 Getting Procedure Information

➲ **Example 1**

To using dmSQL to get procedure information from the system table **SYSPROCINFO**:

```
dmSQL> SELECT * FROM SYSPROCINFO;
```

➲ **Example 2**

To use dmSQL to get procedure information from system table **SYSPROCPARAM**:

```
dmSQL> SELECT * FROM SYSPROCPARAM;
```

NOTE *ODBC functions* SQLProcedure() *and* SQLProcedureColumns() *are used to get procedure and parameter information for programs.*

12.6 Security

Only the owner or a user with DBA or higher authority can initially execute a stored procedure. Other users can execute the procedure when the execution privilege has been granted to them or a group that the user is a member of. Only owner or a user with DBA or higher authority can grant EXECUTE PROCEDURE privilege on a stored procedure for other users.

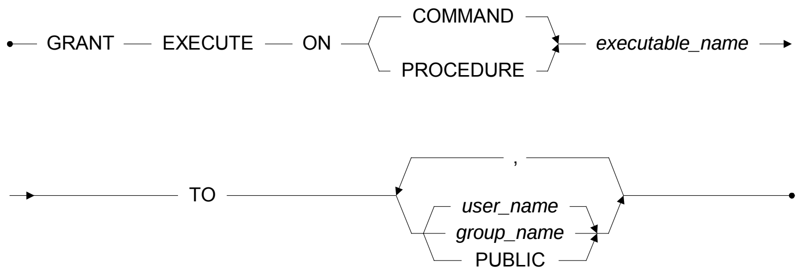


Figure 12-9 Syntax for the GRANT EXECUTE privileges statement

The owner or a user with DBA or higher authority can also revoke execute privilege on a stored procedure for other users.

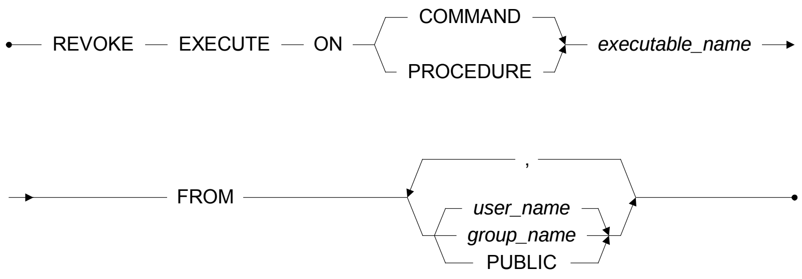


Figure 12-10 Syntax for the REVOKE EXECUTE privileges statement

➲ **Example 1**

user1 creates a stored procedure called **sp_proc1** and grants the execute privilege to **user2** using dmSQL:

```
dmSQL> GRANT EXECUTE ON PROCEDURE sp_proc1 TO user2;
```

➲ **Example 2**

user1 creates a stored procedure called **sp_proc1** and grants the execute privilege to **PUBLIC** using dmSQL:

```
dmSQL> GRANT EXECUTE ON PROCEDURE sp_proc1 TO PUBLIC;
```

➲ **Example 3**

user1 revokes the execute privilege from **user2** using dmSQL:

```
dmSQL> REVOKE EXECUTE ON PROCEDURE sp_proc1 FROM user2;
```

➲ **Example 4**

user1 revokes the execute privilege from **PUBLIC** using dmSQL:

```
dmSQL> REVOKE EXECUTE ON PROCEDURE sp_proc1 FROM PUBLIC;
```

13 Schedules

Organizations have too many tasks and manually dealing with each one can be daunting. To help users simplify these management tasks, as well as offering a rich set of functionality for complex scheduling needs, DBMaker provides advanced job scheduling capabilities. A schedule indicates when a task should run. It has a start time that specifies the date and time when the schedule starts, an end time which indicates the date and time when the schedule expires, as well as a timetable that indicates when it will run.

Schedule enables users to control when and where various tasks take place in the database environment. These tasks can be time consuming and complicated, so using the schedule can help users to improve the management and planning of these tasks. In addition, by ensuring that many routine database tasks occur without manual intervention, you can lower operating costs, implement more reliable routines, minimize human error, and shorten the time system needed.

DBMaker provides advanced job scheduling capabilities with a collection of stored procedures: START_DMSCHSVR, STOP_DMSCHSVR, SCHEDULE_CREATE, SCHEDULE_ALTER, SCHEDULE_DROP, SCHEDULE_RELOAD, SCHEDULE_ENABLE, SCHEDULE_DISABLE, SCHELOG_CLEAN, TASK_CREATE, TASK_ALTER and TASK_DROP. For more details of these procedures, please refer to *SQL Command and Function Reference*. A user with RESOURCE privilege can create/alter/drop a task/schedule, enable/disable a schedule for himself; A user with DBA privilege or higher can create/alter/drop a task/schedule, enable/disable a schedule of all users, start/stop **dmschsvr** and reload all schedules. In addition, users create a schedule only basing on own task.

13.1 Dmschsvr

The daemon **dmschsvr** is mainly used to execute a SQL statement, a procedure, or an executable program, all of that are designed by users in advance. Users can create schedules for routine tasks and **dmschsvr** will automatically execute these tasks, cutting a large amount of routine maintenance.

Dmschsvr executes a job periodically basing on information stored in SYSSCHEDULE and SYSTASK. SCHEDULE_CREATE, SCHEDULE_ALTER, SCHEDULE_DROP, SCHEDULE_ENABLE, and SCHEDULE_DISABLE is separately used to create, alter, drop, enable and disable a schedule, TASK_CREATE, TASK_ALTER and TASK_DROP is separately used to create, alter, and drop a task. After calling of these procedures, the information stored in SYSSCHEDULE and SYSTASK will change.

Dmschsvr uses a special user to connect to a database through ODBC when running, and then scans all schedule information stored in SYSSCHEDULE to find enabled schedules, and finally reads information of the schedule into system. According to these information, **dmschsvr** calculate the task that has the shortest interval between current time and its execution time, and then read this task's data into a queue, and finally, when execution time comes, this task runs. If the data of SYSSCHEDULE change, **dmschsvr** will automatically reload all schedules.

Users can start **dmschsvr** through the command **dmschsvr –d db_name**, setting value of **DB_SchSv** to 1, or calling *start_dmschsvr('taskNum','logPath')*. For more details of START_DMSCHSVR, please refer to *SQL Command and Function Reference*. In addition, the command **dmschsvr –d** also has other parameters: -n, specifying maximum number of task running at the same time; -p, specifying directory where save the schedule logs. However, users can stop **dmschsvr** only by calling *stop_dmschsvr*, and **dmschsvr** will be stopped within a minute after users calling *stop_dmschsvr*.

After start-up, **dmschsvr** will generate a log file every day, and the format of this log file is *<db_name><_><date>.log*, DB_20160304.log for example. These log files mainly record the operation status of **dmschsvr**, schedules and tasks, including start time, end time and abnormal information. By refering to these log files, a user can monitor whether **dmschsvr** is normally running and debug his schedules and tasks until they can be correctly executed. With the increase of log files, a user can automatically delete some old log files by himself, and also can call SCHELOG_CLEAN() to delete log files ahead of the finally generated ones by specified days. For details of SCHELOG_CLEAN(), please refer to Chapter5 *System-Stored Procedures* in the *SQL Command and Function Reference*. In addition, a user can use **DB_SchLgDir** and **DB_SchLgLev** to set the storage directory and level of dmschsvr's log files separately. For details of **DB_SchLgDir** and **DB_SchLgLev**, please refer to Appendix *Keywords in dmconfig.ini*.

13.2 Creating Schedules

Use the following command to create a schedule.

```
dmSQL> CALL SCHEDULE_CREATE('schedule_name','task_name','timetable',
'starttime','endtime');
```

➲ **Example**

Create a task named **insert_t1** to insert values into table **t1**.

```
dmSQL> CALL TASK_CREATE('insert_t1','SQL_STATEMENT','INSERT INTO t1
VALUES(1,2)');
```

Create a schedule named **insert_into_t1** for task **insert_t1**.

```
dmSQL> CALL SCHEDULE_CREATE('insert_into_t1', 'insert_t1', '10 0,1 * * *', '2012-
12-12 12:00:00', '2015-12-12 12:00:00'); // The task 'insert_t1' will run at 0:10
and 1:10 every day from 2012-12-12 12:00 to 2015-12-12 12:00).
```

13.3 Altering Schedules

Use the following command to alter a schedule.

```
dmSQL> CALL SCHEDULE_ALTER('schedule_name','task_name','timetable',
'starttime','endtime');
```

➲ **Example**

Alter schedule **insert_into_t1**. In this example, alter the execution plan "**10 0,1** * * *" to "**20 2,3** * * *". For more information of schedule **insert_into_t1**, please refer to the example in Chapter 13.2, *Creating Schedules*.

```
dmSQL> CALL SCHEDULE_ALTER('insert_into_t1', 'insert_t1', '20 2,3 * * *', '2012-
12-12 12:00:00', '2015-12-12 12:00:00'); // The task 'insert_t1' will run at 2:20
and 3:20 every day from 2012-12-12 12:00 to 2015-12-12 12:00).
```

13.4 Dropping Schedules

Use the following command to drop a schedule.

```
dmSQL> CALL SCHEDULE_DROP('schedule_name');
```

➲ **Example**

Delete schedule **insert_into_t1**. For more information of schedule **insert_into_t1**, please refer to the example in Chapter 13.2, *Creating Schedules*.

```
dmSQL> CALL SCHEDULE_DROP('insert_into_t1');
```

13.5 Reloading Schedules

Use the following command to reload all enabled schedules.

```
dmSQL> CALL SCHEDULE_RELOAD;
```

➲ **Example**

Reload all enabled schedules into system.

```
dmSQL> CALL SCHEDULE_RELOAD;
```

14 Coding User-Defined Functions

DBMaker allows programmers to build their own user-defined functions (UDF). Once a UDF has been written in DBMaker, it is treated as a new built-in DBMaker function with the same usages. Creating a new user-defined function is straight forward and follows the general procedure outlined below.

➲ **To create a user defined function:**

1. Write a user defined function in C (UDF Interface)
   - **a)** Write the include statement
   - **b)** Write the function header
   - **c)** Write the arguments that the function passes
   - **d)** Define allocated memory if necessary
   - **e)** Define an error code, if desired
2. Build the dynamic link library for the UDF
3. Create the UDF in DBMaker, with the data array to be passed to the UDF

14.1 UDF Interface

The first step in creating a UDF is coding it in C. The following sections give an example of a UDF in C, and describe each of the elements of the code that are particular to a DBMaker UDF.

Example

If you want to create a new UDF, **INT2STR()**, to convert integer data to a string, you should build a dynamic link library to include it.

```
dmSQL> SELECT INT2STR(c1) FROM t1;      // c1 is integer type
```

The following C source code, *template.c*, gives a snapshot of code of the **INT2STR()** UDF:

```
#include <memory.h>
#include <string.h>
#include <stdio.h>
#include "libudf.h"

/* Transfer integer type to string type */
#ifdef WIN32
__declspec(dllexport)
#endif
int INT2STR(int narg, VAL args[])
{
  char *ptag;
  int len;
  char p1[11];
  int rc;

  if (args[0].type != NULL_TYP)
     {
     sprintf(p1, "%d", args[0].u.ival);
     len = strlen(p1);
     if (rc = _UDFAllocMem(args, &ptag, len))
         return rc;
     memcpy(ptag, p1, len);
     args[0].type = CHAR_TYP;
     args[0].len = len;
     args[0].u.xval = ptag;
     }
  return _RetVal(args, args[0]);
}
```

Including libudf.h

DBMaker defines some constants, data types and common interfaces, which are needed in UDF coding.

Programmers should include **libudf.h** before any UDF coding:

```
#include "libudf.h"
```

Passing Parameters

The arguments of a UDF used in a SQL command are packaged into the **args** parameter of the UDF coded in C language. Through the **args** array, a UDF gets the input data. **args** is also called the *UDF control block*, which is always used as the first argument of the common interface provided by DBMaker. Some common interfaces, such as the BLOB Common Interface, will be introduced later.

Each UDF header in a C function should follow the form:

```
int FUNCTION_NAME(int narg, VAL args[])
{
...
}
```

NOTE *args[] points to an array. Functions passing only one argument should use the pointer form: *args.*

narg specifies how many arguments the function passes. For example, if a UDF **MYSUBSTRING (c1, c2, c3)** is called in a SQL command, **c1** information is passed by *args[0]*, **c2** by *args[1]* and **c3** by *args[2]*. The value of *narg*, specifying the array size, is 3.

➲ **Example 1**

Using the value of **c1** as 'abcdefghijklmn', *args[0]* would be:

```
args[0].type   = CHAR_TYP
args[0].len    = 14
args[0].u.xval = "abcdefghijklmn"
```

➲ **Example 2**

Using the value of **c2** as integer 30, *args[1]* would be:

```
args[1].type   = INT_TYP
args[1].len    = 4
args[1].u.ival = 30
```

In addition to CHAR_TYP and INT_TYP, BIN_TYP, FLT_TYP, OID_TYP, BLOB_TYP, DEC_TYP and NULL_TYP constants are defined in **libudf.h**:

```
#define BIN_TYP   0x0000                  /* bit string     data type*/
#define CHAR_TYP  0x1000                  /* character      data type*/
#define INT_TYP   0x2000                  /* integer        data type*/
#define FLT_TYP   0x3000                  /* floating point data type*/
#define OID_TYP   0x4000                  /* OID            data type*/
#define BLOB_TYP  0x5000                  /* BLOB           data type*/
#define DEC_TYP   0x6000                  /* decimal        data type*/
#define NULL_TYP  0xF000                  /* set if column is null   */
```

➲ **Example 3**

Through NULL_TYP, the programmer can know whether the input data is NULL:

```
if (args[0].type == NULL_TYP)
{
    /* input data is NULL */
}
else
{
   /* input data is not NULL */
}
```

The complete data structure of VAL, as defined in **libudf.h**:

```
typedef struct tVAL {
   i16 type;                  /* data type          */
   i15 len;                   /* data length        */
   union {
      i31    ival;            /* long integer data  */
      i15    sival;           /* short integer data */
      double fval;            /* double data        */
      float  sfval;           /* float data         */
      dec_t  dval;            /* decimal data       */
      char   *xval;           /* pointer to data    */
   } u;
} VAL;
```

The structure *dec_t*, used for DECIMAL type, in **libudf.h**:

```
typedef struct
{
    i8 pre;
    i8 sca;
    i8 dgt[20];
    i8 exp;
    i8 junk;
} dec_t10;
typedef dec_t10 dec_t;
```

A UDF not only passes input data through *VAL* type, but also returns output data through it. How to return data is discussed later.

Allocating Memory Space

In C functions, you may need to allocate memory and free it before leaving the function. Returned values, such as a string or temporary BLOB ID, need to allocate memory, hold it in the UDF function, and have DBMaker assist in freeing memory space.

➲ **Example**

In the following example of a UDF **UDFAllocMem**, *arg* is the UDF control block, *ppt* is the pointer to get the allocated memory block, and *nb* is the desired allocated size. This function allocates memory and holds it until DBMaker takes care of it:

```
int _UDFAllocMem(VAL *arg, char **ppt, int nb);
```

DBMaker knows to free the memory after a result is returned by using *args[0].u.xval*, a pointer to memory space allocated by **_UDFAllocMem()**.

```
if (rc = _UDFAllocMem(args, &ptag, 10))
   return rc; /* return error code */
memcpy(ptag, "0123456789", 10);
args[0].type = CHAR_TYP;
args[0].len = len;
args[0].u.xval = ptag;
```

Returning Results

There are two types of returned values: one is an error code and the other is the result of the UDF through the argument type VAL. Error codes are returned to DBMaker but their values are hidden from the user; only an error message will be displayed. The following describes how error codes are returned.

The header of UDF in a C function follows the form:

```
int FUNCTION_NAME(int narg, VAL args[]);
```

If **FUNCTION_NAME()** returns a non-zero value there is something wrong, if a 0 is returned it means that the function worked properly.

Before returning from the UDF, call **_RetVal()** to pass the imported result from the UDF to DBMaker with the following declaration:

```
int _RetVal(VAL *arg, VAL rtn);
```

The first argument *arg* is the UDF control block, and the second one *rtn* is the value returned. The following code returns integer 30:

```
int rc;                      /* error code */
VAL rtn;
rtn.type   = INT_TYP;
rtn.len    = 4;
rtn.u.ival = 30;
rc = _RetVal(arg, rtn);      /* pass result back to DBMaker         */
return rc;                   /* return error code (0 means no error) */
```

14.2 Building UDF Dynamic-Link Library

DBMaker provides a library **dmudf.lib** to link with the UDF source file to build the dynamic-link library. Since the dynamic-link library is different on Microsoft Windows and UNIX environments, both cases are discussed separately.

DLL in Microsoft Windows Environment

DBMaker also provides the **template.c** source code in the */udf_templates* directory and the template make files **udf42.mak** (for Microsoft VC++ version 4.2), **udf50.mak** (for Microsoft VC++ version 5.0), or **udf60.mak** (for Microsoft VC++ version 6.0) for WIN32 users and the template make files **udf80.mak** (for Microsoft Visual studio 2005 and higher) for WIN32 and WIN64 users to reference. Users can follow the format of a template C source file to write their UDF.

➲ **In the following statements, the udf60.mak is used.**

1. Ensure where to include the **dmudf.lib** file and then use the IDE that Visual C++ provides to modify the required changes.
2. Copy **udf60.mak** template make file into the desired directory and rename it with a make file name.
3. Choose <Open Workspace> from the <File> menu to open the make file project workspace.
4. Choose <File View> from the <Project Workspace> menu and click **template.c.** To remove the template click Delete
5. Choose <Project> item in the tool bar, choose < Add to project >, <Files>, and, insert your own .c file into the make file of the project workspace.
6. In the <Project> -> <Settings>, choose WIN32 Debug for this example. In the Project Settings <General>, you can change output directories. In the template make file, set 60Deb as the intermediate and output directories.
7. In the Project Settings <Link> item, in Category item <General>, change the output .dll file name directly in <Output file name>. Also, change the link path of the *dmudf.lib* file DBMaker supports in the <Object/Library modules> to the working directory.

After completing the steps above, you can build your own **dll make** file. Using similar steps, you can also build a WIN32 Release version dll file.

Users of VC++, can also create a dll make file using the same steps but setting the structure member alignment to be 4 bytes. In the VC ++ 6.0 IDE project workspace, choose the C/C++ menu item, and then in the **Category** dialog box, choose <Code Generation>. You can find the structure member alignment option, and then choose 4 bytes as the result.

Use the make file template to note the setting when writing a **collect dll**. If you do not want to use **template.c** as the default C filename within the make file, remove

template.c from **udf60.mak** and insert your C file into the **udf60.mak** project workspace.

➲ **Example**

In the DBMaker **template.c**, remember to include the **libudf.h** file provided, and to export your functions. Use the export function method from the VC++ programmer guidebook or the following:

```
__declspec(dllexport) datatype YOUR_FUNCTION_NAME( ...... )
```

Alternatively, create a **def file** in the project workspace to export your functions and note that the function name for the UDF must be in UPPER CASE, in C source code.

After finishing the above, you can build a **debug/release version dll** file, thus creating an **udf60.dll** file.

➲ **In the following statements, udf80.mak is used.**

1. Copy **udf80.mak** and **udf80.def**, **template.c** and **udf60.dsp** into the desired directory, and rename **udf60.dsp** to **udf80.dsp**.
2. Edit **udf80.def**, you can replace **template.c** with you own **.C** file.
3. Edit **udf80.def**, you can change output directories. In the template make file, set 60Deb as the intermediate and output directories.
4. Edit **udf80.def**, you can change the output .dll file name directly to **udf80.mak**.

After completing the above steps, you can build your own **dll make** file with cmd.

Open cmd, switch to the desired directory with the cd command, and then execute the following command.

```
@CALL bat_vs_env
@NMAKE /NOLOGO /C /S /f udf80.mak /x make.err CFG="udf80 - Win32 Debug"> make.msg
```

bat_vs_env : Its value is determined by both Visual Studio version and operation system, for example, if c compile is VS2005 and the operation system is 32bit, replace bat_vs_env with "C:\Program Files\Microsoft Visual Studio 8\VC\bin\vcvars32.bat". For details, please reference the following table.

VS version	OS	bat_vs_env
VS2005	32bit	C:\Program Files\Microsoft Visual Studio 8\VC\bin\vcvars32.bat
	64bit	C:\Program Files (x86)\Microsoft Visual Studio 8\VC\bin\amd64\vcvarsamd64.bat
VS2008	32bit	C:\Program Files\Microsoft Visual Studio 9.0\VC\bin\vcvars32.bat
	64bit	C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\bin\amd64\vcvarsamd64.bat
VS2010	32bit	C:\Program Files\Microsoft Visual Studio 10.0\VC\bin\vcvars32.bat
	64bit	C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\amd64\vcvars64.bat
VS2012	32bit	C:\Program Files\Microsoft Visual Studio 11.0\VC\bin\vcvars32.bat
	64bit	C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\bin\amd64\vcvars64.bat

To build Release version of dll, you can replace Debug with Release in cmd line "@NMAKE /NOLOGO /C /S /f udf80.mak /x make.err CFG="udf80 - Win32 Debug"> make.msg".

UDF so File in UNIX

A **so** file, or UNIX dynamic library, can be created.

➲ **Example**

Write UDF C source code, in the example the file is named **udf.c**. After completion, use the UDF function in a UNIX based OS

```
$ cc -c udf.c
$ ld -o libudf.so udf.o -lm
$ dmsqlt
dmSQL> CREATE FUNCTION libudf.INT2STR(INT) RETURNS CHAR(10);
```

NOTE *The options of the* ***ld*** *command in the above example can vary in UNIX. It may be –G, –shared, or something else. Please refer to your UNIX manual or man pages to check how to use the* ***ld*** *command in building a shared library.*

14.3 Creating, Using, and Dropping UDF

The next step for a user-defined function is to create it within DBMaker. The following sections outline the syntax for creating, querying, and dropping a UDF.

Creating a UDF

➲ **Syntax**

```
dmSQL> CREATE FUNCTION <udf_dll_name.function_name> (<function_datatype>) RETURNS
<function_output_datatype>;
```

Querying a UDF

➲ **Syntax**

```
dmSQL> SELECT <function_name>(<related_table_column_name>)
FROM <related_table>;
```

Dropping a UDF

➲ **Syntax**

```
dmSQL> DROP FUNCTION <function_name>;
```

Example

The following demonstrates how to use a UDF file.

➲ **Example 1**

Using a database named **DMDEMO** containing a table, **tb_UDF,** with the table schema, **number INT, name CHAR(10)**:

```
dmSQL> SELECT * FROM tb_UDF;
 number     name
========  =========
10        1
20        2
30        3

3 rows selected
```

Using the example **template.c** DBMaker supports, we can now build an **udf60.dll** successfully.

In the **dmconfig.ini** file, add one line to the **DMDEMO** section:

```
[DMDEMO]
DB_DbDir = D:\UDFDEMO
DB_FoDir = D:\UDFDEMO\FO
DB_LbDir = D:\UDF\60Deb   ; add this line
```

For more information on **DB_LbDir**, refer to *Keywords in dmconfig.ini*. Set **DB_LbDir** or place the **udf60.dll** in the ***<DBMaker home directory>\shared\udf***, since it is the UDF default directory.

➲ **Example 2**

Start the database **DMDEMO**, and then create the UDF function. In the example, the <*udf_dll_name*> is **udf60**, the <*function_name*> is **INT2STR**, <*function_datatype*> is INT, and <*function_output_datatype*> is CHAR(10):

```
dmSQL> CREATE FUNCTION udf60.INT2STR(INT) RETURNS CHAR(10);
```

The UDF function **INT2STR** returns the following results. The <*function_name*> is **INT2STR**, <*related_table_column_name*> is **number** according to the schema of **tb_UDF** and the <*related_table*> is **tb_UDF**:

```
dmSQL> SELECT INT2STR(number) FROM tb_UDF;

 INT2STR(number)
==================
10
20
30

3 rows selected
```

➲ **Example 3**

Another UDF function, e.g., **STR2INT(),** in the same dynamic-link file:

```
dmSQL> CREATE FUNCTION udf60.STR2INT(CHAR(10)) RETURNS INT;
dmSQL> SELECT STR2INT(name) FROM tb_UDF;

 STR2INT(name)
==============
1
2
3

3 rows selected
```

➲ **Example 4**

When dropping a UDF function, simply drop the UDF function name, there is no need to attach the **UDF dll** name. When dropping a UDF function, wait until the database has terminated, then the UDF function will be cleaned up. Before the database is terminated, the function will continue to exist.

```
dmSQL> DROP FUNCTION INT2STR;
```

14.4 Create XML Validate UDF

Flexml

Kristoffer Rose's flexml, distributed under the GNU General Public License, is an XML process generator. It takes a DTD file and generates a LEX file. Flexml is available at http://flexml.sourceforge.net.

GENERATING THE LEX FILE

```
$ flexml name.dtd
```

ADDING CUSTOMIZED YY_INPUT

The original LEX input is a FILE input stream. The LEX file must be modified to use. UDF Blob as an input source. The following example demonstrates this modification by adding customized YY_INPUT.

➲ **Example**

Modify the definition section of the LEX file by adding YY_INPUT as shown below. The definition section is located at the beginning of the file and between the "%{" and "}%" markers

```
#include "libudf.h"

typedef struct udf_file
        {
        VAL *args;
        i31 handle;
        i31 rc;
  i31 left;
        i31 rlen;
        } UDF_FILE;

#undef YY_INPUT
#define YY_INPUT(buf,result,max_size)  {\
  UDF_FILE * uf =(UDF_FILE *)yyin; \
        errno=0; \
        if ( uf->left <= 0 )\
        {\
          result = (uf->rlen=0);\
        }\
        if ( (uf->rc = _UDFBbRead(uf->args, uf->handle, max_size, &(uf->rlen),
buf))!=0 ) \
  { \
          errno = uf->rc; \
          result = 0;\
        }\
        else\
        {\
    uf->left -= uf->rlen;\
          result = uf->rlen;\
        }\
}\
```

Next, add the UDF function to the end of the LEX file as shown below:

```
#ifdef WIN32
__declspec(dllexport)
#endif
int XXX_VALIDATE(int nArg, VAL args[])
{
  BBObj bbin;
  UDF_FILE uf;
  int rc = 0;
  int rc2 = 0;
  memset(&uf, 0, sizeof(UDF_FILE));
  memcpy((char *)(&bbin), args[0].u.xval, BBOBJ_SIZE);
  uf.args = args;
  rc = _UDFBbOpen( args, bbin, &(uf.handle));
  if( rc != 0 )
    goto EXIT;
  if (rc = _UDFBbSize(args, bbin, &(uf.left)) )
         {
    goto EXIT;
         }
  yyin = (void *)&uf;
  rc = validdtd00udf();
  rc2 = _UDFBbClose(args, uf.handle);
EXIT:
  if( args[0].type != NULL_TYP ) // null column data
         {
           args[0].type = INT_TYP;
           args[0].len = 4;
           args[0].u.ival = (rc == 0? 1:0);
         }
  return _RetVal(args, args[0]);
}
```

BUILD DLL/SO

```
flex name.l
```

```
cc -c -DBUILD_DLL lex.name.c -Idbmaker-installed-dir/include
```

CREATE UDF

```
dmSQL> CREATE FUNCTION dllname.udfname(BLOB) returns int;
```

CREATE COLUMN WITH CHECK CONSTRAINT

```
dmSQL> CREATE TABLE table-name( c1 XMLTYPE CHECK udfname(value) = 1);
```

DBMaker DTD Validation UDF Generator

Command line tool for generating a validation UDF for the specified DTD. The DTD filename is required. If not specified, an error is generated. The output directory is optional. If not specified, the files are created in the current working directory. The prefix is optional. If specified, the generated file uses the prefix in the filename. If not specified, the DTD filename without a file extension is used.

```
$ dmxmludfmk -dtd dtd-file-name [-o output-directory] [-p prefix]
```

Several files are generated as follows;

- Lex file:< user-specified-prefix.l>
- Yacc file:<user-specified-prefix.y>
- UDF function file: <user-specified-prefix> udf.c and <user-specified-prefix> udf.h
- The UDF function is named as <user-specified-prefix>_VALIDATE
- hash.c and hash.h provide hash functions
- Makefile on UNIX platforms or Makefile.msvc on Windows platforms

➲ **Example 1**

```
Make <user-specified-prefix>.so        ;for UNIX
```

➲ **Example 2**

```
Nmake /f Makefile.msvc                 ;for Windows visual studio 2005 or 2008
```

➲ **Example 3**

```
nmake /f Makefile.msvc COMPILER=VC60   ;for Windows visual studio 6.0
```

➲ **Example 4**

```
nmake /f Makefile.msvc OSTYPE=$OSTYPE  ; for cygwin environment
```

Please note that dmxmludfmk supports ASCII, and BIG5, gb, shiftJIS, and utf8 while flexml supports content replacement for internally defined DTD only entities.

Default Validator

I_VALIDATE is provided as a default validator for checking the XML's syntax. I_VALIDATE does not provide validation against the DTD or the XMLSchema. I_VALIDATE is part of libmedia library.

14.5 UDF BLOB Common Interface

Today, multimedia is important and useful to users. DBMaker supports a common interface to access BLOBs using a file handle method, so programmers can easily write UDFs for BLOB type data. FILE, LONG VARCHAR, and LONG VARBINARY are the data types used to store BLOB data in a database.

Many of the new features in DBMaker need a temporary BLOB to process temporary results. DBMaker supports temporary BLOBs for programmers to write a UDF more easily. A programmer can open a permanent BLOB, read the data, execute a conversion function or something else, save the result in a new temporary BLOB and return it back in a UDF. The API fetches this temporary BLOB as a normal BLOB column.

BLOB Common Interface Functions

DBMaker provides BLOB common interface functions for programmers to write UDFs. A DBA should set the **DB_FoDir** in the **dmconfig.ini** file for the temporary BLOB file before starting a database. A temporary BLOB will be created in an external file in the **DB_FoDir** directory, with the file name format "**__??????.TMP**", where "?" represents one character of either [0-9, A-Z]. All file names matching the format will be deleted when the database is shut down and restarted.

_UDFBbOpen()

Opens a BLOB using **bbObj** and returns a handle through **pHandle**. **bbObj** can be retrieved by **Arg[i]**, using the **BLOB** with the input argument of the UDF. The function returns 0 if it successfully opens the **BLOB**, otherwise an error code will be returned:

```
int _UDFBbOpen(VAL *Arg, BBObj bbObj, i31 *pHandle);
```

_UDFBbRead()

This function reads the BLOB that belongs to the specified handle. Before calling this function, allocate a buffer, (**pBuf**), with **szBuf** using the function **_UDFAllocMem()** to get the read data. The returned data will be stored in **pBuf** and the size actually read is in **szRead**. If **szBuf** is non-positive, no characters are read:

```
 int _UDFBbRead(VAL *Arg, i31 handle, i31 szBuf, i31 *szRead, char
*pBuf);
```

_UDFBbSeek()

This function is used to set the position of the next output operation in a **BLOB**. The new position is at the offset bytes from the beginning, the current position, or the end of the file, according to the **ptrname** using the **SEEK_BB_BEG, SEEK_BB_CUR,** or **SEEK_BB_END** value defined in **libudf.h**. The function only works between the period of **_UDFBbOpen()** and **_UDFBbClose()**, but not **_UDFBbCreate()** and **_UDFBbClose()**:

```
int _UDFBbSeek(VAL *Arg, i31 handle, i63 offset, i16 ptrname);
```

_UDFBbCurOffset

The function returns the current position of an open **BLOB** or the offset in a **BLOB** by **pOffset**, but it will return at most 2^{31} - 1 even when the current offset is greater than or equal to 2^{31}.

```
int _UDFBbCurOffset(VAL *Arg, i31 handle, i31 *pOffset);
```

_UDFBbCurOffsetEx

Unlike _UDFBbCurOffset, this function always returns the actual current position of an open **BLOB** or the offset in a **BLOB** by **pOffset**:

```
int _UDFBbCurOffsetEx(VAL *Arg, i31 handle, i63 *pOffset);
```

_UDFBbClose()

Closes the **BLOB** opened by **_UDFBbOpen()** or created by **_UDFBbCreate()**:

```
int _UDFBbClose(VAL *Arg, i31 handle);
```

_UDFBbCreate()

Creates a temporary **BLOB** and returns a handle for **_UDFBbWrite()**. The caller should prepare the space for the **BBObj** structure pointed to by **pBbObj** and written by **_UDFBbCreate()**, **_UDFBbWrite()** and **_UDFBbClose(). BBObj** is used to identify this temporary **BLOB**. For example if you want to delete the temporary **BLOB** called **_UDFBbDrop()** using the **BBObj** argument.

If successful, **pHandle** will return a **BLOB** handle similar to the handle of the opened file written by **_UDFBbWrite()** and closed by **_UDFBbClose()**.

Alternatively, specify the temporary **BLOB** to be created in file (**BB_TEMP_FO**) or in memory (**BB_TEMP_MEM**) . If the caller specifies the temporary **BLOB** in memory, it does not mean that the temporary **BLOB** will be created in memory - a memory limitation may prevent this. The temporary **BLOBs** in memory might be converted to files by the operating system if the original temporary **BLOBs** in memory or the input data are over the size limit. Programmers should not depend on this feature when coding.

The function returns **0** if it is successful and an error code will be returned otherwise.

Before reading the new temporary **BLOB**, you must close it using **_UDFBbClose()**, then reopen it using **_UDFBbOpen()**. **_UDFBbSeek()** cannot be used on temporary **BLOBs** unless they are closed and reopened for reading:

```
int _UDFBbCreate(VAL *Arg, BBObj *pBbObj, i31 *pHandle, i31 Opt);
```

_UDFBbWrite()

After using **_UDFBbCreate()** to make a temporary **BLOB**, write data to it using **_UDFBbWrite()**. The handle is from **_UDFBbCreate()**, **pBuf** points to input data and its length is **szBuf**. The function returns **0** if it is successful, otherwise, an error code will be returned:

```
int _UDFBbWrite(VAL *Arg, i31 handle, i31 szBuf, char *pBuf);
```

_UDFBbDrop()

Normally you do not drop a temporary **BLOB** if it will be returned from a **UDF**; the system will control its life cycle. If you do not return the created **BLOB**, you'd better use this function to drop the temporary **BLOB**. This function cannot work on a permanent **BLOB**; doing so will return the **ERR_BLOB_INV_BLOB** error. The function returns **0** if it is successful, otherwise, an error code will be returned:

```
int _UDFBbDrop(VAL *Arg, BBObj bbObj);
```

_UDFBbSize()

This function returns the data size of a **BLOB** by **pLen**. **BbObj** can be a permanent BLOB or a temporary BLOB. But it will return at most 2^{31} - 1 even if the size is greater than or equal to 2^{31}. The function returns 0 if it is successful, otherwise, an error code will be returned:

```
int _UDFBbSize(VAL *Arg, BBObj bbObj, i31 *pLen);
```

_UDFBbSizeEx()

Unlike _UDFBbSize, this function always returns the actual data size of a **BLOB** by pLen. **BbObj** can be a permanent BLOB or a temporary BLOB. The function returns 0 if it is successful, otherwise, an error code will be returned:

```
int _UDFBbSizeEx(VAL *Arg, BBObj bbObj, i63 *pLen);
```

Example

The following demonstrates how to create the user-defined function, **MYCONVERT** with input in varchar format and output as a temporary BLOB.

➲ **To create the user-defined function, MYCONVERT:**

1. Build a dynamic library in UNIX using **myudf.c**, (the source code follows later):

```
cc -g -c myudf.c
ld -G -o myudf.so myudf.o
```

2. Start the database.

3. At the dmSQL prompt, enter:

```
dmSQL> CREATE FUNCTION myudf.myconvert(VARCHAR(100))  // input string
    2>  RETURNS LONG VARCHAR;                         // output BLOB
dmSQL> SELECT myconvert(c1) FROM mytable;             // output temp BLOB
```

The source code for the UDF **MYCONVERT**:

```
#include "libudf.h"
int MYCONVERT(int nArg, VAL args[])
{
  int     rc = 0, trc;            /* return code                     */
  BBObj   tmpobj;         /* output temp BLOB                */
  i31     handle;         /* handle of created temp BLOB     */
  boolean fgCreate = FALSE;       /* temp BLOB has been created?     */
  char   *pInData, pOutData[4096];/* input/output data buffer        */
  i31     nInData, nOutData;      /* input/output data buffer length */

  if (args[0].type == NULL_TYP)
    goto cleanup;

  pInData = args[0].u.xval;       /* get input data                  */
  nInData = args[0].len;          /* input data length               */

  /* create a temp BLOB in file */
  if (rc = _UDFBbCreate(args, &tmpobj, &handle, BB_TEMP_FO))
    goto cleanup;
  fgCreate = TRUE;

  /* any real processing function */
  RealConvert(pInData, nInData, pOutData, &nOutData);

  /* write result data to temp BLOB */
  if (rc = _UDFBbWrite(args, handle, nOutData, pOutData))
    goto cleanup;

  /* close created temp BLOB ( temp BLOB is still alive) */
  if (rc = _UDFBbClose(args, handle))
    goto cleanup;
  args[0].type = BLOB_TYP;
  args[0].len = BBID_SIZE;
  args[0].u.xval = (char *)&tmpobj;
                /* _RetVal() does a copy from this local buffer  */

cleanup:
  if (rc)
    {
    /* error handle */
    if (fgCreate)
      {
      _UDFBbClose(args, handle);      /* close created temp BLOB     */
      trc = _UDFBbDrop(args, tmpobj); /* drop it because of failure */
      if (trc > rc)
        rc = trc;
      }
    return rc;
    }
  else
    return _RetVal(args, args[0]);

}/* MYCONVERT() */
```

Troubleshooting Errors

Use the following to troubleshoot errors when writing a BLOB UDF using the BLOB common interface.

ERROR (327): THE BLOB COLUMN IS NOT OPENED OR CREATED YET

The function must use **_UDFBbOpen()** to open the BLOB or **_UDFBbCreate()** to create a new temporary BLOB, before using other BLOB function interfaces.

ERROR (328): THE OFFSET OF BLOB COLUMN IS INVALID

When a UDF using **_UDFBbSeek()** seeks to offset by a length greater than the length of the BLOB.

ERROR (331): THIS BLOB WAS NOT IN CREATED STATE

_UDFBbWrite() can only work on a temporary BLOB created by **_UDFBbCreate()** and must not be closed. For example, if you use it on BLOB opened by **_UDFBbOpen()**, this error will occur.

ERROR (330): THIS BLOB WAS NOT IN OPENED STATE

_UDFBbRead() can only work on a BLOB (including temporary BLOBs) opened by **_UDFBbOpen()**.

ERROR (332): THE BLOB OBJECT IS NOT CLOSE YET

Whenever **_UDFBbOpen()** or **_UDFBbCreate()** are used to open a BLOB, programmers should call **_UDFBbClose(),** to close the opened BLOB.

ERROR (322): NO FILE OBJECT DIRECTORY IN CONFIGURATION FILE; CANNOT INSERT FILE OBJECT

If temporary BLOBs are used, the keyword **DB_FoDir** in the **dmconfig.ini** file must be set. If not set, attempting to create a temporary BLOB may fail and this error occurs.

14.6 UDF related dmconfig.ini keywords

DB_StrSz

In addition to **DB_LbDir** and **DB_FoDir**, there is also a related keyword **DB_StrSz** in **dmconfig.ini** file

```
DB_StrSz=<value>
```

This keyword indicates the length of returned data of the STRING type, used only by user-defined function (UDF). Since UDFs can only return data of a fixed size, these keywords can limit the size of STRING data, in order to avoid receiving strings that are to long. The default value is 255, and the valid range is from 1 to 4,096. It can be used on a client or server, the client has a higher priority.

15 Database Recovery, Backup, and Restoration

In every database management system, the possibility of a hardware or software failure always exists. A DBMS may fall victim to failures without warning. After a failure occurs, a DBMS should have some method of recovering the information. This is one of the main advantages a DBMS has over the old file-based systems they replaced.

DBMaker incorporates advanced data protection features to prevent data loss and downtime due to failures. These features allow DBMaker to ensure the reliability of a database and the consistency of data by providing recovery, backup, and restoration features.

15.1 Types of Database Failures

Database failures can be divided into two types: system failures and media failures. When either of these types of failure occurs, there is the possibility of data inconsistency or data loss in a database. A DBMS should provide facilities for recovering from failures and for replacing a damaged database with a backup copy.

System Failures

A system failure, known as an instance failure, is a failure from the main memory in a computer system. System failures may be caused by a power failure, an application or operating system crash, a memory error, or other reason. The result is the unexpected termination of DBMS.

Applications and active transactions can terminate abnormally when a system failure occurs. Since the exact state of a transaction in progress or a transaction that has not been completely written to disk cannot be reliably be determined after a system failure, these types of transactions require recovery. The most common method of protection against system failures is the use of a transaction log, or a journal file.

Media Failures

Media failure (e.g., disk failure) is a failure of the disk storage system of a computer system. Media failures are usually caused by physical trauma to the disk itself, such as a head crash, fire, or exposure to vibration or g-forces outside its physical operating limits.

There is nothing to prevent the loss of data on an affected disk when a media failure occurs. One or more files may be physically damaged because of the failure, requiring restoration of the database. However, the database can be successfully restored if it provides backup and restoration facilities.

15.2 Recovery from Database Failures

The goals of recovery after a database failure are to ensure committed transactions are reflected in the database, ensure uncommitted transactions are not reflected in the database, and to return to normal operation as quickly as possible while insulating users from problems caused by the failure.

DBMaker uses journal files and checkpoints to achieve these goals. The journal files and checkpoints work together ensuring all transactions are recovered in the shortest time, with the least little effect on users.

Journal Files

Journal files provide a real-time, historical record of all changes made to a database, and the status of each change. In the event of a system failure, the historical record of changes maintained in the journal file allows DBMaker to recover and redo changes made by transactions that completed but were not written to disk, or undo changes made by transactions that terminated abnormally.

If a database is running in backup mode, the journal files will also store additional information that DBMaker can use to restoration. This information will remain in the journal files until they are backed up; after you back up the journal files DBMaker will free this space for use by new transactions.

During the restoration process, DBMaker adds information from the backup journal files to a backup copy of the database. Therefore, only the journal files that contain changes made to the database between full backups require backup.

Checkpoint Events

A checkpoint is a system event that brings the database to a clean state. DBMaker writes all journal records and all dirty data pages from its internal memory buffers to disk, and reclaims journal blocks that are no longer required for backup or recovery purposes. DBMaker can reclaim journal blocks that contain non-active transactions that completed before the start of the oldest active transaction.

Startup time after an instance failure is reduced after taking a checkpoint. DBMaker writes the time of the last checkpoint and a list of all transactions active at the time of the checkpoint to the journal file header. During database recovery, DBMaker uses this information to determine which transactions should be undone, which should be redone, and which should be ignored.

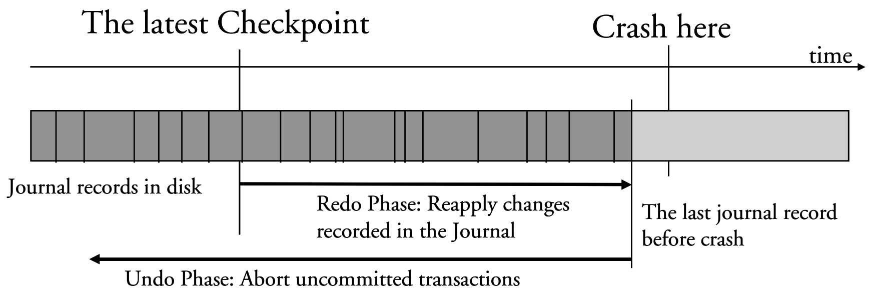
DBMaker will automatically take a checkpoint when the journal files are full to try to reclaim some journal blocks to reuse. If the checkpoint cannot reclaim enough space to complete the current transaction, the transaction will be aborted. DBMaker will also automatically take a checkpoint when the database starts and shuts down, and when an online backup is performed.

Database administrators can manually initiate a checkpoint by executing the CHECKPOINT command. The optimal interval between two checkpoints depends on the frequency of database activity, the average size of transactions, and the size and number of journal files. Since these factors may vary significantly from database to database, the optimal interval is best determined through experience. Manual checkpoints reduce the amount of time required to start, terminate, and backup a database, as well as the possibility that a full journal will be encountered.

Checkpoints may require a significant amount of time to complete, depending on the size and number of transactions since the last checkpoint. Any transactions that are active when a checkpoint occurs need to wait for DBMaker to calculate which journal records it can reclaim, but do not need to wait while DBMaker actually writes journal records and dirty data pages to disk.

Recovery Steps

DBMaker provides support for automatic recovery when the database is started after a system failure or when an error occurs during startup. During the recovery process, DBMaker always performs two separate steps: redo and undo.


The latest Checkpoint
Crash here
time
Journal records in disk
Redo Phase: Reapply changes recorded in the Journal
The last journal record before crash
Undo Phase: Abort uncommitted transactions

The first step in the recovery process is to redo (or reapply) all changes made to the database that are recorded in the journal. This step is necessary since it is possible for a transaction to have completed before the system failure, without having all the changes made by the transaction written to the database. However, these changes *are* stored in the journal, and can be written to the database during this step. After this step, the database contains the changes made by all committed transactions and the changes made by all uncommitted transactions.

The second step in the recovery process rollbacks (i.e., undoes) the changes made by transactions that were not completed before system failure occurred. This step is necessary since the exact state of a transaction in progress cannot be reliably determined in the event of a system failure. These incomplete transactions must be removed since a transaction is self-contained by definition and must either complete successfully and change the data, or fail and leave the data unchanged. At the end of this step, the database contains the changes made by all committed transactions, but does not contain any changes made by uncommitted transactions.

DBMaker also supports starting a database after a media failure or after a system failure, which causes inconsistencies in a database that cannot be repaired during the automatic recovery process. In these cases, the database fails to start and you would normally need to restore your database from a backup. However, if you have never backed up your database, you can force the database to start by setting the forced-start mode using the **DB_ForcS** keyword in the **dmconfig.ini** file. This allows you to start the database and unload the unaffected data. For more information on the forced-start mode, see *Forcing Database Startup*.

Forcing Database Startup

DBMaker automatically performs recovery operations if errors occur when a database starts normally. If the database cannot start up, there may be some disk errors. Disk errors require the database be restored from the most recent backup to repair it. If the database has no backups and cannot start, use the *forced startup* mode provided by DBMaker.

DBMaker supplies a forced startup option for this situation. To set the forced startup mode on, use the **DB_ForcS** keyword in the **dmconfig.ini** file. Setting this keyword to 1 enables forced startup mode, and setting it to 0 disables it. When forced startup mode is on, DBMaker skips errors when starting the database.

If the database still cannot be started, there is one remaining alternative provided in the procedure below. However, before performing this procedure, backup all data and journal files.

➲ **To start a database when it will not start in force start database mode:**

1. Set the Forced Startup Mode to off in **dmconfig.ini** (**DB_ForcS** = 0)
2. Set the Start Mode to New Journal Mode in **dmconfig.ini** (**DB_SMode** = 2)
3. Restart the database
4. Reset Start Mode back to normal in **dmconfig.ini** (**DB_SMode** = 1)

DBMaker provides the option to use a new journal to force the database to start without any recovery operations. Therefore, if errors serious enough to prevent the database from starting have occurred, the database may be in an inconsistent state.

After starting the database with this method, check the consistency of the database. For more information on database consistency checking, refer to section 6.12 *Checking Database Consistency*.

15.3 Types of Backups

Backups are used to protect a database from media failures or other media errors. After a media failure, one or more database files may be damaged and unusable. Use the most recent backup to replace the damaged files and reconstruct the database.

Database backups consist of backup sequences. A backup sequence consists of a full backup, all the differential backups associated with the full backup and incremental backups performed after the full backup.

Full Backups

A full backup is any backup that creates a copy of all data and journal files, providing a copy of the entire database at one point in time. A backup copy of the **dmconfig.ini** file can be created as well; preserving any custom configuration settings there may be for a database. The database administrator may perform a full backup while the database is online or offline.

Full backups archive the entire database, therefore requiring a large amount of storage space. However, a database can be restored relatively quickly using a full backup. A full backup can be used to restore a database to the point in time the full backup was performed.

A valid full backup will be assigned a full backup ID. The full backup ID is a time and date stamp. The backup ID ensures that full backups, differential backups and incremental backups are correctly associated in a backup sequence. Please note that a differential backup records only data that has changed since the most recent full backup. The differential base is necessary when restoring from a differential backup. A differential backup alone is insufficient for rebuilding a database. All incremental backups between the current valid full backup and the next belong only to the current valid full backup. Trying to restore incremental backups against previous (and any other) sequences will fail. Backup sequences are managed by DBMaker. Repairing a database, restoring a database, starting a database in new journal mode, or changing the backup mode will require a new valid full backup.

There are three primary methods of performing full backups. The first is by using the backup server, and is discussed in more detail in section 15.6 *Backup Server*. Full backups by backup server may be performed with dmSQL or with the JServer Manager utility. The second method uses an interactive full backup. Full backup interactively does not require that the Backup Server be started. JServer Manager is the recommended method of performing this type of full backup. For directions on how to perform a full backup interactively, refer to the *JServer Manager User's Guide*. The third method for performing a full backup is offline full backup. Refer to section 15.5, *Offline Full Backups,* for more information.

Differential Backups

Differential backups help us save some time and disk space. Unlike a full backup that simply copies all files, differential backups use a different approach.

A given differential backup is based on the most recent full backup. This full backup is the differential's *base*. A differential base must exist before a differential backup is created.

Differential backups contain only data that has changed since the time that the differential base backup was created. A single differential base is typically used for several successive differential backups. Later, should a database restore become necessary, both the full backup (the differential base backup) and a differential backup is needed.

Data files (all DB files and BB files) are included during differential backups. Journal files, because they change frequently, differential backups copy only useful journal blocks.

A differential backup records only the data that has changed since the most recent full backup. This allows users to make more frequent backups because differential database backups are smaller than full backups. Making frequent backups decreases the risk of data loss. Consider using differential database backups when:

- Only a relatively small portion of the database has changed since the last full backup. Differential backups are particularly effective when the same data is modified many times.
- Frequent backups are desired but not frequent full backups
- Minimizing roll forward time of a transaction journal backups when restoring a database

There are two methods for performing differential backups. The first uses backup server. Configuration of several keywords is necessary before starting the database. See Section 15.6, *Backup Server*. The second method is called on-line differential backups. The JServer Manager utility is recommended for this method and details are explained in the *JServer Manager User's Guide*.

Incremental Backups

An incremental backup is any backup that creates a copy of only the journal files that have changed since the last incremental, differential or full backup. Incremental backups may only be performed after a full backup or a differential backup has been performed. Performing a new full backup starts a backup sequence. Subsequent incremental backups are part of that sequence and may not be used with any other full or differential backup. Note that an incremental backup is composed of journal files which record all transactions since the backup mode (**DB_BMode**) is on. When

DBMaker Database is running on normal mode (**DB_SMode** = 1), before doing an incremental backup, a full backup or a differential backup must has been done; when on replication mode (**DB_SMode** = 4) , an incremental backup can be done without a full backup or a differential backup. The incremental backup file sequence provides a copy of the changes made to the database since the last full backup. The database administrator can perform an incremental backup only while a database is online.

Incremental backups archive only journal files, so they require only a small amount of storage space. However, it may take more time than a full backup to restore a database since the DBMS must rollover all transactions in the backup journal files. Use the incremental backups together with a full and differential backup to restore a database to any point in time between the previous full backup or differential backup and the time the last incremental backup was completed.

There are two methods for performing Incremental Backups (there is a third method Incremental backup to current, which is considered a different type of backup). The first method is Incremental backup by backup server. Backup server must be started on the database to be able to use this method. For more information on performing incremental backup by backup server, refer to section 15.6 *Backup Server*. The second method is incremental backup interactively. This type of Incremental backup does not require that backup server be started. The recommended method of performing incremental backup interactively is with the JServer Manager utility. Incremental backup interactively is explained in the *JServer Manager User's Guide*.

Offline Backups

An offline backup is any backup that must be performed after a database has been shut down. The database administrator must schedule a time to shut down the database, and notify all users so they can disconnect from the database. Offline backups can be inconvenient for users, since they must remember to complete all active transactions and disconnect from the database before it is shut down. The database administrator can perform only full backups while offline.

A DBMS does not need to provide the capability to backup a database offline, since you can backup the database with operating system commands after it is shut down. The database administrator may perform an offline backup using this method, or by using JServer Manager, an easy-to-use graphical tool that performs offline backups without resorting to using operating system commands.

Online Backups

An online backup is any backup that is performed while a database is running. The database administrator does not have to shut down the database, and users do not need to disconnect. Online backups are more convenient for users, since no action is required on their part. The database administrator can perform full, differential and incremental backups while online.

A DBMS must provide the capability to backup a database online, since it is still running and still has users connected. DBMaker allows for online backups to be performed manually using dmSQL and operating system commands, but also provides JServer Manager, an easy-to-use graphical tool that allows online backups to be performed without resorting to operating system commands.

Online Incremental to Current Backups

DBMaker also supports an additional backup type called online incremental to current.

The difference between an online incremental backup and an online incremental to current backup in a database with multiple journal files is minor, but important. In an online incremental backup DBMaker will backup all journal files that have been used since the last backup, excluding the active journal file. In an online incremental to current backup DBMaker will backup all journal files that have been used, including the active journal file. This means that an online incremental backup can restore a database up to the point in time the last committed transaction was written to the last full journal file, while an online incremental to current backup can restore a database up to the point in time the active journal file was backed up.

In a database with only a single journal file, an online incremental backup and an online incremental to current backup are the same. In this case, the only journal file is the active journal file. DBMaker will backup this single journal file in both types of incremental backup.

Online Incremental to current backups may be performed with the JServer Manager Utility. For directions on how to perform an incremental backup to current, please refer to the *JServer Manager User's Guide*.

15.4 Backup Modes

Backup mode determines whether DBMaker can perform online incremental backups, and the type of data that will be backed up during an incremental backup. It also determines when DBMaker will free journal blocks that belong to inactive transactions for use by other transactions. DBMaker has three backup modes: NONBACKUP, BACKUP-DATA, and BACKUP-DATA-AND-BLOB.

Backup Mode	Tablespace Backup Mode	User Defined Tablespace (Data)	User Defined Tablespace (BLOB)	System Tablespace (Data and BLOB)
No Backup		No	No	No
Backup Data		Yes	No	Yes
Backup Data and BLOB	Backup BLOB Off	Yes	No	Yes
	Backup BLOB On	Yes	Yes	Yes

NONBACKUP Mode

NONBACKUP mode provides no protection for any data that was inserted or updated since the last full backup. In this mode, online incremental backups cannot be performed. A database can use the journal to fully recover from instance failure, but a media failure may result in loss of data. Journal blocks not in use by an active transaction can be reused immediately after a checkpoint, but once they are overwritten, the database may only be restored to the point in time of the last full backup.

BACKUP-DATA Mode

BACKUP-DATA mode provides protection for data (excluding BLOB data) that was inserted or updated since the last full backup. In this mode, a database administrator can perform an online incremental backup, but only non-BLOB data will be stored in the backup files. A database can use the journal to fully recover from instance failure, and can partially recover from media failure. Although the last backup can be used to restore the database to the point in time of the media failure, any changes to BLOB data will be lost. Journal blocks not in use by an active transaction can only be reused after a checkpoint has taken place *and* the journal file has been backed up.

