DBMaker

CASEMaker

CASEMaker Inc./Corporate Headquarters

1680 Civic Center Drive

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

www.casemaker.com

www.casemaker.com/support

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.

Trademarks
CASEMaker, the CASEMaker logo, and DBMaker are registered trademarks of CASEMaker Inc. Microsoft, MS-DOS, Windows, and Windows NT are registered trademarks of Microsoft Corp. UNIX is a registered trademark of The Open Group. ANSI is a registered trademark of American National Standards Institute, Inc.

Other product names mentioned herein may be trademarks of their respective holders and are mentioned only form information purposes. SQL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

Notices
The software described in this manual is covered by the license agreement supplied with the software.

Contact your dealer for warranty details. Your dealer makes no representations or warranties with respect to the merchantability or fitness of this computer product for any particular purpose. Your dealer is not responsible for any damage caused to this computer product by external forces including sudden shock, excess heat, cold, or humidity, nor for any loss or damage caused by incorrect voltage or incompatible hardware and/or software.

Information in this manual has been carefully checked for reliability; however, no responsibility is