BACKUP-DATA-AND-BLOB Mode

BACKUP-DATA-AND-BLOB mode provides protection for all data (including BLOB data) that was inserted or updated since the last full backup. In this mode, a database administrator can perform an online incremental backup, and all data will be stored in the backup files. A database can use the journal to fully recover from instance failure, and can fully recover from disk failure. The last backup may be used to completely restore the database to the point in time of the media failure, including all BLOB data. Journal blocks not in use by an active transaction can only be reused after a checkpoint has taken place and the journal file has been backed up.

Tablespace BLOB Backup Mode

DBMaker normally applies the backup mode setting to the entire database; this means all tablespaces in the database will be in the same backup mode. If the database is in BACKUP-DATA-AND-BLOB mode, DBMaker will record all changes to data (including BLOB data) in the journal. Recording BLOB data in the journal can quickly exhaust journal space, producing frequent backups and large backup file sizes.

This may be necessary if all BLOB data is critical, but in many cases, non-critical BLOB data may be backed up at the same time. Situations like this make it difficult for the database administrator to decide which backup mode you should choose. To prevent this type of situation from occurring, DBMaker allows the database administrator to modify the database backup mode for individual tablespaces when creating them.

To backup BLOB data in a specific tablespace, use the BACKUP BLOB ON option when executing the CREATE TABLESPACE command. To avoid backing up BLOB data in a specific tablespace, use the BACKUP BLOB OFF option when executing the CREATE TABLESPACE command.

The backup mode of each tablespace will then depend on the combination of database backup mode and tablespace backup mode as follows:

- If the database is running in BACKUP-DATA-AND-BLOB mode and a tablespace was created with the BACKUP BLOB ON option, DBMaker will backup BLOB data in that tablespace
- If the database is running in BACKUP-DATA-AND-BLOB mode and a tablespace was created with the BACKUP BLOB OFF option, DBMaker will not backup BLOB data in that tablespace
- If the database is running in BACKUP-DATA mode, DBMaker will not backup BLOB data regardless of whether a tablespace was created with the BACKUP BLOB ON or BACKUP BLOB OFF option

DBMaker uses the BACKUP BLOB ON mode by default for newly created tablespaces. All changes to BLOB data in that tablespace will be recorded in the journal file when the database is in BACKUP-DATA-AND-BLOB mode.

NOTE *A new full backup is required after a tablespace is changed to read-only because data files in read-only tablespaces are not backed up during differential backups.*

Backup Mode of File Objects

In addition to backing up regular and BLOB data in the database, users may choose to back up file objects. File objects are backed up only during automatic full backupsinitiated by the backup daemon. Users should first start the backup server, set the full backup schedule, and set the backup directory. For more information full backup settings refer to section 15.6, *Backup Server*.

There are two types of file objects: user file objects and system file objects. The database administrator can choose to back up system file objects, system and user file objects, or neither. The **dmconfig.ini** keyword **DB_BkFoM** specifies the Backup Mode of File Objects.

- **DB_BkFoM** = 0: Do not backup file objects
- **DB_BkFoM** = 1: Backup system file objects only
- **DB_BkFoM** = 2: Backup both system and user file objects

When backing up file objects (**DB_BkFoM** = 1, 2), the backup server copies all external files of file objects to the "FO" subdirectory under the directory specified by **DB_BkDir** keyword. The schedule follows the full backup schedule specified by **DB_FBkTm** and **DB_FBkTv**.

➲ **Example**

An excerpt from a **dmconfig.ini** file containing related keywords is shown below:

```
[MyDB]
DB_BkSvr = 1                         ; starts the backup server
DB_FBkTm = 01/05/01 00:00:00        ; begins from midnight at May 1, 2001.
DB_FBkTv = 1-00:00:00                ; interval is every one day.
DB_BkDir = /home/dbmaker/backup     ; backup directory
DB_BkFoM = 2                        ; backup both system and user file objects
```

Since the Backup Mode of File Objects is 2, the backup server will copy all external database file objects to the "*/home/dbmaker/backup/FO*" directory. If the **FO** subdirectory does not exist, the backup server will create it.

The files in FO subdirectory are renamed with a sequential number. For example, if the name of the original external file is "**/DBMaker/mydb/FO/ZZ000123.bmp**", the backup server would copy it to the **FO** subdirectory and rename it '**fo0000000344.bak**', meaning it is the 344th file object. The mapping between the source full file name and its new name is recorded in the file object mapping list file, **dmFoMap.his**. For more information about the file object mapping list file, refer to section 15.7, *Backup History Files*

The backup server will also move the previous version of file objects to the **FO** subdirectory under the old backup directory specified by **DB_BkOdr**.

Database administrators should consider that enabling file object backup requires more time for a full backup. The cost of complete full backup includes (1) copying the previous full backup if **DB_BkOdr** is set; (2) copying all database files; (3) copying all journal files; and (4) copying all file objects if **DB_BkFoM** is set. Also, ensure that there is enough disk space in the backup directory specified by **DB_BkDir** for all backup files to avoid backup failure.

Backup Mode of Stored Procedures

There are three types of stored procedures, SQL stored procedures, ESQL stored procedures and JAVA stored procedures. Because source codes are written into a database, SQL stored procedures are backed up as regular data during a full backup. In addition to backing up regular, BLOB data, and file objects in the database, users may choose to back up ESQL stored procedures or JAVA stored procedures. The two types of stored procedures are backed up only during automatic full backups initiated by the backup daemon. Users should first start the Backup Server, set the full backup schedule, and set the backup directory. For more information about settings of a full backup, please refer to section 15.6, *Backup Server*.

Users can specify the backup mode of stored procedures by setting **DB_BkSPm**.

- **DB_BkSPm** = 0: Do not backup ESQL stored procedures and JAVA stored procedures
- **DB_BkSPm** = 1: Back up ESQL stored procedures and JAVA stored procedures

➲ **Example**

An excerpt from a **dmconfig.ini** file containing related keywords is shown below:

```
[MyDB]
DB_BkSvr = 1                        ; starts the backup server
DB_FBkTm = 14/05/01 00:00:00       ; begins from midnight at May 1, 2014
DB_FBkTv = 1-00:00:00              ; interval is every one day
DB_BkDir = /home/dbmaker/backup    ; backup directory
```

```
DB_BkSPm = 1                           ; backup  ESQL stored procedures and JAVA
stored procedures
```

The default directory of the two types of stored procedures's backup files is the subdirectory named **SP** under the directory specified by the **DB_BkDir** keyword. If users have set the **DB_BkOdr** keyword in the **dmconfig.ini** file, in the process of doing a full backup, the previous backup sequences of the two types of stored procedures will be moved into the subdirectory also named **SP** under the old backup directory specified by **DB_BkOdr**, and then these backup sequences will be deleted from the default directory of stored procedures.

In the process of backing up the two types of stored procedures, Backup Server firstly generated a file named *dmSpBk.his*. Then the Backup Server will copy stored procedures' files, meanwhile, files that use filename extension .s0 and .b0 and are used to load the backed up stored procedures will be created. For ESQL stored procedures, the files that need to be backed up are source files and object files; for JAVA stored procedures, the files that need to be backed up only are object files. For convenience of rebuilding stored procedures, ESQL stored procedures' source files will be renamed in the format spnameowner.ec, for example, if the ESQL stored procedure's name is **y1**, and its owned by SYSADM, the copied source files will be renamed **y1SYSADM.ec**.

The file **dmSpBk.his** is used to record backup information. For more information about the stored procedures backup list file, please refer to section 15.7, *Backup History Files*.

Database administrators should consider that enabling file object backup requires more time for a full backup. The cost of complete full backup includes (1) copying the previous full backup if **DB_BkOdr** is set; (2) copying all database files; (3) copying all journal files; (4) copying all file objects if **DB_BkFoM** is set; (5) copying ESQL stored procedures and JAVA stored procedures if **DB_BkSPm** is set. Also, ensure that there is enough disk space in the backup directory specified by **DB_BkDir** for all backup files to avoid backup failure.

Compressing Backup Files

Database files can become very large and a large amount of free space is required to store backup files. DBMaker now supports compressing backup files. To enable or disable this feature, you can set the keyword **DB_BkZip** in **dmconfig.ini**, or change BKZIP with the system stored procedure **SetSystemOption** while the database is running.

- **DB_BkZip = 1:** Compresses the backup files
- **DB_BkZip = 0:** Backup files are not compressed (default)

The compression format is GZIP, so you can use any GZIP-compatible tool to read the compressed file.

NOTE *FO files are not compressed, even if you set **DB_BkZip** to enable compressing backup files.*

Setting the Backup Mode

DBMaker provides several different methods to set the backup mode. The method you choose depends on whether your database is online or offline, and whether you are more comfortable editing the configuration file directly, using the dmSQL command line utility, or using the JServer Manager graphical utility.

Modifying the backup mode of a database to provide a higher level of backup protection (i.e. from NONBACKUP to BACKUP-DATA mode, or from BACKUP-DATA to BACKUP-DATA-AND-BLOB mode) has an effect on journal usage. The journal begins recording changes to data that was previously not recorded before modifying the backup mode. As a result, it is necessary to perform a full backup or differential backup when you change the backup mode. This provides a starting point for the backup journal files to update during the restoration process.

No extra steps are required when modifying the backup mode of a database to provide a lower level of backup protection (i.e. from BACKUP-DATA or BACKUP-DATA-AND-BLOB mode to NONBACKUP mode) since the journal simply stops recording changes to data. DBMaker will use the previous full backup or differential backup as a starting point for the backup journal files to update during the restoration process.

However, some changes to data may be lost if the database is restored after changing to a lower level of backup protection.

The database administrator may change the backup mode of the database while offline using the **dmconfig.ini** file or JServer Manager. Since the backup mode affects journal usage, an offline full backup must be performed before starting the database with the new backup mode setting. Backup modes may be changed from one mode to another without restriction when offline, providing a full backup is made when going from a lower level of backup protection to a higher level. For more information on performing an offline full backup, see *Offline Full Backups* later in this chapter.

A database administrator can change the backup mode of a database online using dmSQL. Since the backup mode will affect journal usage, backup mode must be changed from a lower level of backup protection to a higher level (i.e., from NONBACKUP to BACKUP-DATA mode, or from BACKUP-DATA to BACKUP-DATA-AND-BLOB mode) between the start and finish of a full backup period. During runtime, users can't directly change backup mode from NONBACKUP to BACKUP-DATA-AND-BLOB mode, or from BACKUP-DATA-AND-BLOB to BACKUP-DATA mode. However, users can change backup mode from BACKUP-DATA-AND-BLOB to NONBACKUP mode at any time.

➲ **Example**

To use dmSQL to change the backup mode online:

```
dmSQL> BEGIN BACKUP;
dmSQL> SET DATA BACKUP ON;
dmSQL> END BACKUP DATAFILE;
dmSQL> END BACKUP JOURNAL;
```

or:

```
dmSQL> BEGIN BACKUP;
dmSQL> END BACKUP DATAFILE;
dmSQL> SET DATA BACKUP ON;
dmSQL> END BACKUP JOURNAL;
```

DBMaker does not allow the database to go from a higher level of backup protection to a lower level unless it is changed to NONBACKUP mode first. To change from BACKUP-DATA-AND-BLOB to BACKUP-DATA mode, first change to

NONBACKUP mode and then follow the rules above for changing from a lower level of backup protection to a higher level. The backup mode may be changed from BACKUP-DATA-AND-BLOB or BACKUP-DATA to NONBACKUP at any time; it does not need to be done between the start and finish of a full backup period.

USING THE DMCONFIG.INI CONFIGURATION FILE

If the database is offline, change the backup mode directly using the **DB_BMode** keyword in the **dmconfig.ini** file. The next time the database is started, the new backup mode will be used. If the database is online, changing the value of the **DB_BMode** keyword will have no effect until the database is shut down and restarted. Remember to perform an offline full backup if the backup mode is going to be changed from NONBACKUP to BACKUP-DATA or BACKUP-DATA-AND-BLOB mode or from BACKUP-DATA to BACKUP-DATA-AND-BLOB mode.

➲ **To set the backup mode using the dmconfig.ini file:**

1. Open the **dmconfig.ini** file on the database server using any **ASCII** text editor.
2. Locate the database configuration section for the database.
3. Change the value of the **DB_BMode** keyword to one of the following values:

```
0 - NONBACKUP mode

1 - BACKUP-DATA mode

2 - BACKUP-DATA-AND-BLOB mode
```

4. Restart the database to begin using the new backup mode.

If the **DB_BMode** keyword is not present in the database configuration section for the database, you will have to add the **DB_BMode** keyword to that database configuration section. You can add the keyword on a separate line anywhere between the start of the database configuration section and the start of the next configuration section; the order the keywords appear in is not important. If you do not specify a value for **DB_BMode**, the default value of 0 (NONBACKUP mode) will be used.

Using dmSQL

If the database is online and you are comfortable using the dmSQL command line utility, you can change the backup mode using the SQL SET command. You must execute this command during an online full or differential backup. The new backup mode will be enabled as soon as the command is executed.

➲ **To set the backup mode using the dmSQL command line utility:**

1. Connect to the database to change the backup mode using dmSQL.
2. Begin an online full backup using the **BEGIN BACKUP** command.
3. Change the backup mode during the full backup period by issuing one of the following **SET** commands at the dmSQL command prompt:

```
dmSQL> SET BACKUP OFF;

dmSQL> SET DATA BACKUP ON;

dmSQL> SET BLOB BACKUP ON;
```

4. Complete the online full backup.

The SET BACKUP OFF command corresponds to NONBACKUP mode, the SET DATA BACKUP ON corresponds to BACKUP-DATA mode, and the SET BLOB BACKUP ON command corresponds to BACKUP-DATA-AND-BLOB mode.

Using JServer Manager

If the database is offline, you can change the backup mode using the JServer Manager graphical utility. JServer Manager will automatically change the value of the **DB_BMode** keyword in the **dmconfig.ini** file. The next time you start the database, the new backup mode will be used. If the database is online, changing the value of the **DB_BMode** keyword will have no effect until the database is shut down and restarted. You must remember to perform an offline full backup if you are going from NONBACKUP to BACKUP-DATA or BACKUP-DATA-AND-BLOB mode or from BACKUP-DATA to BACKUP-DATA-AND-BLOB mode. For directions on how to set the backup mode offline using the JServer Manager graphical utility refer to the *JServer Manager User's Guide*.

15.5 Offline Full Backups

Offline full backups use operating system commands to back up the database. DBMaker provides this option, however, backup server is recommended. Offline full backups necessitate the database be shut down, furthermore, managing the backup sequence is a more complex process.

To perform an offline full backup, you must have read permission on the database files from the operating system, and write permission on the backup directory from the operating system. If you have to shut down the database first, you must have DBA, SYSDBA or SYSADM security privileges.

You can perform an offline full backup regardless of the backup mode; the database may be running in NON-BACKUP, BACKUP-DATA, or BACKUP-DATA-AND-BLOB mode. Using an offline full backup, you can restore the database to the point in time the database was shut down.

Note that offline full backup using JServer Manager does not back up file objects. File objects must be copied manually. Be sure to exactly replicate the file and directory structure if restoring a database from an offline full backup. For directions on how to perform an offline full backup using JServer Manager, refer to the *JServer Manager User's Guide*.

Offline Full Backup Using dmSQL

➲ **To perform an offline full backup using dmSQL:**

1. Notify all users that the database will be shut down at a specified time and ask them to disconnect before that time.
2. If the database is running, shut it down using the TERMINATE DB command. If there are any errors while shutting down the database, restart the database, correct the problem, and shut it down again.
3. Examine the **dmconfig.ini** file and list all relevant files and directories, including the file object directory, which require backup.

4. Use operating system commands or utilities to copy the database files, including data files, journal files, file objects, and the **dmconfig.ini** file, to the backup directory or backup device.

Offline Full Backup Using JServer Manager

If the database is offline, you can perform an offline full backup using the JServer Manager graphical utility. Note that offline full backup using JServer Manager does not back up file objects. File objects must be copied manually. Be sure to exactly replicate the file and directory structure if restoring a database from an offline full backup. For directions on how to perform an offline full backup using JServer Manager, refer to the *JServer Manager User's Guide*.

15.6 Backup Server

Although DBMaker provides methods for backing up databases manually, you must still remember to perform backups on a regular basis. To help, Backup Server provides a convenient way to fully automate online full, differential and incremental backups.

NOTE *Backup Server can only perform an online backup, since only after database startup， Backup Server can startup.*

Backup Server runs in the background and performs online full, differential and incremental backups on a set schedule, as journal files become full, or both. This flexibility is possible because Backup Server and the database server communicate to determine when a backup should occur, the type of incremental backup to perform, and which backup options to change. Backup Server starts at the same time as the database server, and continues running until you either stop it or shut down the database server.

When performing full backups, Backup Server will copy the last full backup from the backup directory to the old directory. Then, it will copy all database files including journal files and **dmconfig.ini** to the backup directory, over writing the previous full backup.

When performing differential backups, Backup Server copies only data files (DB and BB). Journal files are excluded because they change too frequently. During differential backups, only useful journal blocks are copied. Additionally, data files in read-only tablespaces are excluded from differential backups.

When performing incremental backups, Backup Server will copy necessary journal files to the backup directory.

There are several options used to configure Backup Server. These options control the filename format of the backup files, the location of the backup directory, the location of the old directory, the schedule Backup Server uses to perform backups, the interval and the maximum number of differential backups to retain after a full backup, the amount a journal file must fill before Backup Server performs an incremental backup, and the way Backup Server saves backup files.

Backup Server also allows backup-related configuration settings to be made during the run time with the dmSQL **SetSystemOption** stored procedure, that is to say, BKSVR, BKDIR, BKITV, DBKTV, BKTIM, BKFUL, BKFOM, BKZIP, BKCMP, BKRTS, BKCHK, FBKTM, FBKTV, DBKMX, BKODR, BKFRM can be changed during the run time with the **SetSystemOption** system stored procedure.

Starting Backup Server

Backup Server is a daemon and its life cycle is as long as the database server. Users do not have to explicitly start Backup Server after setting the **DB_BkSvr** keyword, since DBMaker will automatically start Backup Server while starting the database. Backup Server is disabled by default. Backup Server will only be started when the database is starting in muti-user mode.

Backup Server has two states: inactive and active. Users can control the state of backup server with **DB_BkSvr**. When **DB_BkSvr** is set to 0, the Backup Server is inactive. Backup Server will not respond to any backup request, namely the Backup Server will not perform any backup; when **DB_BKSvr** is set to 1, the Backup Server is active. Backup Server will response to a variety of backup requests, and then users can do any backup.

To activate the backup server, there are three methods: setting the value of the **DB_BkSvr** keyword to 1 in the **dmconfig.ini** file, changing BkSvr to 1 with *call* *setsystemoption('bksvr','1')* after the database is started and using **Run Time Setting** in Jserver Manager to change the backup setting when the database is running.

Before doing backup with Backup Server, users need to set some parameters to specify how to do a backup. For example, backup directory, compact backup mode and so on.

The following is how to set these parameters:

- Users can set related keywords in **dmconfig.ini** before starting the database. The next time users start the database, backup server will use these keywords to initialize associated parameters.
- If the database has been started, users can use **Run Time Setting** in Jserver Manager to alter values of parameters. Additionally, users can set parameters with *call SetSystemOption('option_name','value').* Please note that individual parameters only can be set with set syntax, such as set backup OFF; set data backup ON; set blob backup ON.

When Backup Server is activated, and the appropriate backup parameters are set in the **dmconfig.ini** configuration file, you can call the system stored procedure **SetSystemOption** to begin a backup. The stored procedure can be used by any client tool or user application.

➲ **The syntax to do online full, differential and incremental backup is:**

```
dmSQL> CALL SETSYSTEMOPTION('STARTBACKUP','1'); //do full backup
dmSQL> CALL SETSYSTEMOPTION('STARTBACKUP','2'); //do incremential backup
dmSQL> CALL SETSYSTEMOPTION('STARTBACKUP','3'); //do differential backup
```

USING DMCONFIG.INI TO START BACKUP SERVER

If the database is offline, you can enable Backup Server directly using the **DB_BkSvr** keyword in the **dmconfig.ini** file. The next time you start the database, Backup Server will also start. If the database is online, changing the value of the **DB_BkSvr** keyword found in the **dmconfig.ini** configuration file will have no effect until the database is shut down and restarted.

➲ **To start Backup Server using the dmconfig.ini file:**

1. Open the **dmconfig.ini** file on the database server using any ASCII text editor.

2. Locate the database configuration section for a database to enable Backup Server.
3. Ensure the backup mode of the database is either BACKUP-DATA or BACKUP-DATA-AND-BLOB mode. The database is in BACKUP-DATA mode if the value of **DB_BMode** is set to 1, and it is in BACKUP-DATA-AND-BLOB mode if the value of **DB_BMode** is set to 2.
4. Change the value of the **DB_BkSvr** keyword to 1 to enable Backup Server.
5. Restart the database to begin using Backup Server.

USING DMSQL TO START BACKUP SERVER

When a database is online, the Backup Server can be dynamically enabled using the dmSQL command line tool as shown below.

```
dmSQL> CALL SETSYSTEMOPTION('BKSVR','1');
```

Users can change BkSvr with *Call SetSystemOption('BkSvr', '1')*. To change BkSvr and write the value of **DB_BkSvr** in the **dmconfig.ini** configuration file at the same time, users can using *Call SetSystemOptionW('option','value')*.

When Backup Server is activated, and set the appropriate backup parameters in the **dmconfig.ini** configuration file, you can call the system stored procedure **SetSystemOption** to begin a backup, which stored procedure can be used by any client tools and user applications.

```
dmSQL> CALL SETSYSTEMOPTION('STARTBACKUP','1');   //do full backup
dmSQL> CALL SETSYSTEMOPTION('STARTBACKUP','2');   //do incremential backup
dmSQL> CALL SETSYSTEMOPTION('STARTBACKUP','3');   //do differential backup
```

The syntax to change an incremental backup interval is:

```
dmSQL> CALL SETSYSTEMOPTION('bkitv', 'Interval')
```

USING JSERVER MANAGER TO START BACKUP SERVER

Regardless of the online status of a database, you can enable Backup Server dynamically using the JServer Manager graphical utility. JServer Manager automatically changes the value of **DB_BkSvr** keyword in the **dmconfig.ini** configuration file. If the database is offline, the next time you start the database, Backup Server will also start. For directions on starting Backup Server while offline using JServer Manager, refer to the *JServer Manager User's Guide*.

Differential Backup Filename Format

This is the differential backup filename format:

DTimeStamp_DataFileName.dif(2)

D — Required differential backup identification

TimeStamp — Number of seconds since January 1, 1970 (00:00:00 GMT)

DataFileName — The data file's database name

.dif — File extension used for differential backup files. If no corresponding differential backup source file exists for a particular full backup, the file extension name must be **.dif2**.

Here is an example. A first differential backup is performed at 2009/12/01 14:11, then differential backup filenames are generated like this: D1259647860_DBNAME.BB.dif, D1259647860_DBNAME.DB.dif, D1259647860_DBNAME.SBB.dif and D1259647860_DBNAME.SDB.dif. The journal filename is D1259647860_DBNAME.JNL.

Incremental Backup Filename Format

Backup filename format is *<I><TimeStamp>**<_><DB_BkFrm>***, e.g., I1234567890_%2F%4N%4B.JNL. The total length of the filename cann't exceed 256 characters. The timestamp is a system 10 digits valid time numeric data, and the **<DB_BkFrm>** may include both text constants and format sequences (e.g., escape sequences), that represent special character strings.

An incremental backup file name must consist of at least three special character strings: the full backup id, the database name, and the backup identification number. Backup Server assigns a full backup ID when naming incremental files in a backup sequence. When restoring a database, DBMaker uses the full backup ID to correctly recreate the backup sequence. The database name correctly identifies the database to which an incremental backup file belongs. The backup identification number identifies the relative position of the incremental backup file in the backup sequence.

Format sequences have three parts: the escape character, the length value, and the format character. Valid format sequences are:

%[*x*]F — The full backup ID. The variable *x* may have values 1 through 4 where the values represent the following formats;

1: full backup id shown as YYYYMMDD, e.g., 20010917

2: full backup id shown as MMDD, e.g., 0917

3: full backup id shown as MMDDhhmm, e.g., 09171305

4: full backup id shown as DDhhmmss, e.g., 17130558

%[*n*]B — Backup identification number

%[*n*]N — Journal file belongs to this database

The escape character identifies the start of the format sequence, and is represented by the % symbol. If you want to include the % symbol as a text constant in the backup filename format, you must use two % symbols together (i.e. %%). A single digit or one of the valid format characters shown above must immediately follow the % symbol. If any other characters follow the % symbol the backup filename format is invalid, and DBMaker will return an error.

The length value *n* is an integer value between one and nine that determines the length of the character string generated by the format sequence. If the format sequence returns a string that can be represented in fewer characters than the length value provides then zeros will be appended to it. The database name has zeroes added to the right of the name, while all other values have zeroes added to the left. If the format sequence returns a string that requires more characters than the length value provides, it will be truncated. The database name is truncated from the right, while all other values are truncated from the left. The square brackets enclosing the length value indicate the length value is optional; do not include the square brackets when entering the format sequence. If you do not provide a value for the length, Backup Server will use the full length of the character string generated by the format sequence.

The format character identifies the type of special character string the format sequence will return. The format character must be F, B, or N; using any other character will result in an invalid backup filename format, and DBMaker will return an error. A valid format character that does not immediately follow either the escape character or the escape character and a single digit will be treated as a text constant.

Date and time values are taken from the system. These values will only be correct if the system date and time are correct. The value for the backup identification number is the ordinal position of the backup journal file in the backup sequence. DBMaker automatically provides this number for each journal file that is backed up by Backup Server.

DBMaker provides several different methods to set the backup filename format. The method you choose depends on whether you are more comfortable editing the configuration file directly or using the JServer Manager graphical utility.

USING DMCONFIG.INI TO SET BACKUP FILE NAME FORMAT

If the database is offline, you can set the backup filename format used by Backup Server directly using the **DB_BkFrm** keyword in the **dmconfig.ini** file. The next time you start the database, Backup Server will apply this backup filename format to all backup journal files. If the database is online, changing the value of the **DB_BkFrm** keyword will have no effect until the database is shut down and restarted.

➲ **To set the backup file format using the dmconfig.ini configuration file:**

1. Open the **dmconfig.ini** file on the database server using any ASCII text editor.
2. Locate the database configuration section for a database.
3. Change the value of the **DB_BkFrm** keyword to a string containing the format to use for the backup filename format.

   NOTE *The string may contain any valid format sequences and text constants, but the total length of the resulting filename must not exceed 256 characters.*

4. Restart the database to begin using the new backup filename format.

USING DMSQL TO SET BACKUP FILE NAME FORMAT

The stored procedure **SetSystemOption** can be used to change the backup filename format while the database is running. The general syntax for the command is:

```
CALL SETSYSTEMOPTION('bkfrm', 'name')
```

➲ **Example**

To change the backup filename format to I1234567890_%2F%4N%4B.JNL, enter the following line at the dmSQL command prompt.

```
dmSQL> CALL SETSYSTEMOPTION('bkfrm', 'I1234567890_%2F%4N%4B.JNL');
```

USING JSERVER MANAGER TO SET BACKUP FILE NAME FORMAT

The backup filename format can be set for offline or online databases using JServer Manager's graphical utility. JServer Manager will automatically change the value of the **DB_BkFrm** keyword in the **dmconfig.ini** file. The next time you start the database, Backup Server will apply this backup filename format to all backup journal files. For directions on how to set the backup file format using JServer Manager, refer to the *JServer Manager User's Guide*.

Backup Directory

The backup directory specifies where the Backup Server will place backup files. DBMaker supports single backup file path and multiple backup file paths for users. Backup server will automatically create BkDir. However, you should choose one or more backup directory on a different disk than the database files to prevent the loss of both the database and the backup files in the event of a media error.

The backup directory is specified by the **DB_BkDir** keyword in the **dmconfig.ini** file. The value of the **DB_BkDir** keyword may contain either a full or a relative path to the backup directory. If you do not specify a backup directory, the Backup Server will automatically create a default backup directory named **backup** under the database directory. The database directory is specified by the **DB_DbDir** keyword in the **dmconfig.ini** file. The total length of the backup directory path must not exceed 256 characters in length.

However, if the database is running on replication mode (master or slave), BKDIR should be single path. If you set BKDIR multi-path, the only first is used and path size is ignored. Furthermore, it is not a good idea to allow the Backup Server to create and use the default backup directory if you have more than one database in the same directory. In this case, the backup history information from one database may overwrite or append to the backup history information from another database, rendering one or both of the backups unusable. To avoid this type of problem you can put each database in a different database directory, or explicitly specify a backup directory for each database. Placing each database in a different database directory is the preferred method, since this allows you to see exactly which files belong to which database.

DBMaker provides several different methods to set the backup directory. The method you choose depends on whether your database is online or offline, and whether you are more comfortable editing the configuration file directly or using the JServer Manager graphical utility.

USING DMCONFIG.INI TO SET BACKUP DIRECTORY

If the database is offline, you can set the backup directory used by Backup Server directly using the **DB_BkDir** keyword in the **dmconfig.ini** file. The next time you start the database, Backup Server will use this directory as the backup directory. If the database is online, changing the value of the **DB_BkDir** keyword will have no effect until the database is shut down and restarted.

➲ **To set the backup directory using the dmconfig.ini configuration file:**

1. Open the **dmconfig.ini** file on the database server using any ASCII text editor.
2. Locate the database configuration section for a database.
3. Change the value of the **DB_BkDir** keyword to a string containing the name of an existing directory to set the backup directory.
4. Restart the database to begin using the new backup directory.

USING DMSQL TO SET BACKUP DIRECTORY

The stored procedure **SetSystemOption** can be used to change the backup directory while the database is running. The general syntax for the command is:

```
CALL SETSYSTEMOPTION('bkdir', 'path')
```

Path is the full path of the new backup directory. The length of the string in *path* should not exceed 256 characters.

➲ **Example**

To change the directory path to *E:\storage\database\backup\WebDB*, enter the following line at the dmSQL command prompt.

```
dmSQL> CALL SETSYSTEMOPTION('bkdir', 'E:\storage\database\backup\WebDB');
```

USING JSERVER MANAGER TO SET BACKUP DIRECTORY

If the database is offline, you can set the offline backup directory used by Backup Server using the JServer Manager graphical utility. JServer Manager will automatically change the value of the **DB_BkDir** keyword in the **dmconfig.ini** file. The next time you start the database, Backup Server will use this directory as the backup directory. If the database is online, JServer Manager can change the backup directory immediately with Run Time Setting or delay the change until the next time you restart the database when the database making an interactively backup. In either case, JServer Manager will also make a copy of the backup history file in the new backup directory. For directions on how to set the backup directory using JServer Manager, refer to the *JServer Manager User's Guide*.

Setting Multiple Backup Paths

DBMaker also supports multiple backup file paths for users. This function is useful when a user tries to save to a backup path, but the backup path does not provide enough space for the backup to be completed. If the multiple backup option is set DBMaker will then shunt the remaining data to be backed up to secondary backup locations so that the backup can be properly performed. Users are able to use multiple backup paths on full, differential or incremental backups. DBMaker has the following constraints when backing up information using multiple backup paths:

- When a database system attempts to backup files, it will try to store files in the paths one by one for each file. For example, when storing a file to backup directory 1 and the directory does not have enough space to store the file, then the file is shunted to backup directory 2, and so on. If all backup directories are full an error message will be returned.
- Only one backup directory can be used to backup files on the slave sites
- FOs must backup in the first backup directory
- The maximum number of backup paths is 32

➲ **Example**

When setting multiple backup paths DBMaker conforms to the following structure:

```
DB_BkDir = <BKDIR 1> <SIZE 1> < BKDIR 2> <SIZE 2> < BKDIR 3> <SIZE 3>…

< BKDIR n > : the n's backup path
< SIZE n > : the size of the n's backup path
```

So when setting multiple backup paths for the database **DB1** you need to set the paths in **DB_BkDir**.

```
DB_BkDir = /home/usr/dbmaker/bk 5000 /home2/backup 1000
```

When the available space in */home/usr/dbmaker/bk* is full the database will backup at */home2/backup*.

Setting the Old Directory

The old directory is one directory or a group of directories (up to 32), and it is used to saving a backup sequence which is one just before the last one You should choose it on a different disk than the database files to prevent the loss of both the database and the backup files in the event of a media error.

The old directory is specified by the **DB_BkOdr** keyword in the **dmconfig.ini** file. If you do not specify it, the Backup Server will discard the previous backup sequence.

USING DMCONFIG.INI TO SET OLD DIRECTORY

You can set the old directory used by Backup Server directly using the **DB_BkOdr** keyword in the **dmconfig.ini** file. The next time you start the database, Backup Server will use this directory as the old directory. If the database is online, changing the value of the **DB_BkOdr** keyword will have no effect until the database is shut down and restarted.

USING DMSQL TO SET OLD DIRECTORY

The stored procedure **SetSystemOption** can be used to change the old backup directory while the database is running. The general syntax for the command is:

```
CALL SETSYSTEMOPTION('bkodr', 'path')
```

Path is the full path of the new old backup directory. The length of the string in *path* should not exceed 256 characters.

➲ **Example**

To change the old directory path to *E:\storage\database\backup\WebDB*, enter the following line at the dmSQL command prompt.

```
dmSQL> CALL SETSYSTEMOPTION('bkodr', 'E:\storage\database\backup\WebDB');
```

USING JSERVER MANAGER TO SET OLD DIRECTORY

If the database is offline, you can set the location for the previous backup using the JServer Manager graphical utility. JServer Manager will automatically change the value of the **DB_BkOdr** keyword in the **dmconfig.ini** file. The next time you start the database, Backup Server will use this directory as the backup directory. If the database is online, JServer Manager can change the old backup directory immediately or delay the change until the next time you restart the database. For directions on setting the old backup directory while offline using JServer Manager, refer to the *JServer Manager User's Guide*.

Differential Backup Settings

The differential backup schedule specifies when Backup Server performs online differential backups. The schedule includes initial backup time and interval time.

Initial backup time is the date and time when Backup Server will perform the first differential backup. Interval time specifies the wait time between subsequent differential backups.

The initial full backup time is specified by the **DB_FBkTm** keyword found in the **dmconfig.ini** file. The value of the **DB_FBkTm** keyword must be a date and time in this format: YY/MM/DD HH:MM:SS. There is no default value for initial backup time; however when enabling Backup Server with JServer Manager a default value is added to the **dmconfig.ini** file.

Interval time is specified by the **DB_DBkTv** keyword found in the **dmconfig.ini** file. The first differential backup is performed at **DB_FBkTm** + **DB_DBkTv.** The value of the **DB_DBkTv** keyword must be time interval in the this format: D-HH:MM:SS. There is no default value for interval time; however, when enabling Backup Server with JServer Manager a default value of 1-00:00:00 is added to the **dmconfig.ini** file.

Lastly, the keyword **DB_DbKmx** specifies the maximum number of differential backups to retain after a full backup. Backup Server removes the oldest differential backup when the number of differential backups, after a full backup, exceeds **DB_DbKmx**. The system stored procedure **SetSystemOption** can be used to change DBKMX while the database is running. Additional, the keyword **DB_BkChk** specifies whether check database before differential backup, the system stored procedure **SetSystemOption** can be used to change BKCHK while the database is running.

USING DMCONFIG.INI TO CHANGE DIFFERENTIAL BACKUP SETTINGS

When a database is offline, its backup schedule can be set using the **DB_FBkTm** and **DB_DBkTv** keywords found in the **dmconfig.ini** configuration file. The next time the database is started; Backup Server uses these settings for the differential backup schedule. If the database is online, changing the value of the **DB_FBkTm** and **DB_DBkTv** keywords has no effect until the database is shut down and restarted.

➲ **To set the backup schedule using the dmconfig.ini file:**

1. Open the **dmconfig.ini** file on the database server using any ASCII text editor.

2. Locate the database configuration section of the desired database.
3. Change the value of the **DB_FBkTm** keyword to a date and time value using this format: YY/MM/DD HH:MM:SS.
4. Change the value of the **DB_DBkTv** keyword to a time interval value using this format: ndays-HH:MM:SS.
5. Restart the database to activate the new backup schedule.

USING DMSQL TO CHANGE DIFFERENTIAL BACKUP SETTINGS

The stored procedure **SetSystemOption** can be used to activate the Backup Server. The command is:

```
dmSQL> CALL SETSYSTEMOPTION('BKSVR','1');
```

When Backup Server is activated, initiates a differential backup by calling the system stored procedure **SetSystemOption**:

```
dmSQL> CALL SETSYSTEMOPTION('STARTBACKUP','3');
```

The syntax to change the differential backup interval is:

```
CALL SETSYSTEMOPTION('dbktv','Interval')
```

USING JSERVER MANAGER TO CHANGE DIFFERENTIAL BACKUP SETTINGS

If the database is offline, the differential backup schedule can be set using JServer Manager graphical utility. JServer Manager automatically changes the value of the **DB_FBkTm** and **DB_DBkTv** keywords found in the **dmconfig.ini** file. The next time the database is started. Backup Server uses these settings as the new differential backup schedule. If the database is online, JServer Manager can immediately change the backup schedule or optionally delay changes until the next time the database is restarted. For more details please refer to the *JServer Manager User's Guide*.

Incremental Backup Settings

The incremental backup schedule specifies the times when Backup Server will perform an online incremental backup. The schedule is composed of two parts: the initial backup time and the interval time. The initial backup time determines the date and time Backup Server will perform the first incremental backup, and the interval time determines the length of time to wait between subsequent incremental backups.

You can combine the incremental backup schedule with the journal trigger value to backup your database both on a regular schedule and when journal files fill to a specified percentage. If you do not specify an incremental backup schedule, Backup Server will not backup the database on a regular schedule.

The initial backup time is specified by the **DB_BkTim** keyword in the **dmconfig.ini** file. You must enter the value of the **DB_BkTim** keyword as a date and time in the format YY/MM/DD HH:MM:SS. There is no default value for the initial backup time.

The interval time is specified by the **DB_BkItv** keyword in the **dmconfig.ini** file. You must enter the value of the **DB_BkItv** keyword as a time interval in the format D-HH:MM:SS. There is no default value for the interval time. However, if you use JServer Manager to enable Backup Server, JServer Manager will provide a default value of 1-00:00:00 for you and write this value into the **dmconfig.ini** file.

DBMaker provides several different methods to set the incremental backup schedule. The method you choose depends on whether your database is online or offline, and whether you are more comfortable editing the configuration file directly or using the JServer Manager graphical utility.

USING DMCONFIG.INI TO CHANGE INCREMENTAL BACKUP SETTINGS

If the database is offline, you can set the backup schedule used by Backup Server directly using the **DB_BkTim** and **DB_BkItv** keywords in the **dmconfig.ini** configuration file. The next time you start the database, Backup Server will use these settings for the incremental backup schedule. If the database is online, changing the value of the **DB_BkTim** and **DB_BkItv** keywords will have no effect until the database is shut down and restarted.

➲ **To set the backup schedule using the dmconfig.ini file:**

1. Open the **dmconfig.ini** file on the database server using any ASCII text editor.

2. Locate the database configuration section for a database to change the backup schedule.
3. Change the value of the **DB_BkTim** keyword to a date and time using the YY/MM/DD HH:MM:SS value format.
4. Change the value of the **DB_BkItv** keyword to a time interval using the DDDDD-HH:MM:SS value format.
5. Restart the database to begin using the new backup schedule.

USING DMSQL TO CHANGE INCREMENTAL BACKUP SETTINGS

The stored procedure **SetSystemOption** can be used to change the incremental backup start time and interval while the database is running. The general syntax to change the incremental backup start time is:

```
CALL SETSYSTEMOPTION('bktim','StartTime')
```

The general syntax to change the incremental backup interval is:

```
CALL SETSYSTEMOPTION('bkitv','Interval')
```

StartTime is the time to start the first incremental backup, and has the format YY:MM:DD HH:MM:SS. *Interval* is the time interval that incremental backups occur, and has the format D-HH:MM:SS.

When Backup Server is activated, call the system stored procedure **SetSystemOption** to initiate an incremental backup.

```
dmSQL> CALL SETSYSTEMOPTION('STARTBACKUP','2');
```

➲ **Example**

To set the incremental backup interval to 1 hour, enter the following line at the dmSQL command prompt.

```
dmSQL> CALL SETSYSTEMOPTION('bkitv','0-1:00:00');
```

USING JSERVER MANAGER TO CHANGE INCREMENTAL BACKUP SETTINGS

If the database is offline, you can set the incremental backup schedule used by Backup Server using the JServer Manager graphical utility. JServer Manager will automatically change the value of the **DB_BkTim** and **DB_BkItv** keywords in the **dmconfig.ini** file. The next time you start the database, Backup Server will use these settings as the new incremental backup schedule. If the database is online, JServer Manager can change the backup schedule immediately or delay the change until the next time you restart the database. For directions on how to set the incremental backup schedule using JServer Manager, refer to the *JServer Manager User's Guide*.

Journal Trigger Value Settings

The journal trigger value specifies the percentage a journal file must fill before Backup Server will perform an online incremental backup. You can combine the journal trigger value with the backup schedule to backup your database on a regular schedule and when journal files fill to the specified percentage.

The journal trigger value is specified by the **DB_BkFul** keyword in the **dmconfig.ini** file. The value of the **DB_BkFul** keyword may be an integer value in the range 50-100, or zero. Values between 50 and 100 represent the percentage a journal file must fill before Backup Server performs a backup. A value of zero causes Backup Server to perform a backup whenever a journal file fills completely. Setting the value to 0 is effectively the same as setting it to a value of 100, since both will cause Backup Server to perform a backup whenever a journal file fills completely (100% full). If you do not specify a value for the journal trigger value, Backup Server will use the default value of 90.

DBMaker provides several different methods to set the journal trigger value. The method you choose depends on whether your database is online or offline, and whether you are more comfortable editing the configuration file directly or using the JServer Manager graphical utility.

USING DMCONFIG.INI TO CHANGE JOURNAL TRIGGER VALUE

If the database is offline, you can set the journal trigger value used by Backup Server directly using the **DB_BkFul** keyword in the **dmconfig.ini** file. The next time you start the database, Backup Server will use this setting for the journal trigger value. If the database is online, changing the value of the **DB_BkFul** keyword will have no effect until the database is shut down and restarted.

➲ **To set the journal trigger value using the dmconfig.ini file:**

1. Open the **dmconfig.ini** file on the database server using any ASCII text editor.
2. Locate the database configuration section for a database to change the journal trigger value.
3. Change the value of the **DB_BkFul** keyword to an integer value between 50 and 100, or set it to zero.
4. Restart the database to begin using the new journal trigger value.

USING DMSQL TO CHANGE JOURNAL TRIGGER VALUE

The stored procedure **SetSystemOption** can be used to change the journal trigger value while the database is running. The general syntax to change the journal trigger value is:

```
CALL SETSYSTEMOPTION('bkful','n')
```

Where *n* is either 0 or 50-100. Setting *n* to 0 will trigger the backup server whenever a journal file is full. Setting *n* between 50 and 100 specifies the percentage a journal file fills to before the backup server activates.

➲ **Example**

To set the journal trigger value to 75 percent, enter the following line at the dmSQL command prompt.

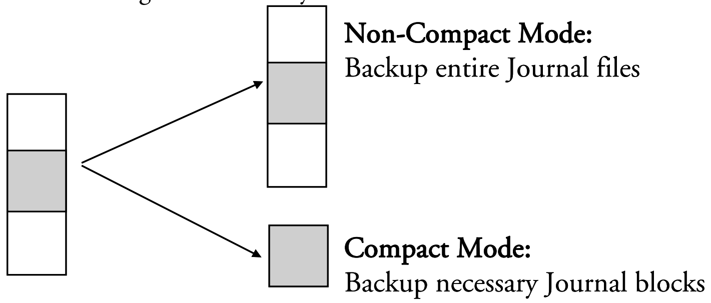
```
dmSQL> CALL SETSYSTEMOPTION('bkful','75');
```

USING JSERVER MANAGER TO CHANGE JOURNAL TRIGGER VALUE

If the database is offline, you can set the journal trigger value used by Backup Server using the JServer Manager graphical utility. JServer Manager will automatically change the value of the **DB_BkFul** keyword in the **dmconfig.ini** file. The next time you start the database, Backup Server will use this setting as the new journal trigger value. If the database is online, JServer Manager can change the journal trigger value immediately or delay the change until the next time you restart the database. For directions on how to set the journal trigger value using JServer Manager, refer to the *JServer Manager User's Guide*.

Compact Backup Mode Settings

Compact backup mode specifies whether Backup Server will backup entire journal files or only full journal blocks when it performs an online incremental or differential backup. This is possible since not every journal block contains data needed to restore a database, so Backup Server will only backup the necessary journal blocks when it performs a backup. This allows you to save storage space on your backup device, but it also means restoring a database may take more time.



The compact backup mode setting is specified by the **DB_BkCmp** keyword in the **dmconfig.ini** configuration file. The value of the **DB_BkCmp** keyword may be zero or one. Setting the value to one enables compact backup mode, and setting it to zero disables compact backup mode. If you do not specify a value for the compact backup mode, Backup Server will use the default value of one (enabled).

DBMaker provides several different methods to set the compact backup mode. The method you choose depends on whether your database is online or offline, and whether you are more comfortable editing the configuration file directly or using the JServer Manager graphical utility.

USING DMCONFIG.INI TO SET COMPACT BACKUP MODE

If the database is offline, you can set the compact backup mode setting used by Backup Server directly using the **DB_BkCmp** keyword in the **dmconfig.ini** file. The next time you start the database, Backup Server will use this setting for the compact backup mode. If the database is online, changing the value of the **DB_BkCmp** keyword will have no effect until the database is shut down and restarted.

➲ **To set the Compact Backup Mode using the dmconfig.ini configuration file:**

1. Open the **dmconfig.ini** file on the database server using any ASCII text editor
2. Locate the database configuration section for a database to change the compact backup mode.
3. Change the value of the **DB_BkCmp** keyword to one to enable compact backup mode, or zero to disable compact backup mode
4. Restart the database to begin using the new journal trigger value

USING DMSQL TO SET COMPACT BACKUP MODE

The stored procedure **SetSystemOption** can be used to change the compact backup mode while the database is running. A successful backup does not require every journal block in a journal file. If this keyword **DB_BkCmp** is set to 1 the backup server will only back up the journal blocks that require backup. The command is:

```
dmSQL> CALL SETSYSTEMOPTION('bkcmp','1');
```

USING JSERVER MANAGER TO SET COMPACT BACKUP MODE

If the database is offline, you can set the compact backup mode setting used by Backup Server using the JServer Manager graphical utility. JServer Manager will automatically change the value of the **DB_BkCmp** keyword in the **dmconfig.ini** file. The next time you start the database, Backup Server will use this setting as the new compact backup mode setting. If the database is online, JServer Manager can change compact backup mode setting immediately or delay the change until the next time you restart the database. For directions on how to set the Compact Backup Mode using JServer Manager, refer to the *JServer Manager User's Guide*.

Full Backup Schedule

The full backup schedule specifies the times when Backup Server will perform an online full backup. The schedule is composed of two parts: the initial backup time and the interval time. The initial backup time determines the date and time Backup Server will perform the first full backup, and the interval time determines the length of time to wait between subsequent full backups.

You can combine full and differential backup schedules with an incremental to backup your database. If you do not specify a full backup schedule, Backup Server will not perform full backups on a regular schedule.

The initial backup time is specified by the **DB_FBkTm** keyword in the **dmconfig.ini** file. You must enter the value of the **DB_FBkTm** keyword as a date and time in the format **YY/MM/DD HH:MM:SS**. There is no default value for the initial backup time.

The interval time is specified by the **DB_FBkTv** keyword in the **dmconfig.ini** file. Enter the value of the **DB_FBkTv** keyword as a time interval in the format **D-HH:MM:SS**. There is no default value for the interval time.

Lastly, the keyword **DB_BkChk** specifies whether check database before full backup and differential backup and the keyword **DB_BkRTs** specifies whether the backup server includes the read-only tablespace files when performing a full-backup. To enable or disable the two features, you can set the keyword **DB_BkChk** and **DB_BkRTs** in **dmconfig.ini**, or change BKCHK and BKRTS with the system stored procedure **SetSystemOption** while the database is running.

USING DMCONFIG.INI TO SET FULL BACKUP SCHEDULE

If the database is offline, you can set the full backup schedule used by Backup Server directly using the **DB_FBkTm** and **DB_FBkTv** keywords in the **dmconfig.ini** file. The next time you start the database, Backup Server will use these settings for the full backup schedule. If the database is online, changing the value of the **DB_FBkTm** and **DB_FBkTv** keywords will have no effect until the database is shut down and restarted.

➲ **To set the full backup schedule using the dmconfig.ini configuration file:**

1. Open the **dmconfig.ini** file on the database server using any ASCII text editor
2. Locate the database configuration section for a database to change the journal trigger value
3. Set the configuration parameter **DB_FBkTm** to a value of the format YY/MM/DD HH:MM:SS, and **DB_FBkTv** to a value of the format D-HH:MM:SS

4. Restart the database to begin using the new full backup schedule

USING DMSQL TO SET FULL BACKUP SCHEDULE

The stored procedure **SetSystemOption** can be used to change the full backup start time and interval while the database is running. The general syntax to change the full backup start time is:

```
CALL SETSYSTEMOPTION('fbktm','StartTime')
```

The general syntax to change the full backup interval is:

```
CALL SETSYSTEMOPTION('fbktv','Interval')
```

StartTime is the time to start the first full backup, and has the format YY:MM:DD HH:MM:SS. *Interval* is the time interval that full backups occur, and has the format D-HH:MM:SS.

When Backup Server is activated, call the system stored procedure **SetSystemOption** to initiate an full backup.

```
dmSQL> CALL SETSYSTEMOPTION('STARTBACKUP','1');
```

➲ **Example**

To set the full backup interval to 1 hour, enter the following line at the dmSQL command prompt.

```
dmSQL> CALL SETSYSTEMOPTION('fbktv','0-1:00:00');
```

USING JSERVER MANAGER TO SET FULL BACKUP SCHEDULE

You can set the full backup schedule with JServer Manager by using the start database setup utility. JServer Manager will automatically change the value of the **DB_FBkTm** and **DB_FBkTv** keywords in the **dmconfig.ini** file. The next time you start the database, Backup Server will use this setting as the new full backup schedule. For directions on how to set the full backup schedule using JServer Manager, refer to the *JServer Manager User's Guide*.

Backup Mode of File Objects

The Backup Mode of File Objects lets the database administrator decide whether Backup Server will back up file objects during a full backup. It is also possible to specify Backup Server to back up just system file objects or system and user file objects.

It is possible to set the Backup Mode of File Objects in a number of ways. The configuration keyword **DB_BkFoM** determines the setting during database startup, but it may also be modified during runtime with dmSQL or the JServer Manager utility.

The backup server will move all files from the previous backup to the old backup directory specified by **DB_BkOdr**.

Starting file object backup will cause the database to require more time to complete a full backup, depending on how many file objects are in the database. The total cost of a complete full backup includes (1) copying the previous full backup if **DB_BkOdr** is set; (2) copying all database files; (3) copying all journal files; and (4) copying all file objects if **DB_BkFoM** is set. Be sure that enough disk space is available in the backup directory specified by **DB_BkDir** (and **DB_BkOdr** if applicable) for all mentioned backup files to avoid backup failure.

File objects are copied into an FO directory that is created in the backup directory at the time a full backup is performed. File objects are renamed sequentially when they are copied to the directory for backed-up file objects. The files in the **/FO** subdirectory are renamed starting with the letters FO followed by a ten digit serial number. All backup file objects are appended with the file extension .BAK. The mapping between the source file name and path and the backup file name is recorded in the file object mapping file **dmFoMap.his**.

BACKUP FILE OBJECT MAPPING FILE

The file object mapping file **dmFoMap.his** is created in the "*DB_BkDir/FO*" directory. It is a pure ASCII text file that records the original external file name and backup file name. The format looks like:

```
Database Name: DBSAMPLE5
Begin Backup FO Time: 2013/04/12 09:21:32
FO Backup Directory: C:\DBMaker\5.4\SAMPLES\DATABASE\backup\FO\
[Mapping List]
s, fo0000000000.bak, "C:\DBMaker\5.4\SAMPLES\DATABASE\backup\FO\ZZ000001.bmp"
```

```
u, fo0000000001.bak, "C:\DBMaker\5.4\SAMPLES\DATABASE\backup\FO\image.jpg"
...
s, fo0000002345.bak, "C:\DBMaker\5.4\SAMPLES\DATABASE\backup\FO\ZZ00AB32.txt"
```

The content before "[Mapping List]" is only a description for user reference. Each line after "[Mapping List]" represents a record that shows the file object type (s = system file object, u = user file object), the new file in */fo* subdirectory and its original file name and path. This mapping file is necessary for restoration of file objects.

USING DMCONFIG.INI TO SET BACKUP MODE OF FILE OBJECTS

The configuration file keyword **DB_BkFoM** determines the backup mode of file objects:

- **DB_BkFoM** = 0: Do not back up file objects
- **DB_BkFoM** = 1: Back up system file objects only
- **DB_BkFoM** = 2: Back up both system and user file objects

If **DB_BkFoM** = 1 or 2, the backup server will copy all file objects to the */fo* subdirectory under the backup directory. The schedule follows the full backup schedule.

➲ **Example**

An entry in a **dmconfig.ini** file for specifying the file object backup parameters.

```
[MyDB]
DB_BkSvr = 1                     ; starts the backup server
DB_FBkTm = 01/05/01 00:00:00    ; begins at midnight, May 1, 2001
DB_FBkTv = 1-00:00:00            ; interval is once every day
DB_BkDir = /home/dbmaker/backup    ; backup directory
DB_BkFoM = 2                    ; backup both system and user file objects
```

Since the backup mode is 2, the backup server will copy all external files (user file objects) and system file objects to the ***/home/dbmaker/backup/FO*** directory. If the **FO** subdirectory does not exist, the Backup Server will create it.

USING DMSQL TO SET BACKUP MODE OF FILE OBJECTS

The stored procedure **SetSystemOption** can be used to change the backup mode of file objects while the database is running. The general syntax to change the Backup Mode of File Objects is:

```
CALL SETSYSTEMOPTION('bkfom','n')
```

Where *n* is 0, 1, or 2. Setting *n* to 0 will turn the Backup Mode of File Objects to off. Setting *n* to 1 configures backup server to back up all system file objects during a full backup. Setting *n* to 2 configures backup server to back up all system and user file objects during a full backup.

➲ **Example**

To configure Backup Server to perform a full backup on all user and system file objects, enter the following line at the dmSQL command prompt.

```
dmSQL> CALL SETSYSTEMOPTION('bkfom','2');
```

USING JSERVER MANAGER TO SET BACKUP MODE OF FILE OBJECTS

The settings under the **Backup File Object Mode** effect how file objects are copied during the full backup process. Selecting **Do Not Backup File Objects** disables file backup during the full backup process. Selecting **Backup System File Objects Only** backs up system file objects during automatic full backups. Selecting **Backup System and User File Objects** will backs up both system file objects and user file objects copied to the backup directory during automatic full backups. For directions on how to set the Backup Mode of File Objects during database startup or with the Run Time Settings dialog in JServer Manager, refer to the *JServer Manager User's Guide*.

Backup Mode of Stored Procedures

The Backup Mode of stored procedures lets the database administrator decide whether Backup Server will back up ESQL stored procedures and JAVA stored procedures during a full backup.

It is possible to set the Backup Mode of stored procedures in a number of ways. The configuration keyword **DB_BkSPm** determines the setting during database startup, but it may also be modified during runtime with dmSQL or the JServer Manager utility.

The backup server will move the previous backup of stored procedures to the old backup directory specified by **DB_BkOdr**.

Starting stored procedures backup will cause the database to require more time to complete a full backup, depending on how many file objects are in the database. The total cost of a complete full backup includes (1) copying the previous full backup if **DB_BkOdr** is set; (2) copying all database files; (3) copying all journal files; (4) copying all file objects if **DB_BkFoM** is set; (5) copying ESQL stored procedures and JAVA stored procedures if **DB_BkSPm** is set. Also, ensure that there is enough disk space in the backup directory specified by **DB_BkDir** for all backup files to avoid backup failure.

Stored procedures are copied into the subdirectory **SP** that is created in the backup directory at the time a full backup is performed. Stored procedures are renamed when they are copied to the directory for backed-up stored procedures. The backup information about copied stored prcedures is recorded in the file **dmSpBk.his**.

BACKUP STORED PROCEDURE INFORMATION FILE

The backup information is composed of three parts, database name, backup time and backup list used to record which rows have been backed up. The backup information file **dmSpBk.his** is created in the subdirectory **SP** under the directory specified by **DB_BkDir**. It is a pure ASCII text file that records backup information about the copied stored procedure. The format looks like:

```
Database Name: MYDB
Begin Backup SP time: 2014/06/20 09:13:06
[Backup List]
ESQLSP, SYSADM, A4, A4SYSADM.dll, A4SYSADM.ec
....
JAR, "", "", employee.jar, ""
```

The content before "[Backup List]" is only a description for user reference. Each line after "[Backup List]" represents a record that shows stored procedures' type, stored procedure's owner, stored procedures' name, the name of stored procedures' object files and the name of its corresponding source files. This backup information file is necessary for restoration of stored procedures.

USING DMCONFIG.INI TO SET BACKUP MODE OF STORED PROCEDURES

The file keyword **DB_BkSPm** determines the backup mode of stored procedures:

- **DB_BkSPm** = 0: Do not back up ESQL stored procedures and JAVA stored procedures
- **DB_BkSPm** = 1: Back up ESQL stored procedures and JAVA stored procedures

➲ **Example**

An entry in a **dmconfig.ini** file for specifying the stored procedure backup parameters.

```
[MyDB]
DB_BkSvr = 1                       ; starts the backup server
DB_FBkTm = 14/05/01 00:00:00      ; begins at midnight, May 1, 2014
DB_FBkTv = 1-00:00:00              ; interval is once every day
DB_BkDir = /home/dbmaker/backup   ; backup directory
DB_BkSPm = 1                       ; backup ESQL stored procedures and JAVA
stored procedures
```

Since the value of **DB_BkSPm** is 1, the Backup Server will backup ESQL stored procedures and JAVA stored procedures to the subdirectory **SP** under the directory specified by **DB_BkDir**. If the subdirectory **SP** does not exist, the Backup Server will create it.

USING DMSQL TO SET BACKUP MODE OF STORED PROCEDURES

The stored procedure **SetSystemOption** can be used to change the backup mode of stored procedures while the database is running. The general syntax to change the Backup Mode of stored procedures is:

```
CALL SETSYSTEMOPTION('BKSPM','n')
```

Users can assign 0 or 1 to *n*. If assign 0 to *n*, the Backup Server will not back up ESQL stored procedures and JAVA stored procedures; if assign 1 to *n*, the Backup

Server will back up ESQL stored procedures and JAVA stored procedures during a full backup.

➲ **Example**

To configure Backup Server to perform a full backup on all stored procedures, enter the following line at the dmSQL command prompt.

```
dmSQL> CALL SETSYSTEMOPTION('BKSPM','1');
```

USING JSERVER MANAGER TO SET BACKUP MODE OF STORED PROCEDURES

Regardless of the online status of a database, you can enable backing up ESQL stored procedures and JAVA stored procedures dynamic using the JServer Manager graphical utility. JServer Manager automatically changes the value of **DB_BkSPm** keyword in the **dmconfig.ini** configuration file. If the database is offline, the next time you start the database, this new setting will take effect. For directions on how to set the Backup Mode of stored procedures during database startup or with the Run Time Settings dialog in JServer Manager, please refer to the *JServer Manager User's Guide*.

Inactivate Backup Server

DBMaker will automatically start Backup Server while starting the database. Backup Server is disabled by default. You can control the state of backup server with **DB_BkSvr**. When **DB_BkSvr** is set to 0, the backup server is inactive; when **DB_BkSvr** is set to 1, the backup server is active. When you no longer want the backup server is active, you can set the value of the **DB_BkSvr** keyword to 0 in the **dmconfig.ini** file or change BkSvr with *call setsystemoption('bksvr','0')* after the database is started.

USING DMCONFIG.INI TO INACTIVATE BACKUP SERVER

If the database is offline, you can disable Backup Server directly using the **DB_BkSvr** keyword in the **dmconfig.ini** file. The next time you start the database, Backup Server will not start. If the database is online, changing the value of the **DB_BkSvr** keyword will have no effect until the database is shut down and restarted.

➲ **To inactivate Backup Server using the dmconfig.ini file:**

1. Open the **dmconfig.ini** file on the database server using any ASCII text editor.
2. Locate the database configuration section for a database to change the backup mode.
3. Change the *value* of the **DB_BkSvr** keyword to 0 to disable the Backup Server.
4. Restart the database.

USING DMSQL TO INACTIVATE BACKUP SERVER

The stored procedure **SetSystemOption** can be used to change the state of backup server while the database is running. The general syntax to change the state of Backup Server:

```
CALL SETSYSTEMOPTION('bksvr','n')
```

Where *n* is 0, or 1. Setting *n* to 0 will inactivate the backup server. Setting *n* to 1 will activate the backup server.

➲ **Example**

To inactivate the backup server, enter the following line at the dmSQL command prompt.

```
dmSQL> CALL SETSYSTEMOPTION('bksvr','0');
```

USING JSERVER MANAGER TO INACTIVATE BACKUP SERVER

If the database is offline, you can disable Backup Server using the JServer Manager graphical utility. JServer Manager will automatically change the value of the **DB_BkSvr** keyword in the **dmconfig.ini** configuration file. The next time you start the database, Backup Server will not start. If the database is online, disabling Backup Server will have no effect until the database is shut down and restarted. For directions on how to inactivate Backup Server using JServer Manager, refer to the *JServer Manager User's Guide*.

15.7 Backup History Files

Automatic backups using the backup server can store the information about which journal files were backed up, when they were backed up, and where the backup files are located in the backup history file by automatically.

Locating the Backup History File

Backup history file is a text file located in the first directory of **DB_BkDir** keyword in the **dmconfig.ini** file. This file is created in the online backup path and is named **dmBackup.his**. The file will automatically be used during restoration of a database, but the offline backup is recorded with **offBackup.his**.

Understanding the Backup History File

Backup history files contain all information pertaining to the id number, file names, and time and date that backups were made. DBMaker uses the backup history file to track backup sequences and ensure the consistency of full, differential and incremental backups within each sequence.

The following is the format of the backup history file:

```
<backup_id>: file_name -> archive_file_name, time, event
```

This denotes that a file named *file_name* was copied to an archive file named *archive_file_name* at time because of event. The event is a text string indicating the reason for the backup. This string can be JOURNAL-FULL, TIME-OUT, ON-LINE-FULL-BACKUP-BEGIN, ON-LINE-FULL-BACKUP, or ON-LINE-FULL-BACKUP-END. The string JOURNAL-FULL indicates an incremental backup was performed because the journal was full. The string TIME-OUT indicates a differential or an incremental backup was performed because the scheduled backup interval has elapsed. The string ON-LINE-FULL-BACKUPxxxx means it is a full backup.

Using the Backup History File

If journal full occurs frequently, lower the backup journal full percentage or shorten the time interval. Also, find out if the backup interval is too short by checking the backup history file. If the same journal file is backed up consecutively in the backup history file, the time interval may be too short. This situation will waste disk space because each file may only contain a few changed blocks. To avoid this, enable compact backup mode or lengthen the backup time interval.

If many journal files are backed up every time, it may mean the time interval is too long. This situation is more dangerous because of the possibility of losing more data when a disk fails. To avoid this, users should shorten the backup time interval.

Performing full backups regularly will reduce recovery time after media failures even when using Backup Server. This also reduces the amount of backup storage needed.

Understanding the File Object Backup History File

The file object backup history file, **dmFoMap.his**, keeps a record of all file objects that have been backed up by setting the file object backup configuration parameter on. **dmFoMap.his**, placed in the *<DB_BkDir>\FO* directory, is a pure ASCII text file that records the original external file name and backup file name.

The following is the file format:

```
Database Name: MYDB
Begin Backup FO Time: 2001.5.13 2:33
FO Backup Directory: /DBMaker/mydb/backup/FO (i.e. DB_BkDir/FO)
[Mapping List]
s, fo0000000000.bak, "/DBMaker/mydb/fo/ZZ000001.bmp"
u, fo0000000001.bak, "/home2/data/image.jpg"
....
s, fo0000002345.bak, "/DBMaker/mydb/fo/ZZ00AB32.txt"
```

In the first column, **s** or **u** represent system or user file objects, respectively. The second column gives the backup name, and the third column gives the full name and path of the original file object.

Understanding the Stored Procedure Backup History File

The stored procedures backup history file, **dmSpBk.his**, keeps a record of ESQL stored procedures and JAVA stored procedures that have been backed up by setting the stored procedures backup configuration parameter on. **dmSpBk.his**, placed under the subdirectory **SP** under the directory specified by **DB_BkDir**, is a pure ASCII text file that records backup information of ESQL stored procedures and JAVA stored procedures.

The following is the file format:

```
Database Name: MYDB
Begin Backup SP time: 2014/06/20 09:13:06
[Backup List]
ESQLSP, SYSADM, A4, A4SYSADM.dll, A4SYSADM.ec
....
JAR, "", "", employee.jar, ""
```

The first column gives stored procedures' type; the second column gives stored procedure's owner; the third column gives stored procedures' name; and the fourth column and the fifth column give the name of stored procedures' object files and the name of its corresponding source files respectively.

15.8 Backup on Replication Databases

On both normal database and master, but not slave database, users can do full backup, differential backup and incremental backup. The method is same as before. However, JServerManager can't do incremental backup interactively on master database. Furthermore, it can't clear incremental backup files when doing full backup interactively on master database.

Please note that on a master database, potentially, many incremental backup files ahead of a full backup are still remained in backup sequence for replication. Meanwhile the replication server may not clear incremental backup files because of full backup, so maybe a large number of files exist in **DB_BkDir** if next full backup will not be done on a long duration.

In one word, the replication sever must cooperate with the backup sever well, and they can't disturb each other. On the one hand, backup should not damage the replication, in other words, replication server always can replication all transactions to slave sites regardless of whether a full backup or a differential backup has been done or is being done, on the other hand, replication can't damage backup sequence.

It is doable to restore the master database with backup sequence. However, after the master database restored, the database replication will not continue. If users want to continue replicating the database, all slave databases must have been replaced by new master database, that is to say, users must copy the master database files to replace all slave databases files.

There are some constraints for backup on replication database:

- When the master database started up, BMODE and BKSVR must be on.
- BMODE, BKSVR, BKDIR can't be changed during runtime both on master and slave databases, for example, call setsystemoption ('bkdir','new-bkdir') will return an error.
- Both on master and slave databases site, **DB_BkDir** keyword in the dmconfig.ini file should be single path. If Users set **DB_BkDir** keyword is multi-path, only the first path is used and the path size is ignored.
- On a master database, it is disable to do incremental backup interactively by JServerManager.
- On a slave database, it is disable to do full backup, differential backup and incremental backup.

15.9 Recovery Options

Restoring a database recreates the database as it existed at the time of the most recent full backup plus changes as applied by the backed up journal files.

Analyzing Recovery Options

What recovery operations are available?

The answer to this question is determined by whether or not a database is in BACKUP mode.

- If the database is operating in NONBACKUP mode, the only option for restoration after a disk failure is to restore the most recent full backup and restart the database. All work performed since the last full backup will be lost, and must be re-entered. If this is the case, there is no need to answer the following questions.

- If the database is operated in BACKUP (BACKUP-DATA or BACKUP-DATA-AND-BLOB) mode, several recovery options are available for reconstructing the damaged database.

Preparing for Restoration

Before you restore a database after a disk error, answer the following questions:

- What point in time do you want a database restored to?

If your answer is the time when the disk error occurs, backup all journal files of the damaged database. These files will help DBMaker to restore the database to the most current time.

- What files have previously been backed up?

Find out where the most current full backup and all subsequent differential and incremental backups are located. For example, suppose you perform a full backup on the 30th day of every month, a differential backup every 15th day and an incremental backup every 10 days. If your system is damaged on May 25th, you need the full backup from April 30th, the differential backup from May 15th and the incremental backups from May 10th and May 20th, and the damaged journal files from May 25th. After locating these files, DBMaker can restore your database to the state it was in before the failure on May 25th. The valid backup sequence that is composed of a group of full backup files and a series of differential backup files and incremental backup files is essential for the restoration. The online backup sequence is identified by a backup history file named **dmbackup.his** and the offline backup sequence is identified by a backup file named **offbackup.his.** This makes the backup history file especially important because DBMaker reads it to get this information when restoring a database.

Performing a Restoration

When executing the restoration process, DBMaker will do the following actions:

- Copying all full backup files; includes data files, blob files and journal files, to the directory specified by the **DB_DbDir** keyword in the **dmconfig.ini**. This operation will overwrite the original database files. So it is strongly recommended that user can manually copy original database files to other place before running the restoration tools, at least make sure the journal files to be saved, to insure that if the restoration failed, there is also another chance for the database to be restored to the most current time.
- Applying the differential or incremental backup files or both into database.

When using restoration tools, users can specify:

- Whether restore database section in system **dmconfig.ini**. To restore it, specify the full path of restored **dmconfig.ini**.
- If you want to use a backup sequence to restore the database to a location different from the original, modify the keywords in the data file *path*, these include **DB_DbDir**, **DB_DbFil**, **DB_UsrDb**, and so on. If the backup sequence is moved to another location or computer, consider the following when restoring the database:
  a) Database names in **dmconfig.ini** must be consistent with the backup database name.
  b) Set keywords BKDIR and others for data and blob files as needed.
  c) The value of **DB_JnFil** must be set if there is more than one jnl file. Ensure that this value matches the number of jnl files in backup database.
  d) If backup files are located in multiple folders the **DB_BkDir** keyword must be set to include all folders where backup files are located. The **dmbackup.his** file must be located in the first BKDIR.

  NOTE *Users can copy an existing **dmconfig.ini** configuration file from the folder where the backup sequence existed, then configure a new dmconfig.ini file by modifying the relevant keywords.*
- Backup full path of a backup history file **dmBackup.his** or **offBackup.his** if the **dmBackup.his** or **offBackup.his** is not located in the default directory.

- Restore time (RTime). RTime denotes what time the database to be restored to, it will determine whether the current backup sequence is available or not, and which differential and incremental backup files will be applied to database. User can specify it in restoration tools, or add keyword **DB_RTime** into system dmconfig.ini or backup dmconfig.ini which will be restored. If RTime is not specified, the default value is the current time.

DBMaker provides two methods to perform restoration. One is by JServer Manager Tool and the other is by Rollover command line tool.

For more information on usage of the JServer Manager, please refer to *JServer Manager User's Guide*. And for more information on usage of the rollover command line tool, please refer to next section 'use rollover to restore database'.

Restoring database by Rollover

User can also use the Rollover which is a command line tool to restore the database. Its principle is same as the Restore Database of JServer Manager.

The usage of rollover is like:

rollover database_name [-i inifile] [-r rtime] [-h hisfile] [-m foMapfile] [-f FOtype] [-t rsSP]

There are six optional parameters in the square bracket:

-i specify full path of **dmconfig.ini**. If user specifies the **dmconfig.ini** to restore, rollover will replace the database section in system dmconfig.ini with the corresponding database section in specified **dmconfig.ini**, otherwise, DBMaker will not restore **dmconfig.ini**.

-r denote the time that database should be restored to. The option –r is the first method to specify rtime, the second method is to add **DB_RTime** keyword into system dmconfig.ini or backup dmconfig.ini which will be specified to restore database. If neither **–r** option nor **DB_RTime** keyword, the rtime will be the current time.

-h give full path of **dmBackup.his** or **offBackup.his**. The default is *DB_BkDir\dmBackup.his* or *"DB_BkDir\offBackup.his"*.

-m give full path of **dmFoMap.his**. The default is *DB_BkDir\FO\dmFoMap.his*.

-f specify which type FO files to restore. There are four values, the value of 0 means no FO files to be restored; the value of 1 will restore system FO; value of 2 will restore user FO and value of 3 will restore all FO. The default value is *3*.

-t specify whether restore stored procedures. There are two values. If set its value to 0, no stored procedures will be restored; if set its value to 1, all stored procedures will be stored. The default value is *1*.

16 Distributed Databases

This chapter introduces the distributed database management functions provided by DBMaker, including distributed databases, the distributed architecture, distributed data access, distributed database object management, and distributed transaction management.

16.1 Introduction to Distributed Databases

Traditional client-server DBMS, as shown in Figure 16-1, locate the database on a specific network computer, and the computer is responsible for handling all client requests.

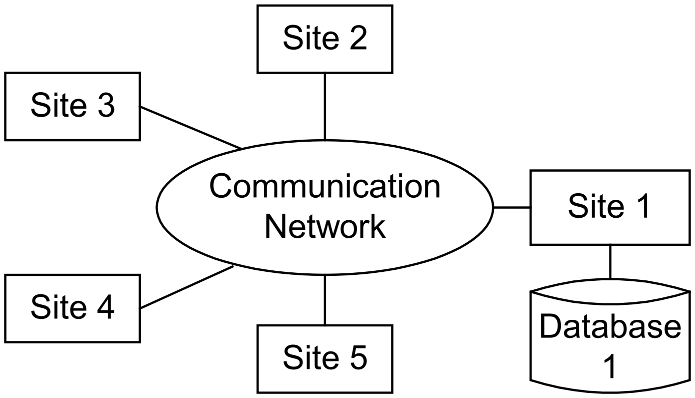


Figure 16-1 Traditional client/server database management system

Distributed databases, as shown in Figure 16-2, locate a copy of the database on several network computers, and each can independently support clients. The distributed database management system manages the databases on these computers, so users can access the data transparently.

DBMaker supports a true distributed architecture to provide a complete and robust distributed database management system (Distributed DBMS). It provides remote database connections, distributed queries, and distributed transaction management. DBMaker also provides table and database replication to keep data automatically up-to-date.

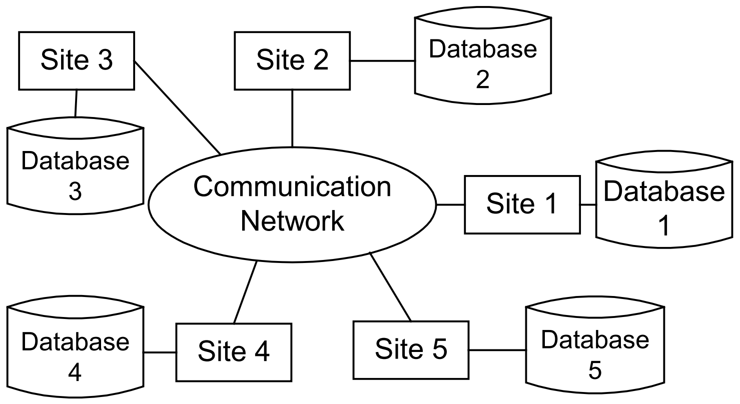


Figure 16-2 Distributed database in client-server

In the DBMaker distributed database environment, you can write application programs using the DBMaker ODBC 3.0 compatible API or perform ad-hoc SQL queries that access data from different parts of the distributed database. DBMaker will transparently integrate the data and return the results, just as if they all came from a local database.

In this chapter, we will briefly describe the system architecture and basic functions of distributed database management using DBMaker. This includes configuring the distributed environment, managing remote data links and distributed transactions, and performing distributed queries. Whether you are a database administrator or an application developer, this chapter will provide you with a thorough overview of the simplicity and power of the DBMaker distributed architecture.

16.2 Distributed Database Structure

The DBMaker distributed database environment builds on the traditional client/server architecture, effectively linking multiple client applications and multiple database servers. Client applications process user requests and display the results, and the database servers handle data management. Each client has a direct connection to a single database server, which is known as the Coordinator Database to that client. Through the Coordinator Database, the client can connect to other remote databases, which are known as Participant Databases.

DBMaker uses a hierarchical distribution structure to connect to remote databases. This allows DBMaker to access data from a remote database with no direct connection to the Coordinator Database by routing through one of the Participant Databases. When this happens, the Participant Database becomes a Local Coordinator Database, and acts as coordinator for any child databases accessed through it.

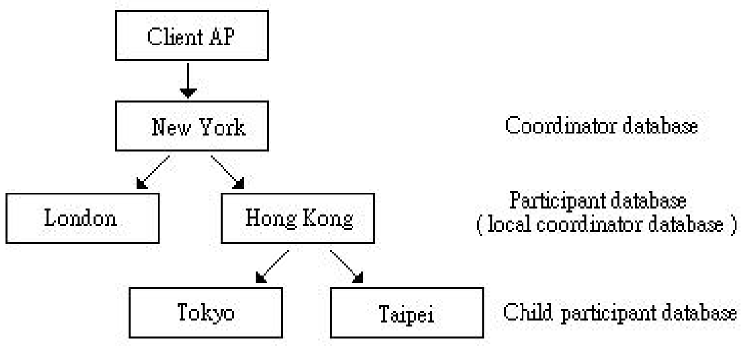


Figure 16-3 DBMaker distribution structure

In Figure 16-3, the client application program connects to the database server in New York, which makes the database in New York the Coordinator Database. If you use the database in New York to access data from London and Hong Kong, then both the London and Hong Kong databases are Participant Databases.

Some of the data you are looking for in Hong Kong might actually be in the databases in Tokyo or Taipei, so the databases in Tokyo and Taipei are Child Participant Databases. This makes the database in Hong Kong a Coordinator Database for the databases in Tokyo and Taipei, so the database in Hong Kong is not only a Participant Database, but also acts as a Local Coordinator Database.

16.3 Distributed Database Environment

Setting up a distributed database environment using DBMaker is as simple as adding some keywords to the **dmconfig.ini** file to set the distributed database configuration options. Optionally, set these parameters using the JConfiguration Tool. For more information, refer to the *JConfiguration Tool User's Guide*.

You must provide values for the following keywords when setting up a distributed database environment in DBMaker. Keywords with the prefix DB_ are for the client/server connection between the client and the Coordinator Database, and keywords with the prefix DD_ are for the distributed database connections between the Coordinator Database and the Participant Databases.

- **DB_SvAdr=<ip_address/host name>** — IP address or host name of the Coordinator Database.
- **DB_PtNum=<port number>** — port number that the client application and the Coordinator Database should use to communicate.
- **DD_DDBMd=<0/1>** — enables distributed database mode for the Coordinator Database. The default value is 0, which means that distributed database mode is disabled.
- **DD_CTimO=<number of seconds>** — time in seconds that the Coordinator Database should wait when trying to establish a connection to a Participant Database. The default value is 5 seconds.

- **DD_LTimO=<number of seconds> —** time in seconds that the Coordinator Database should wait when trying to establish a lock on the requested data in a Participant Database. The default value is 5 seconds.
- **DD_GTSvr=<0/1> —** enables the global transaction recovery daemon (GTRECO). The default value is 1, which means the global transaction recovery daemon is enabled.
- **DD_GTItv=<YYYY/MM/DD hh:mm:ss> —** specifies the time interval that the global transaction recovery daemon (GTRECO) should wait when processing pending global transactions.

DBMaker supports an automatic recovery mechanism for distributed transactions that have failed due to network problems or errors on the Participant Database. The automatic recovery mechanism is handled by the GTRECO daemon, which will check whether a distributed database server has any problems with pending global transactions. If any problems are detected, the GTRECO daemon will attempt to recover the pending global transactions. The GTRECO daemon is enabled using the **DD_GTSvr** keyword in the **dmconfig.ini** file.

To better understand how DBMaker manages distributed databases, refer to the following example.

➲ **Example**

ABC Bank has two branch offices; One in Los Angeles and the other in Seattle. Each branch maintains its own customer and business data, but the Los Angeles branch controls the government and financial databases.

The LA branch database server **dmconfig.ini** file:

```
[BankTranx]                              ;LA branch business database
DB_DbDir = c:\database
DB_SvAdr = 192.168.0.1
DB_PtNum = 21000
DD_DDBMd = 1

[BankMIS]                                ;government and financial database
DB_DbDir = c:\database
DB_SvAdr = 192.168.0.1
```

```
DB_PtNum = 30000
DD_DDBMd = 1

[BankTranx@Seattle]                 ;Seattle branch business database
DB_SvAdr = 192.168.0.2
DB_PtNum = 21000
DD_CTimO = 20
DD_LTimO = 10
```

The Seattle branch database server **dmconfig.ini** file:

```
[BankTranx]                         ;Seattle branch business database
DB_DbDir = c:\database
DB_SvAdr = 192.168.0.2
DB_PtNum = 21000
DD_DDBMd = 1

[BankMIS]                           ;government and financial database
DB_SvAdr = 192.168.0.1
DB_PtNum = 30000
DD_CTimO = 20

[BankTranx@La]                      ;LA branch business database
DB_SvAdr = 192.168.0.1
DB_PtNum = 21000
DD_CTimO = 20
DD_LTimO = 10
```

The LA client application server **dmconfig.ini** file:

```
[BankTranx]                         ;LA branch business database
DB_SvAdr = 192.168.0.1
DB_PtNum = 21000
```

The Seattle client application server **dmconfig.ini** file:

```
[BankTranx]                         ;Seattle branch business database
DB_SvAdr = 192.168.0.2
DB_PtNum = 21000
```

In the files shown above, set **DD_DDBMd** = 1 in the configuration section for the local database to enable distributed database support. In these examples, place the keywords in the **BankTranx** configuration section of the Los Angeles and Seattle **dmconfig.ini** files.

In addition, include a database configuration section for the Participant Database in the Coordinator Database configuration file, and for the Coordinator Database in the Participant Database configuration file. In this case, both the Los Angeles branch and the Seattle branch databases use the same database name. If you use the remote database name for the name of the database configuration section, it will cause a conflict with the local database name in the **dmconfig.ini** file.

To avoid this type of problem when using distributed databases, DBMaker can distinguish the remote database name from the local database name by appending a server host description to the remote database name in the local **dmconfig.ini** file.

The remote database name would look like:

```
database_name@server_host_description
```

The server host description can be any identifying name, such as the IP address or host name of the database server, the domain name, or almost any other descriptive text. In this example, the Los Angeles branch client application would use **BankTranx@Seattle** when it wants to access data in the Seattle branch database, and the Seattle branch client application would use **BankTranx@La** when it wants to access data in the Los Angeles branch database.

Also, set up the server address and port name for both the local database and the remote database in their respective configuration sections in the configuration files at both the Los Angeles and Seattle branches.

In this example, the Los Angeles branch configuration file would contain the local server address in the **BankTranx** configuration section, and would contain the Seattle branch server address in the **BankTranx@Seattle** configuration section. Similarly, the Seattle branch configuration file would contain the Los Angeles branch server address in the **BankTranx** configuration section, and would contain the Seattle branch server address in the **BankTranx@La** configuration section.

You should also set the **DD_CTimO** and **DD_LTimO** remote connection parameters. These parameters go in the configuration section for the Participant

Database in the Coordinator Database configuration file, and for the Coordinator Database in the Participant Database configuration file.

Every database server in the network can operate on distributed database objects. Any of these database servers can be accessed through the Coordinator Database, in a manner similar to the normal client/server architecture. The SQL commands that reference a remote database will be passed to the remote database server through the Coordinator Database. The Coordinator Database will decompose this SQL command into the local and remote portions, and send the appropriate commands to the remote database server. The Coordinator Database will wait for the remote database to return results, and then merge all local and remote data and return the combined results.

16.4 Distributed Database Objects

DBMaker provides several different methods to access a Participant Database:

- Specify the Participant Database name directly
- Using database links defined in the Coordinator Database
- Through remote object mapping such as views or synonyms.

The difference between the first two approaches is that database links contain security information in addition to the remote database name. This allows you to specify the user name and password that you want to use in the database link when you access the remote database.

There is no obvious difference between the statements of a distributed query and a normal query, except in the way database objects are specified. However, when using a remote database, the only remote database objects that can be accessed are tables, views, synonyms and stored commands. To access a remote database object, provide the remote database name or database link when specifying the name of the database object. This provides two ways to identify a remote database object:

- remote_database_name:object_owner.object_name
- database_link:object_owner.object_name

➲ **Example 1**

To specify a remote database object in a query:

```
dmSQL> SELECT * FROM Bank:EmpTable;
dmSQL> DELETE FROM Bank:EmpTable WHERE id = 101;
dmSQL> INSERT INTO Link1:mis.account VALUES (2003,'Kevin Liu','2327-0021');
```

➲ **Example 2**

To access remote database objects in two different Participant Databases:

```
dmSQL> SELECT * FROM ABCBank@La:account a,
                     ABCBankMIS@Seattle:account b
                WHERE a.name = b.name;
```

➲ **Example 3**

Create a stored command named **cmd1** in the remote database DB1, and then execute it in the local database DB2:

```
dmSQL> EXECUTE COMMAND DB1:cmd1(value);
```

Remote Database Connections-Using Names

Users can connect to remote databases with the database name of the Coordinator Database Server. Users must know the remote database name, which is defined in the **dmconfig.ini** file in the Coordinator Database Server.

➲ **Example 1**

A client application in the Los Angeles ABC bank branch accesses the database located in the Taipei ABC bank branch. It appears that the user is connecting to the Taipei branch with the user name SYSADM and the password aa. The user is actually connecting to the Coordinator Database, in other words, the Los Angeles branch database. The Coordinator Database then connects to the remote database with the supplied account and password.

```
dmSQL> CONNECT TO BankTranx SYSADM aa;
dmSQL> SELECT * FROM BankTranx@Taipei:SYSADM.Account ORDER BY AccID;
```

➲ **Example 2**

Using joins to access remote database objects:

```
dmSQL> SELECT * FROM BankMIS:SYSADM.Personnel ORDER BY PID;
dmSQL> SELECT Personnel.* FROM BankTranx@Taipei:Account A,
                               BankMIS:Personnel B
                         WHERE A.CustID = B.CustID;
```

Remote Database Connections-Using Links

A database link creates a connection to a remote database, and contains the login information and password necessary for connecting. The link permits users to connect to a remote database with a different user name than in the Coordinator Database, or to connect to a remote database with no account. It also makes data in a distributed database environment location transparent. The link definition, which also contains the login information and password, is stored in the Coordinator Database.

CREATING DATABASE LINKS

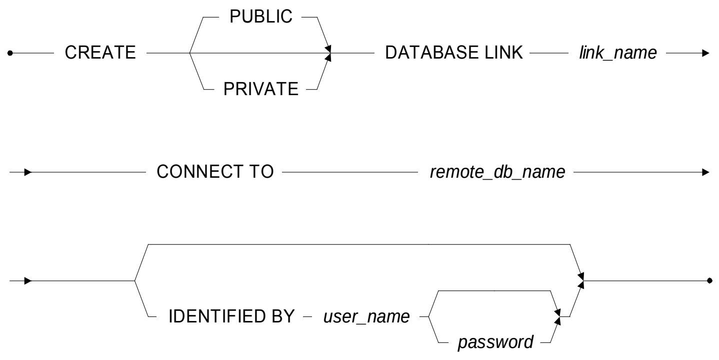


Figure 16-4 Syntax for creating a database link

Only database administrators can create public links to be used by all users in a database. Any user can create private links for themselves. Users may create private links using the same name. A private link will override a public link with an identical name.

DBMaker creates a private link by default if the user does not specify the type. If the user does not specify the login account and password in the IDENTIFY BY clause, the user's current login name and password become the default.

REMOTE DATABASE OBJECTS & DATABASE LINKS

➲ **Example 1**

The following shows how to access remote database objects using database links. In this example, the SYSADM connects to the database and creates a public link named **Bank_Seattle** that connects to the Seattle branch database using the SYSADM account. The SYSADM updates some values and disconnects. Then **user1** connects and performs a query on the **Account** table.

```
dmSQL> CONNECT TO BankTranx SYSADM;
dmSQL> CREATE PUBLIC DATABASE LINK Bank_Seattle CONNECT TO BankTranx@Seattle
    2> IDENTIFIED BY SYSADM;
dmSQL> UPDATE Bank_Seattle:Account SET balance = balance + 100
    2> WHERE id = 1001;
dmSQL> DISCONNECT;
dmSQL> CONNECT TO BankTranx user1 pwd1;
dmSQL> SELECT * FROM Bank_Seattle:Account;
```

➲ **Example 2**

The SYSADM does not specify the account to use when connecting to the public link. This means that when user1 uses the public link to connect to the **Bank_Seattle** database, there must be an account for **user1** in the remote database, and the user1 account must have the authority to query the **SYSADM.Account** table. Otherwise, an error will occur.

```
dmSQL> CONNECT TO BankTranx SYSADM;
dmSQL> CREATE PUBLIC DATABASE LINK Bank_Seattle CONNECT TO BankTranx@Seattle;
dmSQL> SELECT * FROM Bank_Seattle:Account;
dmSQL> DISCONNECT;
dmSQL> CONNECT TO BankTranx user1 pwd1;
dmSQL> SELECT * FROM Bank_Seattle:SYSADM.Account;
```

If a database link name is the same as the remote database name, DBMaker will use the database link name in preference to the remote database name. If you want to access the remote database directly, you must specify the remote database name in the form of dbname@"" to force DBMaker to access the remote database directly instead of through the database link.

The following 2 examples show different ways to access a remote database, one through a database link and the other by specifying the remote database name in the form of dbname@"".

➲ **Example 1**

SYSADM connects to a remote database with a link.

```
dmSQL> CONNECT TO BankTranx SYSADM;
dmSQL> CREATE PUBLIC DATABASE LINK BankMIS CONNECT TO BankMIS
    2> IDENTIFIED BY SYSADM;
dmSQL> DISCONNECT;
```

➲ **Example 2**

user1 connects to a remote database using the **BankMIS@"":SYSADM.Personnel** form.

```
dmSQL> CONNECT TO BankTranx user1 pwd1;
dmSQL> SELECT * FROM BankMIS:Personnel;             //using database link
dmSQL> SELECT * FROM BankMIS@"":SYSADM.Personnel;    //using remote db name
```

NOTE *When access remote database objects using database links and doing some UPDATE or DELETE actions, there should not have sub query. At present, this is not supported by DBMaker.*

Deleting Database Links

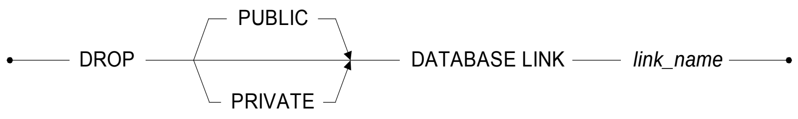


Figure 16-5 Syntax for deleting a database link

Only database administrators can delete public links, and only the owner of a private link can delete it. Ensure to specify the public link to be deleted when it has the same name as a private link or DBMaker will delete the private link by default.

➲ **Example**

To delete a public database link named **BankMIS**:

```
dmSQL> DROP PUBLIC DATABASE LINK BankMIS;
```

Database Object Mapping

Database Object Mapping provides better location transparency in a distributed database environment. There is no difference between the way a user accesses remote database objects with Database Object Mapping and local database objects. This type of Database Object Mapping includes using views and synonyms.

SYNONYMS

Using a synonym to define a remote database object is done by assigning the remote database object an alias name. The privileges you have in the remote database when using a synonym are the same as in a local database.

➲ **Example**

To access a remote database object using a synonym:

```
dmSQL> CONNECT TO BankTranx user1;
dmSQL> CREATE DATABASE LINK LK1 CONNECT TO BankMIS IDENTIFIED BY user2;
dmSQL> CREATE SYNONYM s1 FOR BankTranx:Account;
dmSQL> CREATE SYNONYM s2 FOR LK1:user2.Personnel;
dmSQL> SELECT * FROM s1;
       // SELECT * FROM BankTranx:user1.Account; (BankTranx, user1)
dmSQL> SELECT * FROM s2;
       // SELECT * FROM LK1:user2.Personnel; (BankMIS, user2)
dmSQL> DISCONNECT;
dmSQL> CONNECT TO BankTranx user3;
dmSQL> CREATE DATABASE LINK LK1 CONNECT TO BankMIS IDENTIFIED BY user4;
dmSQL> SELECT * FROM s1;
       // SELECT * FROM BankTranx:user3.Account; (BankTranx, user3)
dmSQL> SELECT * FROM s2;
```

```
       // SELECT * FROM LK1:user2.Personnel; (BankMIS, user4)
```

The comments indicate the equivalent SQL expression, the database being connected to, and the account used to connect.

VIEWS

Using a view to define a remote database object is a bit different than using a synonym. The view is not just an alias, but includes the database name, user account, password, object owner, and object name as part of the definition. The privileges a user has in the remote database depend on the privileges of the user that created it.

➲ **Example**

To access a remote database object using a view:

```
dmSQL> CONNECT TO BankTranx user1;
dmSQL> CREATE DATABASE LINK LK1 CONNECT TO BankMIS IDENTIFIED BY user2;
dmSQL> CREATE VIEW v1 AS SELECT * FROM BankTranx:Account;
dmSQL> CREATE VIEW v2 AS SELECT * FROM LK1:user3.Personnel;
dmSQL> SELECT * FROM v1;
       // SELECT * FROM BankTranx:user1.Account; (BankTranx, user1)
dmSQL> SELECT * FROM v2;
       // SELECT * FROM BankMIS:user3.Personnel; (BankMIS, user2)
dmSQL> DISCONNECT;
dmSQL> CONNECT TO BankTranx user3;
dmSQL> CREATE DATABASE LINK LK1 CONNECT TO BankMIS IDENTIFIED BY user4;
dmSQL> SELECT * FROM v1;
       // SELECT * FROM BankTranx:user1.Account; (BankTranx, user1)
dmSQL> SELECT * FROM v2;
       // SELECT * FROM LK1:user3.Personnel; (BankMIS, user2)
```

The comments indicate the equivalent SQL expression, the database being connected to, and the account used to connect.

Closing Links

Once a user accesses a remote database with a SQL command, the Coordinator Database will build a remote connection to the Participant Database. The remote connection will remain open until all users disconnect from the Coordinate Database or until the link is closed with the CLOSE DATABASE LINK command. DBMaker provides up to 256 remote connections for each database; it is a good idea to close remote connections that are no longer in use to free the connections for other users.

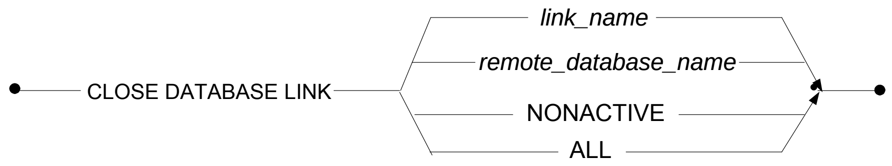


Figure 16-6 Syntax for closing a database link

➲ **Example 1**

To close a database link using the remote database name **BankMIS**:

```
dmSQL> CLOSE DATABASE LINK BankMIS;
```

➲ **Example 2**

To close a database link using the remote database link **BankLink1**:

```
dmSQL> CLOSE DATABASE LINK BankLink1;
```

When a user issues a CLOSE DATABASE LINK command, DBMaker will decrease the remote connection counter by one. When the counter reaches zero, the connection is fully closed and the occupied resources freed. Otherwise, the connection remains open.

➲ **Example 3**

To close all database links and free the connections and resources:

```
dmSQL> CLOSE DATABASE LINK ALL;
```

➲ **Example 4**

To close all NONACTIVE remote connections no longer being used:

```
dmSQL> CLOSE DATABASE LINK NONACTIVE;
```

Link System Catalog Tables

Two system catalog tables relate to database links: SYSDBLINK and SYSOPENLINK. SYSDBLINK records all database link names and their definitions, while SYSOPENLINK, records open connections between databases.

16.5 Distributed Transaction Control

DBMaker supports a distributed transaction mechanism that is transparent to users. Participant Databases do not commit only a part of a distributed transaction; DBMaker handles it.

➲ **Example**

The following shows how distributed transaction control works.

```
dmSQL> CONNECT TO BankTranx user1;                    // ABC Bank in Taipei
dmSQL> SET AUTOCOMMIT OFF;
dmSQL> UPDATE BankTranx:Customer SET money=money-1000 WHERE id=123;
dmSQL> UPDATE BankTranx@"Bank_in_Seattle":Customer SET money=money+1000
    2> WHERE id=123;
dmSQL> COMMIT;
```

Since the Coordinator Database handles all database operations from the client application, the Coordinator Database knows the scope of the instructions via the Distributed System Catalog Manager. The Coordinator Database will handle transactions belonging to the local site in the same manner as a regular client/server transaction. Transactions belonging to remote sites need to reference the appropriate remote database. The Coordinator Database will exchange information with every Participant Database and coordinate the whole transaction until it is either rolled back or committed.

Two-Phase Commit

Database management systems need to maintain data integrity, and this requires that all transactions be *atomic*. All operations in the transaction must commit or roll back together. In the traditional client/server architecture, a journal is used to make sure that the changes are either rolled back or committed.

In the distributed database architecture, a two-phase commit protocol with presumed abort is used as the mechanism for controlling distributed transactions that span multiple database servers. A transaction that modifies data on two or more databases must complete the two-phase commit protocol before it is committed. The two-phase commit mechanism guarantees that all sites commit or roll back globally. It also protects data manipulation operations performed by remote synonyms, integrity constraints, and triggers. To commit a transaction, a user has to ensure every sub-transaction has finished; otherwise, the transaction will be aborted. For the same reason, if any sub-transaction cannot commit, the other sub-transactions must be aborted as well.

Distributed Transaction Recovery

DBMaker uses the two-phase commit protocol to inform all Participant Databases to commit a global transaction. Before entering the commit phase, the Coordinator Database will check the status of the Participant Databases to ensure there are no server or network problems. If the Coordinator Database finds a problem with any of the Participant Databases, it informs the other Participant Databases to roll back their part of the transaction, and returns an error indicating the global transaction has failed. If the two-phase commit protocol has finished the preparation phase, but there is a server or network problem with a Participant Database, the global transaction is regarded as a success. DBMaker records which Participant Database server cannot commit its part of the transaction in the SYSGLBTRANX system catalog table, and records which database contains a pending transaction in the SYSPENDTRANX system catalog table for the crashed database.

DBMaker also provides an automatic recovery mechanism to handle network or site failure during the execution of a distributed transaction. In the Coordinator Database, you start the global transaction recovery daemon (GTRECO). This daemon scans the SYSGLBTRANX system catalog table and periodically recovers any pending global transactions. Then it tries to connect to the crashed Participant Database and informs it to commit or roll back its part of the global transaction.

Heuristic End Global Transaction

After a network or site failure occurs during the two-phase commit, pending transactions continue to hold some resources such as locks or journal blocks. The pending transaction will occupy these resources until the problem is solved by the global recovery daemon (GTRECO). If the network or site failure cannot be solved immediately, then the held resources may block some of the users in the Participant site. To solve this problem, DBMaker supports heuristic end global transaction. A heuristic end transaction is an independent action taken by the database administrator to force a pending transaction in a Participant Database to commit or roll back. The database administrator can use JDBATool to solve this problem. Refer to the *JDBA Tool User's Guide* for more information.

➲ **To resolve a pending transaction manually, perform the following:**

1. In the Participant Database, browse the SYSPENDTRANX table to find out whether there are transactions that have been pending for a long time.
2. In the Coordinator Database, determine the commit status of the pending transaction from SYSGLBTRANX. Also determine the commit status of the pending transaction from the Participant Database. For example, if there are two pending transactions "DB_1-3376aafd" and "DB_2-3376aafd:DB_3-3376ab0f#1", the administrator of the coordinator database should ask the administrator of DB_1 to determine the status of "DB_1-3376aafd" and ask the administrator of DB_3 to determine the status of "DB_2-3376aafd:DB_3-3376ab0f#1".
3. In the Coordinator Database, when the administrator receives the transaction status query he can examine SYSGLBTRANX to determine the transaction status. If the STATE is 2 (COMMIT) or 3 (PENDCOM), reply "**commit**". If the STATE is 4 (PENDABO), reply "**abort**". But if the STATE is 1 (PREPARE), this transaction branch is pending in this site too, and ask the administrator of the parent site to determine its status.
4. In the Participant Database, the administrator uses JServer Manager to perform a heuristic commit or abort the pending transaction based on the reply.

If the administrator initiates a heuristic end transaction on a pending transaction that is different from action taken in the Coordinator Database, the distributed data will be inconsistent.

17 Data Replication

Data Replication, in the broadest sense of the term, refers to the process of representing objects in more than one database.

Only a few years ago, corporate data resided in a central location. Remote departments accessed the information they needed by establishing direct connections to the central sites, or by requesting printed reports from central MIS. The connections however were expensive, unreliable, and limited in number, while the reports were inflexible and untimely.

Open systems brought inexpensive and powerful computing resources to all corners of an enterprise. The ability to share corporate information effectively using these new resources became an important competitive advantage for organizations. The question enterprises face today is not "Why distribute and share corporate data?" but rather "How can one distribute information effectively?" Replication is quickly becoming the architecture of choice for a majority of distributed corporate applications.

17.1 Table Replication

What is Table Replication?

Table replication creates a full or partial copy of a table to a destination location. This allows users in remote locations to work with a local copy of data. The local copy remains synchronized with the databases in other locations. This way each database can service data requests immediately and more efficiently, without having to go to another machine over a slower wide area network connection. This kind of request happens frequently between the headquarters and area companies or branch offices.

For example, after creating a replication from table A on Taipei's database server to table B on Tokyo's database server, modifications made to table A will replicated to table B. Clients in Tokyo can then access Tokyo's database rather than connecting to Taipei's database to acquire the same data.

Destination tables may also reside on databases on the same server. This situation may occur when two databases, designed for different functions, share data, or when sharing data between databases of different types (e.g., Oracle, Sybase). Tables that are receiving data from another database through replication are destination tables rather than remote tables, even though the destination tables may reside on a remote database.

Differences between Database and Table Replication

The major difference between database and table replication is that the replicated data object is different: one is the whole database; the other one is a table. Users can choose one of them depending on their demands. If you choose database replication, since the unit to replicate is the whole database, the target (or slave) database is read-only.

Two Types of Table Replication

There are two types of table replication. One is synchronous. 'Synchronous' means the modification to the destination site shows up immediately; the destination table is modified at the same time as the local table. DBMaker uses a 'two phase commit' and triggers to perform synchronous table replication. Thus, after establishing a replication, any update on the source table will become a DDB (distributed database) action. This will affect the local database's behavior. If the destination database server is unreachable, updates on the local database will fail.

The other table replication type is asynchronous. Modifications to the destination site are delayed. The delay between source and destination database depends on a user-defined schedule. Asynchronous table replication stores changes to the local table and modifies the destination table based on a schedule. In this type of replication, two databases of a replication pair are independent and can work as normal even if the network is not available.

Term Definitions

SOURCE TABLE

The table on the source database that the data is replicated from.

DESTINATION TABLE

The table on the destination database that the data is replicated to.

PUBLICATION

A data set on the source table that is available for replication.

SUBSCRIPTION

The data set on the destination table to receive a publication.

FRAGMENT

Also called a horizontal partition, a fragment is the replication of a given range of data.

PROJECTION

The selected columns from a base table chosen for replication.

REPLICATION DOMAIN

A replication fragment (horizontal partition) and projection (vertical partition) are called a replication domain. It is the range of a table's data to be replicated.

There is a problem when replicating a domain change. It may occur when replicating an UPDATE statement.

➲ **Example**

```
dmSQL> CREATE REPLICATION rp_case1 WITH PRIMARY AS tb_example WHERE number > 0
REPLICATE TO db2:tb_example;
```

```
dmSQL> CREATE REPLICATION rp_case2 WITH PRIMARY AS tb_example WHERE number < 0
REPLICATE TO db2:tb_example;
dmSQL> UPDATE tb_example SET number=-7 WHERE number=7;
```

The **rp_case1** replication domain is number > 0, and the **rp_case2** replication domain is number < 0. When replicating, it is not only replicating the UPDATE statement. The updated tuples replication domain is changed from **rp_case1** to **rp_case2**, subsequently the replication performs a DELETE statement for **rp_case1** (DELETE number = -7) and performs an INSERT statement for **rp_case2** (INSERT number = 7).

DATA INITIALIZATION

When creating replications, users can specify how to initialize the data on both sides automatically. After creating a table replication, any modification (insertion, deletion, update), to a source table will affect the destination tables.

DBMaker provides 4 options:

- **Clear data** — during table replication delete all data from the destination table
- **Flush data** —insert all data tables that satisfy the fragment for the source table into the destination tables
- **Clear and flush data** — clear and then flush data
- **Do nothing** — Keep the destination tables

Creating Table Replication

The following syntax diagram describes both asynchronous and synchronous table replication.

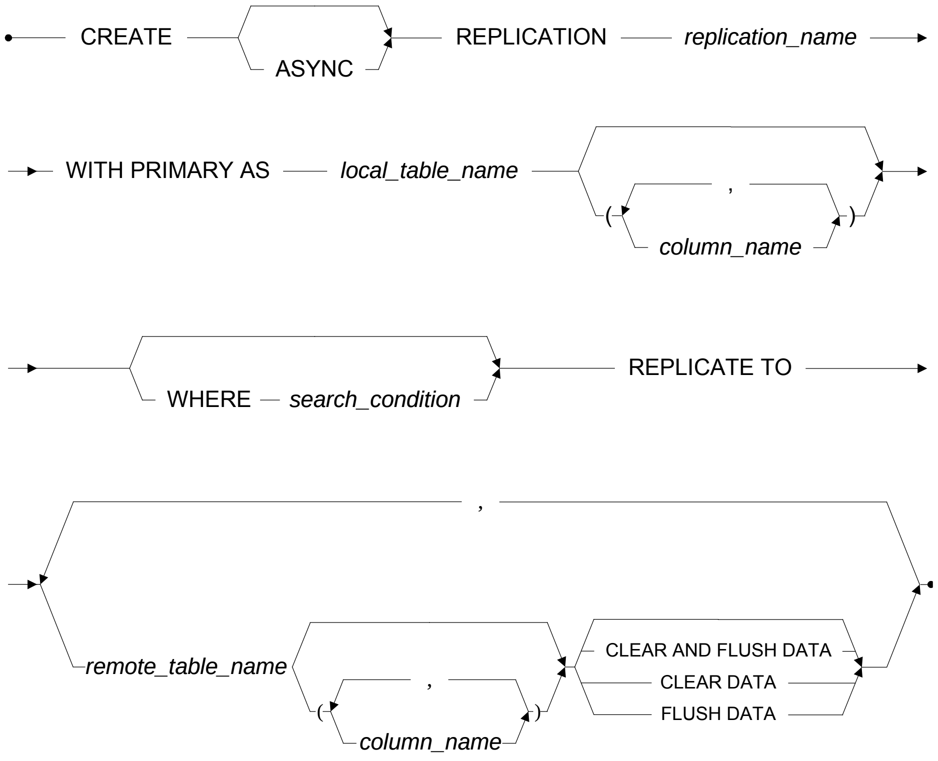


Figure 17-1 Syntax for the CREATE REPLICATION Statement

➲ **Example**

Suppose there is a table **TB1** in database **DB30A**, a table **TB2** in **DB30B**. We want to replicate **TB1**'s data to **TB2** and do not want to modify **TB2**'s current data.

```
dmSQL> CREATE REPLICATION rp_example WITH PRIMARY AS TB1
          REPLICATE TO DB30B:TB2;
```

In the above example, we use a database session name to specify the destination database. Alternatively, a database link name could be used.

Table Replication Rules

- Schemas of source and destination tables must exist. This means DBMaker does not create tables when performing table replication
- Replication names for a table must be unique
- The subscriber name for a replication must be unique using the <*link_name/database_session_name*> + <*table_owner_name*> + <*table_name*> syntax
- Projected columns for every table must contain primary key columns
- Primary key columns must be included with fragment columns
- Only the table owner of the source table or a user with DBA or higher authority has the privilege to create, drop, or alter replications
- If no column-identifier exists with the destination table, the destination column names must be the same as the base table names
- Number of primary key columns must be equal in the base table and destination tables

➲ **Example 1**

The following statement will create a publication that will replicate table **tb_salary** from the local database to table **usr1.tb_salaryA** on **db1**. Without specifying column names, all columns in table **tb_salary** must exist in **tb_salaryA** and column types must be compatible. If **tb_salary** contained 3 columns **id**, **name**, and **basepay**, it would replicate **id**, **name**, and **basepay** from table **tb_salary** to columns **id**, **name**, and **basepay** on table **usr1.tb_salaryA**.

```
dmSQL> CREATE REPLICATION rp_salaryA WITH
       PRIMARY AS tb_salary
       REPLICATE TO db1:usr1.tb_salaryA;
```

➲ **Example 2**

The following statement will create a publication used to replicate tuples where **id** > 100 with columns (**id**, **name**) to (**Aid**, **Aname**) in **tb_salaryA** on **db1**, and (**id**, **name**) in **tb_salaryB** on **db2**. After issuing this command, all data where **id** > 100 on **tb_salary** will be replicated to **tb_salaryB** on **db2** and **Aid** and **Aname** of **tb_salaryA** on **db1**:

```
dmSQL> CREATE REPLICATION rp_salaryAB WITH
                 PRIMARY AS tb_salary (id,name) WHERE id > 100 ,
        REPLICATE TO db1:tb_salaryA (Aid,Aname),
                     db2:tb_salaryB flush data;
```

➲ **Example 3**

The first function of this command is to delete all data from **tb_salaryA** on **db1** and then to create a publication used to replicate the records where **id** > 100 with only column (**id**, **name**) to (**Aid**, **Aname**) in **tb_salaryA** on **db1**:

```
dmSQL> CREATE REPLICATION rp_salaryAClear WITH
                 PRIMARY AS tb_salary (id,name) WHERE id > 100 ,
        REPLICATE TO
                 db1:tb_salaryA (Aid, Aname) clear data;
```

Drop Replication

This command will drop a replication from a source table.

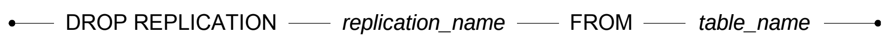


Figure 17-2 Syntax for the DROP REPLICATION Statement

➲ **Example**

To drop the replication **rp_salaryA** from the **tb_salary** table:

```
dmSQL> DROP REPLICATION rp_salaryA FROM tb_salary;
```

Alter Replication

Users may add destination tables to or drop destination tables from an existing table replication. The following syntax diagrams and examples demonstrate how.

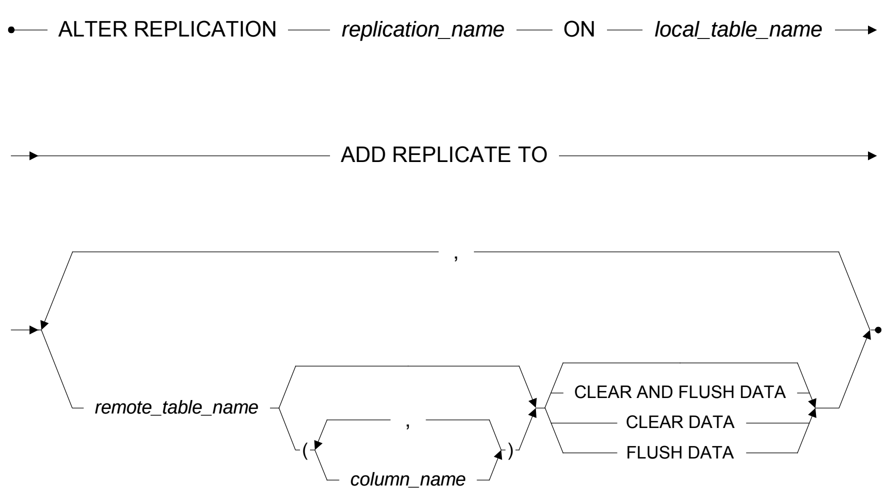


Figure 17-3 Syntax for the ALTER REPLICATION … ADD REPLICATE TO Statement

➲ **Example 1**

The first command creates a replication to replicate table **tb_salary** on the local database to table **tb_salaryX** on database **dbX**. The second command adds 2 subscribers, **tb_salaryA** on **db1** and **tb_salaryB** on **db4**, to the replication of **rp_AlterRp** in table **tb_salary**.

```
dmSQL> CREATE REPLICATION rp_AlterRp WITH PRIMARY AS tb_salary
       REPLICATE TO dbX:tb_salaryX;
dmSQL> ALTER REPLICATION rp_AlterRp ON tb_salary
       ADD REPLICATE TO db1: tb_salaryA,
                   db4: tb_salaryB (Bid, Bname) clear data;
```

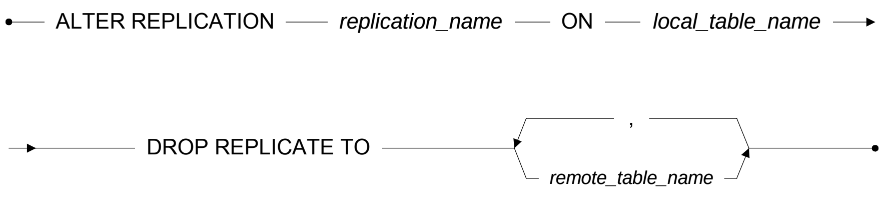


Figure 17-4 Syntax for the ALTER REPLICATION … DROP REPLICATE TO Statement

➲ **Example 2**

To drop subscriber **tb_salaryA** on **db1** from the replication of **rp_AlterRp** for table **tb_salary**:

```
dmSQL> ALTER REPLICATION rp_AlterRp ON tb_salary
       DROP REPLICATE TO db1: tb_salaryA;
```

17.2 Synchronous Table Replication

Two-phase commit allows the synchronization of distributed data. A transaction will only be accepted if all interconnected distributed sites agree. A 'handshake' mechanism across the network allows distributed sites to coordinate their acceptance for each transaction. Therefore, using synchronous table replication guarantees that data will be synchronous whenever an update occurs.

Synchronous Table Replication Setup

DBMaker uses two-phase commits to perform synchronous table replication. Source and destination databases must be in distributed database (DDB) mode. Therefore, the first step is to start the databases in DDB mode (**DD_DDBMd** = 1) and add database sessions to the **dmconfig.ini** file. Refer to Chapter 16, *Distributed Databases* for more information.

After creating a table replication, any modifications (insertion, deletion, updates) to source a table will affect the destination tables.

17.3 Asynchronous Table Replication

Synchronous table replication modifies the destination table at the same time it modifies the local table, while asynchronous table replication stores changes to the local tables and modifies the destination table based on a schedule.

DBMaker uses a file known as a replication log to store changes to local tables. Modifications to local tables are stored in replication logs, and are replicated to the destination table according to a predefined schedule. Using replication logs enables DBMaker to treat the local transaction and the destination transaction independently, allowing you to update local tables normally even if the remote connection is not available. This allows asynchronous table replications to tolerate network and destination database failures, since DBMaker will keep trying until failures are corrected.

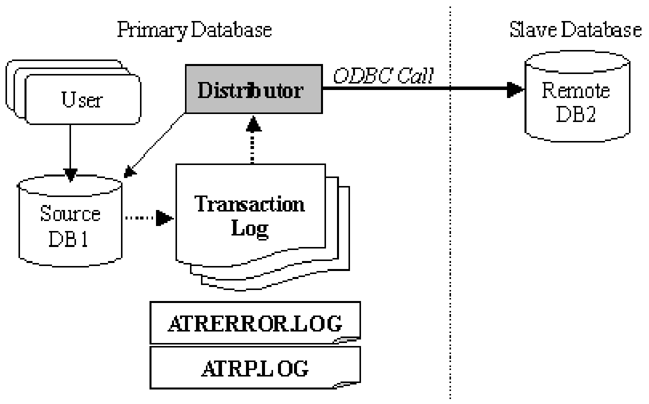


Figure 17-5: Architecture of asynchronous table replication

Asynchronous table replication uses a replication log system and the distributor server to handle data replication. Replication logs are not DBMaker journal files. The level of the replication log is higher than the journal, and it is only for replicating tables. The content of a journal is physical data modification, but the replication log consists of commands that are applied to the destination tables.

When the source database is running, DBMaker logs the modification of source tables into the replication log files. When the distributor server activates, it will redo all changes made to the source tables to the destination tables according to the replication log.

Normally the distributor server uses ODBC function calls to communicate with the destination database servers, so it is possible to replicate tables to heterogeneous database servers such as Oracle, SQL Server, Informix, etc. Heterogeneous table replication will be covered later in this chapter. Express asynchronous table replication is another type of asynchronous table replication that does not use ODBC function calls. It will also be covered later in this chapter.

➲ **There are three primary steps to build an asynchronous table replication:**

1. Enable asynchronous table replication
2. Create a schedule to the destination database
3. Based on the schedule, create the asynchronous table replication

Enabling Asynchronous Table Replication

The distributor server resides in the source database. The distributor connects to the destination databases periodically and performs the table replication.

The **DB_AtrMd** keyword in the **dmconfig.ini** file in the source database specifies whether to start the distributor server for a database. If you do not start the distributor server, the database cannot be the source for asynchronous table replications.

The **RP_LgDir** keyword in the **dmconfig.ini** of the source database specifies the directory where DBMaker will place replication log files for asynchronous table replication. The replication log files are binary and users should not manually remove them. The default directory of **RP_LgDir** is the subdirectory named **/TRPLOG** in the database home directory.

When creating table replications with schema checking, the distributed database mode in the source and the destination database must be enabled (**DD_DDBMd** = 1). If the schedule is set to NO CHECK, DBMaker will not check the schema, and the distributed database mode can be set to OFF (**DD_DDBMd** = 0). See the following section for more information about creating replication schedules.

➲ **Example**

To replicate a table from the **SRCDB** database to the destination **DESTDB** database, add the following lines to the **dmconfig.ini** file on the source database server:

```
[SRCDB]
DB_DbDir = /disk1/DBMaker/src
DB_UsrBb = /disk1/DBMaker/src/SRCDB.BB 3
DB_UsrDb = /disk1/DBMaker/src/SRCDB.DB 150
RP_LgDir = /disk1/DBMaker/src/trplog
DB_AtrMd = 1
DD_DDBMd = 1
DB_SvAdr = srcpc
DB_PtNum = 22222

[DESTDB]
DB_SvAdr = destpc
DB_PtNum = 33333
```

The **dmconfig.ini** file on the destination database:

```
[SRCDB]
DB_SvAdr = srcpc
DB_PtNum = 22222

[DESTDB]
DB_DbDir = /disk3/DBMaker/dest
DB_UsrBb = /disk3/DBMaker/dest/DESTDB.BB 3
DB_UsrDb = /disk3/DBMaker/dest/DESTDB.DB 150
DD_DDBMd = 1
DB_SvAdr = destpc
DB_PtNum = 33333
```

Due to the way distributor server makes use of the ODBC driver manager to perform asynchronous table replication, the ODBC data source name (DSN) of the destination database must be set if the source database is running under the Microsoft Windows environment.

Schedule (Creating and Dropping)

A *schedule* must be defined, by a user with DBA privilege or higher, before creating asynchronous replication to one or more destination table. A schedule defines the starting time, the period, and the account and password used to connect to the destination database. The user can create several schedules for different destination databases on the same source database, but cannot build more than one schedule to the same destination database.

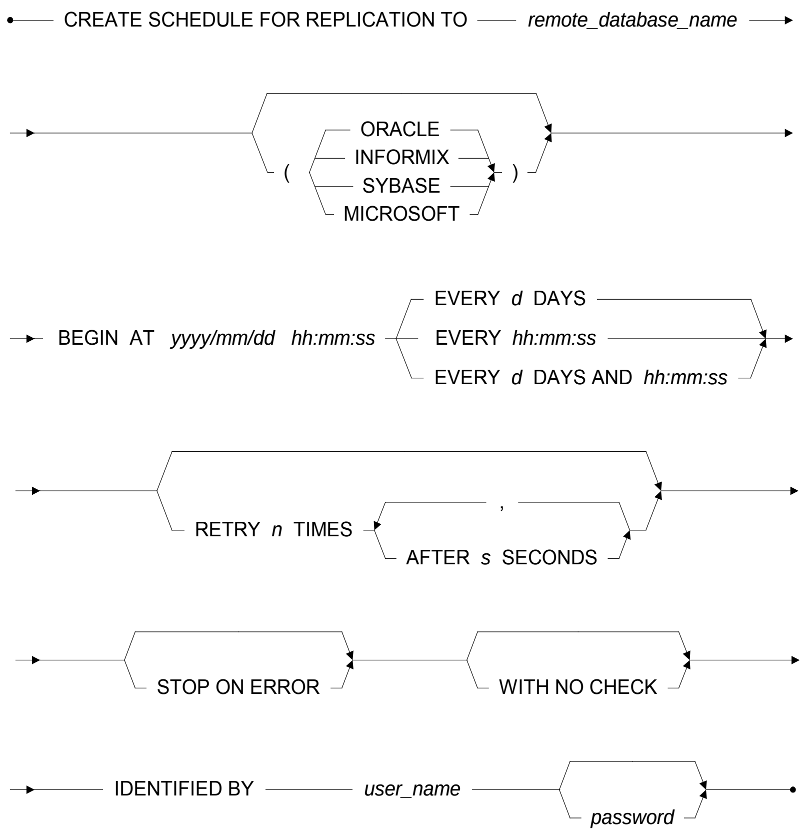


Figure 17-6 Syntax for the CREATE SCHEDULE Statement

➲ **Example 1**

To create a schedule for the **DESTDB** database:

```
dmSQL > CREATE SCHEDULE FOR REPLICATION TO destdb
          BEGIN AT 2000/1/1 00:00:00
          EVERY 12:00:00
          IDENTIFIED BY User Password;
```

From January 1, 2000, the distributor server will activate every 12 hours to perform an asynchronous replication. The *ATRP.LOG* distributor message log in the database home directory will record the starting time and status of the distributor server.

The IDENTIFIED BY keyword specifies the destination account used by the distributor server to connect to the destination database and execute the table replication. The account must have privilege to insert, delete, and update on the destination tables.

If the schedule is not necessary and there is no replication associated with it, users having the DBA privilege on the source database can drop it.

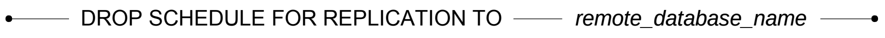


Figure 17-7 Syntax for the DROP SCHEDULE Statement

➲ **Example 2**

To drop a schedule:

```
dmSQL> DROP SCHEDULE FOR REPLICATION TO destdb;
```

Creating Asynchronous Table Replication

All asynchronous table replications depending on the same schedule will be done at the same time. The asynchronous table replication mechanism supports all data types, including LONG VARCHAR, LONG VARBINARY, and FILE types.

The action to create an asynchronous table replication is similar to creating a synchronous table replication. Add a keyword (ASYNC, CREATE ASYNC, REPLICATION command), to create table replication based on one schedule to a destination database.

➲ **Example 1**

To create a replication named **rp_AsyRP** for the **SRCDB** database, based on the schedule to the **DESTDB** destination database:

```
dmSQL> CREATE ASYNC REPLICATION rp_AsyRP
       WITH PRIMARY AS tb_salary
       REPLICATE TO destdb:tb_salary;
       CLEAR AND FLUSH DATA;
```

Users can specify CLEAR DATA, FLUSH DATA, or CLEAR AND FLUSH DATA to perform data initialization for a destination site. When creating replication, DBMaker may use a database link to connect to the destination database for checking and initialization. The action is performed through the current user account or the destination account using the DESTDB destination database or database link name.

After the asynchronous table replication has been created, the job of replicating is moved to the distributor server. The account used to connect to the destination database will be changed to the one specified by the IDENTIFIED option defined in the CREATE SCHEDULE statement.

Any transactions resulting from the asynchronous table replication will be recorded in the distributor message log **ATRP.LOG** under the source database home directory. The distributor message log is a pure text file, and records the start-up and actions of the distributor server.

➲ **Example 2**

Typical content of an ATRP.LOG file:

```
2000/02/09 10:02:30 : start up
2000/02/09 10:02:33 : replicate transactions before 2000/02/09
                      10:02:29 (log:1.856152) to DESTDB
```

NO CASCADE OPTION

The NO CASCADE keywords are optional. It only functions when the replication type is asynchronous. The keyword specifies cascade replication. Commands flow in most organizations from the highest level to lower levels; for example, replicating data from A to B, and then B to C. This is a typical kind of cascade replication. A typical no-cascade model replicates data to B and B replicates data to A. If your data model works like this, you can turn on the NO CASCADE option. The default is setting is CASCADE.

The NO CASCADE option causes table replication to occur only across one site.

➲ **Example**

Replication **Rp1** is from **DB1:t1** to **DB2:t2**, and **Rp2** is from **DB2:t2** to **DB3:t3**. If **Rp1** has the NO CASCADE option set, the changes for **DB1:t1** will be replicated to **DB2:t2**, but **DB2** will stop replicating the same changes to **DB3:t3**. If **Rp1** has the CASCADE option, the changes of **DB1:t1** will be replicated to **DB2:t2**, and then from **DB2:t2** to **DB3:t3**.

Error Handling

In the process of data replication, the distributor may face five kinds of errors: warning, connection, data, statement, and transaction.

WARNING

For example, if a CHAR(10)data type is replicated to a CHAR(5) column type, there will be a data truncation warning. The distributor server will ignore these kinds of errors.

CONNECTION ERRORS

If the distributor server fails to connect to the destination database server, it will abandon the schedule and wait until the next time. All jobs will be kept until then.

DATA ERRORS

Because asynchronous table replication is loosely coupled, it is possible that someone else will update the destination data first. For example, the distributor server wants to insert a record to a destination database, but it already exists. Another situation could be that the distributor server wants to delete a record, but it does not exist. Users can use the STOP ON ERROR option to make the distributor server stop when it incurs such errors. The default behavior of the distributor is to ignore these kinds of errors.

NOTE *The data error mentioned above includes an integrity violation error and affected row error. The integrity violation error calls function SQLError() and returns the '23000'error state. The affected row error calls function SQLRowCount(), the result is not one. For more information on ODBC functions, refer to the ODBC Programmer's Guide.*

STATEMENT ERRORS

When the distributor server faces lock time-out errors when executing a statement, it needs to wait and retry using the RETRY <*n*> TIMES option. The AFTER <*s*> SECONDS option specifies how long to wait before the next try.

TRANSACTION ERRORS

For example, a dead lock causes transactions belonging to transaction errors to be rolled back. If the distributor server faces errors that rollback transactions, it will retry once for each whole transaction. If the outcome still fails, the distributor will leave these actions for the next schedule.

Users can check a text file named **ATRERROR.LOG** for error records that occurred during replication. This file is located in the home directory of the database.

Schedule (Suspending and Resuming)

If we replicate data to a database located in Tokyo, and we know that the database will be shut down on traditional holidays, we can suspend the schedule until the database is ready.

Users with the DBA privilege can suspend and resume schedules. After a schedule is suspended, the distributor server will stop trying to connect for replications.

➲ **Example 1**

To suspend the schedule to the **DESTDB** destination database:

```
dmSQL> SUSPEND SCHEDULE FOR REPLICATION TO destdb;
```

➲ **Example 2**

To restart the schedule for the **DESTDB** destination database:

```
dmSQL> RESUME SCHEDULE FOR REPLICATION TO destdb;
```

Synchronizing a Replication

Users will need synchronized data sometimes; to achieve this DBMaker provides synchronized schedules. The synchronized schedule forces the distributor server to perform local changes to the specified database immediately. Users do not need to wait until the schedule activates the distributor server.

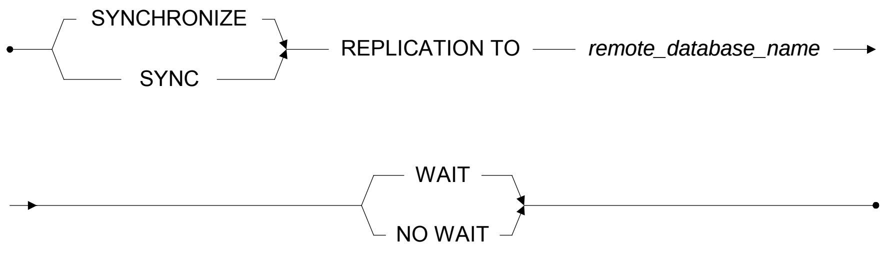


Figure 17-8 Syntax for the SYNCHRONIZE REPLICATION Statement

➲ **Example 1**

To synchronize schedule to the **DESTDB** destination database:

```
dmSQL> SYNC REPLICATION TO destdb WAIT;
```

The default WAIT option causes the distributor server to wait until all changes have been made. The command returns only after the completion of replication. The NO WAIT option instructs the distributor server to perform its job immediately, and the SYNC command will return immediately.

➲ **Example 2**

Using SYNC REPLICATION TO with NO WAIT:

```
dmSQL> SYNC REPLICATION TO destdb NO WAIT;
```

Altering Schedule

After creating a schedule, users having the DBA privilege can alter the attributes of a schedule, including the distributor's activation interval, the account used to connect to the destination database, the RETRY option and STOP/IGNORE ON ERROR option.

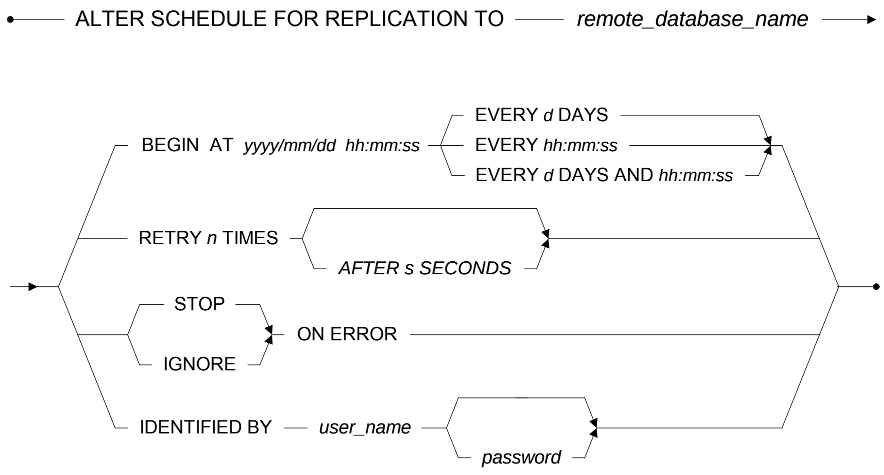


Figure 17-9 Syntax for the ALTER SCHEDULE Statement

➲ **Example 1**

To alter the schedule for replications to the **DESTDB** destination database by adding the IGNORE ON ERROR option:

```
dmSQL> ALTER SCHEDULE FOR REPLICATION TO destdb IGNORE ON ERROR;
```

➲ **Example 2**

```
dmSQL> ALTER SCHEDULE FOR REPLICATION TO destdb IDENTIFIED BY User2 Password2;
dmSQL> ALTER SCHEDULE FOR REPLICATION TO destdb STOP ON ERROR;
dmSQL> ALTER SCHEDULE FOR REPLICATION TO destdb RETRY 5 TIMES AFTER 3 SECONDS;
```

Heterogeneous Asynchronous Table Replication

DBMaker not only allows asynchronous table replication to other DBMaker databases, but also to Oracle, Informix, Sybase, and Microsoft SQL Server databases. This type of replication allows DBMaker to coexist with other databases in a heterogeneous environment, and is known as *heterogeneous table replication*.

DBMaker needs to preprocess the replicated data before sending it to a third-party destination database; users must specify the type of DBMS they are replicating to when creating a schedule in a heterogeneous environment using the ORACLE, INFORMIX, SYBASE, and MICROSOFT keywords.

Due to the way DBMaker makes use of the ODBC Driver Manager to perform asynchronous table replication, the DBMaker server must be located on a computer running Windows, and the definition of the destination database name cannot include a link name. The third-party destination databases may be located on Windows, UNIX, or Linux platforms.

When creating a schedule for heterogeneous table replication, use the WITH NO CHECK keywords to prevent DBMaker from performing schema checking. The user creating the replication must take responsibility for schema checking, and ensure that columns and data types in the destination table are compatible with the columns and data types in the local table.

➲ **Example 1**

To connect to an Oracle database with an ODBC data source name of **orcdb**, user **orcuser** with password **mypassword** enters:

```
dmSQL> CREATE SCHEDULE FOR REPLICATION TO orcdb (ORACLE)
       BEGIN AT 2000/01/01 00:00:00 EVERY 2 DAYS
       WITH NO CHECK
       IDENTIFIED BY orcuser mypassword;
```

The CLEAR DATA, FLUSH DATA, or CLEAR AND FLUSH DATA keywords cannot be used when creating a heterogeneous table replication. Data in the third-party destination database must be manually deleted or inserted to put the table in its initial state before the replication begins. Heterogeneous replication schedules use the same syntax as homogeneous schedules.

➲ **Example 2**

The following shows the heterogeneous replication of table **tb_salary**:

```
dmSQL> CREATE ASYNC REPLICATION rp_HeteroRp
       WITH PRIMARY AS tb_salary
       REPLICATE TO orcdb:orcuser.tb_salary;
```

Express Asynchronous Table Replication

Asynchronous table replication uses ODBC function calls to communicate with destination databases, which might cause poor performance in a WAN environment. To achieve better performance on a WAN, DBMaker provides another mechanism named express asynchronous table replication. DBMaker packs commands into a package to travel the network.

Since other database management systems do not support this protocol, express asynchronous table replications cannot work with on heterogeneous replications. It also does not support the STOP ON ERROR option when creating express schedules.

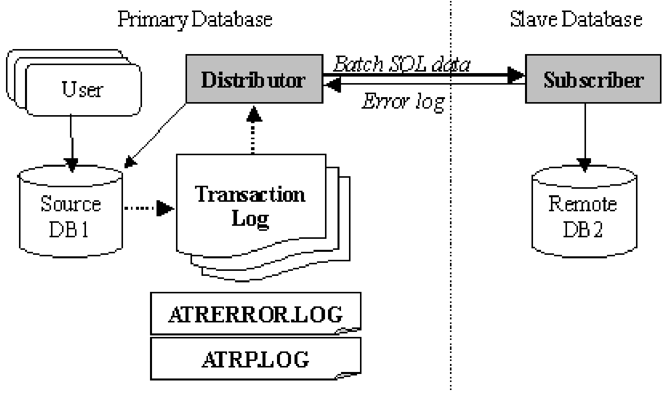


Figure 17-10: Architecture of express asynchronous table replication

There is a distributor server on the source database and a subscriber daemon on the destination database. They co-operate to do the work for express asynchronous table replication. The distributor does not directly apply the changes from the source table to the destination table via ODBC calls. Instead, it only packages the SQL commands and related data applied on the source table, and sends the packet out to the subscriber daemon on the destination database. After getting the packet on the destination database, the subscriber daemon applies the commands to the destination tables.

Express Replication Setup

➲ **To build an express replication:**

1. Enable the distributor server on the source database and subscriber daemon on the destination database(s).
2. Create an express schedule to the destination database(s).
3. Based on the schedule, create the asynchronous table replication(s).

ENABLING SUBSCRIBER DAEMON

In order to start the subscriber daemon, the **DB_EtrPt** keyword in the **dmconfig.ini** file for the destination (subscriber) database should be set. It specifies the port number of the communicating channel between the distributor server and the subscriber daemon.

➲ **Example**

To replicate a table from the **SRCDB** source database to the **DESTDB** destination database by express replication, the subscriber daemon should be started on the destination database.

A sample **dmconfig.ini** file in a destination database follows:

```
[SRCDB]
DB_SvAdr = srcpc    ; tell the target database where the source
DB_PtNum = 22222    ; database is
[DESTDB]
DB_DbDir = /disk3/DBMaker/dest
DB_UsrBb = /disk3/DBMaker/dest/DESTDB.BB 3
DB_UsrDb = /disk3/DBMaker/dest/DESTDB.DB 150
DD_DDBMd = 1
DB_SvAdr = destpc
DB_PtNum = 33333
DB_EtrPt = 44444    ; port number used by Subscriber Daemon
```

In the source database, the **DB_AtrMd** keyword is used to start the distributor server. It is not necessary to let the distributor know which port number the subscriber daemon uses for the destination database.

A sample **dmconfig.ini** file from the source database follows:

```
[SRCDB]
DB_DbDir = /disk1/DBMaker/src
DB_UsrBb = /disk1/DBMaker/src/SRCDB.BB 3
DB_UsrDb = /disk1/DBMaker/src/SRCDB.DB 150
RP_LgDir = /disk1/DBMaker/src/trplog
DB_AtrMd = 1
DD_DDBMd = 1
DB_SvAdr = srcpc
DB_PtNum = 22222
[DESTDB]
DB_SvAdr = destpc
DB_PtNum = 33333
```

SCHEDULE FOR EXPRESS ASYNCHRONOUS TABLE REPLICATION

Express asynchronous table replication uses the EXPRESS option specified in the CREATE SCHEDULE command.

➲ **Example**

To build a schedule for express asynchronous table replication:

```
dmSQL> CREATE SCHEDULE FOR EXPRESS REPLICATION TO destdb
       BEGIN AT 2000/1/1 00:00:00
       EVERY 12:00:00
       IDENTIFIED BY User Password;
```

CREATING EXPRESS ASYNCHRONOUS TABLE REPLICATION

The steps are the same as for creating asynchronous table replications. Refer to the section *Creating Asynchronous Table Replication*, or the *SQL Command and Function Reference* for more information.

17.4 Database Replication

Most enterprises or companies had to put all data in one file server or database in their headquarters, and all terminals needed to connect to the server directly. Under this architecture, the application system might work smoothly if all terminals are in the same building or area. However, the performance was much slower if the remote branch wanted to access the database, because of transmitting speed and network bandwidth.

To share data and increase access speed, DBMaker provides database replication. The database replication will replicate the primary database to the slave database during a set time frame. In other words, when there are changes happening to the primary database like newly added, modified, or deleted data, DBMaker will replicate the changed data into the slave database automatically. There are three advantages to use database replication. First, the performance of application systems will increase since it can access data from the local side directly. Second, the destination database will have the same data modification as the local one; thus, the goal of data sharing can be achieved efficiently. Third, if it fails to connect to the local database for an application, it still could connect to the remote database and continue to work. Although there are advantages to using data replication, more storage for data and more processes to handle the replication are required.

Database Replication Basics

In this section, we will use Figure 17-11 to explain the flow of database replication. Database replication will work with the journal backup server, if you are not familiar with database backup and restoration, please read Chapter 15, *Database Recovery, Backup, and Restoration.*

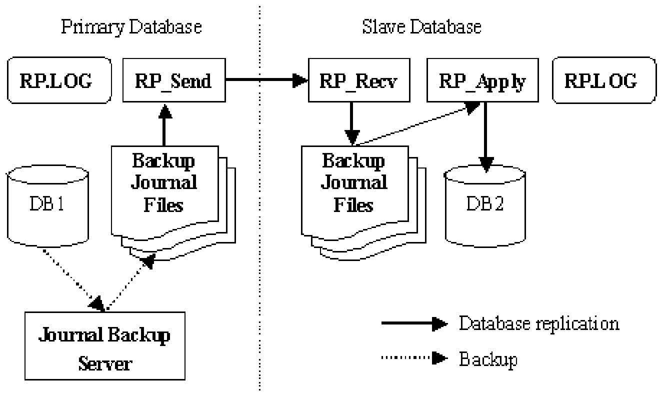


Figure 17-11: Flow of database replication

Database replication is accomplished with 4 servers: journal backup server, RP_SEND server, RP_RECV server, and RP_APPLY server. The journal backup server and the RP_SEND server are started from the primary database. The RP_RECV server and the RP_APPLY server are started from the slave database.

The initial step of database replication is to make the slave database the same as the primary. All subsequent changes, data insertion, deletion, creating schema, etc., have to be made in the primary database first. Then the journal backup server running on the primary database regularly writes changes to the backup journal.

The RP_SEND server periodically sends the backup journal of the primary database to the slave- RP_RECV server. The RP_RECV gives notice to the RP_APPLY to apply the received backup journal to the slave database.

The slave database is read-only for all users, only RP_APPLY can update the slave database.

Database Replication Setup

➲ **To manually set up database replication:**

1. Duplicate the primary database to the slave database
2. Set up the journal backup server and RP_SEND server on the primary database

3. Set up the RP_RECV server and RP_APPLY server on the slave database
4. Start the primary and slave databases
5. Verify the replication log (RP.LOG) and error log (DMERROR.LOG)

FULL BACKUP

As mentioned earlier, the first step of database replication is to make the slave database identical to the primary. Please note the byte ordering of primary and slave database machines must be the same. For example, if the primary database is running on an x86 machine, the slave database machine must be byte ordering-compatible with x86; Sun Sparc's byte ordering is different from an x86's, so the slave database cannot run on a Sparc-based machine or vice versa.

➲ **To duplicate the primary database:**

1. Shut down the primary database
2. Make a copy of the primary database's data files, BLOB files, and journal files
3. Move the copy of files to the machine that will run the slave database
4. Modify the slave database's **dmconfig.ini** configuration file to reflect the corresponding file directory changes

Step 1 and step 2 combined make an offline full backup. Refer to Section *Offline Full Backups* for more detailed information.

Here we will illustrate how to perform an offline full backup manually. In this section, all examples assume an x86 processor and the primary database running on Linux or FreeBSD.

➲ **Example**

To copy the primary database's files, query the SYSFILE system table to get the logical names:

```
dmSQL> SELECT * FROM SYSFILE;
```

An excerpt from the **dmconfig.ini** file to view the physical file names:

```
[MYDB]     ;; Primary Database Configuration, CPU：x86, OS: FreeBSD
DB_DbDir = /home/dbmaker/mydb
```

```
DB_UsrBb = /home/dbmaker/mydb/MYDB.BB 3
DB_UsrDb = /home/dbmaker/mydb/MYDB.DB 150
FILE1 = /home/dbmaker/mydb/data/FILE1.DB 50
FILE2 = /home/dbmaker/mydb/data/FILE2.DB 50
DB_JnFil = JN1.JNL JN2.JNL
...
```

After shutting down the primary database, copy the primary database files to the machine to run the slave database. Here, the slave machine is an x86 running Windows platforms. In the previous example, the files to copy include system files **MYDB.SDB** and **MYDB.SBB**, default user files **MYDB.DB** and **MYDB.BB**, journal files **JN1.JNL** and **JN2.JNL**, and user-defined files **FILE1.DB**, **FILE2.DB**.

Next, copy the primary database section in the **dmconfig.ini** configuration file into the **dmconfig.ini** on the computer serving the slave database. Then modify the slave database **dmconfig.ini** to ensure physical file names logically match, since the actual paths and path name conventions may be different for different machines.

An excerpt from the **dmconfig.ini** of the slave database may look like this:

```
[MYDB]     ;; Slave Database Configuration, CPU：x86, OS: MS Windows NT
DB_DbDir = d:\dbmaker\db\mydb
DB_UsrBb = d:\dbmaker\db\MYDB.BB 3
DB_UsrDb = d:\dbmaker\db\MYDB.DB 150
FILE1    = d:\dbmaker\db\mydb\FILE1.DB 50
FILE2    = d:\dbmaker\db\mydb\FILE2.DB 50
DB_JnFil = JN1.JNL JN2.JNL
...
```

DBMaker supports up to 256 slave databases; if there is more than one slave database, the database administrator will have to repeat the above steps for each slave database.

SETUP PRIMARY DATABASE'S JOURNAL BACKUP SERVER

All changes to a database are logged to the journal files. DBMaker therefore uses backup journal files as the source data for replication. After the journal backup server performs an incremental backup, the RP_SEND server sends backup journal files to the slave database. All the transactions logged on the journal files are then committed to ensure that all the changes to the primary database will also take place on the slave side.

➲ **Example**

Only the primary database needs to start the journal backup server. The **dmconfig.ini** file settings on the primary database server specify the backup directory and schedule:

```
DB_BMode = 1                      ; Database start up in BACKUP-DATA mode
DB_BkSvr = 1                      ; Start up journal backup server
DB_BkDir = /home/dbmaker/mydb/bkdir ; Directory of backup journal files
DB_BkTim = 00/01/01 00:00:00      ; The initial backup time
DB_BkItv = 0-12:00:00             ; Perform journal backup evey 12 hours
```

DB_BMode can be BACKUP-DATA (1) or BACKUP-DATA-AND-BLOB (2) mode. However, even if **DB_BMode** is BACKUP-DATA-AND-BLOB mode, the BACKUP BLOB ON option still must be set for all tablespaces to be replicated; otherwise BLOB data will not be replicated to the slave database.

Data Transmitted and Received

In the process of database replication, there are three resident servers RP_SEND, RP_RECV and RP_APPLY involved (as illustrated in Figure 17-11). RP_SEND is located on the primary database and is responsible for sending backup journal files to the slave side. The RP_RECV and RP_APPLY are on the slave database side. RP_RECV receives backup journal files from the primary database, and RP_APPLY executes the changes according to the journal files. RP_SEND is started and shut down automatically whenever the primary database is started or shut down. RP_RECV and RP_APPLY also start and end simultaneously with the slave database.

After the journal backup server completes an incremental backup, RP_SEND will send all the backup files that have not been sent to the slave database yet from the backup directory (**DB_BkDir**) to the slave database. Meanwhile, RP_RECV on the slave database will receive all the backup files transferred from the primary database, and put these files under the backup directory (**DB_BkDir**) on the slave database. After the files have been received, RP_APPLY will execute the changes recorded in the backup journal files to the slave database, and the flow of database replication will be completed.

Setup RP_SEND Server on the Primary Side

RP_SEND and RP_RECV are responsible for backup journal file transfer and reception, respectively. Thus, RP_SEND must know the IP address and port number of the machine where RP_RECV is located. Note that the port number is specifically for communication between RP_SEND and RP_RECV for data replication, and must be different from the one used by database server. RP_SEND uses the keyword **RP_SlAdr** in **dmconfig.ini** on the primary database to determine where to send replication data.

Syntax for **RP_SlAdr**:

```
RP_SlAdr = {address[:port number]}
```

The default port number is 23001.

➲ **Example 1**

The primary database can support up to 256 slave databases. Use a comma or a space to separate slave databases information. The following example uses three slave databases: **192.168.9.222** (with port number 5100), **Mars** (with port number 5101), and **Scorpio** (with default port number 23001):

```
RP_SlAdr = 192.168.9.222:5100, Mars:5101, Scorpio
```

The initial time and the time interval for RP_SEND to execute database replication can be set because the data replication destination is known. After establishing the schedule, RP_SEND periodically executes data replication.

➲ **Example 2**

To set the initial time to 01/01/2000 AM 01:00 and set the time interval to one day, use the following **dmconfig.ini** settings:

```
DB_SMode = 4                ; Start as primary database,
                            ; and also start up RP_SEND server
RP_BTime = 00/01/01 01:00:00 ; Initial replication time
RP_Iterv = 1-00:00:00       ; Replicate everyday
RP_SlAdr = 192.168.9.222:5100, Mars:5101, Scorpio ; Replicate to 3 machines
RP_ReTry = 3                ; Times to retry if network connection fails
RP_Clear = 1                ; Clear backup journal files after sending
                            ; to the slave side
```

In the configuration file, **RP_BTime** and **RP_Iterv** are used to specify the schedule for data replication. **RP_BTime** is the starting time of sending the backup journal files to the slave side. The format of **RP_BTime** is <*year/month/day hour:minute:second*>. If it is not given, the system will set **RP_BTime** to be the starting time of primary database. The **RP_Iterv** specifies the time interval at which RP_SEND is activated. Its format is <*day-hour:minute:second*>. The default value is one day. The valid value ranges from 0 to 24,855.

NOTE *These values must be set before starting the database.*

RP_ReTry sets how many times to retry if network connection fails. The value of **RP_Clear** determines whether the backup journal files should be cleared after being sent to the slave. Setting **RP_Clear** to 1 clears the backup journal files. The default value is 0. If backup journal files are used for database replication only, clearing those files can reclaim some storage space. However, if a hardware crash occurs, the data from backup journal files is irretrievable and will not be able to be restored. In this case, a full backup against the slave database must be made to restore the primary database. Therefore, if backup journal files will be used for primary database backup as well, **RP_Clear** should be set to 0.

Setup RP_RECV and RP_APPLY Servers on a Slave

In order to start up RP_RECV and RP_APPLY servers, the start-up mode **DB_SMode** of the slave database should be set to 5.

➲ **Example 1**

A slave database can receive replication data only from one primary database; hence, it needs **RP_Primy** to specify the primary database's machine name or address.

```
RP_Primy = FreeBSD       ; Receive replication data from machine 'FreeBSD'
```

➲ **Example 2**

RP_RECV server uses a different port number from **DB_PtNum** to receive data from RP_SEND. For example, suppose there is a slave database located on machine NTPC, with the **RP_PtNum** keyword setting.

```
RP_PtNum = 5100            ; Port number for RP_SEND and RP_RECV connection
DB_PtNum = 3333            ; Port number for user database access
```

➲ **Example 3**

The primary database uses the **RP_SlAdr** keyword to retrieve the slave database's machine name or address and port number.

```
RP_SlAdr = NTPC:5100       ; Slave-side machine name and address
```

In addition, the slave database has to set the **DB_BkDir** backup directory to keep backup journal files received from the primary database. After RP_APPLY has applied changes to the slave database with the backup journal files, DBMaker will automatically remove them.

SLAVE DATABASE IS READ-ONLY

Data in the slave database must be the same as the primary. It cannot accept data definitions (DDL, such as Create Table and Alter Table) and update data (such as INSERT, UPDATE, and DELETE). Thus, we can say that the slave database is read-only.

During the process of database replication, the slave database uses the backup journal files received from the primary database to restore data. The system will not lock data on the primary database during the process of restoring data. Therefore, queries to the slave database are a kind of *dirty read*. In other words, at any given point in time, the same query on both the primary and slave databases may return different values because RP_APPLY is restoring data.

START THE PRIMARY AND SLAVE DATABASES

The start mode of the primary database is different from the slave. If you want to set a database to be a primary, set the start mode to be in primary database mode. On the other hand, if you want to set a database to be a slave, set the start mode to be in slave database mode. Using the **DB_SMode** located in the **dmconfig.ini** file can specify all of the setup options. The start mode **DB_SMode** for the primary database mode is 4. The **DB_SMode** for a slave database mode is 5.

You can start the primary and slave databases separately, in no particular order.

A summary of all **dmconfig.ini** keywords, for replication, mentioned in this section follows.

➲ Example

In the following **dmconfig.ini** file, the primary database name is **FreeBSD**, and the slave database is **NTPC**, using the minimum settings for the primary database.

```
[MYDB] ;; Primary Config, CPU：x86, OS：FreeBSD, Name：FreeBSD
;; Database related settings
DB_DbDir = /home/dbmaker/mydb
DB_UsrBb = /home/dbmaker/mydb/MYDB.BB 3
DB_UsrDb = /home/dbmaker/mydb/MYDB.DB 150
FILE1 = /home/dbmaker/mydb/data/FILE1.DB 50
FILE2 = /home/dbmaker/mydb/data/FILE2.DB 50
DB_JnFil = JN1.JNL JN2.JNL
DB_SvAdr = FreeBSD                  ; Machine name of primary database
DB_PtNum = 3333                     ; Port number of primary database
;; journal backup server related settings
DB_BMode = 1                        ; Database start up in BACKUP-DATA mode
DB_BkSvr = 1                        ; Start up journal backup server
DB_BkDir = /home/dbmaker/mydb/bkdir ; Directory of backup journal files
DB_BkTim = 00/01/01 00:00:00        ; The initial backup time
DB_BkItv = 0-12:00:00               ; Perform journal backup every 12 hours
;; RP_SEND server related settings
DB_SMode = 4                        ; Start as primary database
                                            ; and also start up RP_SEND server
RP_BTime = 00/01/01 01:00:00        ; Initial replication time
RP_Iterv = 1-00:00:00               ; Replicate everyday
RP_SlAdr = NTPC:5100                ; Replicate to NTPC with port no. 5100
RP_ReTry = 3                     ; retry 3 timesif network connection fails
RP_Clear = 1                     ; Clear backup journal files after sending
```

The following is the minimum settings for the slave database:

```
[MYDB];; Slave Config., CPU：x86, OS：MS Windows NT, Name：NTPC
;; Database related settings
DB_DbDir = d:\dbmaker\db\mydb
DB_UsrBb = d:\dbmaker\db\MYDB.BB 3
DB_UsrDb = d:\dbmaker\db\MYDB.DB 150
FILE1    = d:\dbmaker\db\mydb\FILE1.DB 50
FILE2    = d:\dbmaker\db\mydb\FILE2.DB 50
DB_JnFil = JN1.JNL JN2.JNL
DB_SvAdr = NTPC
DB_PtNum = 3333
```

```
;; RP_RECV and RP_APPLY servers related settings
DB_SMode = 5               ; Start as slave database,
                           ; also start up RP_RECV and RP_APPLY servers
RP_Primy = FreeBSD         ; Receive replication data only from this machine
RP_PtNum = 5100            ; Port number for RP_SEND and RP_RECV connection
DB_BkDir = e:\mydb\bkdir   ; Directory of temporary backup journal files
```

EXECUTE DATABASE REPLICATION IMMEDIATELY

We mentioned that the resident server for database replication will detect whether there are changes to data and will replicate the data automatically.

➲ **Example**

Use the dmSQL tool to enter a SQL command to have DBMaker execute the database replication immediately.

```
dmSQL> SET FLUSH;
```

Use this command when you need to synchronize the data between a primary and slave database.

When you execute this command, the journal backup server will immediately execute the incremental backup and backup the current journal files. Then three resident servers (RP_SEND, RP_RECV, and RP_APPLY) will follow the procedure and replicate data.

VERIFY REPLICATION LOG (RP.LOG) AND ERROR LOG (DMERROR.LOG)

In the process of database replication, if there are any network failures or any error messages, the system will generate a log file, DMERROR.LOG, in the current database directory. Error log entries follow the following syntax:

```
yy/mm/dd hh:mm:ss Daemon name:Error number:Error message
```

➲ **Example 1**

An DMERROR.LOG:

```
97/12/31 11:40:59 - RP_SEND:rc = 1503, cannot connect to server 192.72.116.130
97/12/31 11:43:36 - RP_SEND:rc = 1503, cannot connect to server 192.72.116.130
97/12/31 11:45:00 - RP_SEND:rc = 1503, cannot connect to server 192.72.116.130
```

```
97/12/31 11:50:00 - RP_SEND:rc = 1503, cannot connect to server 192.72.116.130
97/12/31 11:50:45 - RP_SEND:rc = 1503, cannot connect to server 192.72.116.130
```

Both primary and slave databases may generate the DMERROR.LOG.

If the replication succeeds, the system will also generate a log file named RP.LOG. The format is:

On the primary database:

```
RP_SEND:RPID id ~ id sent at yy/mm/dd hh:mm:ss
```

On the slave database:

```
RP_RECV:RPID id ~ id sent at yy/mm/dd hh:mm:ss
RP_APPLY:RPID id ~ id applied at yy/mm/dd hh:mm:ss
```

The replication log, RP.LOG, will be on the primary and slave database servers. Every backup journal file has an ID, the RPID. In the above format, RPID indicates which journal file is being processed – RPID in the RP_SEND lines indicate which one is being sent, in the RP_RECV lines they indicate which one is being received, and in the RP_APPLY they indicate which one is being restored. Normally the RPID is the same as the incremental backup file ID.

➲ **Example 2**

RP.LOG on a primary database:

```
RP_SEND : RPID  7 ~ 10 sent to 192.72.116.130 at 97/12/16 15:36:17
RP_SEND : RPID 11 ~ 11 sent to 192.72.116.130 at 97/12/16 15:59:42
RP_SEND : RPID 12 ~ 12 sent to 192.72.116.130 at 97/12/31 11:52:28
```

➲ **Example 3**

RP.LOG on a slave database:

```
RP_RECV : RPID  7 ~ 10 received at 97/12/16 15:35:53
RP_APPLY : RPID  7 ~ 10 applied at 97/12/16 15:35:55
RP_RECV : RPID 11 ~ 11 received at 97/12/16 15:59:18
RP_APPLY : RPID 11 ~ 11 applied at 97/12/16 15:59:18
RP_RECV : RPID 12 ~ 12 received at 97/12/31 11:52:01
RP_APPLY : RPID 12 ~ 12 applied at 97/12/31 11:52:02
```

RP.LOG is used to record all actions performed by three resident servers, RP_SEND, RP_RECV, and RP_APPLY. The file is located in the database directory,

DB_DbDir, on all replication participant databases. Database administrators should monitor these files periodically to ensure that the replication is running smoothly.

For example, if the connection to a remote database always fails, check whether the network works normally or if the remote database was engaged at the time of failure.

JServer Manager Environment Settings

To replicate a database, you need to establish the initial database environment, Use a text editor to modify the **dmconfig.ini** file directly, as shown in the previous section. In this section, the database replication environment is set up using DBMaker's JServer Manager.

PRODUCE A FULL BACKUP

First, generate an offline full backup of the primary database, and then copy the backup to the slave database. For more information on offline full backup, refer to Section 15.5, *Offline Full Backups*.

PRIMARY DATABASE SETUP

To allow the database replication server to function, first set up the database so that incremental backups are being made. Refer to Chapter 15, *Database Recovery, Backup, and Restoration* for more information. Next, set up the environment for the primary database.

➲ **To setup for the primary database environment:**

1. Start the JServer Manager application on the primary database server.
2. Click on the **Setup** button in the **Start Database** window.
3. Click the **Replication** tab in the **Start database advanced settings** window.
4. To enable database replication:
   a) Enter slave database IP and port numbers into the **IP and port number of target database.**
   b) Enter a date and time into the **Begin time of database replication** time fields.
   c) Enter a value into the **Times to retry upon failure** field.

   d) If desired, enable **Remove backup journal files after replication.**
   e) Enter the number of days, hours, minutes, and seconds between each successive database replication in the **Time interval to start database replication** time fields.

5. Click the **Save** button.
6. Click the **Cancel** button to return to the **Start Database** window.

SLAVE DATABASE SETUP

After copying a full backup of the primary database to the slave database, use JServer Manager to setup the configuration of the slave database.

1. Start the JServer Manager application on the slave database server.
2. Click on the **Setup** button in the **Start Database** window.
3. Click the **Replication** tab in the **Start database advanced settings** window.
4. To enable database replication:
   a) Enter the IP Address of Source Database.
   b) Enter a port number for RP_RECV in the **Port number of receive daemon on target DB**.
5. Click the **Save** button.
6. Click the **Cancel** button to return to the **Start Database** window

Database Configuration File

This section is a summary of database replication related keywords used in the **dmconfig.ini** file.

PRIMARY DATABASE CONFIGURATION

Primary database keywords for data replication:

- **DB_SMode** — Setting **DB_SMode** to 4 means that this database will start in primary mode.
- **RP_BTime** — beginning time for the primary database full backup files to be sent to the slave database. The format is yr/mon/day hr:min:sec. The default

value is the starting time of the primary database. For example: 97/12/31 12:00:00.

- **RP_Iterv** — time interval to send the backup journal files. The format is day-hr:min:sec. For instance, 1-12:00:00 means to send the backup files every one and half day. The valid range of days is 0 through 24,855.
- **RP_ReTry** — number of times to retry a failed network connection
- **RP_Clear** — clears journal backup files after sending them. A value of 1 clears the files; a value of 0 does not clear them. The default value is 0. If the value is set to 1, the primary database cannot be restored if there are hardware damages unless the slave database is used to restore it.
- **RP_SlAdr** — the address, or machine number, and the port number of the slave database. DBMaker supports 1 to 8 slave databases for each primary database.

  Syntax for **RP_SlAdr**:

  ```
  RP_SlAdr = {Address[:Port Number]}
  ```

  The primary database needs to start up the journal backup server to perform incremental backups:

- **DB_BMode** — 1 (BACKUP-DATA) or 2 (BACKUP-DATA-AND-BLOB).
- **DB_BkSvr** — Set **DB_BkSvr** to 1 to start the journal backup server
- **DB_BkDir** —directory to store backup journal files
- **DB_BkTim** —starting time of the journal backup server. The format is <*yr/mon/day hr:min:sec*>. The default value is the starting time of the primary database.
- **DB_BkItv** —time interval to perform incremental backups, and the format is <*day-hr:min:sec*>. For instance, 0-12:00:00 means to backup every 12 hours.

SLAVE DATABASE CONFIGURATION

The slave database keywords for data replication:

- **DB_SMode** — Setting **DB_SMode** to 5 starts database in slave mode

- **RP_Primy** — address or machine name of the primary database
- **RP_PtNum** — port number used by RP_RECV. The value must be different from **DB_PtNum,** and it has to be the same as the port number specified in **RP_SlAdr** of the primary database.
- **DB_BkDir** — Directory to store temporary backup journal files received from the primary database. The default directory is **<*database directory*>/backup**.

Database Replication Limitations

Summary of database replication limitations:

- Byte ordering must be the same on the primary and slave machines
- Slave database is read-only
- One primary database supports up to 256 slave databases
- It is doable to restore the master database with backup sequence. However, after the master database restored, the database replication will not continue. If users want to continue replicating the database, all slave databases must have been replaced by new master database, that is to say, users must copy the master database files to replace all slave databases files.
- The replication server may not clear incremental backup files because of full backup, so maybe a large number of files exist in BKDIR directory if next full backup will not be done on a long duration.
- Replication of FILE data type is not currently supported
- To replicate BLOB data, i.e., columns of LONG VARCHAR or LONG VARBINARY data type, set **DB_BMode** to 2 (BACKUP-DATA-AND-BLOB) and remember to set the BACKUP BLOB option while creating tablespaces.

18 Performance Tuning

DBMaker is a highly tunable database system. Tuning DBMaker will increase its performance level to satisfy individual needs. This chapter presents the goals and methods used in the tuning process to demonstrate how to diagnose a system's performance.

18.1 The Tuning Process

Before tuning DBMaker, you must define goals for improving performance. Keep in mind that some goals may conflict. You must decide which of the conflicting goals are most important.

The following outlines some of the goals for tuning DBMaker:

- Improving the performance of SQL statements
- Improving the performance of database applications
- Improving the performance of concurrent processing
- Optimizing resource utilization

After determining the goals, you are ready to begin tuning DBMaker.

Start by performing the following steps:

- Monitor database performance
- Tuning I/O
- Tuning memory allocation

- Tuning concurrent processing
- Monitor database performance and compare with previous statistics

The methods used to perform tuning in each of these steps may have a negative influence on other steps. Following the order shown above can reduce this influence. After performing all of the tuning steps, monitor the performance of DBMaker to see whether the best overall performance has been achieved.

Before tuning DBMaker, make certain that the SQL statements are written efficiently and the database applications employ good design. Inefficient SQL statements or badly designed applications can have a negative influence on database performance that tuning cannot improve. To write efficient statements and applications, refer to the *SQL Command and Function Reference* and the *ODBC Programmer's Guide*.

18.2 Monitoring a Database

This section shows how to monitor information about the status of a database, including resource status, operation status, connection status, and concurrency status. This section also shows how to kill a connection.

NOTE *The user must have DBA authority to login.*

The Monitor Tables

DBMaker stores the database status in four system catalog tables: SYSINFO, SYSUSER, SYSLOCK and SYSWAIT.

The SYSINFO table contains database system values including total DCCA size, available DCCA size, number of maximum transactions and the number of page buffers. It also includes statistics on system actions such as the number of active transactions, the number of started transactions, the number of lock and semaphore requests, the number of physical disk I/O, the number of journal record I/O, and more. Use this table to monitor the database system status, and use the information to tune the database.

The SYSUSER table contains connection information, including connection ID, user name, login name, login IP address, and the number of DML operations that have been executed. Use this table to monitor which users are using a database.

The SYSLOCK table contains information about locked objects, such as the ID of the locked object, lock status, lock granularity, ID of the connection locking this object, and more. Use this table to monitor which objects are being locked by which connection, and which users are locking which objects.

The SYSWAIT table contains information on the wait status of connections, including the ID of the connection that is waiting and the ID of the connection it is waiting for. Use this table to monitor the concurrency status of connections. Once a connection is waiting for those resources locked by an idle or dead connection, you can determine which connection is locking those objects from this table. Then you can kill the idle or dead connection to release the resources.

Browse these four system catalog tables in the same way ordinary tables are browsed.

➲ **Example**

SQL SELECT command used to browse the SYSUSER table:

```
dmSQL> SELECT * FROM SYSUSER;
```

Refer to *System Catalog Reference* for more information on these four system catalog tables.

Killing Connections

A connection should be killed when the connection is holding resources and is idle for a long time, or when the resources are urgently required. In addition, all active connections should be killed before shutting down a database. Before killing a connection, browse the SYSUSER table to determine its connection ID.

➲ **Example 1**

To kill the connection for **Eddie**, retrieve the connection ID first:

```
dmSQL> SELECT CONNECTION_ID FROM SYSUSER WHERE USER_NAME = 'Eddie';

CONNECTION_ID
```

```
============
       352501
```

➲ **Example 2**

Then to kill the connection for Eddie use:

```
dmSQL> KILL CONNECTION 352501;
```

18.3 Tuning I/O

Disk I/O requires the most time in DBMaker.

To avoid disk I/O bottlenecks, perform the following:

- Determine data partitions
- Determine journal file partitions
- Separate journal files and data files onto different disks
- Use raw devices
- Pre-allocate space in an autoextend tablespace
- Turn I/O and checkpoint daemon on

Determining Data Partitions

You can use tablespaces to partition data instead of storing all of the data together. If tablespaces are used properly, DBMaker will have greater performance when performing space management functions or full table scans. Small tables that contain data of a similar nature can be grouped in a single tablespace, but very large tables should be placed in their own tablespace.

You can achieve speed improvement in disk I/O by using disk striping. Striping is the practice of separating consecutive disk sectors so they span several disks. This can be used to divide the data in a large table over several disks. This helps to avoid disk contention that may occur when many processes try to access the same files concurrently.

Determining Journal File Partitions

DBMaker gives the flexibility to use one or more journal files. A single journal file is easier to manage, but using multiple journal files has some advantages as well. If you run DBMaker in backup mode and use the backup server to perform incremental backups, using multiple journal files can improve the performance of incremental backups. Only full journal files are backed up. In addition, spreading multiple journal files across different disks can increase disk I/O performance.

You may determine the size of journal files by examining the needs of transactions. However, if you run DBMaker in backup mode and perform backups according to the journal full status, the journal size will also affect the backup time interval. A larger journal file increases the interval between backups.

Separating Journal Files and Data Files

Separating the journal files and data files onto different disks will increase disk I/O performance, permitting files to be accessed concurrently to some degree. If the disks have different I/O speeds, consider which files to put on the faster disks. In general, if you run on-line transaction processing (OLTP) applications often, put the journal files on the faster disks. However, if you run applications that perform long queries, such as a decision support system, put data files into faster disks.

Using Raw Devices

If you run DBMaker on a UNIX system, construct raw device files to store DBMaker data and journal files. Since DBMaker has a good buffer mechanism, it is much faster to read/write from a raw device than a UNIX file. For more information on how to create a raw device, refer to the operating system manual or consult your system administrator. The one disadvantage of using raw devices is that DBMaker cannot extend tablespaces on them automatically; so more planning is required when using raw device files.

Pre-Allocating Autoextend Tablespaces

DBMaker supports autoextend tablespaces to simplify tablespace management. However, if you are able to estimate the required size of a tablespace, it is better to fix the size when creating the tablespace. This improves performance, as extending pages takes a lot of time. You can extend the pages of a file later by using the alter file command. Pre-allocating the size of a tablespace can also avoid disk full errors when DBMaker attempts to extend a tablespace that already occupies all available disk space.

I/O and Checkpoint Daemons

I/O Daemon

DBMaker has an I/O daemon to periodically write dirty pages from the least recently used page buffers to disk. This helps reduce the overhead incurred when swapping data pages into the page buffers, and increases performance. One configuration parameter in the **dmconfig.ini** file controls the I/O daemon.

DB_IOSvr — enables and disables the I/O daemon. Setting this keyword to a value of 1 enables the I/O daemon, and setting it to a value of 0 disables the I/O daemon.

➲ **Example**

A typical excerpt from the **dmconfig.ini** file:

```
[MYDB]
…
DB_IOSvr = 1
```

MYDB database has 400 (**DB_NBufs**) page buffers in DCCA. Every 10 minutes, the I/O daemon performs the following steps:

- Scan the least recently used page buffers
- Collect the dirty pages during scan processing
- Write these collected dirty pages to disk

CHECKPOINT DAEMON

DBMaker has a checkpoint daemon (based on the I/O daemon) that periodically takes a checkpoint. This helps reduce the time spent waiting for a checkpoint that occurs during a command, when a journal is full, or when starting or shutting down a database. The checkpoint daemon is actually a sub-function of the I/O daemon, which can perform I/O alone, checkpoints alone, or both together.

To turn on the checkpoint daemon, turn on the I/O daemon using the **DB_IoSvr** keyword. If the I/O daemon is activated, it will automatically take a checkpoint every 10 minutes after the database starts successfully.

In fact, the I/O and checkpoint daemon will expend some I/O resources. After starting the database server, error messages generated by the I/O and checkpoint daemon are written to the file **DMERROR.LOG,** the warning message is written to the **DMEVENT.LOG.**

18.4 Tuning Memory Allocation

DBMaker stores information temporarily in memory buffers and permanently on disk. Since it takes much less time to retrieve data from memory than disk, performance will increase if data can be obtained from the memory buffers. The size of each of DBMaker's memory structures will affect the performance of a database. However, performance becomes an issue only if there is not enough memory.

This section focuses on tuning the memory usage for a database. It includes information on how to calculate the required DCCA size, and how to monitor and allocate enough memory for the page buffers, journal buffers and system control area.

➲ **To achieve the best performance, follow the steps in the order shown:**

1. Tune the operating system
2. Tune the DCCA memory size
3. Tune the page buffers
4. Tune the journal buffers
5. Tune the SCA

DBMaker's memory requirement varies according to the applications in use; tune memory allocation after tuning application programs and SQL statements.

Tuning an Operating System

The operating system should be tuned to reduce memory swapping and ensure that the system runs smoothly and efficiently.

Memory swapping between physical memory and the virtual memory file on disk takes a significant amount of time. It is important to have enough physical memory for running processes. Measure the status of an operating system with the operating system utilities. An extremely high page-swapping rate indicates that the amount of physical memory in a system is not large enough. If this is the case, remove any unnecessary processes or add more physical memory to the system.

Tuning DCCA Memory

The *Database Communication and Control Area (DCCA)* is a group of shared memory allocated by DBMaker servers. Every time DBMaker is started, it allocates and initializes the DCCA.

The UNIX client/server model of DBMaker allocates the DCCA from the UNIX shared memory pool. Ensure that the size of the DCCA is smaller than the maximum-shared memory size permitted by the operating system. If the requested size for the DCCA is larger than the operating system limit, refer to the operating system administration manual for information on how to increase the maximum size of shared memory.

DBMaker's shared memory size is 2^{31} pages (2^{31} × PAGE SIZE bytes) on 64 bit platforms. Shared memory on 32-bit platforms is 2 GB bytes. This includes page buffer size, journal buffer size and SCA size.

For 64 bit Linux OS, **shmmax** size must be set when requesting shared memory beyond 2^{31}. Please note, sufficient RAM is necessary. Edit ***/etc/sysctl.conf*** to set **kernel.shmmax = n** where n specifies the maximum share memory size.

➲ **Example**

```
kernel.shmmax = 8405194752.
```

CONFIGURING THE DCCA

The DCCA contains the process communication control blocks, concurrency control blocks, and the cache buffers for data pages, journal blocks and catalogs. DBMaker maintains the concurrency control blocks and communication status of each DBMaker process in the DCCA. Each DBMaker process accesses the same disk data through the cache buffers in the DCCA.

Setting the appropriate parameters in **dmconfig.ini** before starting the database configures the size of each of the DCCA components.

➲ **Example 1**

A sample configuration for the DCCA in the **dmconfig.ini** file:

```
DB_NBuFs = 200
DB_NjnlB = 50
DB_ScaSz = 50
```

DB_NBufs specifies the number of data page buffers (4,096 bytes per buffer if you set the page size to 4 KB), **DB_NJnlB** specifies the number of journal block buffers (4,096 bytes per buffer if you set the page size to 4 KB), and **DB_ScaSz** specifies the size of the SCA in pages (4,096 bytes per page if you set the page size to 4 KB). DBMaker reads these DCCA parameters only when starting a database. To adjust the parameters, terminate the database, modify the values in the **dmconfig.ini** file and restart the database. For more information on setting these parameters, refer to *Keywords in dmconfig.ini*.

The total memory allocation for the DCCA is the sum of the size of **DB_NBufs**, **DB_NJnlB** and **DB_ScaSz**.

➲ **Example 2**

If the page size is 4 KB, to calculate the total size of the DCCA:

```
size of DCCA  = (200 + 50 + 50) * 4 KB
                            = 1200 KB
```

ALLOCATING SUFFICIENT DCCA PHYSICAL MEMORY

The DCCA is the resource most frequently accessed by DBMaker processes. It is important to ensure there is enough physical memory to prevent the operating system from swapping the DCCA to disk too often or it will seriously degrade the performance of a database. Measure the page-swapping rate using operating system utilities.

➲ **Example**

To determine the size of memory allocated for the DCCA from the system table SYSINFO:

```
dmSQL> SELECT INFO, VALUE FROM SYSINFO WHERE INFO = 'DCCA_SIZE'
                                         OR INFO = 'FREE_DCCA_SIZE';
            INFO                          VALUE
============================= ===============================
DCCA_SIZE                     1228800
FREE_DCCA_SIZE                189024
```

DCCA_SIZE — the memory size, in bytes, of the DCCA.

FREE_DCCA_SIZE — the size, in bytes, of free memory remaining in the DCCA.

The free memory in the DCCA is reserved for use by dynamic control blocks, such as lock control blocks.

Usually a larger number of buffers are better for system performance. However, if the DCCA is too large to fit in physical memory, the system performance will degrade. Therefore, it is important to allocate enough memory for the DCCA but still fit the DCCA in physical memory.

Tuning Page Buffer Cache

DBMaker uses the shared memory pool for the data page buffer cache. The buffer cache allows DBMaker to speed up data access and concurrency control. DBMaker automatically configures the number of page buffers by default. Setting the **dmconfig.ini** keyword **DB_Nbufs** to zero allows DBMaker to automatically set the number of page buffers. DBMaker can dynamically adjust the number of page buffers on systems that allow DBMaker to detect physical memory usage. The number will be no less than 500 pages on Windows 95/98, or no less than 2,000 pages for Windows others platforms or UNIX. If DBMaker cannot detect the system's physical memory usage, it will allocate the minimum amount.

Adjusting the size of the page buffers will have the greatest effect on performance. The next sections show how to monitor the buffer cache performance and calculate the buffer hit ratios.

➲ **To improve buffer cache performance:**

1. Update statistics on schema objects
2. Set NOCACHE on large tables
3. Reorganize data in poorly clustered indexes
4. Enlarge cache buffers
5. Reduce the effect of checkpoints

MONITORING PAGE BUFFER CACHE PERFORMANCE

DBMaker places buffer cache access statistics in the SYSINFO system table.

➲ **Example**

To get buffer cache values use the following SQL statements:

```
dmSQL> SELECT INFO, VALUE FROM SYSINFO WHERE INFO = 'NUM_PAGE_BUF';

              INFO                              VALUE
================================ ================================
NUM_PAGE_BUF                     4000

1 rows selected

dmSQL> SELECT INFO, VALUE FROM SYSINFO WHERE INFO = 'NUM_PHYSICAL_READ'
    2>  OR INFO = 'NUM_LOGICAL_READ'
    3>  OR INFO = 'NUM_PHYSICAL_WRITE'
    4>  OR INFO = 'NUM_LOGICAL_WRITE';

              INFO                              VALUE
================================ ================================
NUM_PHYSICAL_READ                  64
NUM_PHYSICAL_WRITE                 1
NUM_LOGICAL_READ                   509
NUM_LOGICAL_WRITE                  0

4 rows selected
```

NUM_PAGE_BUF — number of pages used for data buffer cache

NUM_PHYSICAL_READ — number of pages read from disk

NUM_LOGICAL_READ — number of pages read from the buffer cache

NUM_PHYSICAL_WRITE — number of pages written to disk

NUM_LOGICAL_WRITE — number of pages written to the buffer cache

Calculate the page buffer read/write hit ratio with the following formulas:

$$\text{read hit ratio} = 1 - \left(\frac{\text{NUM\_PHYSICAL\_READ}}{\text{NUM\_LOGICAL\_READ}}\right)$$

$$\text{write hit ratio} = 1 - \left(\frac{\text{NUM\_PHYSICAL\_WRITE}}{\text{NUM\_LOGICAL\_WRITE}}\right)$$

Using the example above, can calculate the read/write hit ratio:

$$
\begin{aligned}
\text{read hit ratio} &= 1-\left(\frac{13207}{331595}\right)\\
&= 0.960\\
&= 96.0\%\\
\text{write hit ratio} &= 1-\left(\frac{7361}{127423}\right)\\
&= 0.942\\
&= 94.2\%
\end{aligned}
$$

Based on the read/write hit ratio, determine how to improve the buffer cache performance. If the hit ratio is too low, tune DBMaker with the methods described in the following subsections.

If the hit ratio is always high, for example higher than 99%, the cache is probably large enough to hold all of the most frequently used pages. In this case, try to reduce the cache size to reserve memory for applications. To ensure good performance, monitor the cache performance before and after making the modifications.

Statistics Values are Outdated

If the read/write hit ratio is too low, it may be that the statistics values of schema objects (tables, indexes, columns) are out of date. The wrong statistics may cause the DBMaker optimizer to use an inefficient plan for SQL statement. If users have inserted large amounts of data into the database after the last time the statistics values were updated, update the values again.

➲ **Example 1**

To update the statistics values for all schema objects:

```
dmSQL> UPDATE STATISTICS;
```

If a database is extremely large, it will take a lot of time to update statistical values for all of the schema objects. An alternative method is to update statistics on specific schema objects that have been modified since the last update, and set the sampling rate.

➲ **Example 2**

To update specific schema objects:

```
dmSQL> UPDATE STATISTICS tabel1, table2, user1.table3 SAMPLE = 30;
```

After successfully updating the statistical values for schema objects, monitor the performance of the page buffer cache with the method specified in *Monitoring Page Buffer Cache Performance*.

Swap Out Cache

DBMaker determines which page buffers to swap with the *Least Recently Used (LRU)* rule. This keeps the most frequently accessed pages in the page buffers and swaps pages that are used less frequently. However, if a large table is browsed all page buffers may be swapped out just to perform one table scan.

For example, in a database with 200 page buffers, if a table with 250 pages is browsed, DBMaker might read all 250 pages into the page buffers and discard the 200 most frequently used pages. In the worst case, DBMaker must read 200 pages from disk when accessing other data after a full table scan. However, if the table cache mode is set to NOCACHE, DBMaker will place the retrieved pages at the end of the LRU chain when a full table scan is performed. Therefore, 199 of the 200 most frequently used pages are still kept in the buffer cache.

Normally the tables with page numbers that exceed the page buffers should be set to NOCACHE. Tables that are not used frequently or with page numbers close to the number of page buffers should also be set to NOCACHE.

➲ **Example 1**

To determine the number of pages and cache mode for a table:

```
dmSQL> SELECT TABLE_OWNER, TABLE_NAME, NUM_PAGE, CACHEMODE FROM SYSTEM.SYSTABLE
WHERE TABLE_OWNER != 'SYSTEM';

TABLE_OWNER     TABLE_NAME      NUM_PAGE   CACHEMODE
=========== ================== ========== =========
BOSS        salary                      5 T
MIS         asset                      45 T
MIS         department                  3 T
MIS         employee                  29 T
MIS         worktime                 450 T
TRADE       customer                 350 T
TRADE       inventory                167 T
TRADE       order                    112 T
TRADE       transaction             1345 F

9 rows selected
```

NUM_PAGE — the number of pages in a table

CACHEMODE — cache mode of full table scan, 'T' means table scan is cacheable, and 'F' means table scan is non-cacheable

In the above sample, the table **TRADE.transaction** is already set to NOCACHE. The other tables still are cacheable. If there are 200 page buffers, the **MIS.worktime** and **TRADE.customer** tables should be set to NOCACHE, and the **TRADE.order** and **TRADE.inventory** tables should be set to NOCACHE if they are rarely used.

➲ **Example 2**

To set the cache mode for a table to NOCACHE:

```
dmSQL> ALTER TABLE MIS.worktime SET NOCACHE ON;
```

If there are no valid indexes for a table, or the predicate in a query references non-indexed columns, DBMaker may also perform a full table scan. To prevent this, try to write SQL statements as efficiently as possible, and make use of indexed columns when possible.

POOR CLUSTERING OF RECORDS

When fetching many records that must be ordered by an index key, or when the predicate references an indexed column, index clustering becomes an important factor that affects the buffer cache performance.

➲ **Example 1**

To select all columns from the **tb_customer** table and sort it using the **custid** primary key:

```
dmSQL> SELECT * FROM tb_customer ORDER BY custid;
```

Suppose there are 3500 records in the **tb_customer** table distributed over 350 pages, and there are 200 page buffers in the system. If the records are clustered using **custid** and the clustering is very good (arranged sequentially on all pages), DBMaker only needs to read 350 pages from disk. On the other hand, if the clustering is poor (no sequential records on the same page), DBMaker may have to read 3,500 pages from disk in the worst case (every record needs a disk read)! To determine the state of an index cluster, update statistics on the table first.

➲ **Example 2**

To build an index called **custid_index** on the **custid** column for the **tb_customer** table:

```
dmSQL> SELECT CLSTR_COUNT FROM SYSTEM.SYSINDEX
                          WHERE TABLE_OWNER = 'TRADE'
                          AND TABLE_NAME = 'tb_customer'
                          AND INDEX_NAME = 'custid_index';
```

Result:

```
CLSTR_COUNT
===========
        385

1 rows selected
```

CLSTR_COUNT—cluster count, the number of data pages that will be fetched by a fully indexed scan with few buffers. DBMaker performs 385 page reads from disk at most, when the full customer table is scanned and results ordered by the **custid** column.

➲ **Example 3**

To fetch the number of pages and rows:

```
dmSQL> SELECT NUM_PAGE,NUM_ROW FROM SYSTEM.SYSTABLE
                               WHERE TABLE_OWNER = 'TRADE'
                               AND TABLE_NAME = 'tb_customer';
```

Result:

```
NUM_PAGE    NUM_ROW
=========== ===========
        350       4375

1 rows selected
```

NUM_PAGE — the number of pages allocated by a table

NUM_ROW — the number of records in a table

With CLSTR_COUNT, NUM_PAGE and NUM_ROW, estimate the clustering factor with the following formula:

$$\text{clustering factor} = \frac{(\text{CLSTR\_COUNT - NUM\_PAGE})}{\text{NUM\_ROW}}$$

In the above example, the clustering factor is 0.8%.

$$\begin{aligned}\text{clustering factor} &= \frac{(385 - 350)}{4375}\\ &= 0.008\\ &= 0.8\%\end{aligned}$$

The clustering factor will be between 0 and 100%. In cases where CLSTR_COUNT is only a little less than NUM_PAGE, it can be treated as zero. If the clustering factor is zero, it means that the data is fully clustered for the index. If the clustering factor is too high, for example larger than 20% (what determines a high rate depends on the table size, average record size, etc.), the index has poor clustering. When DBMaker finds an index that has poor clustering, the DBMaker optimizer may use a full table scan when a SQL statement executed, even if an index scan seems more appropriate.

➲ **To improve poor clustering for a frequently used index:**

1. Unload all data from the table (ordered by the index)
2. Rearrange the unloaded data by order
3. Drop indexes on the table
4. Delete all data in the table
5. Reload the data into the table

6. Recreate indexes on the table

After data reloading, the index is fully clustered. Note however, a table can only be clustered with one index. If one table has many indexes, maintain clustering on the most important index. Usually, the most important index is the primary key. Since unloading and reloading data takes a great deal of time and storage, tune index clustering only on the tables that are very large and frequently browsed.

Low Data Page Buffers

If allocated data page buffers are not enough for database access, add page buffers to the DCCA.

➲ **To modify the number of page buffers:**

1. Terminate the database server
2. Reset **DB_NBufs** in **dmconfig.ini** to a larger value
3. Restart the database

After successfully enlarging the data buffers, run the database for a period and then monitor the buffer cache performance again. If the buffer hit ratio has gone up, adding buffer pages has resulted in a performance improvement. If not, add more pages to the buffer cache or check for other reasons why system performance may be reduced.

Checkpoints Occurring Too Often

If the write hit ratio is much lower than the read hit ratio, the checkpoints may be processed too often.

When a checkpoint is processed, DBMaker will write all dirty page buffers to disk. Since checkpoints require a lot of CPU time, specify the checkpoint daemon to perform a checkpoint on a regular schedule. Another advantage of performing checkpoints periodically is to reduce the recovery time taken by DBMaker to start a database after a system crash.

Except when the checkpoint daemon makes checkpoints regularly, DBMaker will perform checkpoints automatically when running out of free journal space in NON-

BACKUP mode or when an incremental backup is performed in BACKUP mode. To increase the time interval between these kinds of checkpoints, enlarge the journal size.

➲ **Example**

To determine how many checkpoints have been processed:

```
dmSQL> SELECT INFO, VALUE FROM SYSINFO WHERE INFO = 'NUM_CHECKPOINT';

              INFO                             VALUE
================================ ================================
NUM_CHECKPOINT                   26

1 rows selected
```

RE-MONITOR THE BUFFER CACHE PERFORMANCE

After tuning a system with the above methods, monitor the cache buffer performance.

➲ **To monitor cache buffer performance:**

1. Run the database for a period to ensure the information in the database is in a stable state.
2. Reset the statistics values in the SYSINFO system table with the following:

```
dmSQL> SET SYSINFO CLEAR;
```

3. Run the database for a period time.
4. Get the read/write counter from the SYSINFO table and check the hit ratio.

Tuning Journal Buffers

The journal buffers store the most recently used journal blocks. With enough journal buffers, the time required to write journal blocks to disk when updating data and reading journal blocks from disk when rolling back transactions is reduced.

If you seldom run a long transaction that modifies (inserts, deletes, updates) many records, you may skip this section. Otherwise, should determine whether there are sufficient journal buffers for the system. The optimum number of journal buffers is the sum of journal blocks needed by the longest running transactions at the same time.

➲ **To estimate the number of journal buffers, perform the following:**

1. Make sure there is only one active user in the database.
2. Clear the counters in the SYSINFO table with the following command:

```
dmSQL> SET SYSINFO CLEAR;
```

3. Run the transaction that will update the most records.
4. Run the following SQL statement to determine the number of used journal blocks:

```
dmSQL> SELECT INFO, VALUE FROM SYSINFO WHERE INFO =
'NUM_JNL_BLK_WRITE';


              INFO                                VALUE

================================ ================================

NUM_JNL_BLK_WRITE                 626



1 rows selected
```

NOTE *NUM_JNL_BLK_WRITE—the blocks used in this transaction. The journal block size used in this example is 512 bytes. In the above example, you need approximately 41 journal buffer pages (if you set 1 page = 4 KB).*

Another measurement that can be used to determine the journal buffer utilization is the journal buffer flush rate. The journal buffer flush rate is the percentage of journal buffers flushed to disk when DBMaker writes to the journal. If the journal buffer flush rate is too high (for example, more than 50%), increase the number of journal buffers.

➲ **Example**

To calculate the journal buffer flush rate:

```
dmSQL> SELECT INFO, VALUE FROM SYSINFO WHERE INFO = 'NUM_JNL_BLK_WRITE'
```

```
    2>                                    OR INFO = 'NUM_JNL_FRC_WRITE';

              INFO                                VALUE
================================ ================================
NUM_JNL_BLK_WRITE                41438
NUM_JNL_FRC_WRITE                159

2 rows selected
```

NUM_JNL_BLK_WRITE — number of journal blocks written to the buffer

NUM_JNL_FRC_WRITE — number of forced writes of the journal buffers to disk

Suppose **DB_NJnlB** is set to 50 pages (i.e., there are 400 journal buffers). In the example below, the journal flush rate (0.65) is a little too high. Add journal buffers to improve the journal buffer performance.

$$\begin{aligned}\text{journal flush rate} &= \frac{(\text{NUM\_JNL\_BLK\_WRITE/NUM\_JNL\_FRC\_WRITE})}{(\text{DB\_NJNLB}\times 8)}\\ &= \frac{(41438/159)}{(50\times 8)}\\ &= 0.65\end{aligned}$$

Tuning the System Control Area (SCA)

Cache buffers and some control blocks, such as session and transaction information, have a fixed size, and are pre-allocated from the DCCA when a database is started. However, some concurrency control blocks are allocated dynamically from the DCCA while the database is running; their size is specified by **DB_ScaSz**

If a database application gets the error message *database request shared memory exceeds database startup setting* it means that DBMaker cannot dynamically allocate memory from the SCA area. Usually, this error is due to a long transaction using too many locks. If this situation happens often, solve it with the methods illustrated below.

AVOID LONG TRANSACTIONS

A long transaction will occupy many lock control blocks and journal blocks. If there is a long transaction in progress when the above error occurs, analyze whether the transaction can be divided into multiple small transactions.

AVOID EXCESSIVE LOCKS ON LARGE TABLES

Selecting many records from a large table using an index scan requires many lock resources. To decrease the amount of lock resources used by the transaction, escalate the lock mode before performing the table scan.

For example, if the table's default lock mode is row, escalate the default lock mode to page or table. Although this will reduce the resources used, it will also sacrifice concurrency to some degree.

INCREASE THE SCA SIZE

If both of the above conditions have not occurred, increase the size of the SCA. Reset the value of **DB_ScaSz** in **dmconfig.ini** to a larger value and then restart the database.

Tuning the Catalog Cache

DBMaker stores the catalog cache in the SCA. If schema objects are seldom modified, turn on the data dictionary turbo mode by setting **DB_Turbo** = 1 in the **dmconfig.ini** file. When turbo mode is on, DBMaker will extend the lifetime of the catalog cache. This can improve the performance of on-line transaction processing (OLTP) programs.

18.5 Tuning Concurrent Processing

Resource contention occurs in a multi-user database system when more than one process tries to access the same database resources simultaneously. This can also lead to a situation known as a deadlock, which occurs when two or more processes wait for each other. Resource contention causes processes to wait for access to a database resource, reducing system performance.

DBMaker provides the following methods to detect and reduce resource contention:

- Reducing lock contention
- Limiting the number of processes
- Setting CPU Affinity
- Setting Job Priority

Reducing Lock Contention

When accessing data in a database, DBMaker processes will lock the target objects (records, pages, tables) automatically. When two processes want to lock the same object, one must wait. If more than two processes wait for the other processes to release the lock, a deadlock occurs. When a deadlock occurs, DBMaker will sacrifice the last transaction that helped cause the deadlock by rolling it back. Deadlock reduces system performance. Monitor the lock statistics to avoid a deadlock in DBMaker.

➲ **Example**

To view deadlock statistics:

```
dmSQL> SELECT INFO, VALUE FROM SYSINFO WHERE INFO = 'NUM_LOCK_REQUEST'
    2> OR INFO = 'NUM_DEADLOCK'
    3> OR INFO = 'NUM_STARTED_TRANX';

              INFO                           VALUE
================================ ================================
NUM_STARTED_TRANX                33
NUM_LOCK_REQUEST                 173
```

```
NUM_DEADLOCK                    0

3 rows selected
```

NUM_LOCK_REQUEST — number of lock requests

NUM_DEADLOCK — number of deadlocks

NUM_STARTED_TRANX — number of transactions issued

In the above example, on average one transaction is in deadlock per 51 (9287/181) transactions and one transaction requests approximately 83 (772967/9287) locks.

If the deadlock frequency is high, examine the schema design, SQL statements and applications. Setting the table default lock mode lower, such as ROW lock, could reduce the lock contention, but it will require more lock resources.

Another method is to use the browse mode to read a table on a long query if the data does not need to remain consistent after the point in time that it was retrieved. This is useful when viewing the data or performing calculations using the data while not performing any updates. It provides a snapshot of the requested data at a particular point in time, but with the benefit of increased concurrency and fewer lock resources consumed, because locks are freed as soon as the data is read.

Limiting the Number of Processes

DBMaker allows up to 4,800 simultaneous session connections to a server. If server resources, (such as memory, CPU power) are not sufficient, limit the maximum number of connections to avoid resource contention. The configuration parameter **DB_MaxCo** affects the maximum number of connections in the database.

When a database is initially created, the journal file is formatted for a specific number of connections. The journal file needs to be able to preserve a transaction information array for each connection. The number of connections available according to the journal file is also known as the *hard connection number*. This value is determined by the value of **DB_MaxCo** when the database is created. The *hard connection number* has a minimum value of 240, a maximum value of 4840, and must be a multiple of

40. If **DB_MaxCo** is set to a value that is not a multiple of 40, then the *hard connection number* is rounded up to a value that is a multiple of 40. The hard connection number is a limitation of the journal file, therefore, to change it the user must reconfigure **DB_MaxCo** and started in new journal mode (**DB_SMode**=2).

The following formula is used to calculate the number of *hard connections*.

Hard connection numbers = (**DB_MaxCo** + reserved connection numbers + 40 - 1)/40 * 40

At present DBMaker owns 20 reserved connections to support internal connections, such as daemon, administration use, etc.

The hard connection number does not directly affect the size of the DCCA. This is determined by a value known as the *soft connection number*. The larger connection number means more memory at both server and client is necessary, so even if the *hard connection number* of a database is large, users can start the database with a smaller soft connection number to save memory. The *soft connection number* is determined by the value of **DB_MaxCo** when the database starts. The soft connection number determines the number of connections that the DCCA will support, and consequently the memory usage of the DCCA. The soft connection may be any value less than or equal to the hard connection value. To change the soft connection number, restart the database normally after changing **DB_MaxCo**.

The following formula is used to calculate the number of *soft connections*.

Soft connection numbers = **DB_MaxCo** + reserved connection numbers.

➲ **Example 1**

In the following configuration file, the hard connection number for **DB1** is 240. For database **DB2**, it is 1120.

```
[DB1]
DB_MaxCo = 50    ;; the hard connection number is 240
                 ;; the soft connection number is 70
[DB2]
DB_MaxCo = 1100  ;; the hard connection number is 1120
                 ;; the soft connection number is 1120
```

➲ **Example 2**

After starting the database successfully, the new hard connection number for **DB1** becomes 320.

```
[DB1]
DB_SMode = 2                ;; startup with new journal file
DB_MaxCo = 280             ;; the new hard connection number is 320
```

➲ **Example 3**

Assuming **DB2** has already been created as in example 1, the following entry in the **dmconfig.ini** file will result in a hard connection number of 1120 and a soft connection number of 40.

```
[DB2]
DB_SMode = 1                ;; normal start
DB_MaxCo = 20              ;; the new soft connection number is 40
```

Setting CPU Affinity

To improve the multitasking's performance, operation system dispatch processes and threads among different CPUs. Although it is efficient from the operating system point of view, this activity will reduce DBMaker's performance under heavy system loads, as each processor cache is repeatedly reloaded with data. If each process runs on the CPU which it run on last time, DBMaker will get better performance. At this time, user can get the process's affinity by querying from sysuser, and then use the system stored procedure SETAFFINITY to set new affinity mask. In this way, the process will only run on the specific CPU.

Different from the old Operation System, CPU affinity is a feature of modern Operation Systems, such as Linux 2.6, Windows NT and Windows 98. There are two kinds of CPU affinity: one is soft affinity that user can't change, and the other is hard affinity that user can change.

CPU affinity is defined by affinity mask. Affinity mask is a bit vector in which each bit represents one processor. DBMaker define affinity mask as char(64), so it most set 64 CPU.

➲ **Example 1**

There are affinity mask values for a 8-CPU system. (The continuous zeros in high position are omitted.)

Decimal value	Binary bit mask	Allow run on CPU
1	'1'	0
3	'11'	0 and 1
7	'111'	0, 1, and 2
15	'1111'	0, 1, 2, and 3
31	'11111'	0, 1, 2, 3, and 4
63	'111111'	0, 1, 2, 3, 4, and 5
127	'1111111'	0, 1, 2, 3, 4, 5, and 6
255	'11111111'	0, 1, 2, 3, 4, 5, 6, and 7

Using GETCPUNUMBER, SETAFFINITY, users can get the current system state and set a connection's CPU affinity without restarting DBMaker during runtime.

The prototype for GETCPUNUMBER is:

```
GETCPUNUMBER (INT CPU_NUMBER OUTPUT)
```

CPU_NUMBER: output parameter, the number of logical processors in the machine

The prototype for SETAFFINITY is:

```
SETAFFINITY (INT CONNECTION_ID INPUT, CHAR(64) AFFINITY_MASK INPUT)
```

CONNECTION_ID: Input parameter, the ID of connections or servers. User can get it with "select connection_id from sysuser" or checking system monitor. It is thread's ID in windows and process ID in Unix-like system.

AFFINITY_MASK: input parameter, CPU affinity mask. The valid affinity mask is composed of '1' or '0'. '1' means the CPU is valid for connection; '0' means the CPU is invalid for connection.

➲ **Example 2**

Users must get some system information before setting CPU affinity, such as the number of CPU on the server, the CPU usage of every connection, correct affinity mask.

To get the number of CPU by calling GETCPUNUMBER:

```
dmSQL> CALL GETCPUNUMBER(?);
```

To get the CPU usage of every connection, correct affinity mask:

```
dmSQL> SELECT connection_id, affinity_mask, priority_level, cpu_usage FROM
sysuser;
```

To set CPU affinity for sysuser and allow the connection running on CPU 0 and 1:

```
dmSQL> SELECT connection_id , user_name FROM sysuser;

   CONNECT*             USER_NAME
===============   ========================
         30420   BACKUP_SERVER
         30418   SYSADM

2 rows selected
dmSQL> CALL SETAFFINITY(30418,'11');
```

NOTE *Only sysadm can call SETAFFINITY.*

To get CPU affinity mask by querying sysuser for a precise connection:

```
dmSQL> SELECT affinity_mask FROM sysuser WHERE connection_id = ?;
```

Setting Job Priority

The priority of processors and threads is a basic feature in multi-tasking operation systems. The scheduler will check system load and change a job's priority level if need be. DBMaker allow user setting one connection's priority to improve the performance of the whole system. For example, there is a long time job and it takes up much of CPU time, as a result, other connections will wait for a long time. If users reduce the priority of the long job, other connections will get more CPU time to execute, and then the performance of the whole system is improved. Users can set new values for some important connections to improve the connections' performance, meanwhile other connections may get low performance.

Using GETCPUNUMBER, SETPRIORITY, users can get the current system state and set a connection's priority without restarting DBMaker during runtime.

The prototype for SETPRIORITY is:

```
SETPRIORITY (INT CONNECTION_ID INPUT,INT PRIORITY_LEVEL INPUT)
```

CONNECTION_ID: Input parameter, the ID of connections or servers. User can get it with "select connection_id from sysuser" or checking system monitor. It is thread's ID in windows and process ID in Unix-like system.

PRIORITY_LEVEL: input parameter, there are five levels, the normal and default priority level is three. Valid priority levels are '1', '2', '3', '4' and '5'. '1' means lowest priority; '2' means lower priority; '3' means normal priority; '4' means higher priority; '5' means highest priority.

NOTE *Users can't set a higher priority in Linux because it need root privilege, So you can only set lower level in Linux, but there is no limits in Windows.*

➲ **Example**

User must get some system information before setting CPU affinity, such as the number of CPU on the server, the CPU usage of every connection, the priority.

To get the number of CPU by calling GETCPUNUMBER:

```
dmSQL> CALL GETCPUNUMBER(?);
```

To get the CPU usage of every connection, the priority:

```
dmSQL> SELECT connection_id, affinity_mask, priority_level, cpu_usage FROM
sysuser;
```

To set priority level for sysuser:

```
dmSQL> SELECT connection_id, user_name FROM sysuser;

    CONNECT*              USER_NAME
================   ========================
          30420    BACKUP_SERVER
          30418    SYSADM

2 rows selected
dmSQL> CALL SETPRIORITY(30418,3);
```

NOTE *Only SYSADM can call SETPRIORITY.*

To get the priority level by querying sysuser for a precise connection:

```
dmSQL> SELECT priority_level FROM sysuser WHERE connection_id = ?;
```

19 Query Optimization

This chapter is an introduction to the query optimizer for DBMaker. The query optimizer makes a query on SQL commands much faster and efficient by choosing the best internal execution method.

The contents in this chapter cover the following topics:

- What is query optimization and why do we need it? When you understand the goal of query optimization, you will find the role it plays in a SQL query.
- What is the Query Execution Plan (QEP) and how do you read a QEP? When you know the QEP, you will learn how DBMaker executes a SQL query command.
- How does the query optimizer operate? When you understand the way the query optimizer searches for a QEP, you can help it to find a more efficient QEP by rewriting an equivalent SQL query.
- What is the cost function? When you know how much time it takes for an operation in QEP, you will learn how the query optimizer chooses a proper operation. Use some commands provided by DBMaker to help the query optimizer find a better operation.
- What are the statistics values and where have these values been used? When you understand the usage of statistics values in query optimization, you will see the reason why the query optimizer chooses a particular execution plan.

- How can the execution speed of a query be accelerated? When you know how to write an efficient query, you can enhance the execution efficiency by rewriting the query command.

19.1 What is Query Optimization

Data Manipulation Language (DML) commands, such as SELECT, INSERT, DELETE, and UPDATE, are a very important stage in query optimization. DBMaker may have several execution methods for one SQL query. The goal of query optimization is to find the most efficient execution plan. The main job of the query optimizer is to decide each operation, and the order in which it operates.

➲ **To find the most efficient operation:**

1. *Read the data from a table* -- can be read with a sequential or index scan.
2. *Join tables* -- tables can be joined with a nested loop or a sort merge join.
3. *Sort* -- when is a sort needed, before an operation or after it? Can sorting be avoided by using an alternative solution?

The query optimizer must estimate the number of rows affected by outer joins in order to optimize the sequence of the tables to be joined. Some database users get familiar enough with the data characteristics that they become more efficient than the query optimizer at executing a query.

Query optimizer for DBMaker will estimate all possible execution plans for each plan, compute the number of rows, how many disk page I/O required, and CPU time it takes for a single table. From the above factors mentioned, find a plan with the lowest cost.

When DBMaker seeks for a query execution plan, it will consider some major operations:

- ***Table scan*** -- or called sequential scan, which means receiving each row from data pages from a database in sequential order
- ***Index scans*** -- the order to retrieve data is referenced by the address of the data page that is pointed to by the index page

- *Nested join* -- compare two or more tables, row by row, to achieve the goal of merging
- *Merge join* -- sort two tables and compare row by row to achieve merging
- *Sort* -- executes sort
- *Temporary table* -- in the process of query execution, establish a temporary table

➲ **Example 1**

Using the ORDER BY clause to sort:

```
dmSQL> SELECT * FROM tb_staff, tb_salary WHERE
tb_salary.basepay=3000 AND tb_staff.id = tb_salary.id ORDER BY
tb_staff.name;
```

➲ **Example 2**

Query Execution Plan 1:

```
sort tb_staff.name
    merge join tb_staff.id = tb_salary.id
       index scan tb_staff on idx_ID(id)
       sort tb_salary.id
          table scan tb_salary, filter tb_salary.basepay=3000;
```

➲ **Example 3**

Query Execution Plan 2:

```
nested join
    index scan tb_staff on idx_name(name)
    table scan tb_salary, filter tb_salary.basepay and tb_staff.id =
tb_salary.id;
```

19.2 How Does the Optimizer Operate

When the optimizer for DBMaker performs the optimization process for a query, it will use the following rules:

- Analyze the query, and then cut the WHERE predicate into several factors
- Search all possible execution sequences and the join sequence

- Decide whether to use a nested or sort merge join
- Decide whether to use a table or index scan
- Decide the sort order

Input of Optimizer

The precision of estimation is the critical factor deciding whether the optimizer will be successful or not. However, there is only finite information for the optimizer to estimate the time required for an operation, only a small part compared with the actual execution time. All the information needed by the optimizer comes from the system catalog tables. To make sure that this information is useful and not out of date, use the UPDATE STATISTICS command. Refer to section 19.4, *Statistics* for more information.

The data listed in system catalog tables includes:

- Number of rows in a table
- Number of data pages used by a table
- Average bytes for a row in a table
- Average bytes that a column uses
- The distinct value of each index column
- The second maximum and minimum value for each column; the reason that we do not choose the maximum and minimum value is to avoid special large and small values from affecting precision
- Number of index scan pages occupied by the B-tree index
- Number of levels (height) in the B-tree index
- Number of leaf pages in the B-tree index
- Cluster count for the B-tree index

The premise for the optimizer to use this information is that that the data values are distributed uniformly. If the distribution of data is askew, not uniform, the optimizer will choose a poor plan.

Factors

The first job of the optimizer is to examine all expressions in the WHERE predicate. We can decompose these expressions into several smaller independent expressions called factors.

➲ **Example 1**

The optimizer will decompose the WHERE predicate into two factors **tb_staff.id = tb_salary.id** and **tb_salary.basepay=3000**:

```
dmSQL> SELECT * FROM tb_staff, tb_salary WHERE tb_staff.id = tb_salary.id AND
tb_salary.basepay=3000;
```

➲ **Example 2**

Using the WHERE predicate for one factor **tb_staff.id = tb_salary.id** or **tb_salary.basepay=3000**:

```
dmSQL> SELECT * FROM tb_staff, tb_salary WHERE tb_staff.id = tb_salary.id OR
tb_salary.basepay=3000;
```

➲ **Example 3**

Using the WHERE predicate for two factors **tb_staff.id = tb_salary.id** and **tb_salary.basepay=3000** or **tb_staff.name='joy'**:

```
dmSQL> SELECT * FROM tb_staff, tb_salary WHERE tb_staff.id = tb_salary.id AND
(tb_salary.basepay=3000 OR tb_staff.name='joy');
```

➲ **Example 4**

Using the WHERE predicate for one factor **tb_staff.id = tb_salary.id** or **tb_staff.name='joy'**:

```
dmSQL> SELECT * FROM tb_staff, tb_salary WHERE tb_staff.id = tb_salary.id AND
tb_salary.basepay=3000 OR tb_staff.name='joy';
```

From the above example, we find that when the expression contains the binary operation **and** that it can be divided into different factors. However, when it contains the binary operation **or**, the decomposition is not allowed.

To find the factors, the optimizer needs to estimate the selectivity of each factor. The selectivity is the ratio of data filtered by each factor; its value is between 0 and 1. Table1 contains 100 rows.

➲ **Example 5**

Using 5 rows for query on table **tb_staff**, **tb_staff.id = 3** is 5 / 100, that is 0.05

```
dmSQL> SELECT * FROM tb_staff WHERE tb_staff.id=3;
```

If there is more than one factor in an expression, then the selectivity of this expression is the product of these factors because they are independent of each other.

Join Sequence

The join sequence is the access order of the original table to be merged. Different join sequences will produce different execution sequences and different execution times. However, no matter how we execute, we will always retrieve the correct result.

➲ **Example 1**

```
dmSQL> SELECT * FROM tb_staff, tb_salary WHERE tb_staff.id = tb_salary.id;
```

Query Execution Plan 1:

```
nested join
     table scan tb_staff
     table scan tb_salary, filter tb_staff.id = tb_salary.id;
```

Query Execution Plan 2:

```
nested join
     table scan tb_salary
     index scan tb_staff on tb_staff (id), filter tb_staff.id = tb_salary.id;
```

➲ **Example 2**

```
dmSQL> SELECT * FROM tb_staff, tb_salary,tb_dept WHERE tb_staff.id = tb_salary.id
AND tb_salary.id=tb_dept.id;
```

Query Execution result, there will be 3! (=6) join sequences:

```
(tb_staff, tb_salary), tb_dept
(tb_staff, tb_dept), tb_salary
(tb_salary, tb_staff), tb_dept
(tb_salary, tb_dept), tb_staff
(tb_dept, tb_staff), tb_salary
(tb_dept, tb_salary), tb_staff
```

DBMaker will search all these join sequences, then compute the cost, and choose the best one.

Nested Join and Merge Join

Nested and merge joins are the two join methods supported by DBMaker.

- Nested join uses nested loop over two layers to accomplish the join purpose. The analysis algorithm for its time complexity is $O(n^2)$.
- Merge join will sort two tables in advance, and then merge these two tables with the sorted order row by row. The time complexity for sort is $O(n \times \log(n))$. The data that has already been sorted has the time complexity to perform a join of $O(n)$. Sort merge join can only be used for equal joins.

From the view of the time complexity, merge join is better than a nested join. However, there are still exceptions; for example, the difference in the number of rows in two tables is great. Regardless, the optimizer will decide the best way to perform a join with cost functions and statistical values.

Table Scan and Index scan

Table scans acquire rows from a table sequentially, row by row. For instance, if a user wants to find all rows in a table that match the condition **age > 50**, then it will receive each row from each data page, and then compare each row with the matched condition to retrieve the desired one.

Another scan type is an index scan, which builds the index on columns in a table, then finds all required data by referencing the index. The index method used by DBMaker is a B-tree, and the precondition to use an index scan is to build an index on the column for use with the predicate.

Sort

Another important operation of the query optimizer is to determine how to sort, before or after a join or to try avoiding a sort.

➲ **Example**

Creating a sort:

```
dmSQL> SELECT * FROM tb_staff, tb_salary WHERE tb_staff.id=tb_salary.id ORDER BY
tb_staff.basepay;
```

Query Execution Plan 1, the optimizer performs sorting after merging:

```
sort tb_staff.basepay
      merge join tb_staff.id=tb_salary.id
           index scan tb_staff on idx_id(id)
           sort tb_salary.id
               table scan tb_salary;
```

Query Execution Plan 2, the optimizer performs sorting before merging:

```
nested join
      index scan tb_staff on idx_base(basepay)
      table scan tb_salary, filter tb_staff.id=tb_salary.id;
```

19.3 Time Cost of a Query

Time to read data from disk and time to compare the column values are two major parts in performing a query.

CPU Cost

The database server must process data in memory. It has to read a row into memory, and then use a filter expression for testing. It needs to load data from two tables into memory first and then test their join condition. In addition, the database server has to collect data for the selected columns from each row. More time is required for sorting when wild cards such as **like** and **match** are used in SQL statements.

I/O Cost

It takes much more time to read a row from the disk than to check a row in memory, so one of the main purposes of the optimizer is to reduce disk I/O.

The basic unit of disk storage that a database server processes is called a page. A page is composed of clustered blocks, and the size of a page is related to the database server. DBMaker supports 4 KB, 8 KB, 16 KB or 32 KB sizes. The row capacity of a page is related to the size of a row. In the case of the page size is 4 KB, there are usually from 10 to 100 rows in a data page. The entity of an index page contains a key value and a four-byte pointer. There are usually from 100 to 1000 entries in an index page.

The database server needs memory to store copies of disk page reads from the disk for processing. Due to memory limitations, some pages might be reread. The memory is called a page buffer. If the page needed happens to be in the page buffer, the server will not read the row from the disk anymore, and it will increase performance. The database server and the operating system decide the size of a disk page and the number of page buffers. The real cost to read a page is a variable and hard to estimate.

Buffers are combinations of the following factors:

- **Buffers** – It is possible for the target page to be in the page buffer. The access cost can be almost omitted in this condition.
- **Contention** – If there is more than one application program attempting to use hardware devices, such as a disk, requests from the database server will be delayed.
- **Seek time** – This is the most time-consuming operation for a disk. It means the elapsed time to move the read/write head to the location of the desired data. It is affected by the speed of the disk and the initial position of the disk read/write head. The variation also depends largely on seek time.
- **Latency time** – Also known as the rotation delay time, is related to the speed of the disk and location of the read/write head.

Cost of Table Scan

The time spent to scan all data in a table. Whether there are predicates in the query or not, it needs to compare all data contained in the pages. The cost of a table scan equals to the number of data pages.

Cost of Index Scan

Index scans read data through B-tree index pages. There are two kinds of index scans: one will read data pages referenced by the B-tree, and the other will read data directly from the index leaf; known as a leaf scan.

➲ **Example**

Using table **tb_staff** with columns **id** and **name**, and scanning an index built on **id**:

```
dmSQL> SELECT * FROM tb_staff WHERE id > 0;
```

Alternatively use:

```
dmSQL> SELECT id FROM tb_staff WHERE id > 0;
```

We can use a leaf scan and there will be no need to read data from the data pages because the leaf pages contain all desired data.

- When all data is read, the cost of an index scan is:

  cost = B-tree level I/O + number of leaf page I/O + cluster count

- When all data is read but only a leaf scan required, the cost of the index scan is:

  cost = B tree level I/O + no. of leaf page I/O

- When a row is read, the cost of an index scan is:

  cost = B tree level I/O + one leaf page I/O + one data page I/O

- When a row is read but only a leaf scan required, the cost of an index scan is:

  cost = B tree level I/O + one leaf page I/O

- When partial data is read, the cost of an index scan is:

  cost = B tree level I/O + (no. of leaf page × S) + (cluster count × S), where S equals selectivity

Cost of Sort

Reading data from disk into memory is the only thing that takes more time. The computing cost is proportional to **$c \times w \times n \times \log_2(n)$**, where c is the number of columns being sorted, w is the bytes of the sorted key, and n is the number of rows being sorted.

Cost of Nested Join

More than two loops are required to access data pages for a nested join. In a nested join, the outer table is different from the inner. Generally, the cost of a nested join is: outer table I/O + inner table I/O × number of rows in outer table

Cost of Merge Join

It is necessary to sort tables before performing a merge join. Suppose two tables have already been sorted for merging with the merge keys. The cost of the merge join is the sum of the I/O for these two tables. If sorting is not performed on the merge keys, the cost of the sorting will still need to be added.

19.4 Statistics

Statistics represent the amount and distribution of data for a table. They provide the information for the cost functions to find the best access plan. However, the statistics will be out of date if data in the table is being inserted, deleted, or updated. The UPDATE STATISTICS command should be used to update statistical values and find real time statistics to enhance the efficiency of a query.

Types of Statistics

DBMaker will collect the following statistics:

FOR A TABLE

- **nPg** – number of data pages
- **nRow** – number of data rows
- **avLen** – average bytes for a row

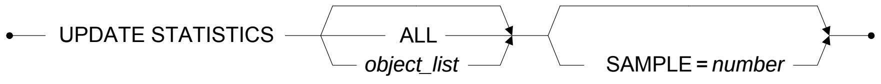
FOR A COLUMN

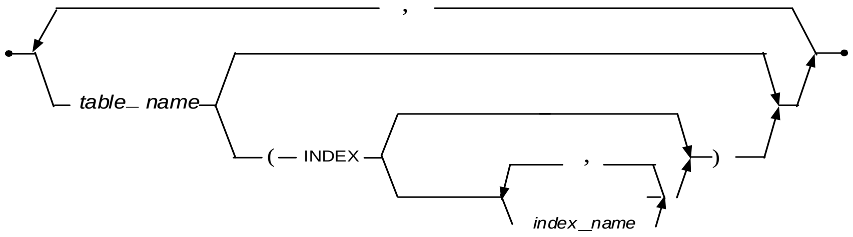
- **distVal** – number of distinct values
- **avLen** – average bytes for each column
- **loVal** – the second minimum value for a column
- **hiVal** – the second maximum value for a column

FOR AN INDEX

- **nPg** – number of index pages
- **nLevel** – number of levels in an index tree
- **nLeaf** – number of leaves in an index tree
- **distKey** – number of distinct keys
- **distC1** – number of distinct keys for the first index column
- **distC2** – number of distinct keys for the first two index columns
- **distC3** – number of distinct keys for the first three index columns
- **nPgKey** – number of index pages for each key
- **cCount** – number of clusters counted; the number of data pages accessed through an index

UPDATE STATISTICS Syntax





The *object_list* Clause

Figure 19-1 Syntax for the UPDATE STATISTICS Statement

- **ALL** – means forcibly update the statistics values for all schema objects.
- **SAMPLE** – means the sampling rate expressed as a percentage of the whole, an integer between 1 and 100.

➲ **Example 1**

To update the statistics values for all schema objects:

```
dmSQL> UPDATE STATISTICS;
```

If a database is extremely large, it will take a lot of time to update statistical values for all of the schema objects. An alternative method is to update statistics on specific schema objects that have been modified since the last update, and set the sampling rate.

➲ **Example 2**

To forcibly update the statistics values for all schema objects:

```
dmSQL> UPDATE STATISTICS ALL;
```

➲ **Example 3**

To update statistics for all tables including columns, indexes, and system tables with the default sampling rate value of 30:

```
dmSQL> UPDATE STATISTICS SAMPLE=30;
```

➲ **Example 4**

To update statistics for the **jeff.tb_staff** table:

```
dmSQL> UPDATE STATISTICS jeff.tb_staff;
```

➲ **Example 5**

To update statistics for the **jeff.tb_staff**, and **jeff.tb_dept** tables:

```
dmSQL> UPDATE STATISTICS jeff.tb_staff, jeff.tb_dept;
```

Auto Update Statistics Daemon

DBMaker provides a daemon to automatically update statistics for the entire database. Not all statistics on all tables are regenerated, but rather an optimum sample ratio based on how recently statistics were updated for a table and how much the table has been changed since the last time its statistics were updated. Users can select update statistics mode by themselves, set different sample ratio for different table, and change the initial time and interval of update statistics. Users can query system table SYSUSER to get update statistics status. The status is a character string stored in column SQL_CMD. Additional, If the update statistics command is executing, users can abort it with the stored procedure **setSystemOption()**, and the connection is not broken.

Please note, update statistics daemon will not work if update statistics server is not existed or has been terminated. And users cannot set table statistics on temporary table.

Users can enhance the performance of update statistics by the following methods: dmconfig.ini setting, every table setting, **setSystemOption,** obtain update statistics status and abort update statistics command.

DMCONFIG.INI SETTING

DBMaker have some configure keywords to activate update statistics daemon. Setting the configuration parameter **DB_StSvr** equal to 1 in the **dmconfig.ini** file activates the Auto Update Statistics Daemon. The **DB_StMod** keyword specifies update statistics mode (**general** mode or **every table setting** mode) for a database, the **DB_StsTm** keyword specifies the start time for update statistics, the **DB_StsTv** keyword specifies the update statistics daemon interval, and the **DB_StsSp** keywords is used to set the update statistics sample ratio.

➲ **Example 1**

Before starting database, configure some keywords related to update statistics daemon in the **dmconfig.ini** file:

```
[DBNAME]
; Here omit other keywords
DB_StSvr = 0
DB_StMod = 1
DB_StsTm = 2010-10-10 10:00:00
DB_StsTv = 12:00:00
DB_StsSp = 70
```

Now, start database, then update statistics daemon will auto update statistics according to the schedule.

➲ **Example 2**

Allow user to set all parameters about update statistics daemon during runtime:

```
dmSQL> CALL SETSYSTEMOPTION('STSVR','1');                 //activate automatic
update statistic server
dmSQL> CALL SETSYSTEMOPTION('STMOD','1');                 //reset DB_StMod
dmSQL> CALL SETSYSTEMOPTION('STSTM','2009/6/6 20:30:00');  //reset DB_StsTm
dmSQL> CALL SETSYSTEMOPTION('STSTV','7-00:00:00');        //reset DB_StsTv
dmSQL> CALL SETSYSTEMOPTION('STSSP','60');                //reset DB_StsSp
```

EVERY TABLE SETTING

If users set update statistics option for every table by executing the SQL statement UPDATE STATISTICS SET, there are four filter conditions as follows:

- If it is a new table, that is to say, the table did not perform update statistics, then execute automatic update statistics.
- If the total number of pages in the table is less than 20 pages, then execute automatic update statistics.
- If the total number of pages in the table is more than 20 pages, the new page number that is larger than 2 pages since the last automatic update statistics, then execute automatic update statistics.
- If the table doesn't update statistics more than 10 days, execute automatic update statistics.

If startup update statistics daemon in **every table setting** mode, that is to say, the value of **DB_StMod** is 1, users can set update statistics option for every table by executing the SQL statement UPDATE STATISTICS SET. If value of MODE in the SQL statement UPDATE STATISTICS SET is 0, the table's update statistics ratio is decided by value of **DB_StsSp** in **dmconfig.ini**, but need to consider the above four filter conditions. If value of MODE in the SQL statement UPDATE STATISTICS SET is 1, the table's update statistics ratio is decided by value of SAMPLE in the SQL statement UPDATE STATISTICS SET, but need to consider the above four filter conditions. If value of MODE in the SQL statement UPDATE STATISTICS SET is 2, the table's update statistics ratio is decided by value of SAMPLE in the SQL statement UPDATE STATISTICS SET and regardless of the above four filter conditions.

Every table setting info will stored into system table SYSTABLE, the column UPD_STS_MODE stored the table's update statistics mode, and the column UPD_STS_SAMPLE stored the table's update statistics sample.

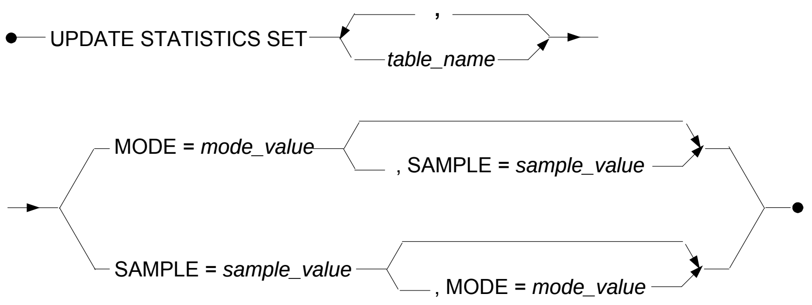


Figure 19-2 Syntax for the UPDATE STATISTICS SET statement

➲ **Example 1**

Setting update statistics mode and sample ratio for table **jeff.tb_staff**.

```
dmSQL> UPDATE STATISTICS SET jeff.tb_staff MODE = 1, SAMPLE = 80;
dmSQL> SELECT TABLE_NAME, TABLE_OWNER, UPD_STS_MODE, UPD_STS_SAMPLE FROM
SYSTABLE;
        TABLE_NAME             TABLE_OWNER       UPD_STS_MODE   UPD_STS_SAMPLE
======================== ======================== ============== ================
TB_STAFF                 JEFF                                 1               80
1 rows selected
```

➲ **Example 2**

Setting update statistics mode and sample ratio for table **jeff.tb_staff** and table **jim.tb_salary**.

```
dmSQL> UPDATE STATISTICS SET jeff.tb_staff, jim.tb_salary MODE = 1, SAMPLE = 60;
dmSQL> SELECT TABLE_NAME, TABLE_OWNER, UPD_STS_MODE, UPD_STS_SAMPLE FROM
SYSTABLE;
        TABLE_NAME            TABLE_OWNER        UPD_STS_MODE   UPD_STS_SAMPLE
======================== ======================== ============== ================
TB_STAFF                 JEFF                                  1               60
TB_SALARY                JIM                                   1               60
2 rows selected
```

SETSYSTEMOPTION

The system option of automatic update statistics include STSVR, STMOD, STSTM, STSTV and STSSP. DBMaker support setting these system option at run time by calling the system stored procedure **setSystemOption()**, and now add new system stored procedure **setSystemOptionW()** to support setting these system option at run time and writing run time setting to **dmconfig.ini** file. Additionally, users can obtain these system option info by calling the system stored procedure **getSystemOption()**. For more information on how to use the three system stored procedures **setSystemOption()**, **setSystemOptionW()** and **getSystemOption()**, please refer to the *SQL Command and Function Reference* and *ODBC Programmer's Guide*.

➲ **Example 1**

Set system option **STSVR** to 0 when the database is running.

```
dmSQL> CALL SETSYSTEMOPTION('STSVR','0');
```

Set system option **STSVR** to 1 and write **DB_StSvr = 1** into current database section in **dmconfig.ini** file when the database is running.

```
dmSQL> CALL SETSYSTEMOPTIONW('STSVR','1');
```

Obtain value of system option **STSVR** when the database is running.

```
dmSQL> CALL GETSYSTEMOPTION('STSVR',?);
```

➲ **Example 2**

Set system option **STSSP** to 70 when the database is running.

```
dmSQL> CALL SETSYSTEMOPTION('STSSP','70');
```

Set system option **STSSP** to 30 and write **DB_StsSp = 30** into current database section in **dmconfig.ini** file when the database is running.

```
dmSQL> CALL SETSYSTEMOPTIONW('STSSP','30');
```

Obtain value of system option **STSSP** when the database is running.

```
dmSQL> CALL GETSYSTEMOPTION('STSSP',?);
```

NOTE *Not all configure keyword could be changed in the run time.*

UPDATE STATISTIC STATUS

User can query system table SYSUSER to get update statistics status that is a character string stored in column: SQL_CMD.

Following is update statistic status information:

```
[EXEC] update statistics command // (Total, start_time, execute_time,
remain_time, complete_percent) (table_name, start_time, execute_time,
remain_time, complete_percent) (index_name, start_time, execute_time,
remain_time, complete_percent)
```

The update statistics status includes 3 parts: total, table and index. Total means all update statistics objects. Table means one table statistics, includes indexes belong to it and data page. Index means index statistics or data page statistics.

➲ **Example**

```
dmSQL> SELECT CONNECTION_ID, SQL_CMD FROM SYSUSER;
CONNECTION_ID                    SQL_CMD
============== ===========================================================
         3264  [EXEC]  Update t1 set c1 = c1 + 1;
          3267  [EXEC]  update statistics // for table SYSADM.HUNDRED index
HUNDRED_CODE start at 2011/02/21 09:19:54
2 rows selected
```

MONITOR AND ABORT UPDATE STATISTICS

DBMaker supports display update statistics status, you can query the **SQL_CMD** column of the SYSUSER table to monitor update statistics progress. Beside, you can calling a system stored procedure ***setSystemOption('STS_ABORT', 'connection_id')*** to abort an ongoing update statistics. About the **SQL_CMD** column of the SYSUSER system table information and the usage of the system stored procedure ***setSystemOption()***, please refer to the *JDBA Tool User's Guide* and the *SQL Command and Function Reference*.

➲ **Example 1**

Query the **SQL_CMD** column of the SYSUSER table to monitor update statistics progress:

```
dmSQL> SELECT SQL_CMD FROM SYSUSER;
```

➲ **Example 2**

Abort update statistics command which connection ID is 14076:

```
dmSQL> CALL SETSYSTEMOPTION('STS_ABORT','14076');
```

➲ **Example 3**

The value 0 is a special connection ID. It means abort all connection related to update statistics.

```
dmSQL> CALL SETSYSTEMOPTION('STS_ABORT','0');
```

Load and Unload Statistics

Users can use the UNLOAD STATISTICS command to dump the statistics values into an external text file. Users can also use the LOAD STATISTICS command to copy statistics values to a database from an external text file.

➲ **Example 1**

To use UNLOAD STATISTICS:

```
dmSQL> UNLOAD STATISTICS TO file1;//dump statistics from database to file1
```

➲ **Example 2**

To use LOAD STATISTICS:

```
dmSQL> LOAD STATISTICS FROM file1;//read statistics from external text file1
```

An experienced user can enhance the efficiency of query by means of modifying files with statistics, and input it into database.

➲ **Example 3**

An external text file generated from UNLOAD STATISTICS:

```
DBname = TESTDB

TBowner = jeff
TBname = tb_staff
TBpage = 5
TBrows = 30
Tbavlen = 50

COname = age
COtype = INTEGER
COdist = 12
COavlen = 4
```

```
COlow = 25
COhigh = 42

IXname = idxage
IXpages = 5
IXlevel = 2
IXleaf = 3
IXdist = 12
IXdistC1 = 12
IXdistC2 = 12
IXdistC3 = 12
IXpgkey = 8
IXcount = 7
```

19.5 Accelerating Execution of Query

The execution of a query can be accelerated by making the following modifications:

- Read fewer data rows
- Avoid sorting or sort on fewer data rows and columns
- Read data sequentially

Data Model

The definition of a data model includes all tables, views and indices on the database, especially the existence of indices. It describes whether an index should be used in the conditions such like join, sort and views.

Query Plan

Use the **Set Dump Plan ON** command to check the query execution plan for DBMaker.

Characteristics of an execution plan:

- **Index —** checks the output data to see whether an index has been used or not and if so, how to use it

- **Filter** — checks the predicate factors to see how much data the predicate can filter
- **Query** — checks the query after completion to see whether the access plan is the best

➲ **Example**

Use the following to see the execution plan:

```
dmSQL> SET DUMP PLAN ON;
```

Index Check

Check whether proper indexes on query columns exist. Use methods mentioned in the following sections to improve query efficiency.

Filter Columns

It only takes a small part of source information to make an efficient query. Users can use the WHERE predicate in a SELECT command to control the amount of output information, known as a data filter.

Following are some methods for using the advanced WHERE predicate:

AVOIDING CORRELATED SUB-QUERIES

Correlated sub-queries exist when duplicate columns appear in the main and the sub-query of a WHERE predicate. A different result is returned for a duplicate sub-query for each data row contained in the main query. If the data for columns in each row is different from the one in the previous row in a sub-query, then it is the equivalent to executing a new query for each row gained from the main query.

If you have found a time-consuming sub-query, first check whether it is a correlated sub-query. If so, rewrite the query to avoid the condition. If it is not easy to rewrite the query, try another method to reduce the number of data rows.

AVOIDING DIFFICULT REGULAR EXPRESSIONS

The keyword LIKE provides a comparison of wild cards, known as a regular expression. When a wild card is used at the beginning of an expression, a database server will check each row because it cannot use an index filter. This will instruct DBMaker to sequentially access and check every row in a table.

➲ **Example**

Using the LIKE keyword with the "*" wildcard:

```
dmSQL> SELECT * FROM tb_salary WHERE name LIKE '*st';
```

Query Results

When you understand what a query really does, you can find another equivalent query to get the same result. We give some suggestions for users to rewrite queries that are more efficient.

- Rewrite joins using views
- Avoid or reduce sorting
- Avoid accessing a large table sequentially
- Use unions to avoid sequential access

Temporary Tables

It is useful to create a temporary, ordered table to accelerate a query. It also can help to avoid sorting operations on multiple columns to simplify the operation of the optimizer.

- Use a temporary table to avoid sorting on multiple columns
- Replace sorting on non-sequential access

19.6 Syntax-Based Query Optimizer

Optimizer chooses a query execution plan with the cost function and statistics automatically. In some special cases when data distribution is skewed, the optimizer may choose a poor query execution plan. To solve the problem, DBMaker supports an optimizer mechanism called the *syntax based query optimizer*.

You can now manually specify the type of scan to use in a query, and the indexes to be used in an index scan. In addition, the DBMaker query optimizer automatically determines the most efficient type of scan to use, even if you have not recently updated the database statistics. There are five different cases where the type of index to be used.

Forced Index Scans

General syntax used to force an index scan:

```
tablename (INDEX [=] idxname [ASC|DESC])
```

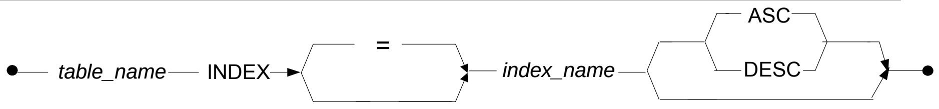


Figure 19-3 Force Index Scans syntax

➲ **Example 1**

To force a table scan specify the value 0:

```
dmSQL> SELECT * FROM tb_staff (INDEX=0);
```

➲ **Examples 2**

To force an index scan on a primary key specify the value 1:

```
dmSQL> SELECT * FROM tb_staff (INDEX=1);
```

➲ **Example 3**

To force an index scan on the index **idx_id**:

```
dmSQL> SELECT * FROM tb_staff (INDEX idx_id);
```

➲ **Example 4**

Allows the query optimizer to decide what type of scan to use on table **tb_staff**, but forces an index scan on the **idx_id** index for table **tb_salary**:

```
dmSQL> SELECT * FROM tb_staff, tb_salary (INDEX idx_id);
```

Forced Index Scan and "Alias"

General syntax used to force an index scan and provide an alias for the table:

```
tablename (INDEX [=] idxname) aliasname
```

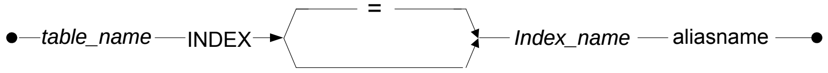


Figure 19-4 Force Index Scans and 'Alias' syntax

➲ **Example**

To force an index scan on the **idx_id** index, and provides an alias for the table:

```
dmSQL> SELECT * FROM tb_staff (INDEX idx_id) a, tb_staff b WHERE a.id = b.id;
```

Forced Index Scan and "Synonym"

General syntax used to force an index scan using a synonym:

```
synonymname (INDEX [=] idxname)
```

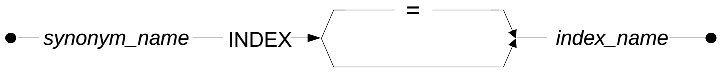


Figure 19-5 Force Index Scans and 'Synonym' syntax

➲ **Example**

To force an index scan on the **idx_id** index using synonym **staff**:

```
dmSQL> SELECT * FROM staff (INDEX idx_id);
```

Forced Index Scan and "View"

General syntax used to force an index scan when creating a view:

```
viewname (INDEX [=] idxname)
```

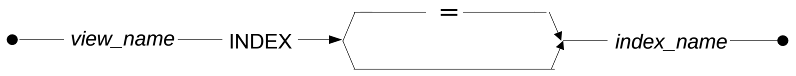


Figure 19-6 Force Index Scans and 'View' syntax

➲ **Example 1**

To force an index scan on the **idx_id** index when creating view **vi_staff**:

```
dmSQL> CREATE VIEW vi_staff as SELECT * FROM tb_staff (INDEX idx_id);
```

You cannot force an index when selecting a view.

➲ **Example 2**

A wrong usage that will return errors:

```
dmSQL> SELECT * FROM vi_staff (INDEX idx_id);
```

Forced Text Index Scans

General syntax used to force a text index scan:

```
tablename (TEXT INDEX [=] idxname)
```

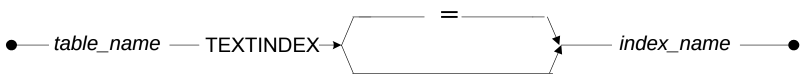


Figure 19-7 Force Text Index Scans syntax

➲ **Example**

To force a text index scan on the **tidx1**index:

```
dmSQL> SELECT * FROM tb_staff (TEXT INDEX tidx1);
```

Forced Loop Join (Nested Join)

General syntax used to force a Nested Join between two tables:

```
tablename { INNER | OUTER } LOOP JOIN tablename
```

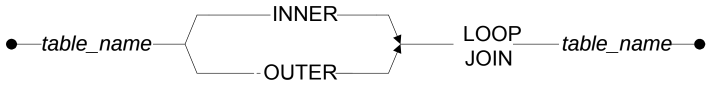


Figure 19-8 Force Loop Join Syntax

NOTE *A forced join of this type must use INNER JOIN or OUTER JOIN syntax.*

➲ **Example 1**

```
dmSQL> SELECT * FROM tb_staff INNER LOOP JOIN tb_salary ON
tb_staff.id=tb_salary.id;
```

➲ **Example 2**

```
dmSQL> SELECT * FROM tb_staff OUTER LOOP JOIN tb_salary ON
tb_staff.id=tb_salary.id;
```

Forced Merge Join

General syntax used to force a Merge Join between two tables:

```
tablename { INNER | OUTER } MERGE JOIN tablename
```

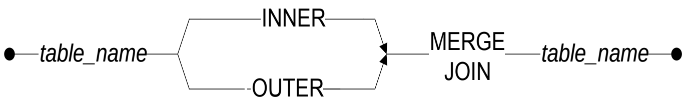


Figure 19-9 Force Merge Join Syntax

NOTE *When the join cannot use Merge Join, a Force Merge Join is useless, but will not return an error message.*

➲ **Example 1**

```
dmSQL> SELECT * FROM tb_staff INNER MERGE JOIN tb_salary ON tb_staff.id=
tb_salary.id;
```

➲ **Example 2**

```
dmSQL> SELECT * FROM tb_staff OUTER MERGE JOIN tb_salary ON tb_staff.id=
tb_salary.id;
```

Forced Join Sequence

Force all tables join sequence, and then the join sequence cannot swap. General syntax used to force Join Sequence:

```
SELECT ..... FROM [SEQUENCE | SEQ] tablename_list;
```

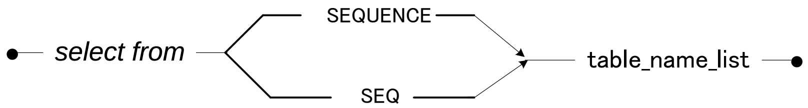


Figure 19-10 Force Join Sequence Syntax

Example 1

```
dmSQL> SELECT * FROM sequence tb_staff, tb_salary, tb_dept WHERE tb_staff.id=
tb_salary.id AND tb_salary.basepay=tb_dept.basepay;
```

Example 2

```
dmSQL> SELECT * FROM seq tb_staff INNER JOIN tb_salary ON tb_staff.id=
tb_salary.id INNER JOIN tb_dept ON tb_staff.basepay=tb_detp.basepay;
```

Forced Group by Method

General syntax used to force a Join Sequence:

```
GROUP BY column_name_list [USING SORT | USING HASH] having .....
```

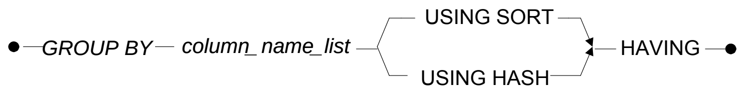


Figure 19-11 Force Group by Method Syntax

Example 1

```
dmSQL> SELECT id,name,count(*) FROM tb_staff GROUP BY id,name USING HASH;
```

Example 2

```
dmSQL> SELECT id,name,count(*) FROM tb_salary GROUP BY id,name USING SORT HAVING
sum(basepay)>0;
```

19.7 How to Read a Dump Plan

The first step to check a slow query is to read the execution plan. DBMaker supports a dump and read execution plan function.

There are three dmSQL commands for dump plans:

```
dmSQL> SET DUMP PLAN ON;
```

Turns on the dump plan option. The latter queries will dump plan and then execute commands.

```
dmSQL> SET DUMP PLAN OFF;
```

Turns off the dump plan option. The latter queries will only execute commands but not dump. This is the default option.

```
dmSQL> SET DUMP PLAN ONLY;
```

Turns on the dump plan only, but does not execute commands.

At first glance, it appears that a dump plan is composed of several blocks called ON. Query optimizer divides a query into several ON blocks, and each block is a logical optimization unit. The optimizer will then optimize every ON block. Simple and joined queries usually have only one ON block, but a complex query like a sub query may generate more than one ON block, including a main block and sub blocks.

Optimizer finds the best execution method based on cost for every ON block. It will divide an ON block into several PL blocks, where each PL block represents an operation, such as scan, join, etc.

Be familiar with the terms introduced in previous sections of this chapter:

- table scan
- index scan
- nested join
- merge join
- factor

Table Scan

➲ **Example**

To set the dump plan for a table scan of **tb_staff**:

```
dmSQL> SET DUMP PLAN ON;
dmSQL> SELECT * FROM tb_staff WHERE id>1;
```

Result, the dump plan for the table scan of **tb_staff**:

```
----- begin dump plan -----

{ON Block 0}
ON Type     : SCAN

[PL Block 0]
Method     : Scan
Table Name : tb_staff
Type       : Table Scan
Order      : <none>
Factors    : (1) tb_staff.id > 1
I/O Cost   : 101.0
CPU Cost   : 25.3
Sub Cost   : 0.0
Result Rows: 330.0

----- end dump plan -----
```

The first two lines give the information for an ON block:

{ON Block 0} — ON block with a block ID of 0

ON Type: SCAN — ON block type is a scan

The ON block contains one PL block:

[PL Block 0] — A PL block with a block ID of 0

Method: Scan — This PL block will perform a scan operation

Table Name: tb_staff — Scan on table tb_staff defined

Type: Table Scan — Scan type is a table scan

Order: <none> — Scan order, there is no use for a table scan

Factors: (1) tb_staff.id > 1 —scan uses the filter tb_staff.id > 1

I/O Cost: 101.0 — Estimated I/O cost is 101.0 pages

CPU Cost: 25.3 — Estimated CPU cost is 25.3 pages

Sub Cost: 0.0 — Estimated sum of costs for the PL block's sub-block. In this example, there is no PL sub-block

Result Rows: 330.0 — Estimated result rows after the scan and filter

Index Scan

➲ **Example**

To set the dump plan for **id** and **name** from table **tb_salary** using WHERE:

```
dmSQL> SET DUMP PLAN ON;
dmSQL> SELECT id,name FROM tb_salary WHERE id>1 AND name='john';
```

Result, the dump plan for **id** and **name** from table **tb_salary** using WHERE:

```
----- begin dump plan -----

{ON Block 0}
ON Type     : SCAN

[PL Block 0]
Method      : Scan
Table Name  : tb_salary
Scan Type   : Index Scan on idx21(name, id)
Order       : ASC
Index EQFA#: 1
Index FA#   : 2
Index FACOL: 1, 2
Index Cost  : 2
Factors     : (1) tb_salary.name = 'john'
            : (2) tb_salary.id > 1
I/O Cost    : 2.0
CPU Cost    : 0.6
Sub Cost    : 0.0
```

```
Result Rows: 13.0

----- end dump plan -----
```

The first two lines give the information for an ON block:

{ON Block 0} — ON block with a block ID of 0

ON Type: SCAN — ON block type of scan. The ON block also contains one PL block: **[PL Block 0]** — A PL block with a block ID of 0

Method: Scan — The block executes a scan

Table Name : tb_salary — Scan of table tb_salary

Scan Type: Index Scan on idx21(name, id) — Scan type is an index, applying an index to idx12 using the index columns name and id

Order: ASC — Ascending index scan order

Index EQFA#: 1 — Equal factor number applied in the index scan, in this example using tb_salary.name = 'john'

Index FA#: 2 — Factor number that applied in the index scan, in this example using tb_salary.name = 'john' and tb_salary.id > 1

Index FACOL: 1, 2 — Factor ID mapping from index columns. In this example, it maps the first index column, name, to factor, (1) tb_salary.name = 'john', and the second index column, id, maps to factor, (2) tb_salary.id > 1.

Index Cost: 2 — Estimated index page cost of 2

Factors: (1) tb_salary.name = 'john'

(2) tb_salary.id > 1 - Applies filters tb_salary.name = 'john' and tb_salary.id > 1

I/O Cost: 2.0 — Estimated I/O cost of 2.0 pages

CPU Cost: 0.6 — Estimated CPU cost of 0.6

Sub Cost: 0.0 — Estimated sum of costs for the PL block's sub-block

Result Rows: 13.0 — Estimated result of rows after the scan and filter

Equal Join

➲ **Example**

To set the dump plan from **tb_staff** and **tb_salary** using WHERE:

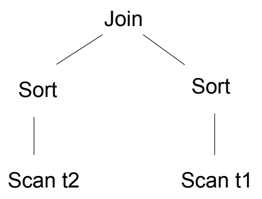
```
dmSQL> SET DUMP PLAN ON;
dmSQL> SELECT * FROM tb_staff, tb_salary WHERE tb_staff.id=tb_salary.id;
```

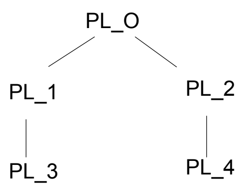
Result, the dump plan from **tb_staff** and **tb_salary** using WHERE:

```
----- begin dump plan -----

{ON Block 0}
ON Type     : JOIN

[PL Block 0]
Method      : Join
Type        : Merge Join
Factors     : (1) tb_staff.id = tb_salary.id
I/O Cost    : 8.5
CPU Cost    : 573.8
Sub Cost    : 231.6
Result Rows: 500.0
Sub Block 1: [PL Block 1]
Sub Block 2: [PL Block 2]

[PL Block 1]
Method      : Sort
I/O Cost    : 4.2
CPU Cost    : 274.4
Sub Cost    : 120.0
Result Rows: 1000.0
SUB Block   : [PL Block 3]

[PL Block 3]
Method      : Scan
Table Name : tb_salary
Type        : Table Scan
Order       : <none>
Factors     : <none>
I/O Cost    : 101.0
```

```
CPU Cost    : 25.3
Sub Cost    : 0.0
Result Rows: 1000.0

[PL Block 2]
Method      : Sort
I/O Cost    : 4.2
CPU Cost    : 274.4
Sub Cost    : 120.0
Result Rows: 1000.0
SUB Block   : [PL Block 4]

[PL Block 4]
Method      : Scan
Table Name  : tb_staff
Type        : Table Scan
Order       : <none>
Factors     : <none>
I/O Cost    : 101.0
CPU Cost    : 25.3
Sub Cost    : 0.0
Result Rows: 1000.0

----- end dump plan -----
```

In this example, there is more than one PL block. The PL block relationship is combined using the sub-block information.





A simple tree representing blocks

Replace each node with the names

Descriptions of Join blocks:

[PL Block 0] -- A PL block with a block ID of 0

Method: Join -- The block is a Join

Type: Merge Join -- Join is of the type merge

Factors: (1) tb_staff.id = tb_salary.id -- Apply Join filter tb_staff.id = tb_salary.id using a Join block

I/O Cost: 8.5-- Estimated I/O cost is 8.5 pages

CPU Cost: 573.8 -- Estimated I/O cost is 573.8 pages

Sub Cost: 231.6 -- Estimated sum of costs for the PL block's sub-block

Result Rows: 500.0 -- Estimated result rows after the Join block

Sub Block 1: [PL Block 1] -- The block's first child links to [PL Block 1]

Sub Block 2: [PL Block 2] -- The block's second child links to [PL Block 2]

Description for a sort block:

[PL Block 1] -- A PL block with a block id of 1

Method: Sort -- A sort block

I/O Cost: 4.2 -- Estimated I/O cost is 4.2 units

CPU Cost: 274.4 -- Estimated CPU cost is 274.4 units

Sub Cost: 120.0 -- Estimated sum of costs for the PL block's sub-block

Result Rows: 1000.0 -- Estimated result rows after the sort block

SUB Block: [PL Block 3] -- This block's child block link to [PL Block 3]

Listed above are the most common dump plan cases that users will encounter. Many changes will be evident in dump plans, but they all consist of the same elements, I/O cost, CPU cost, and Result Rows. If a dump plan is too complex, use the syntax-based optimizer discussed previously to try other methods.

A. Keywords in dmconfig.ini

A.1 General Concept

When the DBMaker database engine is started or when a user connects to a database server, DBMaker must initialize several parameters to configure itself. These parameters are read from an ASCII text configuration file named **dmconfig.ini**. This text file contains the keywords and corresponding values that are used for configuration. This file is in ASCII format. A DBA can edit it with a text editor to change the parameters as required.

In most cases, the keywords are required when a database starts. Keywords changed after the database has been started will not take effect until the database has been shut down and restarted.

However, some of the keywords are only required when users connect to database servers. Users can change these keywords after the server has started, but before the connect command is issued and these new values will be valid during the current session.

The configuration parameters play an important role in the performance of DBMaker. To ensure DBMaker operates smoothly, understand the effects of changing configuration parameters and estimate the best values to use. It is recommended that the database administrator back up the **dmconfig.ini** file and the database files on a regular basis.

A.2 dmconfig.ini File Format

The **dmconfig.ini** file is an ASCII text file. It can be edited with any text editor capable of opening and saving ASCII text files. The **dmconfig.ini** file includes many sections. Each section comprises the configuration information used to start a particular database. Each section begins with the section name, followed by a list of keywords and their values.

➲ **Example**

The format of a dmconfig.ini file:

```
[section_name_1]
keyword1 = value1         ; here is a comment
keyword2 = value2
         .
         .
[section_name_2]
keyword3 = value3 value4  ; spaces or commas may be used
keyword4 = value5         ; as delimiters
         .
         .
```

Section Names

Each section name corresponds to the name of the database that uses the configuration options found in that section when it starts up. Section names begin with a left square bracket ([) followed by the name of the database, and end with a right square bracket (]). The brackets must enclose the section name, and the left bracket must be the first character on its line.

Keywords

Following each section name is a list of keywords and their values. These configuration values are used by the database corresponding to the section heading when it starts. The statement keyword=value assigns the specified value to a keyword. The value of a keyword can be an integer or a string, depending on the keyword itself.

Comments

Any string or symbol that is written after the semi-colon (;) is considered a comment and therefore ignored by DBMaker.

➲ **Example**

dmconfig.ini file using (;) to include comments:

```
[SDB]
DB_DbFil=SDB.DB
DB_JnFil=SDE.JNL
DB_SMode=1             ;normal start mode
DB_BMode=1
DB_BkSvr=1
DB_BkTim=96/03/19 00:00:00
DB_BkItv=7-00:00:00
DB_NBufs=100
DB_NJnlB=200
DB_MaxCo=100
DB_JnlSz=20000
file1=SDE.FIL 40

[EMP]
DB_DbFil=EMP.DB
DB_JnFil=EMP.JNL
DB_SMode=1             ;normal start mode
DB_NBufs=100
DB_NJnlB=400
DB_MaxCo=100
DB_JnlSz=20000
file1=EMP.FL1 100
file2=EMP.FL2 200
```

In the example, the **dmconfig.ini** contains a first section for the **SDB** database and a second section for the **EMP** database.

A.3 Search Path for dmconfig.ini

DBMaker running on UNIX platforms looks in three locations for the **dmconfig.ini** file.

➲ **The locations and the search order are:**

1. The current directory.
2. The directory specified in the environment variable DBMAKER.
3. The installation directory: **~dbmaker/** ***Version*** .

On the Microsoft Windows platform, the **dmconfig.ini** file is placed in the installation directory; this is typically **C:\DBMaker\*Version***.

When starting a database, DBMaker scans the directories listed above to locate a **dmconfig.ini** file with a section name that corresponds to the database. When a **dmconfig.ini** file is found and the section name exists, the keywords defined in the section are used. If the section name cannot be found in **dmconfig.ini** file, DBMaker continues searching sequentially in the directories for a **dmconfig.ini** file sequentially in the directories until the section name is found.

A.4 Default Values for Keywords

When DBMaker needs a parameter, it searches the corresponding keywords in the proper section of the **dmconfig.ini** file. The pattern matching for section names or keywords in a search are case insensitive, except for user-defined files. When a keyword cannot be found in the **dmconfig.ini** file, a default value is used. Most of the keywords have their own default values. Refer to the last section in this chapter for more information about this topic.

A.5 Creating dmconfig.ini

Normally, before a database is created, a database administrator creates the corresponding section in the **dmconfig.ini** file using a text editor and the parameters take effect while creating that database. However, if DBMaker cannot find the section in **dmconfig.ini** while creating a database, it automatically creates a section in the first **dmconfig.ini** file found. If a **dmconfig.ini** file is not found then one is created containing the section for the new database Therefore, a section for each database should always be found at startup time, if not found, DBMaker will returns an error.

A.6 Keyword Reference

DB_AtCmt=<value>

This keyword specifies the on or off status of the auto-commit mode. Setting this value to 1 actives auto-commit mode and setting it to 0 deactivates auto-commit mode. When auto-commit mode is on, DBMaker automatically issues a *`COMMIT TRANSACTION`* after each successfully executed SQL command. This keyword is set from the client side.

default value: 1

valid range: 0, 1

see also: DB_LTimO

where to use: client side

DB_AtrMd=<value>

This keyword defines the database as a source database of asynchronous table replication. Setting this value to 1 activates the logging of base table operations and enables the Distributor Daemon. Therefore, it can be a source database of a replication.

default value: 0

valid range: 0, 1

see also: DB_EtrPt, RP_LgDir

where to use: server side

DB_BbFil=<string>

This keyword specifies the name of the system BLOB file. It expands as necessary as more BLOB data is inserted into this file.

default value: database name with the file extension ***.SBB***. For example: ***db.SBB***.

valid range: string with length < 256

see also: DB_DbDir, DB_DbFil, DB_UsrBb, DB_UsrDb

where to use: server side

DB_BfrSz=<value>

This keyword specifies, in kilobytes, the size of each BLOB frame. This keyword is used when the database is created.

default value: 32 (KB)

valid range: 8 ~ 256 (KB)

see also: DB_BbFil

where to use: server side (only for creating a database)

DB_BkChk=<value>

This keyword specifies whether check database before full backup and differential backup. If this keyword is set to 0 denotes DBMaker will not check database before full backup and differential backup; and 1 denotes DBMaker will check database before full backup and differential backup. Backup server will write error message and stop this backup if find the database has been damaged; 2 denotes DBMaker will check database before full backup and differential backup. Backup server will write error message and continue to back up bad database to BKDIR/BADDB directory if find the database has been damaged. It will be done only once to back up bad database when backup server first finds the damaged database. After then, if checking db is ok, backup server will remove bad backup resided in BKDIR/BADDB directory and continue normal backup, otherwise, only write error message and stop this backup.

Incremental backup files always follow the last full backup or differential backup. So, incremental backup file will be placed into BKDIR/BADDB after finding database has damaged, and they are switched back into BKDIR after database is ok.

default value: 0

valid range: 0, 1, 2

see also: DB_DbKtv, DB_DbKmx

where to use: server side

DB_BkCmp=<value>

This keyword specifies the compact backup mode. A successful backup does not require every journal block in a journal file. If this keyword is set to 1 the backup server will only back up the journal blocks that require backup. This method can be used to save disk space. Please also see the chapter 15 *, Database Backup, Recovery, and Restoration* for more detailed information about this topic.

default value: 1

valid range: 0, 1

see also: DB_BkSvr

where to use: server side

DB_BkDir=<string>

This keyword specifies one directory or a group of directories (up to 32) where the backup server puts the most recent backup sequence. A valid backup sequence is composed of a full backup, one or more differential backup (optional) and a series of incremental backup (optional). These directories must already exist and can be different from **DB_DbDir**.

default value: a default backup directory named **backup** under the database directory

valid range: string with length < 256

see also: DB_BkSvr, DB_BMode

where to use: server side

➲ **Example**

< BKDIR n >: the n's backup path

< SIZE n >: the size of the n's backup path

DB_BkDir = <BKDIR 1> <SIZE 1> < BKDIR 2> <SIZE 2> < BKDIR 3> <SIZE 3>…

```
[MYDB]
……
DB_BkDir = /home/usr/dbmaker/bk 5000 /home2/backup 1000
……
```

When */home/usr/dbmaker/bk* is full, it will backup files to */home2/backup*.

DB_BkFoM=<value>

This keyword specifies the file object (FO) backup mode. File objects are only backed up during a full backup of the database. **DB_BkFoM** has three possible values; 0, 1 and 2. Setting **DB_BkFoM** equal to 0 disables the FO backup feature; file objects are not backed up during a full backup. Setting **DB_BkFoM** to a value of 1 enables system file objects to be backed up during a full backup. Setting **DB_BkFoM** equal to 2 enables both system file objects and user file objects to be backed up.

default value: 0

valid range:	0: File objects are not backed up
	1: System file objects are backed up
	2: System and user file objects are backed up.

see also: DB_BkSvr, DB_FBkTm, DB_FBkTv, DB_BkDir

where to use: server side

DB_BkFrm=<value>

The keyword specifies the format Backup Server used to name incremental backup journal files. The backup filename format is ***<I><Timestamp><_><DB_BkFrm>***, the timestamp is a system 10 digits valid time numeric data, and the <**DB_BkFrm**> may include both text constants and format sequences (e.g., escape sequences), that represent special character strings.

The format sequences represent the year, month, or date the backup was performed, the name of the database, or the backup identification number. Format sequences and text constants can be combined in any way, provided the result is a valid filename supported by the operating system. The format sequences have three parts: the escape character, the length value, and the format character. The valid format sequences are:

%[*n*]Y—The year the journal file was backed up

%[*n*]M—The month the journal file was backed up

%[*n*]D—The day the journal file was backed up

%[*n*]B—The backup identification number

%[*n*]N—The name of the journal file's corresponding database

➲ **Example**

DB_BkFrm = %N.%B

If the database is test1, the incremental backup files are named: test1.1, test1.2…

default value: *I<timestamp>*_%4N%4B.jnl

see also: **DB_BkSvr**, **DB_BkTim**, **DB_BkItv**

where to use: **server side**

DB_BkFul=<value>

This keyword specifies the percentage full of the journal files that triggers the backup server to perform an incremental backup. Setting this value to 0 triggers the backup server to perform a backup when a journal files are 100% full. Setting this value between 50 and 100 triggers the backup server to perform a backup when the total space used in all of the journal files exceeds the specified percentage. For example, if there are two journal files of 500 journal blocks each and **DB_BkFul** is set to 80, then after every $500 \times 2 \times 0.8 = 800$ blocks are used, the backup server will automatically perform an incremental backup.

default value: 90

valid range: 0, 50 ~ 100

see also: **DB_BkSvr**, **DB_BkTim**, **DB_BkItv**

where to use: **server side**

DB_BkItv=<string>

This keyword specifies the backup time interval. Please refer to **DB_BkTim** for more information.

default value: none (no backup is scheduled when **DB_BkItv** is unassigned)

valid range: 0-00:00:01 ~ 24854-23:59:59

see also: **DB_BkSvr**, **DB_BkTim**, **DB_BMode**

where to use: **server side**

DB_BkOdr=<string>

This keyword specifies the old backup directories, which is one directory or a group of directories (up to 32). These directories are used to save a backup sequence which is one just before the last one. A valid backup sequence is composed of a full backup, one or more differential backup (optional) and a series of incremental backup (optional). Please also see chapter 15, *Database Backup, Recovery, and Restoration*, for more detailed information about this topic.

default value: none.

valid range: string with length < 256 *see also:* DB_BkSvr, DB_BMode, DB_FBkTm, DB_FBkTv

where to use: server side

➲ **Example**

DB_BkOdr = <BKDIR 1> <SIZE 1> < BKDIR 2> <SIZE 2> < BKDIR 3> <SIZE 3>…

< BKDIR n > : the n's backup path

< SIZE n > : the size of the n's backup path (8 KB per unit)

```
[MYDB]
          ……
DB_BkOdr = /home/usr/dbmaker/bk 5000 /home2/backup 1000
```

DB_BkRTs=<value>

This keyword specifies whether the backup server includes the read-only tablespace files when performing a full-backup. The default value is 1 and will back up the read-only tablespace files. Assign 0 to **DB_BkRTs** when the read-only tablespace files are already backed up. Please note, when 0 is assigned to the keyword a full-backup must be performed after setting a tablespace to read-only tablespace. If you don't backup the latest files, serious errors may result when restoring the full backup database containing the read-only tablespace. Please use the default value of 1 unless there is a specific reason to do otherwise.

default value: 1

valid range: 0, 1

see also: DB_BkSvr

where to use: server side

DB_BkSPm=<value>

This keyword specifies whether backup ESQL stored procedures and JAVA stored procedures in the process of doing a full backup. Its value can be set to 1 or 0, and the default value is 0. If users assign 0 to **DB_BkSPm**, ESQL stored procedures and

JAVA stored procedures will not be backed up; if users assign 1 to **DB_BkSPm**, ESQL stored procedures and JAVA stored procedures will be backed up. In addition, because source codes are written into a database, SQL stored procedures are backed up as regular data during a full backup.

default value: 0

valid range: 0, 1

where to use: server side

DB_BkSvr=<value>

This keyword is used to control the state of backup server after DBMaker is started. When **DB_BkSvr** is set to 0, the backup server is inactive; when **DB_BkSvr** is set to 1, the backup server is active. To activate the backup server, you can set the value of the **DB_BkSvr** keyword to 1 in the **dmconfig.ini** file or change BkSvr with *call setsystemoption('bksvr', '1')* after the database is started. Please also see chapter 15, *Database Backup, Recovery, and Restoration*, for more detailed information about this topic.

default value: 0

valid range: 0, 1

see also: **DB_BkCmp**, **DB_BkDir**, **DB_BkFul**, **DB_BkTim**, **DB_BkItv**

where to use: server side

DB_BkTim=<string>

This keyword along with **DB_BkItv** specifies backup server schedules. **DB_BkTim** specifies the first time a backup server performs an incremental backup. Incremental backups are there after performed at time interval specified in **DB_BkItv**.

➲ **Example**

```
DB_BkTim = 96/05/01 00:00:00    ;backup begins from May 1, 1996.
DB_BkItv = 1-12:30:00           ;interval is every one day, 12 hours
                                 and 30 minutes.
```

The keywords **DB_BkTim** and **DB_BkItv** are referenced only when the backup server is started. Please also see chapter 15, *Database Backup, Recovery, and Restoration*, for more detailed information about this topic.

default value: none (no backup is scheduled when **DB_BkTim** is unassigned)

valid range: 1970-01-01 00:00:01 ~ 2037-12-31 23:59:59

see also: **DB_BkItv**, **DB_BkSvr**, **DB_BMode**

where to use: **server side**

DB_BkZip=<value>

This keyword specifies whether the backup files are compressed by a backup server when performing full backups.

Setting the keyword to 1 generates compressed files during a full backup. Setting the keyword to 0 does not compress files during a full backup.

default value: 0

valid range: 0, 1

see also: **DB_BkSvr, DB_BkCmp, DB_BkDir, DB_BkFul, DB_BkTim, DB_BkItv**

where to use: **server side**

DB_BMode=<value>

This keyword specifies a database's backup mode. Setting the value to 0 enables NON-BACKUP mode, 1 enables BACKUP-DATA mode, and 2 enables BACKUP-DATA-AND-BLOB mode. This keyword only affects the incremental backup. When the value is set to 1 or 2, users can do an incremental backup. If the value is set to 0, not all jobs are saved to journals. Please also see chapter 15, *Database Backup, Recovery, and Restoration*, for more detailed information about this topic.

default value: 0

valid range: 0, 1, 2

see also: DB_BkSvr

where to use: server side

DB_Brows=<value>

This keyword specifies the lock behavior of a select statement. Setting the value to 0 denotes DBMaker will take **S** lock on the result set of select statement, and 1 denotes DBMaker will not lock the result set of select statement. This value is required while connecting to a database.

default value: 1

valid range: 0, 1

where to use: client side

DB_CBMod=<value>

This keyword specifies the behavior of the cursor after the end of a transaction. A value of 1 specifies that all still open cursors are closed after transactions are committed. A value of 2 or 3 specifies that all still open cursors remain open after a transaction is committed. Additionally, 2 specifies that all locks are released after the transaction and 3 indicates that all locks are reserved but all exclusive locks become shared. In these cases (1, 2 or 3), the cursor is closed when any transaction is aborted.

default value: 2

valid range: 1, 2, 3

where to use: client side

DB_ChkFl=<value>

This keyword specifies whether check a user's files when the database starts in the way of warm start. Setting this value to 0 disables the function, and the server will not check the user's files. Setting this value to 1 enables the function. The database starting, if some configuration items or files are missing, a warning message used to notify users appears and it is also recorded into the log file *DMEVENT.LOG*.

default value: 1

valid range: 0, 1

where to use: server side

DB_CliLCODE=<string>

This keyword specifies the language code in the client side. When use multilingual database and the LCODE of the Database Server is set to 10 (i.e. it is a UTF-8 database), client side can use any local codes DBMaker supported to connect to the UTF-8 database server.

If the Server LCODE is not 10 (UTF-8), then the value of CLILCODE must be same as the code of LCODE in server.

User can use the command SELECT GETSYSINFO('CLILCODE') to return the language code setting of the client side.

default value:

When the Server LCODE is not 10 (UTF-8), the default value would be same with server LCODE.

When the server LCODE is 10 (UTF-8), for Windows platform, client side would get the Windows language as the default CLILCODE value; for Linux platform, client side will get the value of the environment variable **LANG** as the default CLILCODE value, if the **LANG** did not be set, the value could be ASCII.

Anyway, if client side did not do any setting for this keyword, system will ensure the connection between client and server by default status.

valid range: all the codes of DBMaker supported

ASCII (English)
BIG5 (Traditional Chinese)
Shift-JIS (Japanese Shift-JIS + Half Corner)

GBK (Simplified Chinese)
ISO-8859-1 (Latin1 code)
ISO-8859-2 (Latin2 code)
ISO-8859-5 (Cyrillic code)
ISO-8859-7 (Greek code)
EUC-JP (Japanese code)
GB18030 (Simplified Chinese)
Unicode (UTF-8)

ISO-8859-{3,4,9,10,13,14,15,16},KOI8-R, KOI8-U, KOI8-RU,CP{1250,1251,1252,1253,1254,1257}, CP{850,866},Mac{Roman,Central Europe, Iceland, Croatian, Romania }, Mac{Cyrillic, Ukraine, Greek, Turkish }, Macintosh(European Language)

ISO-8859-{6,8}, CP{1255,1256}, CP862, Mac{Hebrew, Arabic} (Semitic languages)

CP932, ISO-2022-JP, ISO-2022-JP-2, ISO-2022-JP-1(Japanese)

EUC-CN, CP936, EUC-TW, CP950(Chinese)

EUC-KR, CP949, JOHAB(Korean)

Georgian-Academy, Georgian-PS(Georgian)

KOI8-T(Tajik)

PT154(Kazakh)

TIS-620, CP874, MacThai(Thai)

MuleLao-1, CP1133(Laotian)

VISCII, TCVN, CP1258(Vietnamese)

see also: **DB_LCode, DB_ErrLCODE**

where to use: **client side**

➲ **Example 1**

```
DB_CliLCODE=GBk;
```

➲ **Example 2**

```
DB_CliLCODE= PT154;
```

DB_CmChe=<value>

This keyword specifies the on or off status of a client command cache. Setting this value to 1 activates the client command cache. Setting this value 0 deactivates it. When the client command cache is turned on, the related information of previously executed SQL commands is cached and reused by subsequent identical SQL commands. In this way, performance gains are achieved

default value: 1

valid range: 0, 1

where to use: client side

DB_CTbLM=<value>

This keyword specifies the default lock mode used when creating a table.

Setting the value to 0 indicates a page level default lock mode. If a lock mode is not specified when creating a table, it defaults to page level.

Setting the value to 1 indicates the default lock mode is row. If a lock mode is not specified when creating a table, it defaults to row.

default value: 1

valid range: 0, 1

where to use: server side

DB_CTimO=<value>

This keyword specifies the connection time-out value, in seconds, for clients attempting to connect to the database server. If a database has not been started or the server IP address is incorrect, users may be forced to wait a long time until the connection times out. Users can set the value of this keyword to shorten the waiting time.

default value: 5 (seconds)

valid range: 5 ~ 1:00:00 (1 hour)

where to use: client side

DB_DaiFm=<value>

This keyword specifies the date input format for SQL statements. Please refer to *Appendix B* in the *ODBC Programming Guide* for more information.

default value: none (accept all date input formats)

valid range:	mm/dd/yy
	mm-dd-yy
	dd/mon/yy
	dd-mon-yy
	mm/dd/yyyy
	mm-dd-yyyy
	yyyy/mm/dd
	yyyy-mm-dd
	dd/mon/yyyy
	dd-mon-yyyy
	dd.mm.yyyy
	yyyy.mm.dd
	yyyymmdd

see also: DB_DaoFm

where to use: client or server side (client has higher priority)

DB_DaoFm=<value>

This keyword specifies the date output format in SQL statements. Please refer to *Appendix B* in the *ODBC Programming Guide* for more information.

default value: yyyy-mm-dd

valid range: see DB_DaiFm

see also: DB_DaiFm

where to use: **client or server side (client has higher priority)**

DB_DbDir=<string>

This keyword specifies where the database file resides. The directory string can be a relative or a full path name.

There are seven types of files in DBMaker:

- system database file defined by **DB_DbFil**
- default user data file defined by **DB_UsrDb**
- system journal file defined by **DB_JnFil**
- system BLOB file defined by **DB_BbFil**
- default user BLOB file defined by **DB_UsrBb**
- system temporary files defined by **DB_TpFil**
- user defined files

Relative path names or simple file names can be used when defining full path names for these keywords. If a path name is used in defining the keywords, DBMaker uses that name to reference the defined file. If a simple file name is used, DBMaker searches for the **DB_DbDir** keyword. If this keyword is found, DBMaker prepends the string specified in **DB_DbDir** to the simple file name when referencing the file. If it is not found, DBMaker uses the file name and assumes it is located in the current directory.

➲ **Example 1**

```
[DB1]
DB_DbDir = /disk1/db
DB_DbFil = mydb1
DB_JnFil = /disk2/usr/DB1.JNL
```

Using the physical file names:

```
DB_DbFil -- /disk1/db/mydb1
DB_JnFil -- /disk2/usr/DB1.JNL
DB_BbFil -- /disk1/db/DB1.BB (using default file name)
```

➲ **Example 2**

```
[DB2]
DB_DbFil = mydb2
DB_JnFil = /disk2/usr/DB2.JNL
```

Using the physical file names:

```
DB_DbFil -- mydb2 ( in current directory )
DB_JnFil -- /disk2/usr/DB2.JNL
DB_BbFil -- DB2.BB ( in current directory )
```

default value: (current directory)

valid range: string with length < 256

see also: DB_DbFil, DB_JnFil, DB_BbFil, DB_TpFil, DB_UsrDb, DB_UsrBb

where to use: server side

DB_DbFil=<string>

This keyword specifies the physical name of the system database file used by the operating system.

default value: database name with the file extension **.SDB**. For example: **db.SDB**.

valid range: string with length < 256

see also: DB_BbFil, DB_DbDir, DB_UsrBb, DB_UsrDb

where to use: server side

DB_DbKmx=<value>

This keyword specifies the maximum number of differential backup after a full backup. Backup server will remove the oldest differential backup if the number of differential backup after a full backup exceeds **DB_DbKmx**. Value 0 denotes not to use this keyword, in other words, the number of differential backup is unlimited.

default value: 10

valid range: 0 ~ 65535

see also: **DB_ FBkTm**, **DB_ FBkTv**, **DB_DbKtv**

where to use: **server side**

DB_DbKtv=<string>

This keyword specifies the differential backup time interval. The first differential backup is done at **DB_FBkTm** + **DB_DbKtv**. The format is **nDays-hh:mm:ss**. Please refer to **DB_FBkTm** and **DB_FBkTv** for more information.

default value: none (no full backup schedule if **DB_DbKtv** is not set)

valid range: 0-00:00:01 ~ 24854-23:59:59

see also: **DB_ FBkTm**, **DB_ FBkTv**, **DB_DbKmx**

where to use: **server side**

DB_DsCmt=<value>

This keyword specifies whether to commit a transaction when an application is disconnecting from the database. Setting the value to 0 indicates that when a client's application issues a SQLDisconnect command, DBMaker will not issue a commit before disconnecting from the database. The transaction rolls back when an autocommit is not set or a commit is not issued before disconnecting.

Setting the value to 1 indicates that DBMaker will also issue a commit before disconnecting from the database when the client's application issues a SQLDisconnect command. The transaction commits when an autocommit is not set or a commit is not issued before disconnecting.

default value: 0

valid range: 0 (does not issue commit when disconnecting from database)

1 (issues commit when disconnecting from database)

where to use: client side

DB_DtClt=<value>

This keyword specifies the time interval when DBMaker clients set self idle time-out to server. Default value of this keyword is 0, that is to say, this feature is disabled.

Sometimes the server cannot release resources allocated to a client when client machine is suddenly powered off or the network is not operating properly. Activating this keyword solves this problem because server can check the client in an interval time. When the server finds a dead (idle) client, it releases all of the client's resources. If an area network is not stable, DBA can set this area client idle time out, it can let the server to check client in a period time. If the client is dead, server will release all client allocated resources.

The maximum idle time of communication between client and server is decided by value of **DB_DtClt** (set on client) and value of **DB_ITimo** (set on server). The rule as follows:

- If users only set one of the two keywords, the maximum idle time of communication between client and server is value of the set keyword. In other words, if users only set **DB_DtClt** on client, the maximum idle time of communication between client and server is value of **DB_DtClt**; if users only set **DB_ITimo** on server, the maximum idle time of communication between client and server is value of **DB_ITimo**;
- If users set the two keywords at the same time, there are two conditions:

  a) If value of **DB_DtClt** is smaller than value of **DB_ITimo**, the maximum idle time of communication between client and server is value of **DB_DtClt**.

b) If value of **DB_DtClt** is bigger than value of **DB_ITimo**, the maximum idle time of communication between client and server is value of **DB_ITimo**. Meanwhile, DBMaker will automatically reset value of **DB_DtClt** to value of **DB_ITimo**.

- If users don't set anyone of the two keywords, the maximum idle time of communication between client and server is unlimited.

default value: 0 (seconds)- disable

valid range: 0, 5 ~ 1:00:00 (1 hour)

see also: DB_ITimO

where to use: **client side**

DB_ERMRv=<string>

This keyword specifies a list of e-mail addresses to receive notification when the database experiences an error. Up to eight recipient e-mail addresses may be specified; a comma must separate each address string. If no recipient is specified, the error report system is disabled and no e-mail is generated.

default value: null

see also: DB_ ERMSv

where to use: **server side**

DB_ERMSv=<string>

This keyword specifies the SMTP server for relaying e-mail messages. Only one SMTP server may be specified. DBMaker assigns the 'localhost' value to this keyword if the SMTP server is unspecified but recipients are specified.

default value: localhost

see also: DB_ERMRv

where to use: **server side**

DB_ErrLCODE=<string>

This keyword is used to set client's error message character set. In multilingual database, client side can set their own output locale codes of error message.

The value of the keyword adopts the combination form of language definition, locale definition and character set coding. I.e. 'language *[_locale] [.code]* ', '*language*' string follows ISO-639 standards, it is must be lowercase; and '*locale*' string follows ISO-3166 standards, it is must be capital letter; '*code*' string is the encoding names that DBMaker supported.

For a language which has more than one locale, it should be specified which locale. For example, zh_CN or zh_TW, the zh is not valid. At present, DBMaker supports 4 languages error message coding: English, simplified Chinese, traditional Chinese and Japanese.

Error table is put in *dbmaker\5.4\shared\locale\locale_LANG\* directory.

User can use the command SELECT GETSYSINFO('ERRLCODE') to return the error message character set of the client side.

default value: For Windows platform, the client side will get the value of **LANG** in the Registry as the default value, if there is not installation language value in the Registry, then the value will be the language code of the Operation System. For Linux platform, client side will get the value of the environment variable **LANG** as the default value, if the **LANG** did not be set, DBMaker would set client error message character set to be ASCII.

locale definition valid: en, ja, zh_CN, zh_TW

The valid values: en, zh_CN, zh_TW and ja or the combination of them and character sets.

such as:

en
en.ASCII
en.ISO-8859-1
en.ISO-8859-2

en.ISO-8859-5
en.ISO-8859-7
en.UTF-8
ja
ja.SHIFT-JIS

ja.SHIFT_JIS
ja.UTF-8
ja.EUC-JP

ja.EUCJP
zh_CN
zh_CN.GBK
zh_CN.UTF-8
zh_CN.GB18030
zh_TW
zh_TW.BIG5
zh_TW.UTF-8
see also: DB_LCode, DB_CliLCODE

where to use: client side

➲ **Example 1**

```
DB_ErrLCODE=ja;
```

➲ **Example 2**

```
DB_ErrLCODE=ja.EUC_JP;
```

➲ **Example 3**

```
DB_ErrLCODE=ja.UTF-8;
```

DB_EtrPt=<value>

This integer keyword specifies the database server's Subscriber Daemon TCP/IP port number. This is used for express asynchronous table replication. In the source database, this keyword indicates how to connect to the subscriber of the destination database. In the destination database, this keyword starts the Subscriber Daemon.

default value: none

valid range: 1024 ~ 65535

see also: DB_AtrMd, RP_LgDir

where to use: server side (for the source and destination of databases)

DB_ExtHd=<value>

This keyword specifies the threshold value for expanding a file repeatedly. Setting the value to -1 indicates that the autoextend tablespace is automatically expanded from the first file always; setting the value to 0 indicates that the autoextend tablespace is automatically expanded from the smallest file always; setting the value to a number of 1~4294967296/**DB_PgSiz**, 1M~4096M or 1G~4G indicates that the autoextend tablespace is automatically expanded from the smallest file firstly, and don't switch to the second smallest file until the current smallest file's size is bigger than the sum of the second smallest file's size and the value of **DB_ExtHd**, in this way, both performance and balance of file size can be taken into account. The default value of **DB_ExtHd** is 100M. Please refer to section 5.3, *Expanding an Autoextend Tablespace* for more information.

default value: 100M

valid range: -1, 0, 1 ~ 4294967296/**DB_PgSiz**, 1M ~ 4096M, 1G ~ 4G

see also: DB_ExtNp

where to use: server side

DB_ExtNp=<value>

This keyword specifies a size for DBMaker to extend autoextend tablespace. When an autoextend tablespace is exhausted, DBMaker automatically extends by the number of pages/frames specified.

default value: 20 (pages/frames)

valid range: 1 ~ 32767

see also: DB_ExtHd

where to use: **server side**

DB_FBkTm=<string>

This keyword, combined with **DB_FBkTv,** specifies the Backup Server's schedule to perform an on-line full backup. **DB_FBkTm** specifies the first time the Backup Server will perform a full backup. On-line full backup is performed after every time interval specified in **DB_FBkTv**.

➲ **Example**

```
DB_FBKTm = 96/05/01 00:00:00  ;begins May 1, 1996.
DB_FBkTv = 1-12:30:00         ;interval is every one day, 12 hours and 30
minutes.
```

The keywords **DB_FBkTm** and **DB_FBkTv** are only used with the Backup Server.

default value: none

valid range: 1970-01-01 00:00:01 ~ 2037-12-31 23:59:59

see also: **DB_FBkTv**, **DB_BkSvr**, **DB_BkOdr**

where to use: **server side**

DB_FBkTv=<string>

This keyword specifies the full backup time interval. Refer to **DB_FBkTm** for more information.

default value: none (no full backup schedule if **DB_FBkTv** is not set)

valid range: 0-00:00:01 ~ 24854-23:59:59

see also: **DB_BkSvr**, **DB_FBkTm**, **DB_BkOdr**

where to use: **server side**

DB_FltDb=<string>

This keyword specifies the internal storage and value range of the FLOAT column.

Setting the value to 0 indicates the storage for the FLOAT column is 4 bytes and the type name is REAL.

Setting the value to 1 indicates the storage for FLOAT column is 8 bytes and the type name is DOUBLE.

default value: 1

valid range: 0, 1

where to use: server side

DB_FoDir=<string>

This keyword specifies the path for system file objects and system file object subdirectories (depending on the setting of **DB_FoSub**). A full path name for **DB_FoDir** must be specified.

➲ **Example 1**

```
DB_FoDir = /usr/DBMaker/fileobj             ;for UNIX
```

➲ **Example 2**

```
DB_FoDir = c:\dbmaker\fileobj               ;for DOS
```

➲ **Example 3**

```
DB_FoDir = \\NTMachine\dbmaker\fileobj      ;for Microsoft UNC name
```

A new system file object can only be inserted when **DB_FoDir** is set. This keyword is used when a database is started.

default value: null string (the system file object cannot be inserted)

valid range: string with length < 256

see also: **DB_UsrFo, DB_FoSub**

where to use: server side

DB_ForcS=<value>

This keyword instructs DBMaker to force start a database even if an error occurs. Set the value to 1 to enable a forced startup.

default value: 0

valid range: 0, 1

see also: DB_SMode

where to use: server side

DB_ForUX=<value>

This keyword specifies the lock behavior of the "select … for update" statement on the server site.

DBMaker takes U locks on result sets of "select … for update" statements. Special applications that set this value to 1 instruct DBMaker to take X locks on result sets of "select … for update" statements. This setting is required when starting the database.

default value: 0

valid range: 0, 1

where to use: server side

DB_FoSub=<value>

This keyword specifies the maximum number of file objects that may be stored in each system file object subdirectory. Subdirectories are created in the directory specified by **DB_FoDir**. A new file object subdirectory is automatically created when the number of file objects in an existing subdirectory exceeds the threshold value.

➲ **Example**

To set the number of files per file object subdirectory to the value 500,

```
DB_FoSub = 500
```

A new system file object can only be inserted when **DB_FoDir** is set. This keyword is used when a database is started.

default value: 0 (file objects are stored directly in the directory specified by DB_FoDir)

valid range: 100 ~ 1000000, 0 (FO directory has no sub-directories)

see also: DB_FoDir

where to use: server side

DB_FoTyp=<value>

This keyword specifies the ODBC type of the FILE. ODBC does not define the FILE types that are supported by DBMaker. Additionally, some development tools like Borland Delphi or Microsoft Visual Basic may not support FILE type. If you want to allow tools to access data of the FILE type, DBMaker should internally map the FILE type to LONG VARBINARY by setting DB_FoTyp to 1. There is no mapping if **DB_FoTyp** is 0.

default value: 1

valid range: 0 (no mapping)

1 (FILE type mapped to LONG VARBINARY)

where to use: client side

DB_GcChk=<value>

This keyword specifies the minimum number of transaction per second (TPS) that will initiate the group commit transaction protocol.

DBMaker always tracks the current number of transactions. The number of transactions per second is equivalent to the number of sync requests per second. DBMaker uses **DB_GcChk** as the TPS threshold. The group-commit protocol is activated when the server's TPS reaches this threshold. The group-commit is turned off when TPS is below the threshold. For example, if **DB_GcChk** = 20, the group-commit protocol will be turned on if there are over 20 transactions per second.

DB_GcChk allows DBMaker to dynamically switch between activating and deactivating the group-commit protocol. Since transaction activity is not constant, e.g. sometimes very high, sometimes low, DBMaker switches the protocol to avoid unnecessary waiting.

default value: 20

valid range: >0

see also: **DB_GcWtm,DB_GcMxw**

where to use: server side

DB_GcMxw=<value>

This keyword specifies the number of waiting transactions needed to trigger execution of a group-commit. It works in conjunction with **DB_GcWtm**, which affects the maximum waiting time for group-commit.

If the group-commits protocol is activated (refer to **DB_GcChk** for details), DBMaker checks every sync request. If the check meets one of the following conditions, the tube sync process proceeds, otherwise it waits for another sync request before performing the group-commit.

- Transactions reach the maximum waiting time specified by **DB_GcWtm** keyword;

The numbers of transactions waiting for a group-commit exceeds the value specified by **DB_GcMxw** keyword.

➲ **Example**

The maximum waiting time is 30 milliseconds and the number of transactions waiting is 5. DBMaker performs the sync operation when there is at least one transaction waiting over 30 milliseconds or when there are 5 transactions waiting. Set **DB_GcMxw** to 0 to disable group commit.

default value: 0 (disabled)

valid range: 1 ~ 32768

*see also:*DB_GcChk,DB_GcWtm

where to use: server side

DB_GcWtm=<value>

This keyword specifies the maximum waiting time when any transaction waits for group-commit. The longer waiting time may reduce the respond time of the transaction, but may increase the whole throughput for group-commit.

This keyword works with **DB_GcMxw**. Refer to **DB_GcMxw** for detail.

default value: 30 (milliseconds)

valid range: >0

see also: **DB_GcChk, DB_GcMxw**

where to use: server side

DB_IDCap=<value>

This keyword specifies the case sensitivity of all identifiers in a database. Setting the value to 0 indicates that all identifiers are case sensitive. Setting the value to 1, indicates that all identifiers in a database are case insensitive; under this mode, all identifiers are converted to uppercase when defined. This keyword can only be set before database creation, which means changing this keyword value for an existing database has no effect.

default value: 1

valid range: 0 (case sensitive)

1 (case insensitive)

where to use: server side

DB_IdxDp=<value>

This keyword specifies the threshold value for automatically dropping an auto index by auto index daemon, if an index not in use reaches or exceeds the number of days specified by **DB_IdxDp**, the auto index will be dropped by auto index daemon. With the database running, users can set the value of **IDXDP** by calling *setSystemOption()*.

default value: 30 (days)

valid range: 0 ~ 365 (days)

see also: **DB_IdxLg, DB_IdxLn, DB_IdxSv, DB_IdxTm, DB_IdxTv**

where to use: **server side**

DB_IdxLg=<string>

This keyword specifies the auto index's directory where the log file will be saved. The default path is the database installation directory.

default value: database directory

valid range: string with length < 256

see also: **DB_IdxDp, DB_IdxLn, DB_IdxSv, DB_IdxTm, DB_IdxTv**

where to use: **server side**

DB_IdxLn=<value>

This keyword specifies the threshold value for automatically creating an auto index by auto index daemon. If the number of scan log (these logs with the same table id, the table version, the column id list and the expression string list) reaches or exceeds the value specified by **DB_IdxLn**, auto index daemon will create an auto index according to the serial log. With the database running, users can set the value of **IDXLN** by calling *setSystemOption()*.

default value: 1

valid range: 1 ~ 65535

see also: **DB_IdxDp, DB_Idx Lg, DB_IdxSv, DB_IdxTm, DB_IdxTv**

where to use: **server side**

DB_IdxSv=<value>

This keyword is used to activate the auto index server. A value of 1 indicates that the server starts. A value of 0 indicates that the server does not start. With the database running, users can set the value of **IDXSV** by calling *setSystemOption()*.

default value: 0

valid range: 0, 1

see also: **DB_IdxDp, DB_IdxLg, DB_IdxLn, DB_IdxTm, DB_IdxTv**

where to use: server side

DB_IdxTm=<string>

This keyword specifies the first time of the auto index daemon to perform an auto index. The format for **DB_IdxTm** is **yyyy-mm-dd hh:mm:ss**. With the database running, users can set the value of **IDXTM** by calling *setSystemOption()*.

default value: 1970-01-01 00:00:00

valid range: 1970-01-01 00:00:00 ~ 2037-12-31 23:59:59

see also: **DB_IdxDp, DB_IdxLg, DB_IdxLn, DB_IdxSv, DB_IdxTv**

where to use: **server side**

DB_IdxTv=<string>

This keyword specifies auto index daemon interval. The value like "1-12:30:00" means time interval is every one day, 12 hours and 30 minutes. With the database running, users can set the value of **IDXTV** by calling *setSystemOption()*.

default value: 0-01:00:00

valid range: 0-00:00:00 ~ 24854-23:59:59

see also: DB_IdxDp, DB_IdxLg, DB_IdxLn, DB_IdxSv, DB_IdxTm

where to use: server side

DB_IFMem=<value>

This keyword specifies the approximate upper bound of memory allocated for IVF text index search routine. While creating inverted-file text indexes, it requires large amount of memory resource. DBMaker will take a simple rule to decide the maximum memory usage for creating text indexes. If DBMaker cannot detect free memory or free memory resource less than 128 MB, then the maximum memory usage will be 64 MB, otherwise will be half of free memory resource. Users can specify the approximate upper bound of memory usage manually through **dmconfig.ini** by adding a keyword entry **DB_IFMem** in megabyte (MB). This is recommended if your operating system does not provide memory usage information, or if you want to allocate a larger maximum cache size to improve IVF text index query performance.

➲ **Example**

To specify a maximum amount of 256 MB to use for the IVF text index buffer:

```
DB_IFMem = 256
```

default value: None – DBMaker automatically configures the buffer

valid range: 64 ~ (maximum allowed by operating system)

where to use: server side

DB_IOSvr=<value>

This keyword specifies if DBMaker should turn the I/O server and checkpoint daemon on or off. Setting the value to 1 starts the I/O and checkpoint daemons after starting the database server. The keyword is used when the database is started.

default value: 1

valid range: 0, 1

see also: DB_NBufs

where to use: server side

DB_IsoLv=<value>

This keyword specifies the default transaction isolation level when a user connects to the database.

Valid values and the options are:

- **1:** Read Uncommitted
- **2:** Read Committed
- **3:** Repeatable Read
- **4:** Serializable

default value: 1

valid range: 1~ 4

where to use: client side

DB_ItcMd=<value>

This keyword specifies whether implicit data conversion is turned on or off. Setting the value to 1 turns on implicit data conversion; setting the value to 0 turns off implicit data conversion.

default value: 0

valid range: 0, 1

where to use: server side

DB_ITimO=<value>

This keyword specifies the idle timeout interval, specified in seconds. DBMaker will automatically disconnect connections that have no database operations with a higher value than the specified timeout interval. The feature forces idle connections to release all database resources, including buffers, pages, locks, and memory. Default value of this keyword is 0, that is to say, this feature is disabled.

The maximum idle time of communication between client and server is decided by value of **DB_DtClt** (set on client) and value of **DB_ITimo** (set on server). The rule as follows:

- If users only set one of the two keywords, the maximum idle time of communication between client and server is value of the set keyword. In other words, if users only set **DB_DtClt** on client, the maximum idle time of communication between client and server is value of **DB_DtClt**; if users only set **DB_ITimo** on server, the maximum idle time of communication between client and server is value of **DB_ITimo**;
- If users set the two keywords at the same time, there are two conditions:
    - **a)** If value of **DB_DtClt** is smaller than value of **DB_ITimo**, the maximum idle time of communication between client and server is value of **DB_DtClt**.
    - **b)** If value of **DB_DtClt** is bigger than value of **DB_ITimo**, the maximum idle time of communication between client and server is value of **DB_ ITimo**. Meanwhile, DBMaker will automatically reset value of **DB_DtClt** to value of **DB_** ITimo.
- If users don't set anyone of the two keywords, the maximum idle time of communication between client and server is unlimited.

default value: 0 (disable)

valid range: 0 ~ 4294967 (seconds)

see also: DB_DtClt

where to use: **server side**

DB_IttDir=<string>

This keyword specifies the storage directories of the system temporary files. Users may specify up to eight storage directories. If a user doesn't set this keyword, system temporary files will be stored in the directory specified by **DB_DbDir**. Also a user can set the value of **DB_IttDir** in **dmconfig.ini** during the database's operation and this set will take effect immediately.

When use storage directories specified by **DB_IttDir**, because a system temporary file's size limitation is 4GB, in order to avoid the error of full disk in the file's growth process, the system will calculate the size of a storage directory's free space in sequence before creating a system temporary file and stored this file in the storage directory whose free space is more than 4GB. However, if all the storage directories' free space is less than 4GB, this file will be stored in the first storage directory.

default value: directory specified by **DB_DbDir**

valid range: up to eight strings with any length, separated by one space

see also: **DB_DbFil**, **DB_BbFil, DB_TpFil**

where to use: **server side**

DB_JnFil=<string>

This keyword specifies the names of the system journal files. It also specifies the amount of journal files allocated for the database. Up to eight journal files can be specified.

default value: database name with the file extension ***.JNL***. For example, ***DB.JNL***.

valid range: string with length < 256

see also: **DB_JnlSz**, **DB_DbDir**

where to use: **server side**

DB_JnlSz=<value>

This keyword specifies the journal file size. User can specify M (megabytes) or G (gigabytes) as the unit. If M or G is not used, the unit is page. If user specifies the value in M or G, the actual size will be one page less than the specified value. For example, if the page size is 16K and user sets DB_JnlSz to 8M, the size of journal file will be 8176K rather than 8192K.

default value: 1000 (default journal size is 1000*DB_PgSiz KB)

valid range: 100 pages~ 8G

see also: DB_JnFil, DB_DbDir

where to use: server side

DB_LbDir=<string>

This keyword specifies the directory where the user-defined functions (UDF), *dll* in Windows, or *so* in Unix files should be loaded when the database starts.

default value: DB_DbDir

valid range: string with length < 256

see also: DB_JnlSz, DB_DbDir

where to use: server side

DB_LCDec=<value>

This keyword specifies whether detect decimal point setting characters in operating system. Setting this value to 1 will turn on this option and setting it to 0 will turn it off. The default value is 0 and use '.' as DBMaker default decimal point's characters. When the value is set to 1, DBMaker will detect decimal point setting characters in operating system and use it as decimal point setting characters in DBMake, that is to say, if the point setting characters used in operating system is a comma (','), DBMaker will also use a comma as point setting characters; if the the point setting characters used in operating system is '.', DBMaker will also use '.' as point setting characters. It is recommend for users to set this value to 1 when the decimal point characters is ',' in operating system.

default value: 0

valid range: 0,1

see also: DB_LCode

where to use: client side

DB_LCode=<value>

This keyword specifies the language code in the server side. The language code will affect the result of LIKE operations in a query. A 0 indicates the language is ASCII compatible. A 1 indicates the language code is Chinese BIG5 code compatible. A 2 indicates the language code is SHIFT-JIS code compatible. A 3 indicates the language code is GB code compatible. Please refer to the *SQL Command and Function Reference* for more information. This value is required when the database is started.

User can use the command SELECT GETSYSINFO('LCODE') to return the language code setting of the server side.

default value: (set during setup)

valid range:	
0	English (ASCII)
1	Traditional Chinese (BIG5)
2	Japanese (Shift-JIS + Half Corner)
3	Simplified Chinese (GBK)
4	Latin1 code (ISO-8859-1)
5	Latin2 code (ISO-8859-2)
6	Cyrillic code (ISO-8859-5)
7	Greek code (ISO-8859-7)
8	Japanese code（EUC-JP）
9	Simplified Chinese (GB18030)
10	Unicode（UTF-8）

see also: DB_CliLCODE, DB_ErrLCODE

where to use: server side

DB_LetPT=<value>

This keyword specifies the *Lock Escalation Threshold* for escalating a page lock to a table lock. When the number of locks on pages in the same table exceeds the lock escalation threshold, DBMaker will automatically escalate the lock to a table lock.

default value: 60

valid range: 0 , 3 ~ 32767

see also: DB_LetRP

where to use: server side

DB_LetRP=<value>

This keyword specifies the *Lock Escalation Threshold* for an escalating rowlock to a page lock. When the number of locks on rows in the same page exceeds the lock escalation threshold, DBMaker will automatically escalate the lock to a page lock.

default value: 30

valid range: 3 ~ 32767

see also: DB_LetPT

where to use: server side

DB_LgDay=<value>

This keyword specifies the number of days to keep the log files available. The expired log files would be removed by service daemon which is set by **DB_StSvr**. If **DB_LgDay** = 0, the original rule (filenames does not include date) of log system would be adapted. When **DB_LgDay** is greater than 0, the setting of **DB_LgFNo** will be ignored.

Default value: 30

Valis range: 0, 1 ~ 365

see also: DB_LgZip, DB_LgSvr, DB_LgFNo

where to use: server side

DB_LgDir=<string>

This keyword specifies the server log's directory where the log file will save.

default value: the directory specified by *DB_DbDir|lgdir.*

valid range: string with length < 256

see also: **DB_DbDir, DB_LgSvr**

where to use: server side

DB_LgErr=<value>

This keyword sets logged error level when only need to log error when **DB_LgSvr** is 1-4. Set this value to 0 will log the core dumped or DB crash error, which error code>30000; set this value to 1 will log the disconnect or DB crash error, which rc>20000; set this value to 2 will log the abort, disconnect or DB crash error, which rc>10000; set this value to 3 to log the normal, abort, disconnect or DB crash error, which rc>100; set this value to 4 to log the warning or any error, which rc>0.

default value: 3

valid range: 0, 1, 2, 3, 4

see also: DB_LgSvr

where to use: server side

DB_LgFNo=<value>

DBMaker stores the log as DBNAME_1.LOG. When the log file size reaches the default value of 100MB or the size specified by **DB_LgFSz**, the log information is logged to a subsequent log file as follows: DBNAME_2.LOG, DBNAME_3.LOG…, DBNAME_n.LOG. Where n is the number of log files that are generated. This keyword specifies the n value.

default value: 20

valid range: 2 ~ 255

see also: DB_LgSvr,DB_LgFSz, DB_LgDay

where to use: server side

DB_LgFSz=<value>

This keyword specifies, in megabytes, the maximum file size for log and text files. DBMaker stores the log as DBNAME_1.LOG. When the log file size reaches the specified value, subsequent logs are generated as follows: DBNAME_2.LOG, DBNAME_3.LOG … DBNAME_n.LOG, where n is the value specified by **DB_LgFNo** or when **DB_LgFNo** is unspecified the default value of 20 is used. Log files are overwritten starting from DBNAME_1.LOG after the final log file reaches the maximum file size.

*default value:*100

valid range: 10 ~ 1500

see also: DB_LgSvr, DB_LgFNo

where to use: server side

DB_LgLck=<value>

This keyword specifies whether logging status of extra lock information. This log information is generated when a lock time out or deadlock occurs. A value of 0 will not log the extra lock information for lock time outs or deadlocks. A value of 1 will log the extra lock time out information.

default value: 0

valid range: 0, 1

see also: DB_LgSvr

where to use: server side

DB_LgPar=<value>

This keyword specifies the logging status for the value of the input parameter. A value of 0 disables the logging function. A value of 1 enables logging of the input parameter's value. A value of 2 enables logging of the input parameter's value and the SQL commands executed by stored procedures. A value of 3 enables logging trigger's sql statement and parameter. A value of 4 enables logging both stored procedure and trigger's sql statement and parameter.

default value: 0

valid range: 0, 1, 2, 3, 4

see also: DB_LgSvr

where to use: server side

DB_LgPln=<value>

This keyword specifies the logging options of the select, update, and delete statements. A value of 0 disables the logging function. A value of 1 will log the execution plan for these statements.

default value: 0

valid range: 0, 1

see also: DB_LgSvr

where to use: server side

DB_LgSQL=<value>

This keyword specifies the logging status of SQL commands when **DB_LgSvr** = 4. A value of 0 disables logging of SQL commands. A value of 1 enables logging of non-select SQL commands when **DB_LgSvr** = 4. A value of 2 enables logging of all SQL commands when **DB_LgSvr** = 4.

default value: 2

valid range: 0, 1, 2

see also: DB_LgSvr

where to use: server side

DB_LgSTm=<value>

This keyword specifies the interval time for logging. This is only needed to log slow statement when **DB_LgSvr** is set to a value of 2, 3, or 4.

default value: 5 (in seconds)

valid range: 1 ~ 65536 (in seconds)

see also: DB_LgSvr

where to use: server side

DB_LgSvr=<value>

This keyword specifies the status and detail level for the server log. A value of 0 disables logging. A value of 1 enabled logging of errors and the default error level as specified by **DB_LgErr**. A value of 2 enabled logging of slow operations and the default time as specified by **DB_LgSTm**. A value of 3 enables logging of both errors and slow operations. A value of 4 enables logging of connects, disconnects, and commits, rollbacks, SQL commands, errors, and slow operations. The logging status of SQL commands is specified by **DB_LgSQL**. A value of 5 enables logging of exit operations. A value of 6 enables logging of all enter and exit operations. Please refer to section 4.2, *Turning on the Log System* for more information about the Log System.

default value: 0

valid range: 0, 1, 2, 3, 4, 5, 6

see also: **DB_LgErr, DB_LgSTm, DB_LgSQL, DB_LgFSz, DB_LgFNo, DB_LgDir, DB_LgPln, DB_LgPar, DB_LgLck, DB_LgSys**

where to use: server side

DB_LgSys=<value>

A value of 0 enables logging of execution time, rc, service functions, connection ids, usernames, statement ids, login information, error arguments, and SQL statements. A value of 1 is equivalent to **DB_LgSys** = 0 and enables logging of SYSUSER and SYSINFO information A value of 2 is equivalent to **DB_LgSys** = 1 and enables logging of all SYSTEM memory information when the memory can be detected.

default value: 0

valid range: 0, 1, 2

see also: DB_LgSvr, DB_LgSQL,DB_LgFSz,DB_LgErr,DB_LgSTm

where to use: server side

DB_LgZip=<value>

This option is used to packing/zipping the need of closed log files would be necessary in order to save some storage.

default value: 0

valid range: 0, 1

where to use: server side

see also: DB_LgSvr, DB_LgDay, DB_LgFNO

DB_LTimO=<value>

This integer keyword specifies the lock time-out value in seconds. For example, acquiring a lock on a database object already allocated to another transaction, such as a table or a tuple, requires a wait time until the object is released.

Specifying a custom value causes DBMaker will wait for the object until the specified waiting time expires. Once expired, a lock time-out error is returned, or until a lock is acquired on the object before the lock time-out expires. A value of -1 disables the lock time-out. This will cause DBMaker to wait indefinitely until the lock is released.

A value of 0 indicates that you don't want to wait at all. This keyword is used at connection time rather than database startup time. Each connection may have its own **dmconfig.ini** file, especially in client/server mode, so each user can have a lock time-out value.

default value: 5

valid range: -1 ~ 65535

where to use: client side

DB_MaxCo=<value>

This keyword specifies the *hard connection number* when creating databases or starting databases with the new journal operation, and specifies the *soft connection number* when starting a database normally. The *soft connection number* indicates the maximum allowable number of simultaneously active transactions in the database system. Since a connection can own at most a single transaction, it also indicates the maximum number of allowable simultaneous connections for the database system.

The *hard connection number* specifies the maximum number of connections recordable by the journal file. The *hard connection number* must be a multiple of 40 between 240 and 4840, so **DB_MaxCo** is rounded up to the closest multiple of 40 (or 240) to determine the hard connection number. The *hard connection number* may be related to the DBMaker license. When creating databases or starting databases with the new journal operation, the *hard connection number* is the max (default value of **DB_MaxCo**, max users of license) when **DB_MaxCo** is unspecified in the dmconfig.ini; otherwise the *hard connection number* is the max (default value of **DB_MaxCo**, the assigned value) when **DB_MaxCo** is specified in the dmconfig.ini. Please refer to section *Tuning Concurrent Processes* for more information regarding the hard and soft connection numbers and their relation to database performance.

default value: 240

valid range: 240 ~ 4840

see also: DB_SMode

where to use: server side

DB_MTimO=<value>

This integer keyword specifies, in seconds, the client latch time-out value. The client latch is set when entering each API function's entry point and released when exiting the API function. This feature prevents multi-thread applications for using the same connection handle to connect to or perform database operations.

A value of 0 specifies no waiting for the client latch to release. A value greater than 0 specifies how long DBMaker waits until the client latch is released. This minimizes the chance of a problem arising when more than one thread uses the same connection handle to connect and call API functions simultaneously.

default value: 0

valid range: 0 ~ 65535

where to use: client side

DB_MxCmd=<value>

This keyword specifies the maximum number of statement handles for ODBC applications or the maximum number of opened tables (value-1) for the DCI COBOL application. Setting the value to n indicates that all ODBC applications can allocate at most n statement handles in one connection. For DCI at most (n-1) tables plus one statement handle for executing SQL command can be opened.

default value: 257

valid range: 1 ~ 32767

where to use: server side

DB_NBufs=<value>

This keyword specifies the size of data buffers. User can specify M (megabytes) or G (gigabytes) as the unit. If M or G is not used, the unit is page. The actual size will be one page less than the specified value when the value is given in M or G. For example, if the page size is 16 KB and **DB_NBufs** is set to 8 MB, the size of page buffer is 8176 KB rather than 8192 KB. In most cases, DBMaker operates more efficiently given more buffers. This keyword is used when the database is started.

Setting the value to 0 for platforms where DBMaker can detect the physical memory usage allows DBMaker to configure the buffer size automatically. If the physical memory information is unavailable, DBMaker sets the buffer size to 2000 pages.

➲ **Example**

After starting the database, query the SYSINFO table to determine the number of buffers that the database uses. The following command shows that the currently running database uses 500 page buffers:

```
dmSQL> SELECT * FROM SYSINFO WHERE INFO = 'NUM_PAGE_BUF';

 ID       INFO            VALUE
==== ============== ==================
0107 NUM_PAGE_BUF   500

1 rows selected
```

For information on determining the optimal value, please refer to section *Tuning Memory Allocation* in Chapter18, *Performance Tuning*.

default value: 0 (auto-configure)

valid range: 0, 15 (according to the system)

see also: **DB_NJnlB**, **DB_ScaSz**

where to use: **server side**

DB_NetEc=<value>

This keyword specifies the on/off status of network encryption. If network encryption is turned on, all network data transmitted between DBMaker server and clients is encrypted. DBMaker's encryption technique is a combination of DES and RSA.

default value: 0 (off)

valid range: 0 (off) , 1 (on)

where to use: server side

DB_NetZc=<value>

This keyword specifies the on/off status of the data compression when deliver data between server and client. When this function is turned on, data is compressed when transmitted from the server and decompressed when received by the client. This operation reduces the amount of data transmitted and minimizes the transmitted time. A value of 1 enables this function while 0 disables it.

default value: 0 (off)

valid range: 0 (off) , 1 (on)

where to use: client side

DB_NJnlB=<value>

This keyword specifies the number of Journal buffers in shared memory.

User can specify M (megabytes) or G (gigabytes) as the unit. If M or G is not used, the unit is page. When using M or G, the actual size is one page less than the specified value. For example, if the page size is 16 KB and **DB_NJnlB** is set to 8MB, the size of journal file is 8176 KB rather than 8192 KB. The keyword is used when the database is started.

default value: 64 (64 × **DB_PgSiz** KB)

valid range: 16 (according to the system)

see also: **DB_JnlSz**, **DB_NBufs**, **DB_ScaSz, DB_PgSiz**

where to use: server side

DB_OptRt=<value>

This keyword specifies a base for query optimizer from which to choose nested join or merge join in both inner and outer join. A value of 0 is the default. A value of 1 indicates optimizer considers response time instead of execution time. A value of 0 indicates optimizer considers execution time instead of response time.

default value: 0

valid range: 0, 1

where to use: server side

DB_Order=<string>

This keyword specifies the name of the order definition file and is located in the **shared/codeorder** subdirectory of DBMaker's installation directory. The order definition file is a pure text file that affects the sorting results in DBMaker. This keyword is only used when databases are created. Without this keyword, the default sorting sequence is binary sequence.

default value: none

valid range: file name of the user-defined order definition file

see also: DB_LCode

where to use: server side (only for creating a database)

DB_PasWd=<string>

This keyword specifies the password for the default login user ID. If no default login user ID is specified, the value is ignored. The keyword is used when the database starts or at connection time.

default value: null string

valid range: string with length < 16

see also: DB_UsrId

where to use: client side

DB_PgSiz=<value>

This keyword specifies the page size. DBMaker supports 4 KB, 8KB, 16KB or 32 KB as the data file page sizes. The keyword is only used when creating databases.

default value: 8

valid range: 4, 8, 16, 32

where to use: server side (only for creating a database)

DB_PtNum=<value>

This integer keyword specifies the TCP/IP port number used by the database server. It is used at connection time on the client side and at startup time on the server side. The connection will fail if the number for a specific database does not match on all clients and servers.

default value: none

valid range: 1024 ~ 65535

see also: DB_SvAdr

where to use: both client and server sides

DB_ResWd=<value>

Users may need to add (e.g., create, import) objects to the database that use DBMaker reserved words as identifiers (see the *SQL Command and Function Reference* for a full list of DBMaker reserved words). Attempting to add an object that uses a reserved word as an identifier will return an error if **DB_ResWd** is set to a value of 1. If **DB_ResWd** is set to 0, DBMaker will not return an error. This keyword allows objects to be imported that contain reserved words.

default value: 1

valid range: 0, 1

where to use: server side

DB_RmPad=<value>

This keyword specifies whether the space padding for CHAR type data is removed. A value of 0 indicates the space padding for all CHAR type data in a result set is kept. A value of 1 indicates the space padding for all CHAR type data to be removed before copying to a user buffer. It allows a user application to retrieve fixed length CHAR data, excluding the trailing space padding generated in the DBMS during a data insert.

default value: 0

valid range: 0, 1

where to use: client side

DB_RstSn=<value>

This option is used to auto reset serial number to first value when it hit maximum serial value.

default values: 0

valid range: 0, 1

where to use: server side

DB_RTime=<string>

This keyword specifies the target time for a database to be restored from a backup. When performing a database restoration, DBMaker will *roll* forward on the backup files from the earliest time in the backup files to the time specified by **DB_RTime**. If **DB_RTime** is not given, DBMaker will restore the database to the latest time in the backup files, which is the time the backup was performed.

If the **DB_RTime** is later than the backup time, the backup time is used as the value for **DB_RTime**.

The format for **DB_RTime** is **yy/mm/dd hh:mm:ss** or **yyyy/mm/dd hh:mm:ss**.

default value: 0 (70/1/1 00:00:00)

where to use: server side

DB_ScaSz=<value>

This keyword specifies the size of the System Control Area (SCA). Users can specify M (megabytes) or G (gigabytes) as the unit. If M or G is not used, the unit is page. If the user specifies the value in M or G, the actual size is one page less than the specified value. For example, if the page size is 16KB and user sets **DB_ScaSz** to 8M, the size is 8176 KB rather than 8192 KB.

If the minimum memory required by DBMaker for the SCA is larger than the value of **DB_ScaSz**, DBMaker ignore the value and allocate the minimum memory required for the SCA. This keyword is used when the database is started.

default value: 200 (200 × DB_PgSiz KB)

valid range: 1 ~ (according to the system)

see also: DB_NBufs, DB_NJnlB

where to use: server side

DB_SchLgDir=<string>

This keyword specifies the storage directory where **dmschsvr**'s log files are saved. If a user doesn't set this keyword, these log files will be stored in the directory which is specified by **DB_DbDir**.

default value: the directory specified by **DB_DbDir**.

valid range: string with length < 256

see also: DB_DbDir, DB_SchSv, DB_SchLgLev

where to use: server side

DB_SchLgLev=<value>

This keyword sets dmschsvr's log level. There are five log levels:

- **Log dmschsvr's operation status** — Value of **DB_SchLgLev** should be set to 0, and only log **dmschsvr**'s operation status.
- **Log tasks' error message and dmschsvr's operation status** — Value of **DB_SchLgLev** should be set to 1, and log tasks' error message and **dmschsvr**'s operation status.
- **Log tasks' warning, error message and dmschsvr's operation status** — Value of **DB_SchLgLev** should be set to 2, and log tasks' warning, error message and **dmschsvr**'s operation status.
- **Log all information of tasks, schedules and dmschsvr's operation status** — Value of **DB_SchLgLev** should be set to 3, and log all information of tasks, schedules and **dmschsvr**'s operation status.
- **Log dmschsvr's calculating information** — Value of **DB_SchLgLev** should be set to 4, and log dmschsvr's calculating information, so that a user can monitor whether **dmschsvr** is running normally.

default value: 1

valid range: 0, 1, 2, 3, 4

see also: DB_SchSv, DB_SchLgDir

where to use: server side

DB_SchSv=<value>

This keyword specifies whether enable dmschsvr service. Setting this keyword to 1 enables the dmschsvr service. Setting this keyword to 0 disables the dmschsvr service.

default value: 0

valid range: 0, 1

see also: DB_TskNo, DB_SchLgDir, DB_SchLgLev

where to use: server side

DB_SMode=<value>

This keyword indicates the database startup mode. There are six startup modes:

- **normal startup** — Starts a system normally. If the database crashed, DBMaker automatically performs crash recovery to return the database to a consistent and stable state.
- **startup with new journal** — Starts a system normally, but creates a new journal file with a name given by the value for the **DB_JnFil** keyword. If a file with the same name already exists, it is overwritten.
- **startup with rollover** — Uses the backup database files, including the journal file, to start the database. DBMaker will rollover the operations to the point in time specified by **DB_RTime**. This mode is used for database restoration. See the chapter on *Database Recovery, Backup, and Restoration* for more detailed information about rollover.
- **startup as a primary database** — This mode is used for database replication. Starting a system with this mode makes it a primary database, i.e. a source database. Refer to the chapter on *Database Replicati*on for more information.
- **startup as a slave database** — This mode is used for database replication. Starting a system with this mode makes it a slave database, i.e. a destination database. For more detailed information refer to the chapter on *Database Replication .*
- **startup as a read-only database** — Starts up a system normally, but the database is read-only or only provides the read privilege. Starting a database with write permissions in read-only mode prohibits modifications.

default value: 1

valid range: 1 (normal startup)

2 (startup with new journal)

3 (startup with rollover)

4 (startup as a primary database)

5 (startup as a slave database)

6 (startup as a read-only database)

see also: DB_ForcS

where to use: server side

DB_SPDir=<string>

This keyword specifies the path for stored procedure files. The stored procedure files include the generated dynamic link library files and the entire temp files generated during stored procedure creation. A full path name must be specified for **DB_SPDir**.

➲ **Example 1**

The path for **DB_SPDir** in UNIX:

```
DB_SPDir = /usr/DBMaker/data/spdir          ;in UNIX
```

➲ **Example 2**

The path for **DB_SPDir** in Windows:

```
DB_SPDir = c:\dbmaker\data\spdir            ;in Windows
```

default value: <Database Directory>

valid range: string with length < 256

see also: DB_SPInc

where to use: server side

DB_SPInc=<string>

This keyword specifies the path for stored procedure include files. It is used when extra include files are required in a generated stored procedure. A full path name must be specified for **DB_SPInc**.

➲ **Example 1**

The path for **DB_SPInc** in UNIX:

```
DB_SPInc = /usr/DBMaker/data/sp/include      ;in UNIX
```

➲ **Example 2**

The path for **DB_SPInc** in Windows:

```
DB_SPInc = c:\dbmaker\data\sp\include          ;in Windows
```

default value: (current directory where DmServer is running)

valid range: string with length < 256

see also: DB_SPDir

where to use: server side

DB_SPLog=<string>

This keyword specifies the path for stored procedure log files. The stored procedure log files include the error log files sent from the database server while creating a stored procedure and the trace log file for the stored procedure execution. A full path name must be specified for **DB_SPLog**.

➲ **Example 1**

The path for **DB_SPLog** in UNIX:

```
DB_SPLog = /usr/joe/mydata/splog               ;in UNIX
```

➲ **Example 2**

The path for **DB_SPLog** in Windows:

```
DB_SPLog = c:\user\joe\mydata\splog            ;in Windows
```

default value: (current directory client application is running)

valid range: string with length < 256

where to use: client side

DB_SQLSt=<value>

This keyword specifies the display mode of the SQL command monitor. It will affect the display content of the **SQL_CMD** and **TIME_OF_SQL_CMD** columns in the **SYSUSER** system table. The SQL command monitor can contain precise or rough information for executing SQL commands. The precise information consumes more CPU time than the rough information. You can also disabled the SQL command monitor to avoid CPU overhead.

default value: 1

valid range: 0 (turn off SQL command monitor)

1 (display SQL command and rough SQL command executed time)

2 (display SQL command and precise SQL command executed time)

where to use: **server side**

DB_StACL=<value>

This keyword specifies whether a database server checks a user's IP address setting when the user connects to the database.

Setting the keyword to 1 enables ACL checking. Setting the keyword to 0 disables ACL checking.

default value: 0

valid range: 0, 1

where to use: **server side**

DB_StMod=<value>

This keyword specifies the update statistics daemon mode. Setting the value to 0 enables startup update statistics daemon in **general** mode and the sample ratio will be decided by value of the keyword **DB_StsSp** in **dmconfig.ini**; setting the value to 1 enables startup update statistics daemon in **every table setting** mode, that is to say, the sample ratio will be decided by every table's mode and sample ratio. Users can set every table's mode and sample ratio with the command "UPDATE STATISTICS SET".

default value: 0

valid range: 0, 1

see also: DB_StSvr, DB_StsSp, DB_StsTm, DB_StsTv

where to use: server side

DB_StpWd=<string>

This keyword indicates the name of the stopword list definition file that is put in the *shared\stopword* subdirectory of DBMaker's installation directory. The stopword list definition file is a text file, which can affect the text index results in DBMaker. This keyword is used when the database is creating and retrieving text indexes. Without this keyword, database search pre-defined stopword list definition is based on LCode.

➲ **Example**

```
[DB1]
DB_LCode = 2
DB_StpWd = /home/usr/jp.tab
```

default value:

DB_LCode	Stopword List Definition
0: English (ASCII)	en.tab
1: Traditional Chinese (BIG5)	tw.tab
2: Japanese (Shift-JIS + Half Corner)	jp.tab
3: Simplified Chinese (GB)	cn.tab
4: Latin1 code (ISO-8859-1)	en.tab
5: Latin2 code (ISO-8859-2)	en.tab
6: Cyrillic code (ISO-8859-5)	en.tab
7: Greek code (ISO-8859-7)	en.tab
8： Japanese code（EUC-JP）	en.tab
9： Simplified Chinese (GB18030)	cn.tab

10： Unicode(UTF-8)   en.tab

valid range: file name of the user-defined stopword list definition file

see also: DB_LCode

where to use: server side

DB_StrOP=<value>

This keyword specifies whether space padding is removed before applying the string concatenation operator (||). A value of 0 indicates space padding for fixed length CHAR type data is kept before applying the string concatenation operator. A value of 1 indicates the space padding is removed before applying the string concatenation operator.

The keyword can be set on the client or server. If this keyword value is not set in the **dmconfig.ini** file on the client, the option value will rely on the **dmconfig.ini** file on the server. The default value on the server is 0.

default value: 0

valid range: 0, 1

where to use: client and server side

DB_StrSz=<value>

This keyword indicates the length of returned data of STRING type, used only by user-defined function (UDF). Since UDFs can only return data of fixed size, these keywords can limit the size of STRING data for clients to avoid receiving too large string.

default value: 255

valid range: 1 ~ 4096

where to use: client or server side (client has higher priority)

DB_StsSp=<value>

This keyword specifies update statistics page data sample ratio. A value of -1 indicates the database intelligently obtain sample ratio; A value of 0 indicates the database don't update statistics value. Additional, users can set sample ratio to a number between 1 and 100.

default value: 100

valid range: -1, 0, 1 ~ 100

see also: **DB_StSvr, DB_StsTv, DB_StsTm, DB_StMod**

where to use: **server side**

DB_StsTm=<string>

This keyword specifies the first time of the update statistics daemon will perform auto update statistics. The format for **DB_StsTm** is **yyyy-mm-dd hh:mm:ss**.

default value: 1970-01-01 03:00:00

valid range: 1970-01-01 00:00:01 ~ 2037-12-31 23:59:59

see also: **DB_StSvr, DB_StsTv, DB_StsSP, DB_StMod**

where to use: **server side**

DB_StsTv=<string>

This keyword specifies update statistics daemon time interval. The value like "1-12:30:00" means time interval is every one day, 12 hours and 30 minutes.

default value: 1-00:00:00

valid range: 0-00:00:01 ~ 24854-23:59:59

see also: **DB_StSvr, DB_StsTm, DB_StsSP, DB_StMod**

where to use: **server side**

DB_StSvr=<value>

This keyword is used to activate the auto update statistics server. A value of 1 indicates that the server is started. A value of 0 indicates that the server is not running. If the auto update statistics server is activated, it will recalculate database statistics according to the value of **DB_StsTm** and **DB_StsTv**.

default value: 1

valid range: 0, 1

see also: **DB_StsTv, DB_StsTm**, **DB_StsSp**, **DB_StMod**

where to use: **server side**

DB_SvAdr=<string>

This keyword can be a string that specifies the TCP/IP address of the server machine or the host name of that machine. If DNS (Domain Name Service) has been set up properly on the client machine, you can even specify the domain name in this keyword. This keyword is required at connection time on all client and server machines. If this address is not correct, the connection will fail. See a network administrator or a manual on TCP/IP networking for further information.

default value: none

valid range: a.b.c.d or host (domain) name (1 <= a,b,c,d <= 254)

see also: **DB_PtNum**

where to use: **both client and server sides.**

DB_SvLog =<value>

This keyword specifies whether the text of DmServer command line is saved to file. After this function started, *<database directory>**<database name>*.log is saved as ASCII file format. Users with DBA authority can use this function to handle errors of connections. A value of 1 indicates that file is saved, 0 indicates that file is not saved.

default value: 0

valid range: 0, 1

where to use: server side

DB_TCPIP=<value>

This keyword specifies whether or not for dmserver to only allow TCP/IP network protocol. A value of 0 indicates that DmServer allows connections from both named pipe and TCP/IP protocol. A value of 1 indicates that dmserver only allow connections from TCP/IP protocol.

default value: 0

valid range: 0, 1

where to use: server side

DB_TmiFm=<string>

This keyword specifies the time input format for SQL statements. Please refer to the *ODBC Programmer's Guide*, *Appendix B* for more information.

default value: none (accept all input time formats)

valid range: hh:mm:ss.fff

hh:mm:ss

hh:mm

hh

hh:mm:ss.fff tt

hh:mm tt

hh tt

tt hh:mm:ss.fff

tt hh:mm:ss

tt hh:mm

tt hh

hhmmss

see also: DB_TmoFm

where to use: **client or server side (client has higher priority)**

DB_TmoFm=<string>

This keyword specifies the time output format for SQL statements. Refer to the *"ODBC Programmer's Guide*" for more information.

default value: hh:mm:ss

valid range: same as valid range of **DB_TmiFm**

see also: DB_TmiFm

where to use: **client or server side (client has higher priority)**

DB_TMPDir=<string>

This keyword specifies directory where the database puts files saved to TMPTABLESPACE. The system will create data files and BLOB files in this directory. Data files' logical name is **DB_TMPDB**, and the physical name is **DB_TMPDir/DBNAME.TDB**; BLOB files' logical name is **DB_TMPBB**, and the physical name is **DB_TMPDir/DBNAME.TBB**.

DB_TMPDB and **DB_TMPBB** are not keywords but save words, so we can not define **DB_TMPDB** and **DB_TMPBB** in **dmconfig.ini** file.

default value: **DB_DbDir**\tmpDir

see also: DB_DbDir

where to use: **server side**

DB_TpFil=<string>

This keyword specifies the names of the system temporary files. The file size limitation is 4GB. Users may specify up to 128 system temporary files. If a user specifies a system temporary file with a full name, this system temporary file will be stored in the directory specified by this full name, otherwise, in the directory specified by **DB_IttDir**.

A user should set this keyword before starting the database. If it is set during the database's operation, this set will take effect next time when the database starts.

ITTs (internal temporary tables) are initially stored in memory, however, if the size grows to a certain degree, DmServer will write the data of ITTs into the system temporary files on disks and then free the memory allocated to them. The system can automatically generate up to 128 system temporary files and the file's size limitation is 4GB. The format of system temporary files' name is <*the first 8 characters of database name*><*sequence number*>.TMP (sequence number is an integer between 1 and 128, including 1 and 128). When are not used by any user, these system temporary files will be automatically deleted to save disks' space, but they will be recreated when needed. In addition, when a database is shuted down, these files will also be deleted.

Generally speaking, system temporary files are automatically generated by system, but with **DB_Tpfil**, a user also can specify system temporary files with names and directories by himself. However, it's too troublesome and time-consuming to set 128 system temporary files manually, and therefore we don't recommend this keyword. In addition, this keyword will not be displayed in the system table **SYSCONFIG**.

default value: the format of system temporary files' name is <*the first 8 characters of database name*><*sequence number*>.TMP (sequence number is an integer between 1 and 128, including 1 and 128) (DB1.TMP for example)

valid range: up to eight strings with any length, separated by one comma followed by one space

see also: DB_DbFil, DB_BbFil, DB_IttDir

where to use: server side

DB_TskNo=<value>

This keyword indicates that how many tasks can be aroused by **dmschsvr** at the same time.

default value: 30

valid range: 1 ~ 50

see also: DB_SchSv

where to use: server side

DB_Turbo=<value>

This keyword indicates that DBMaker will run with normal catalog cache operation. If your applications rarely modify the structure of a database, you can use **DB_TurBo** mode to speed up data access. See *Performance Tuning* for more information. This keyword is used when the database is started.

default value: 0

valid range: 0, 1

where to use: server side

DB_UsrBb=<string>

This keyword, a special user-defined filename, specifies the physical name of the default user BLOB file used by the operating system.

Follow the physical name; user should specify the size for this file. When specify the file size, there are three unit options: Frame, M (megabytes), and G (gigabytes). Please note that the actual size will be 1 frame than the specified value when using M or G. And the default unit is Frame.

➲ **Example**

To specify the default user BLOB file name with 20 frames:

```
[MY_DB]                                         ;database name
```

```
DB_UsrBb = /disk1/usr/f1.bb  20              ;blob file
```

default value: database name with the extension **.BB** and a size of 3 frames.

valid range: string with length < 256

see also: **DB_BbFil**, **DB_DbDir**, **DB_DbFil**, **DB_UsrDb**, User-defined filename

where to use: server side

DB_UsrDb=<string>

This keyword, a special user-defined filename, specifies the physical name of the default user data file used by the operating system.

Follow the physical name; user should specify the size for this file. When specify the file size, there are three unit options: Page, M (megabytes), and G (gigabytes). Please note that the actual size will be 1 page less than the specified value when using M or G. And the default unit is Page.

➲ **Example**

To specify the default user data file name with 200 pages:

```
[MY_DB]                                      ;database name
DB_UsrDb = /disk1/usr/f1.db  200             ;data file
```

default value: database name with the file extension *.DB and* 200 pages. For example: *db.DB*.

valid range: string with length < 256

see also: **DB_BbFil**, **DB_DbDir**, **DB_DbFil**, **DB_UsrBb**, User-defined filename

where to use: server side

DB_UsrFo=<string>

This keyword indicates whether user file objects can be inserted in a database. Setting this value to 1 enables the user file object function. Refer to the chapter on *Large Object Management* for more information. This value is required when starting a database.

default value: 0

valid range: 0, 1

see also: DB_FoDir

where to use: server side

DB_UsrId=<string>

This keyword specifies the default user ID used to login at database startup or connection time.

default value: null string

valid range: string with length <= 128

see also: DB_PasWd

where to use: client side

DB_WsorT=<value>

This keyword specifies the case sensitivity of word sort order. A value of 0 is the default. A value of 1 indicates the word sort code is case insensitive, A value of 2 indicates the word sort code is case sensitive.

default value: 0

valid range: 0,1,2

see also: DB_LCode, DB_Order

where to use: server side

DD_CTimO=<value>

This keyword specifies the connect time-out value while connecting to a remote database during a DDB operation. In the DDB environment, the coordinator database server may need to establish distributed connections to remote databases.

default value: 5 (seconds)

valid range: >= 1

see also: DD_DDBMd, DD_LTimO, DB_CTimO

where to use: server side

DD_DDBMd=<value>

This keyword specifies whether the DDB (Distributed DataBase) function is enabled on the database server. Turn the value on in order to use DDB operations or table replication functions.

default value: 0

valid range: 0, 1

see also: DD_GTSvr

Where to use: server side

DD_GTItv=<string>

This keyword specifies the schedule of the GTRECO demon to solve pending global transactions. It is used only when the GTRECO server is on.

The input format is 'D hh:mm:ss'.

default value: 00:10:00

valid range: 0 days ~ 24855 days

see also: DD_GTSvr

where to use: server side

DD_GTSvr=<value>

This keyword specifies whether to start up the GTRECO (Global Transaction Recovery) demon while DDB mode is on. The GTRECO demon will automatically solve the pending global transactions that cross the DBMaker database servers.

default value: 1

valid range: 0, 1

see also: **DD_DDBmd**, **DD_GTItv**

where to use: **server side**

DD_LTimO=<value>

This keyword specifies the lock timeout value for the distributed data access during the DDB operation. It takes effect on server-to-server data access only. Refer to **DB_LTimO** for more information on the timeout value.

default value: 5 (seconds)

valid range: >= -1

see also: **DD_DDBmd**, **DD_CTimO**, **DB_LTimO**

where to use: **server side**

DM_DifEn=<value>

This keyword needs to be set under the global section in **DM_COMMON_OPTION,** to denote whether or not to allocate a new environment handle when requested. The global section in **DM_COMMON_OPTION,** in the **dmconfig.ini** file, is for global settings across databases. Keywords like the **DM_DifEn**, apply to all databases in the **dmconfig.ini** file when it is specified. Do not change it unless advice from DBMaker customer support for a special case arises.

default value: 1

valid range: 0, 1

where to use: client side

LG_NPFun=<string>

This keyword is set only in the **DM_COMMON_OPTION** section and specifies the unlogged functions. Its value is an empty string or some ODBC function names separated by commas (,). This keyword is valid only if **LG_PTFun** is not defined in the **dmconfig.ini** file. Once this keyword is specified, the functions listed in the string will not be logged.

default value: "" (empty string, all functions will be logged)

valid range: function list string (e.g. "SQLError, SQLGetDiagRec")

see also: **LG_Path**, **LG_PTFun**, **LG_Trace**, **LG_Time**

where to use: client side

LG_Path=<string>

This keyword is only set in the **DM_COMMON_OPTION** section and specifies the file path name of the output log file.

default value: c:\odbclog.log (for Windows), ./odbclog.log (for UNIX)

valid range: string with length < 256

see also: **LG_NPFun**, **LG_PTFun**, **LG_Time**, **LG_Trace**

where to use: client side

LG_PTFun=<string>

This keyword is set in the **DM_COMMON_OPTION** section and specifies the logged functions. Its value is an empty string or some ODBC function names, separated by commas (,). Once this keyword value is specified, only the functions listed in the string will be logged. If **LG_PTFun** and **LG_NPFun** are set, only the **LG_PTFun** will take effect.

default value: none (all functions will be logged)

valid range: function list string (e.g. "SQLError, SQLGetDiagRec")

see also: **LG_NPFun, LG_Path, LG_Time, LG_Trace**

where to use: **client side**

LG_Time=<value>

This keyword is set in the **DM_COMMON_OPTION** section and specifies whether to log time spent on each function. Setting this value to 1 logs time spent for each function in the output log file, setting it to 0 does not log time. This information can help users find performance bottlenecks in an ODBC program.

default value: 0

valid range: 0, 1

see also: **LG_NPFun**, **LG_Path**, **LG_PTFun**, **LG_Trace**

where to use: **client side**

LG_Trace=<value>

This keyword is set in the **DM_COMMON_OPTION** section and specifies whether the ODBC log is turned on or off. Setting this value to 1 will turn on the ODBC log and setting it to 0 will turn it off. When the ODBC log is on, the called ODBC function, input parameters, output parameters, and returned code or error information will be logged to a specified file. See the following keywords for more detailed information.

default value: 0

valid range: 0, 1

see also: **LG_NPFun, LG_Path, LG_PTFun, LG_Time**

where to use: **client side**

RP_BTime=<value>

This keyword is used for database replication and specifies the starting time of database replication. The format is YYYY/MM/DD hh:mm:ss, e.g. 2000/1/1 01:30:00. You can use **RP_Iterv** to specify the schedule of the database replication.

default value: time of the primary database starting

valid range: YYYY/MM/DD hh:mm:ss (e.g. 2000/1/1 00:00:00)

see also: **DB_SMode**, **RP_Clear**, **RP_Iterv**, **RP_Primy**, **RP_PtNum**, **RP_ReTry**, **RP_SlAdr.**

where to use: **server side of the primary database.**

RP_Clear=<value>

This keyword is used for database replication and specifies whether DBMaker should clear the backup files after replicating them to remote databases. The value 1 clears the files, 0 will keep them.

default value: 0 (no)

valid range: 0 (no), 1 (yes)

see also: **DB_SMode**, **RP_BTime**, **RP_Iterv**, **RP_Primy**, **RP_PtNum**, **RP_ReTry**, **RP_SlAdr**.

where to use: **server side of the primary database.**

RP_Iterv=<value>

This keyword specifies the schedule for the database replication. The format is dd-**hh:mm:ss**, e.g. 1-12:00:00; one day and 12 hours. You can use **RP_BTime** to specify the starting time of database replication.

default value: 1-00:00:00 (one day)

valid range: 0 days ~ 24855 days

see also: **DB_SMode**, **RP_BTime**, **RP_Clear**, **RP_Primy**, **RP_PtNum**, **RP_ReTry**, **RP_SlAdr.**

where to use: **server side of the primary database**

RP_LgDir=<string>

This keyword, used for the asynchronous table replication, specifies the directory where DBMaker puts replication log files for the asynchronous table replication. The replication log files are binary and users should not manually remove them.

default value: the subdirectory named **TRPLOG** under the database home directory

valid range: string with length < 256

see also: **DB_AtrMd**, **DB_EtrPt**

where to use: **server side of the primary database**

RP_Primy=<string>

This keyword, used for the database replication, specifies the address of the primary database.

default value: none

valid range: a,b,c,d or host (domain) name (1 < a,b,c,d < 254)

see also: **DB_SMode**, **RP_BTime**, **RP_Clear**, **RP_Iterv**, **RP_PtNum**, **RP_ReTry**, **RP_SlAdr.**

where to use: **server side of the slave database**

RP_PtNum=<value>

This keyword, used for the database replication, specifies the port number of the RP_RECV_SERVER daemon the slave database. It must be different from the **DB_PtNum** used by the slave database and the same as the port number specified in **RP_SlAdr** on the primary database.

default value: 23001

valid range: 1024 ~ 65535

see also: DB_SMode, RP_BTime, RP_Clear, RP_Iterv, RP_Primy, RP_ReTry, RP_SlAdr

where to use: server side of the slave database

RP_Reset=<value>

RP_Reset is used to indicate to DBMaker to reset the asynchronous table replication system while starting a database. If **RP_Reset** is set to 1, DBMaker will clear all unsent asynchronous table replication logs, remove all **.TRP** files in the **RP_LgDir** (default is **DB_DbDir**/TRPLOG) directory, and reset the asynchronous table replication status while starting the database. That is, DBMaker will ignore all the unsent asynchronous table replication data. After starting database, DBMaker server will reset the **RP_Reset** value to 0 to prevent duplicate resetting asynchronous table replication in next time of starting database.

default value: 0

valid range: 0, 1

see also: RP_LgDir

where to use: server side of primary database

RP_ReTry=<value>

This keyword, used for the database replication, specifies how many times DBMaker will try to connect to remote databases after a network failure.

default value: 0

valid range: 0 ~ 2147483647

see also: DB_SMode, RP_BTime, RP_Clear, RP_Iterv, RP_Primy, RP_PtNum, RP_SlAdr *where to use:* **server side of the primary database**

RP_SlAdr=<string>

This keyword is used for database replication and specifies the slave databases for the primary database to send data. DBMaker supports 1 to 8 slave databases for each primary.

➲ **Example 1**

Syntax for **RP_SlAdr** as follows:

```
RP_SlAdr = address[:port number] {, address[:port number]}
```

The default port number is 23001. In slave databases, commas or blank spaces can separate the information.

➲ **Example 2**

RP_SlAdr port numbers as follows:

```
RP_SlAdr = 192.168.9.222:5100, Server2:5101, Server3
```

There are three slave databases. One is 192.168.9.222 with port number 5100, another is Server2 with port number 5101, and the other is Server3 with default port number 23001.

default value: none.

see also: **DB_SMode**, **RP_BTime**, **RP_Clear**, **RP_Iterv**, **RP_Primy**, **RP_PtNum**, **RP_ReTry**

where to use: **server side of the primary database**

User-defined filename=<physical filename> <pages>

DBMaker allows users to create data or BLOB files and add them to a tablespace when the original tablespace has been filled. Users generally specify a logical file name without a full path when creating a file. Users can map the logical file name to a physical file path name that is used by the operating system to access the file.

➲ **Mapping a file in the dmconfig.ini file:**

```
FILE1 = /disk1/usr/datafile 100
```

Although you specify **FILE1** in DBMaker, DBMaker will create a file */disk1/usr/datafile* with 100 pages, where the page size is determined by the DB_PgSiz when creating the database. If this file has to be moved to another directory, change the physical file name in the **dmconfig.ini** file. There is no need for modification to your programs or SQL scripts. The rule for **DB_DbDir** also applies to user-defined files.

valid range: file name — string with length < 256

size —3 ~ 2147483647

see also: **DB_DbDir**, **DB_UsrBb**, **DB_UsrDb**

where to use: **server side**

B. System Catalog Reference

Part of the definition for a relational database is that all database information must be represented at the logical level in the same way as user data. This information is stored in the *system catalog*, allowing authorized users to use SQL to access information on the database in the same way they access data in SQL tables. This chapter contains descriptions of the system catalog tables and views, organized alphabetically by name. You can query these system catalog tables to view detailed status of a database.

B.1 The System Catalog

The system catalog is a set of tables that contains information on all objects in the database. The system catalog is also known as the *data dictionary*.

All system catalog tables are owned by SYSTEM, and can be read by any user that has at least the Connect Authority. Since system catalog tables belong to SYSTEM, you cannot DROP a system catalog table or a system-defined column, and you cannot INSERT or DELETE rows in a system catalog table.

For users with DBA, SYSDBA and SYSADM authorization, all system catalog tables should be read entirety. However, for users with RESOURCE and CONNECT authorization, the following ten system catalog tables cannot be read: SYSAUTHGROUP, SYSCONFIG, SYSFILE, SYSFILEOBJ, SYSGLBTRANX, SYSLOCK, SYSPENDTRANX, SYSTRPJOB, SYSTRPPOS, SYSWAIT.

Users with different authorization have different read privilege for the following thirty-two system catalog tables: SYSACL, SYSAUTHCOL, SYSAUTHEXE, SYSAUTHMEMBER, SYSAUTHTABLE, SYSAUTHUSER, SYSCMDINFO, SYSCOLUMN, SYSCONINFO, SYSDBLINK, SYSDBDESCOL, SYSFOREIGNKEY, SYSINDEX, SYSINDEXREF, SYSINFO, SYSJARFILE, SYSJAVAARGU, SYSOPENLINK, SYSPROCINFO, SYSPROCJAVA, SYSPROCPARAM, SYSPROJECT, SYSPUBLISH, SYSSCHEMA, SYSSUBSCRIBE, SYSSYNONYM, SYSTABLE, SYSTABLESPACE, SYSTEXTINDEX, SYSTRIGGER, SYSUSER, SYSVIEWDATA.

All users can read the following six system catalog tables: SYSDOMAIN, SYSSCHEDULE, SYSSCHELOG, SYSTASK, SYSTRPDEST, SYSUSERFUNC.

The number of system catalog tables is 48.

B.2 DBMaker System Catalog Tables

The following table lists all of the system catalog tables in DBMaker, and a brief description of what is contained in each table.

TABLE NAME	CONTENTS
SYSACL	IP address privilege information
SYSAUTHCOL	Column privilege information
SYSAUTHEXE	Executable object privilege information
SYSAUTHGROUP	Group information
SYSAUTHMEMBER	Group member information
SYSAUTHTABLE	Table privilege information
SYSAUTHUSER	Security level information
SYSCMDINFO	Stored command information
SYSCOLUMN	Column information
SYSCONFIG	Configuration information
SYSCONINFO	Connection information
SYSDBLINK	Database link information

SYSDESCOL	Dynamic column information
SYSDOMAIN	Domain information
SYSFILE	File information
SYSFILEOBJ	File object information
SYSFOREIGNKEY	Foreign key information
SYSGLBTRANX	DDB global transaction information
SYSINDEX	Index information
SYSINDEXREF	Index and Auto Index information
SYSINFO	Database system information
SYSJARFILE	JAR information
SYSJAVAARGU	Java argument information
SYSLOCK	Lock information
SYSOPENLINK	Open link information
SYSPENDTRANX	Pending distributed transaction information
SYSPROCINFO	Stored procedure information
SYSPROCJAVA	Java SP information
SYSPROCPARAM	Stored procedure parameter information
SYSPROJECT	ESQL project information
SYSPUBLISH	Table replication source information
SYSSCHEDULE	Schedule task information
SYSSCHELOG	Information for recording schedule tasks to be executed
SYSSCHEMA	Schema information
SYSSUBSCRIBE	Table replication destination information
SYSSYNONYM	Synonym information
SYSTABLE	Table information
SYSTABLESPACE	Tablespace information
SYSTASK	Schedule's actions information
SYSTEXTINDEX	Text index information
SYSTRIGGER	Trigger information

SYSTRPDEST	Table replication information
SYSTRPJOB	Information for recording all jobs to be replicated
SYSTRPPOS	Information for distributor to prune transaction log files
SYSUSER	Information on users logged into the database
SYSUSERFUNC	User-defined function information
SYSVIEWDATA	View information
SYSWAIT	Waiting connection information

SYSACL

The SYSACL table lists the IP addresses from where users can connect. Users with RESOURCE or CONNECT authorization only can read information of himself. Users with DBA, SYSDBA or SYSADM authorization can read all information. All users can read the public information.

The table contains user names and IP addresses. The user name "PUBLIC" means that all users must satisfy the setting. "*" means that all IP addresses are allowed to connect.

Column Name	**Description**
USER_NAME	User's name.
ADDRESS	IP address.
PRIVILEGE	ALLOW — This ip address is allowed. BLOCK — This ip address is blocked.

SYSAUTHCOL

The SYSAUTHCOL table lists the columns in all tables on which a user has been granted object privileges. Users with RESOURCE or CONNECT authority can read privilege information of columns on which the privilege is granted to users. Users with DBA, SYSDBA or SYSADM authorization can read all privilege information. If a user is allowed to perform some operations such as INSERT, UPDATE or REFERENCE on the authorized table with all columns (i.e. the return value of INS_ALL, UPD_ALL, or REF_ALL in SYSAUTHTABLE is 1), then the return values of the columns, INS, UPD, REF in SYSAUTHCOL should be ignored.

COLUMN NAME	**DESCRIPTION**
COLUMN_NAME	Name of the column on which privileges have been granted.
TABLE_NAME	Name of the table the column belongs to.
GRANTEE	Name of the user granted privileges on the column. Must be a valid user or group name.
TABLE_OWNER	Name of the user who created the table.
INS	**1** — User has the privilege to insert data into the specified column. **0** — User does not have the privilege to insert data into the specified column.
UPD	**1** — User has the privilege to update data in the specified column. **0** — User does not have the privilege to update data in the specified column.
REF	**1** — User has the privilege to create a constraint that refers to the specified column. **0** — User does not have the privilege to create a constraint that refers to the specified column.

SYSAUTHEXE

The SYSAUTHEXE table contains executable object information. Users with RESOURCE or CONNECT authority can read privilege information of columns on which the privilege is granted to users. Users with DBA, SYSDBA or SYSADM authorization can read all privilege information.

COLUMN NAME	**DESCRIPTION**
OBJNAME	Name of the executable object.
OWNER	User who created the executable object.

Column Name	Description
OBJTYPE	Type of executable object, such as "Procedure", "Command", "Project", etc.
GRANTEE	Name of the user granted privileges on the executable object.

SYSAUTHGROUP

The SYSAUTHGROUP table gives the names of all valid groups in the database. Users with RESOURCE or CONNECT authority cannot read any information. The system may make error 6829 showing no select privilege. Only the user with DBA or higher authority can create group; Users with DBA, SYSDBA or SYSADM authority can read all information.

Column Name	Description
GROUP_NAME	Name of the group.
GROUP_OWNER	User who created the group.
NUM_MEMBERS	Number of members in this group.

SYSAUTHMEMBER

The SYSAUTHMEMBER table lists all members who belong to a group. Users with RESOURCE or CONNECT authority only can read group information about himself. Users with DBA, SYSDBA or SYSADM authority can read all information.

Column Name	Description
MEMBER_NAME	Name of the member who belongs to the group.
GROUP_NAME	Name of the group.

SYSAUTHTABLE

The SYSAUTHTABLE table is a list of all privileges which have been granted on tables, and who they were granted to. Users with RESOURCE or CONNECT authority can read privilege information of columns on which the privilege is granted to users. Users with DBA, SYSDBA or SYSADM authority can read all privilege information.

Column Name	**Description**
TABLE_NAME	Name of the table or view on which privileges have been granted.
GRANTEE	Name of the user granted privileges on the table.
TABLE_OWNER	User who created the table or view.
NUM_RPI_COLS	Number of columns that have privileges granted on them in the table or view.
SEL_ALL	**1** — User has the privilege to select data from all columns in the specified table or view. **0** — User does not have the privilege to select data from columns in the specified table or view.
DEL_ALL	**1** — User has the privilege to delete data in all columns in the specified table or view. **0** — User does not have the privilege to delete data from columns in the specified table or view.
INS	**1** — User has the privilege to insert data into specific columns in the specified table or view. **0** — User does not have the privilege to insert data into any columns in the specified table or view.
INS_ALL	**1** — User has the privilege to insert data into all columns in the specified table or view. **0** — User does not have the privilege to insert data into all columns in the specified table or view, but may still have privileges on individual columns (see INS).
UPD	**1** — User has the privilege to update data in specific columns in the specified table or view. **0** — User does not have the privilege to update data in any columns in the specified table or view.

Column Name	Description
UPD_ALL	**1** — User has the privilege to update data in all columns in the specified table or view. **0** — User does not have the privilege to update data in all columns in the specified table or view, but may still have privileges on individual columns (see UPD).
ALT_ALL	**1** — User has the privilege to alter the definition of the specified table or view. **0** — User does not have permission to alter the definition of the specified table or view.
IDX_ALL	**1** — User has the privilege to create or drop indexes on the specified table or view. **0** — User does not have the privilege to create or drop indexes on the specified table or view.
REF	**1** — User has the privilege to create a CONSTRAINT, WHICH refers to specific columns in the specified table or view. **0** — User does not have the privilege to create a constraint on any columns in the specified table or view.
REF_ALL	**1** — User has the privilege to create a CONSTRAINT, WHICH refers to columns in the specified table or view. **0** — User does not have the privilege to create a constraint which refers to columns in the specified table or view, but may still have privileges on individual columns (see REF).

SYSAUTHUSER

The SYSAUTHUSER table lists the names and authority levels of valid users in a database. Users with RESOURCE or CONNECT authority can read the information about himself. Users with DBA, SYSDBA or SYSADM authority can read all information.

COLUMN NAME	DESCRIPTION
USER_NAME	User ID of each valid user in the database; a user is considered valid if they have been granted CONNECT authority.
DBA	**0** — User does not have DBA authority. **1** — User has DBA authority. **2** — User has SYSDBA authority.
RESOURCE	**0** — User does not have resource authority. **1** — User has resource authority.
ACLORDER	**0** — white-list based. **1** — black-list based.

SYSCMDINFO

The SYSCMDINFO table contains stored command information. Users with RESOURCE or CONNECT authority can read information which the user creates and is granted from other users. Users with DBA, SYSDBA or SYSADM authority can read all information.

COLUMN NAME	DESCRIPTION
MODULENAME	Module name.
CMDNAME	Command name.
CMDOWNER	Command owner.
STATEMENT	Command statement.
NUM_PARM	Number of parameters.
STATUS	**1**: valid. **0**: invalid.
REBTIME	Time of rebuild stored command.
CMDPLAN	Dump stored command execution plan string.

SYSCOLUMN

This table lists every column for each table and view, including the columns in the system catalog tables. Users with RESOURCE or CONNECT authority can read information which the user creates and is granted from other users. Users with DBA, SYSDBA or SYSADM authority can read all information. In the SCALE and RADIX columns, a value of -1 is returned where SCALE and RADIX are not applicable to the data type found in that column.

Column Name	Description
COLUMN_NAME	Name of the column.
TABLE_NAME	Name of the table that owns this column.
TABLE_OWNER	Name of the user who created the table.
COLUMN_ORDER	Order of the column in the table.
NULLABLE	**1** — Column allows null values. **0** — Column does not allow null values.
TYPE_NAME	Type name of the column. Can be any of the following: BINARY, CHAR, NCHAR, DATE, DECIMAL, DOUBLE, FILE, FLOAT, INTEGER, LONG VARCHAR, NCLOB, LONG VARBINARY, SERIAL, SMALLINT, TIME, TIMESTAMP, VARCHAR, or NVARCHAR.
PRECISION	Precision of the column.
SCALE	Scale of the column.
RADIX	Radix of the column.
ASCII_DEF	Default value of ASCII form for the column.
CONSTR	Constraint for the column.
REMARKS	A description of the column.

SYSCONFIG

The SYSCONFIG table lists all server configuration setting. Users with RESOURCE or CONNECT authority cannot read any information in this table. The system may make error 6829 showing no select privilege. Users with DBA, SYSDBA or SYSADM authority can all information.

COLUMN NAME	DESCRIPTION
KEYWORD	The keyword used in dmconfig.ini.
VALUE	The value of the current setting.

SYSCONINFO

The SYSCONINFO table contains information about the connection of the database. Users with RESOURCE or CONNECT authority can read the connect information about himself. Please note that users with DBA, SYSDBA or SYSADM authority also only can read the connect information about himself.

COLUMN NAME	DESCRIPTION
CONNECTION_ID	Connection ID.
INFO1	Reserved.
LAST_SERIAL	The last serial number working on the updated column, that is of a SERIAL type.
LAST_OID	The object ID (OID) of the last inserted record.
INFO2	Reserved.
INFO3	Reserved.

SYSDBLINK

The SYSDBLINK table contains information on remote database links. Users with RESOURCE or CONNECT authority can read information which the user creates. Users with DBA, SYSDBA or SYSADM authority can read all information.

COLUMN NAME	DESCRIPTION
OWNER	Link Owner.
DB_LINK	Link Name.

Column Name	Description
DBSVR	A database section, which contains the remote database information.
USER_NAME	User Name in the remote database.

SYSDESCOL

The SYSDESCOL table contains information on dynamic columns. Users with RESOURCE or CONNECT authority can read information which the user creates and is granted from other users. Users with DBA, SYSDBA or SYSADM authority can read all information.

Column Name	Description
COLUMN_NAME	Name of the dynamic column.
TABLE_NAME	Name of the table that owns JSONCOLS type.
TABLE_OWNER	Name of the user who created the table.
DATA_TYPE	Type of the dynamic column.
COLUMN_NUM	Number of the dynamic column.
LENGTH	Length of the dynamic column.

SYSDOMAIN

The SYSDOMAIN table contains information on domains created in the database. The user with RESOURCE, CONNECT, DBA, SYSDBA or SYSADM can look all information.

Column Name	Description
DOMAIN_NAME	The name of the domain.
DOMAIN_OWNER	The name of the user who created the domain.
ASCII_DEF	Default value of ASCII form for the domain.

COLUMN NAME	DESCRIPTION
TYPE_NAME	Type name of the column. Can be any of the following: BINARY, CHAR, NCHAR, DATE, DECIMAL, DOUBLE, FILE, FLOAT, INTEGER, LONG VARCHAR, NCLOB, LONG VARBINARY, SERIAL, SMALLINT, TIME, TIMESTAMP, VARCHAR, or NVARCHAR.
DATA_LEN	The size of the data type for the domain.
PRECISION	Precision of the domain.
SCALE	Scale of the domain.
CONSTR	Constraint of the domain.
TEXT_CONVERTER	Converts the CLOB, NCLOB, BLOB or FILE data to pure text for creating a text index and PURETEXT() UDF.

SYSFILE

The SYSFILE table contains information on files in the database. The user with RESOURCE or CONNECT authority cannot read any information in this table. The system may make error 6829 showing no select privilege. The user with DBA, SYSDBA or SYSADM authority can read all information.

COLUMN NAME	DESCRIPTION
FILE_NAME	Logical file name.
FILE_TYPE	File type: 1 (data file) . 2 (BLOB file).
FILE_OID	OID of file.
TS_NAME	Name of the tablespace the file is in.
FILE_NPAGES	Number of pages in the file. If the tablespace is an AUTOEXTEND tablespace, then FILE_NPAGES may be less than the physical FILE_NPAGES in the file.
RAWDEV_OFFSET	Not supported in this version of DBMaker.

Column Name	Description
CREATE_TIME	Time when the file was created.

SYSFILEOBJ

The SYSFILEOBJ table contains information on the file objects in the database. This includes both system and user file objects. The user with RESOURCE or CONNECT authority cannot read any information in this table. The system may make error 6829 showing no select privilege. The user with DBA, SYSDBA or SYSADM authority can look all information.

Column Name	Description
FILE_TYPE	**00** — system file object. **01** — user file object.
SHARE	The number of records that share the file object.
FILE_NAME	The full path with the filename to show where the file object is located.

SYSFOREIGNKEY

The SYSFOREIGNKEY table contains information on all foreign keys in the database. The user with RESOURCE or CONNECT authority can read the foreign key information about the table which the user creates and is granted from other users. The user with DBA, SYSDBA or SYSADM can read all information.

Column Name	Description
FK_TBL_NAME	Name of the child table. (the table the foreign key is defined on)
PK_TBL_NAME	Name of the parent table for the foreign key.
FK_TBL_OWNER	Owner of the child table.
PK_TBL_OWNER	Owner of the parent table.
FK_NAME	Name of the foreign key.

COLUMN NAME	DESCRIPTION
UPD_ACT	Update referential action: 0 — No action 1 — Set NULL 2 — Cascade 3 — Set default value
DEL_ACT	Delete referential action: 0 — No action 1 — Set null 2 — Cascade 3 — Set default value

SYSGLBTRANX

The SYSGLBTRANX table contains information on global transactions. The user with RESOURCE or CONNECT authority cannot read any information in this table. The system may make error 6829 showing no select privilege. The user with DBA, SYSDBA or SYSADM authority can read all information.

COLUMN NAME	DESCRIPTION
STATE	Global transaction state: 0 (ISSUE) — A transaction branch is issued, but the participant has not prepared all participants. 1 (PREPARE) — The participant is prepared, but waits for it's father participant to decide whether to commit or abort. 2 (COMMIT) — The participant has decided to commit the global transaction. 3 (PEND_TO_COMMIT) — After crash recovery, this transaction branch has been added to the commit queue and is waiting to be committed. 4 (PEND_TO_ABORT) — After crash recovery, this transaction branch has been added to the abort queue and is pending an abort.
PARTICIPANT	Global transaction participant.

Column Name	Description
GLBTRANXID	Global transaction ID.

SYSINDEX

The SYSINDEX table contains information on indexes in the database. A -1 in the NUM_PAGE, NUM_LEVEL, NUM_LEAF, DIST_KEY, NUM_PAGE_KEY, or CLSTR_COUNT columns means those values are not applicable. The user with RESOURCE or CONNECT authority can read the index information about the table which the user creates and is granted from other users. The user with DBA, SYSDBA or SYSADM authority can read all information.

Column Name	Description
INDEX_NAME	Name of the index.
TABLE_NAME	Name of the table that the index is defined on.
TS_NAME	Specifies the tablespace that the index is stored on.
TABLE_OWNER	Owner of the table that the index is defined on.
UNIQUE	Status flag to indicate uniqueness of the index: **0** — non-unique **1** — unique **3** — primary key **4** — auto index
NUM_COL	Number of columns in the index.
INDEX_OID	OID of index.
NUM_PAGE	Number of index pages.
NUM_LEVEL	Number of levels.
NUM_LEAF	Number of leaf pages.
DIST_KEY	Number of distinct keys.
NUM_PAGE_KEY	Number of pages per key.
CLSTR_COUNT	Cluster count, the number of page I/O while we use the index to access data page. It is related to the number of buffers.

Column Name	Description
CREATE_TIME	Time when the index is created.

SYSINDEXREF

The SYSINDEXREF table contains information on indexes and auto indexes in the database. The user with DBA, SYSDBA or SYSADM authority can read all information.

Column Name	Description
INDEX_NAME	Name of the index.
TABLE_NAME	Name of the table that the index is defined on.
TABLE_OWNER	Owner of the table that the index is defined on.
UNIQUE	Status flag to indicate uniqueness of the index: 0 – non-unique 1 – unique 3 – primary key 4 – auto index
TOTAL_COUNT	The total number of the used auto index.
LASTREF_TIME	The last reference time of the auto index.

SYSINFO

The SYSINFO table gives information on the current state of a database. The user with RESOURCE or CONNECT authority can read information:

SYSINFO.ID	SYSINFO.INFO
0108	DB_PAGE_SIZE
0711	VERSION
0712	FILE_VERSION

The user with DBA, SYSDBA or SYSADM authority can read all information.

The SYSINFO table's schema is different from other system tables. The schema is given below:

```
SYSTEM.SYSINFO (char(4) ID, varchar(32) INFO, varchar(32) VALUE);
```

The meaning of each column follows:

- **ID**: Item identifier. The system information is cataloged according to this ID. The first two characters represent the category, and the following 2 characters identify the item within the category. E.g. for the ID '0105' representing NUM_LOGICAL_READ, the '01' means it belongs to the page and I/O category and the '05' is its sequence in the page and I/O category. Users can sort or filter the SYSINFO by ID.
- **INFO**: Item name of the system information. E.g. the name 'NUM_LOGICAL_READ' means the number of logical disk read.
- **VALUE**: All values of system information are returned as VARCHAR data.

➲ **Example**

The statement below will show the number of logical disk read:

```
dmSQL> select ID, INFO, VALUE from SYSTEM.SYSINFO where INFO =
'NUM_LOGICAL_READ';
 ID              INFO                         VALUE
==== ======================== ==========================
0105 NUM_LOGICAL_READ         338

1 rows selected
```

Below lists all items in the SYSINFO catalog.

PAGE AND I/O INFORMATION:

SYSINFO.ID	SYSINFO.INFO	DESCRIPTION
0101	NUM_IDX_SPLIT	Number of index page splits occurring.
0102	NUM_PAGE_COMPRESS	Number of data pages compressed, i.e. page reorganization.

SYSINFO.ID	SYSINFO.INFO	DESCRIPTION
0103	NUM_PHYSICAL_READ	Number of physical disk reads. I/O unit is Page.
0104	NUM_PHYSICAL_WRITE	Number of physical disk writes. I/O unit is Page.
0105	NUM_LOGICAL_READ	Number of logical reads. I/O unit is Page.
0106	NUM_LOGICAL_WRITE	Number of logical writes. I/O unit is Page.
0107	NUM_PAGE_BUF	Number of page buffers. Counting unit is Page.
0108	DB_PAGE_SIZE	Page size of the database (in bytes).
0109	DB_SCA_SIZE	The size of the System Control Area in the shared memory (page).
0110	NUM_JOURNAL_BUF	The number of the journal buffers in the shared memory (block).
0111	DB_SYSCB_SIZE	The size of internal system control block in the shared memory(byte).
0112	READ_HIT_RATIO	The page buffer read hit ratio.
0113	WRITE_HIT_RATIO	The page buffer write hit ratio.

NOTE *1 Page size is determined by the keyword* ***DB_PgSiz****.and can be 4 K,8 K,16 K or32 K.*

JOURNAL INFORMATION:

SYSINFO.ID	SYSINFO.INFO	DESCRIPTION
0201	NUM_JNL_BLK_READ	Number of journal blocks read from journal files expressed in blocks.
0202	NUM_JNL_BLK_WRITE	Number of journal blocks

SYSINFO.ID	SYSINFO.INFO	DESCRIPTION
		written to journal files expressed in blocks.
0203	NUM_JNL_REC_WRITE	Number of journal records generated. The new journal record is placed in the journal buffer first.
0204	NUM_JNL_FRC_WRITE	Number of journal forced writes. This number is the I/O number of flushing dirty journal buffer to disk.
0205	NUM_JOURNAL_FILE	Number of journal files.
0206	NUM_JOURNAL_BLOCKS	Number of journal blocks in a file. The total number of journal blocks in a database is: NUM_JOURNAL_FILE* NUM_JOURNAL_BLOCKS
0207	NUM_JNR_BLOCK_FREE	Number of free journal blocks.
0208	CURRENT_JOURNAL_FN	The file number of the currently used journal file.
0209	CURRENT_JOURNAL_BN	The current block number of the journal file. Each journal block of journal files has a unique address that is formed by CURRENT_JOURNAL_FN and CURRENT_JOURNAL_BN. The block number of a journal file is counted from 0.
0210	JOURNAL_FLUSH_RATE	(the number of journal file blocks written by DmServer/the times of flush) the number of journal files in buffer.

NOTE *1 Block = 512 bytes.*

Transaction Information:

SYSINFO.ID	SYSINFO.INFO	Description
0301	NUM_STARTED_TRANX	Number of started transactions.
0302	NUM_COMMITED_TRANX	Number of committed transactions.
0303	NUM_ABORTED_TRANX	Number of aborted transactions.
0304	NUM_CHECKPOINT	Number of checkpoints.
0305	NUM_COMMIT_WAITER	Number of transactions awaiting group commit.

Lock Information:

SYSINFO.ID	SYSINFO.INFO	Description
0401	NUM_ROW_LOCK_UPG	Number of page locks escalated (i.e. row locks escalated to a page lock).
0402	NUM_PAGE_LOCK_UPG	Number of table locks escalated (i.e. page locks escalated to a table lock).
0403	NUM_LOCK_TIMEOUT	Number of failed locks due to timeout.
0404	NUM_LOCK_WAIT	Number of locks waiting.
0405	NUM_LOCK_REQUEST	Number of locks requested.
0406	NUM_DEADLOCK	Number of deadlocks detected.

Connection Information:

SYSINFO.ID	SYSINFO.INFO	Description
0501	NUM_MAX_HARD_CONNECT	Maximum number of allowed connections for a database (hard limitation of connection, i.e. **DB_MaxCo** when database start with new journal or creating a new database).

SYSINFO.ID	SYSINFO.INFO	DESCRIPTION
0502	NUM_MAX_SOFT_CONNECT	Maximum number of allowed connections at a time (soft limitation of connection, i.e. **DB_MaxCo** when database started normally). The soft limitation of connections is less or equal to the hard limitation of connections (*In previous versions of DBMaker this was called NUM_MAX_TRANX*).
0503	NUM_CONNECT	Number of currently active connections (*In previous versions of DBMaker this was called NUM_ACT_TRANX*).
0504	NUM_PEAK_CONNECT	Maximum number of active connections at a time (peak number of active connections).

DATA OPERATION INFORMATION:

SYSINFO.ID	SYSINFO.INFO	DESCRIPTION
0601	NUM_SQL_SELECT	Number of SELECT operations.
0602	NUM_SQL_INSERT	Number of INSERT (including INSERT INTO) operations.
0603	NUM_SQL_UPDATE	Number of UPDATE operations.
0604	NUM_SQL_DELETE	Number of DELETE operations.
0605	NUM_SQL_PREPARE	Number of SQLPrepare() calls to server.
0606	NUM_SQL_EXECUTE	Number of SQLExecute() calls to server.
0607	NUM_SQL_EXEDIRECT	Number of SQLExecDirect() calls to server.
0608	NUM_SQL_FETCH	Number of fetched data passing across the network.

DATABASE INFORMATION:

SYSINFO.ID	SYSINFO.INFO	DESCRIPTION
0701	SYSINFO_RESET_TIME	Time the counter of SYSINFO was restarted (*new*). This is used for record the time the SYSINFO is reset. The setting happen when: 1. dmSQL> set SYSINFO clear; 2. One counter is overflow, and then reset all counters. The checking is done when: 2-1. Each time to select SYSINFO table. 2-2. Every about 5 seconds by I/O Server.
0702	DCCA_SIZE	Total size of DCCA. **Byte unit**.
0703	FREE_DCCA_SIZE	Available size of DCCA. **Byte unit**.
0704	DDB_MODE	Distributed database mode: **ON**: Enable **OFF**: Disable.
0705	BACKUP_MODE	Backup mode: . **NON-BACKUP**: non backup mode (**DB_BMode** = 0). . **BACKUP-DATA**: backup data only mode (**DB_BMode** = 1). . **BACKUP-DATA-AND-BLOB**: backup data and BLOB mode (**DB_BMode** = 2).
0706	USER_FO_MODE	User file object mode: **ON**: Enable **OFF**: Disable.
0707	READ_ONLY_MODE	Read-only mode: **ON**: Enable

SYSINFO.ID	SYSINFO.INFO	DESCRIPTION
		OFF: Disable.
0708	FRAME_SIZE	BLOB frame size. **Byte unit**.
0709	CREATE_DB_TIME	Time the database was created.
0710	START_DB_TIME	Time the database was started.
0711	VERSION	DBMaker version.
0712	FILE_VERSION	Database file version.
0713	FORCE_NEW_JNL_TIME	Time the database startup with force new journal.
0714	START_NO_JNL_TIME	Time of turning the journal off.
0715	END_NO_JNL_TIME	Time of turning the journal on.
0716	MAX_ITT_SIZE	The max size of allowed memory for internal temporary table. (in bytes)
0717	CURRENT_ITT_SIZE	The current size of internal temporary table. (in bytes)
0718	FULL_BACKUP_COST	The last full backup time cost.
0719	DIFF_BACKUP_COST	The last differential backup time cost.
0720	DIFF_BACKUP_PCT	(The last diff backup size) / (Database size) * 100%

SYSTEM INFORMATION:

SYSINFO.ID	SYSINFO.INFO	DESCRIPTION
0801	CPU_USAGE	The average CPU load during a short period (about 5 seconds) (*new*). (**0 ~ 100 %**) SUPPORTED PLATFORM: Solaris, LINUX, Windows (only count the first CPU, and need pdh.dll library). You must start I/O Server to enable this item.
0802	TOTAL_MEMORY	Total physical memory. **Byte unit**.

SYSINFO.ID	SYSINFO.INFO	DESCRIPTION
		SUPPORTED PLATFORM: Solaris, LINUX, Windows, UNIX that supports POSIX standard.
0803	TOTAL_FREE_MEMORY	The current free physical memory (*new*). **Byte unit**. SUPPORTED PLATFORM: Solaris, LINUX, Windows, UNIX that supports POSIX standard.
0804	TOTAL_SWAP_SPACE	Total swap space. **Byte unit**. SUPPORTED PLATFORM: Solaris, LINUX, Windows.
0805	TOTAL_FREE_SWAP_SPACE	The current free swap space. (*new*). **Byte unit**. SUPPORTED PLATFORM: Solaris, LINUX, Windows.

For unsupported versions the value is NULL. The SYSINFO catalog is the collection of accumulated counters. Users should know that there are two ways to reset SYSINFO:

1. A user executes the "set SYSINFO clear" command.
2. When one counter overflows, all counters in the SYSINFO will be reset. DBMaker checks for overflow:
   a) Each time the SYSINFO table is selected.
   b) Every 5 seconds by the I/O Server.

Users can get the reset time by executing the following statement:

```
dmSQL> select VALUE from SYSTEM.SYSINFO where INFO = 'SYSINFO_RESET_TIME';
```

SYSJARFILE

The SYSJARFILE table contains information on java jar package.

The user with RESOURCE or CONNECT authority can read table or view information which the user creates and is granted from other users. The user with DBA, SYSDBA or SYSADM can read all information.

COLUMN NAME	DESCRIPTION
JAR_NAME	JAR name.
JAR_OWNER	JAR owner.
JARFILE_NAME	Name of JAR file.

SYSJAVAARGU

The SYSJAVAARGU table contains java argument information. The user with RESOURCE or CONNECT authority can read table or view information which the user creates and is granted from other users. The user with DBA, SYSDBA or SYSADM authority can read all information.

COLUMN NAME	DESCRIPTION
PROC_NAME	Procedure name.
PROC_OWNER	Procedure owner.
DATATYPE_NAME	Name of data type.
ORDER	The order of java argument.
DATATYPE	Type of data.

SYSLOCK

The SYSLOCK table contains information on status for locks on objects in s database. The user with RESOURCE or CONNECT authority cannot read any information in this table. The system may make error 6829 showing no select privilege. The user with DBA, SYSDBA or SYSADM authority can read all information.

NOTE *Lock granularity can be SYSTEM, TABLE, PAGE, or TUPLE, lock status can be GRANTED, WAITING, or CONVERT, and lock mode can be NONE, IS, S, IX, SIX, or X.*

COLUMN NAME	DESCRIPTION
LK_OBJECT_ID	OID of locked object.

TABLE_ID	OID of the table containing the locked object.
LK_GRAN	Lock granularity, SYSTEM, TABLE, PAGE, or TUPLE.
HOLD_LK_CONNECTION	Connection ID that is holding the lock on the object.
LK_STATUS	Lock status, GRANTED, WAITING, or CONVERT.
LK_CURRENT_MODE	Current lock mode of object.
LK_NEW_MODE	New lock mode of object.

SYSOPENLINK

The SYSOPENLINK table contains information on open database links. The user with RESOURCE or CONNECT authority can read information which the user creates. The user with DBA, SYSDBA or SYSADM authority can read all information.

Column Name	**Description**
DB_LINK	Open link.
DBSVR	Server.
USER_NAME	User name.
TXN_STATUS	Transaction status: 'R' — Read 'W' — Write 'N' — No transaction

SYSPENDTRANX

The SYSPENDTRANX table contains information on uncommitted transactions in a distributed database environment. The user with RESOURCE or CONNECT authority cannot read any information in this table. The system may make error 6829 show no select privilege. The user with DBA, SYSDBA or SYSADM authority can read all information.

COLUMN NAME	DESCRIPTION
XIDFORMAT	Format ID denoting the type of the global coordinator, DBMaker is 22873.
PREPAREDTIME	Time of the prepared commit transaction.
GLBTRANXID	Global transaction ID.

SYSPROCINFO

The SYSPROCINFO table contains information on stored procedures. The user with RESOURCE or CONNECT authority can read procedure information which the user creates and is granted from other user. The user with DBA, SYSDBA or SYSADM authority can read all information.

For local temp procedure, user only can see the procedure which is created in current connection.

For global temp procedure, all users (contain different connection) can see the procedure's information in the global temp procedure's life cycle. (means until be dropped)

COLUMN NAME	DESCRIPTION
QUALIFIER	Qualifier.
PROC_OWNER	Procedure owner.
NAME	Procedure name.
NUM_INPUT_PARAMS	Number of input parameters.
NUM_OUTPUT_PARAMS	Number of output parameters.
NUM_RESULT_SETS	Number of result sets.
REMARKS	Remarks.

COLUMN NAME	DESCRIPTION
PROC_TYPE	Procedure type: **1** (SQL_PT_PROCEDURE) — Procedure **2** (SQL_PT_FUNCTION) — Function
TEMP_TYPE	Temporary tablespace type: **0** permanent **1** global **2** local
SID	Connection ID.

SYSPROCJAVA

The SYSPROCJAVA table contains java procedure information. The user with RESOURCE or CONNECT authority can read table or view information which the user creates and is granted from other user. The user with DBA, SYSDBA or SYSADM authority can read all information.

COLUMN NAME	DESCRIPTION
PROC_NAME	Procedure name.
PROC_OWNER	Procedure owner.
CLASS_ID	Class ID.
NUM_ARGU	Argument number.
NUM_RELATED_JARFILE	The number of related JAR file.
JAR_NAME	Name of java procedure.
JAR_OWNER	Owner of java procedure.

SYSPROCPARAM

The SYSPROCPARAM table contains information on stored procedure parameters. The user with RESOURCE and CONNECT authority can read procedure parameter information about the procedure which the user creates and is granted from other user. The user with DBA, SYSDBA or SYSADM authority can read all information.

For local temp procedure, users only can see the procedure's parameter information which is created in current connection.

For global temp procedure, all the user (contain different connection) can see the procedure's parameter information in the global temp procedure's life cycle. (means until be dropped)

Column Name	**Description**
QUALIFIER	Qualifier.
OWNER	Procedure owner.
PROC_NAME	Procedure name.
PARAM_NAME	Parameter name.
PARAM_TYPE	Parameter type: 1 (SQL_PARAM_INPUT) — Input. 3 (SQL_PARAM_OUTPUT) — Output. 4 (SQL_RETURN_VALUE) — Return value. 5 (SQL_RESULT_COL) — Result set.
DATA_TYPE	Data type.
TYPE_NAME	Type name.
PRECISION	Precision.
LENGTH	Length.
SCALE	Scale.
RADIX	Radix.
NULLABLE	Nullable column: 1 — Allows null values. 0 — Does not allow null values.
REMARKS	Remarks.
SID	Connection ID.

SYSPROJECT

The SYSPROJECT table contains information on ESQL projects. The user with RESOURCE and CONNECT authority can read ESQL project information about the procedure which the user creates. The user with DBA, SYSDBA or SYSADM authority can read all information.

COLUMN NAME	DESCRIPTION
PROJECT_NAME	Project name.
PROJECT_OWNER	Project owner.
MODULE_NAME	Module name.
MODULE_OWNER	Module owner.
MODULE_SOURCE	Module source.
REF_CMD	Internal usage.

SYSPUBLISH

The SYSPUBLISH table contains information on table replication sources. The user with RESOURCE and CONNECT authority can read the replication information about the user's table. The user with DBA, SYSDBA or SYSADM authority can read all information.

COLUMN NAME	DESCRIPTION
REPLICATION_NAME	Name of replication.
TYPE	**S** — Synchronous **A** — Asynchronous
TABLE_OWNER	Owner of table being replicated.
TABLE_NAME	Name of table being replicated.
NUM_PROJECT	Number of projected columns.
FRAGMENT	Fragment string.
NUM_SUBSCRIBER	Number of subscribers.

SYSSCHEDULE

The SYSSCHEDULE table contains information on scheduled tasks. When a database is created, system will automatically add a record about schedule **schelogcl** into SYSSCHEDULE which is used to run task **tasklogcl** at 1:10 a.m. every day. This schedule is disabled by default, and only a user with DBA authority or higher has the right to enable it with SCHEDULE_ENABLE or disable it with SCHEDULE_DISABLE.

Column Name	**Description**
SCHEDULE_OWNER	Owner of the schedule.
SCHEDULE_NAME	Name of the schedule.
TASK_NAME	Name of the task called by the schedule.
TIMETABLE	The interval between a schedule and the previous schedule.
START_TIME	Start time of executing the schedule.
END_TIME	End time of executing the schedule.
STATUS	Status of a schedule: **0** – disable **1** – enable

SYSSCHELOG

The SYSSCHELOG table contains information on executing records of scheduled tasks. It is recommended that users should clear excessive logs stored in it and keep logs of recent days regularly.

Column Name	**Description**
SCHEDULE_OWNER	Owner of the schedule.
SCHEDULE_NAME	Name of the scheduled tasks.
CONNECTION_ID	Connection id of a task.
BEGIN_TIME	Start time of executing a task.
END_TIME	End time of executing a task.
STATUS	Executing status of a task.
ERROR_MSG	The error message occurring when a task fails to execute.

SYSSCHEMA

The SYSSCHEMA table contains the relationship between schema name and schema owner; the user with RESOURCE and CONNECT authority can read schema information about the user. The user with DBA, SYSDBA or SYSADM authority can read all information.

COLUMN NAME	DESCRIPTION
SCHEMA_NAME	Name of schema.
SCHEMA_OWNER	Owner of schema.

SYSSUBSCRIBE

The SYSSUBSCRIBE table contains information on table replication targets. The user with RESOURCE and CONNECT authority can read the replication information about the user's table and related table. The user with DBA, SYSDBA or SYSADM authority can read all information.

COLUMN NAME	DESCRIPTION
BASE_TABLE_OWNER	Base table owner.
BASE_TABLE_NAME	Base table name.
REPLICATION_NAME	Replication name.
DB_LINK	Database link.
TABLE_OWNER	Table owner.
TABLE_NAME	Table name.

SYSSYNONYM

The SYSSYNONYM table contains information on synonyms defined in a database. The user with RESOURCE and CONNECT authority can read synonym information about the table or view which the user creates. The user with DBA, SYSDBA or SYSADM authority can read all information.

Column Name	Description
SNAME	Synonym name.
OWNER	Synonym owner.
TV_NAME	The source table/view name of the synonym.
TV_OWNER	Table/view owner.
TV_LINK	Link name of a table or view in a remote database.
TV_SERVER	Database name of a table or view in a remote database.

SYSTABLE

The SYSTABLE table contains information on tables in the database. The user with RESOURCE and CONNECT authority can read table or view information which the user creates and is granted from other user. The user with DBA, SYSDBA or SYSADM authority can read all information.

Column Name	Description
TABLE_NAME	Name of the table.
TABLE_OWNER	Owner of the table.
TABLE_TYPE	Table type: SYSTEM TABLE, SYSTEM VIEW, TABLE, or VIEW.
LOCKMODE	Lock mode applied to the table: **T** — table lock **P** — page lock **R** — row lock The default lock mode is row lock.
CACHEMODE	Cache mode of the full table scan: **T** — there is caching (true). **F** — there is no caching (false).
TS_NAME	Name of the tablespace the table is in.
TABLE_OID	The OID of the table.
NUM_COL	Number of columns in the table.
NUM_INDEX	Number of indexes on the table.

Column Name	Description
NUM_PAGE	Number of pages in the table. The default value is -1. When the user updates statistics on the table, the true value of NUM_PAGE will be returned.
NUM_FRAME	Number of BLOB frames in the table.
NUM_ROW	Number of rows in the table. The default value is -1. When the user updates statistics on the table, the true value of NUM_ROW will be returned.
NUM_INDIRECT_ROW	Number of indirect rows.
AVERAGE_LENGTH	Average length for data in the table. Default value is -1. When the user updates statistics on the table, the true value of AVERAGE_LENGTH will be returned.
CREATE_TIME	Time that the table was created.
UPD_STS_TIME	Last time table's statistics were updated.
CONSTR	Constraint for the table.
FILLFACTOR	The **FILLFACTOR** specifies the percentage of the page that can be filled before it stops allowing new data to be inserted (to allow room for updates). The default value is 100 (%).
SERIAL_COL_ID	Serial column is located in the nth column in the table.
SERIAL_START_NO	Serial column starting number. The default value is 1.
REMARKS	A description of the table.
NUM_TRIG	Number of triggers on table.
NUM_TEXTINDEX	Number of text indexes on table.
NUM_PUBLICATION	Number of publications on table.
NUM_DEST	Number of target databases for asynchronous replication.
UPD_STS_MODE	Update statistics mode for the table.
UPD_STS_SAMPLE	Update statistics sample ratio for the table.

SYSTABLESPACE

The SYSTABLESPACE table contains information on all tablespaces in the database. The user with CONNECT privilege cannot look any information in this table. The system may make error 6829 show no select privilege. The user with RESOURCE, DBA, SYSDBA or SYSADM can look all information.

COLUMN NAME	DESCRIPTION
TS_NAME	Name of the tablespace.
TS_TYPE	Type of tablespace: **0** (fixed(W))— NORMAL. **1** (ext(W)) — AUTOEXTEND. **2** (fixed(R))—READ ONLY (NORMAL). **3** (ext(R))—READ ONLY (AUTOEXTEND).
TS_OID	OID of tablespace.
NUM_FILES	Number of files in the tablespace.
NUM_PAGES	Number of pages in the tablespace. If the tablespace is autoextend, then NUM_PAGES may be less than the real NUM_PAGES in the tablespace.
NUM_FREE_PAGES	Number of free pages available in the tablespace.
NUM_PE	Number of PE.
NUM_FRAMES	Number of BLOB frames in the tablespace.
CREATE_TIME	Time of tablespace created.
NUM_FREE_FRAMES	Number of free BLOB frames available in the tablespace.

SYSTASK

The SYSTASK table contains information on a schedule's actions. When a database is created, system will automatically add a record about task **tasklogcl** into SYSTASK. This task is used to clean logs of which creation is later than the most recent logs by 10 days by default by calling SCHELOG_CLEAN. Only a user with DBA authority or higher has the right to alter the number of days between creation time of logs to delete and that of the most recent logs, namely the value of reserve_day.

COLUMN NAME	DESCRIPTION
TASK_OWNER	Owner of a task.
TASK_NAME	Name of a task.
TASK_TYPE	Type of a task.
ACTIONS	Actions of a task.

SYSTEXTINDEX

The SYSTEXTINDEX table contains information on text indexes. The user with RESOURCE and CONNECT privilege can look the text index information about the table which the user creates and is granted from other user. The user with DBA, SYSDBA or SYSADM can look all information.

COLUMN NAME	DESCRIPTION
TEXTINDEX_NAME	Text index name.
TABLE_NAME	Table name.
TABLE_OWNER	Table owner.
INDEX_OID	Index OID.
TYPE	Two kinds of text indexes: signature text index and IVF text index.
FACTOR1	Internal management data.
FACTOR2	Internal management data.
FACTOR3	Internal management data.
FACTOR4	Internal management data.
FACTOR5	Internal management data.
FACTOR6	Internal management data.
FACTOR7	Internal management data.
FACTOR8	Internal management data.

COLUMN NAME	DESCRIPTION
FACTOR9	Internal management data.
FACTOR10	Internal management data.
CREATE_TIME	Time that the text index was created.
REBUILD_TIME	Time that the text index was rebuild.

SYSTRIGGER

The SYSTRIGGER table contains information on triggers. The user with RESOURCE and CONNECT privilege can look trigger information about the table which the user creates. The user with DBA, SYSDBA or SYSADM can look all information.

COLUMN NAME	DESCRIPTION
TBNAME	Table name.
TBOWNER	Table owner.
TRIGNAME	Trigger name.
TRIGEVENT	Trigger event: 1 — Insert event 2 — Delete event 3 — Update event trigger 4 — Update column event
NUM_COL	Number of columns.
SCOL_NUM	The lowest column number updated for trigger.
TRIGTYPE	Trigger type: 1 — BEFORE and FOR EACH STATEMENT 2 — BEFORE and FOR EACH ROW 4 — AFTER and FOR EACH STATEMENT 8 — AFTER and FOR EACH ROW
STATUS	Status: 1: enable 0: disable

COLUMN NAME	DESCRIPTION
OLD	Old value.
NEW	New value.
MODE	**1**: valid trigger **0**: invalid trigger.
TRIGDEF	Trigger definition.

SYSTRPDEST

The SYSTRPDEST table contains information on schedules used by asynchronous table replication. Because the user with RESOURCE privilege need to get the information on schedules used by asynchronous table replication.The user with RESOURCE, CONNECT, DBA, SYSDBA or SYSADM can look all information.

COLUMN NAME	DESCRIPTION
SVRNAME	Remote database name.
USER_NAME	User account in the remote database.
STATUS	Status of the remote database: **0**: ok **1**: error **2**: retry **3**: suspend
BEGTIME	Beginning time of replication.
INTERVAL	Interval between replicating.

SYSTRPJOB

The SYSTRPJOB table contains information on logs used by asynchronous table replication. The user with RESOURCE and CONNECT privilege cannot look any information in this table. The system may make error 6829 show no select privilege. The user with DBA, SYSDBA or SYSADM can look all information.

Column Name	Description
DESTINATION	The database to which data is replicated.
FN	The file numbers of a transaction log record.
OFFSET	The Offset in the transaction log record.

SYSTRPPOS

The SYSTRPPOS table contains information used by asynchronous table replication. The user with RESOURCE and CONNECT privilege cannot look any information in this table. The system may make error 6829 show no select privilege. The user with DBA, SYSDBA or SYSADM can look all information.

Column Name	Description
RECORD_ID	Internal usage.
POSARRAY	Internal usage.

SYSUSER

The SYSUSER table contains information on the status of all users currently connected to a database. The user with RESOURCE and CONNECT authority only can read the information of himself. The user with DBA, SYSDBA or SYSADM authority can read all information. Before killing a connection, you should query the SYSUSER table for the ID of the connection you want to kill. If your login host name is not registered in the network, LOGIN_HOST is `anonymous`. If you want to monitor the update statistics status, you should query the SQL_CMD column of the SYSUSER table.

Column Name	Description
CONNECTION_ID	Connection ID.
USER_NAME	Login user name.

Column Name	Description
LOGIN_TIME	Login time.
LOGIN_IP_ADDR	Login IP address.
LOGIN_HOST	Login host name.
NUM_SCAN	Number of **SELECT** operations.
NUM_INSERT	Number of **INSERT** operations.
NUM_UPDATE	Number of **UPDATE** operations.
NUM_DELETE	Number of **DELETE** operations.
NUM_TRANX	Number of processed transactions.
NUM_JBYTE_PER_TRAN	Number of journal bytes per transaction.
FIRST_W_JNR_FN	First journal file number of one active transaction.
FIRST_W_JNR_BN	First journal block number of one active transaction.
NUM_BYTE_JNR_DATA	Total journal bytes used in the active transaction.
NUM_J_BLOCK_DURATN	The span between first journal block number used by the active transaction and one used most recently.
SQL_CMD	The most recently executed SQL command and the command status. The command status could be the following: **[PRE]** - The SQL command is preparing. **[EXEC]** - The SQL command is executing from a SQLExecute call. **[EXDIR]** - The SQL command is executing from a SQLExecDirect call. **[FETCH]** - The operation is in a fetch data phase. **[EXIT]** - The SQL command has finished a prepare, execute or fetch operation.
TIME_OF_SQL_CMD	The time when the most recently used SQL command was executed.

Column Name	Description
AFFINITY_MASK	CPU affinity.
PRIORITY_LEVEL	Priority of connections.
CPU_USAGE	CPU usage.

SYSUSERFUNC

The SYSUSERFUNC table contains information on user-defined and built-in functions. Because users need data in SYSUSERFUNC to execute UDF function, users that has at least the CONNECT authority can read all information.

Column Name	Description
MODE	Type of function: **1** — built-in function **0** — not built-in function
FILE_NAME	The name of the file that the built-in function is in.
FUNC_NAME	The name of the built-in function.
LANGUAGE_TYPE	Type of UDF: **C** — C UDF **LUA** — LUA UDF
RETURN_TYPE	Data type the built-in function returns.
NUM_OF_PARAMETER	The number of parameters in the function.
PARAMETER	Data type of each parameter. The number of parameters is given by the value of NUM_OF_PARAMETER.

SYSVIEWDATA

The SYSVIEWDATA table gives information on the table views in the database. The user with RESOURCE and CONNECT authority can read the view information which the user creates and is granted from other user. The user with DBA, SYSDBA or SYSADM authority can read all information.

COLUMN NAME	DESCRIPTION
VIEW_NAME	Table view name.
VIEW_OWNER	Table view owner.
STATUS	**0** — invalid view. **1** — valid view.
VIEW_DEFINITION	View definition.

SYSWAIT

The SYSWAIT table gives the status for locks that are currently waiting for another lock to release an object. The user with RESOURCE or CONNECT authority cannot read any information in this table. The system may make error 6829 showing no select privilege. The user with DBA, SYSDBA or SYSADM can read all information.

COLUMN NAME	DESCRIPTION
WAITING_CONNECTION	ID of connection which is waiting.
WAITED_CONNECTION	ID of connection which is being waited for.

C. System Limitations

DBMaker has certain limitations to the length of names in the database, the size of indices, tables, and memory buffers, the size of files and the number of concurrent transactions. These limitations and others are summarized in the following sections.

C.1 Naming Limitations

The ANSI/ISO standard for the SQL language specifies that database objects should be given unique names, and defines the database objects that require names. These names are used in SQL statements to identify which objects the statements act on.

The database objects that require names are:

- Tables
- Columns
- Users

DBMaker conforms to the ANSI/ISO standard with the exception of user names and passwords, which may only contain 1 to 128 characters. In the ANSI/ISO standard, SQL database object names have a maximum length of 128 characters, and may contain letters and numbers. They may not contain spaces or punctuation characters.

DBMaker also supports names for several database objects and extends the range of characters that can be used:

- Indexes
- Cursors

- Tablespaces
- Primary/Foreign keys

In DBMaker, database names may contain 1 to 128 alphanumeric characters or the underscore character, in any position including the first.

All other identifiers, with the exception of passwords, may include 1 to 128 alphanumeric characters, Chinese double-byte characters, spaces, underscores, and the symbols $ and #, in any position including the first. If spaces are used, the name must be enclosed in double quotes (" "), and any trailing spaces are ignored. Passwords also follow these rules, but are limited to a maximum length of 16 characters.

The size limitations on all of the named database objects supported by DBMaker are shown in the table below.

Item	**Minimum**	**Maximum**
Database Name	1	128
Tablespace Name	1	128
Table Name	1	128
Column Name	1	128
Index Name	1	128
Cursor Name	1	128
User Name	1	128
User Password	1[1]	16
Physical File Name with file path	1	256[2]
Logical File Name	1	128
SQL Statement	N/A	2097152

Table C-1: Minimum/maximum lengths of database object names (in characters)

[1] If a user does not set the password, then the length of the password is NULL.

[2] Including null terminator.

C.2 Storage Limitations

The following table shows the storage limitations placed on database objects by DBMaker. While the maximum value is shown as a specific number for most of these limitations, you should keep in mind that physical system limitations, such as system memory or disk space, and operating system limitations, such as system resources or other limitations, may impose a restriction before the specified value is reached. Unless otherwise noted, all limitations are the same for all platforms supported by DBMaker.

ITEM	MINIMUM	MAXIMUM
Size of a database	—	256PB
Number of files in a database	1	32767
Number of tablespaces in a database	1	32767
Number of files in a tablespace	1	32767
Number of tables in a tablespace	0	no limit[3]
Number of pages in a data file	3	2^{31}-1
Number of columns in a table	1	2000 (The max column of a table also depends on the page size)
Size of a tuple (row) in a table	0	3968 (4KB page size)[4] 8064 (8KB page size)[4] 16256 (16KB page size)[4] 32640 (32KB page size)[4]
Number of indices on a table	0	no limit[3]

[3] The number of tables and indexes is currently restricted only by operating system limitations.

ITEM	MINIMUM	MAXIMUM
Number of columns in an index	1	32
Length of the key in an index	0	4000
Column ID which can be used in an index	1	no limit[4]
Number of system temporary files	1	8
Number of journal files	1	8
Size of the journal file	100 pages	8GB
Number of projection columns	1	Same as max number of columns of a table
Number of GROUP BY columns	1	Same as max number of columns of a table
Number of ORDER BY columns	1	Same as max number of columns of a table
Number of ODBC binding parameters	0	Same as max number of columns of a table
Number of SQL sources	1	127
Number of bytes in a BLOB file	0	8TB
Number of pages in a data buffer	15	depends on OS
Number of pages in a journal buffer	16	depends on OS
Number of operators in predicate for a scan	1	depends on expression, worst case is 1022
Constant data in expression or predicate	0	64k
Host data in expression or predicate	0	64k

Table C-2: Minimum and maximum size of database objects (in bytes)

[4] All the columns in a table can be used in an index.

C.3 Processing Limitations

The following table shows the processing limitations placed on a database by DBMaker.

ITEM	MINIMUM	MAXIMUM
Number of concurrent transactions (connections) in a running database	0	4800
Length of CHAR or BINARY data items	0	3968 (4KB page size)[4] 8064 (8KB page size)[4] 16256 (16KB page size)[4] 32640 (32KB page size)[4]
Length of VARCHAR data items	0	3968 (4KB page size)[4] 8064 (8KB page size)[4] 16256 (16KB page size)[4] 32640 (32KB page size)[4]
Length of LONG VARCHAR or LONG VARBINARY data items	0	8TB
Number of items in the projection list of a selected command	1	Same as max number of columns of a table

Table C-3: Minimum and maximum size of processing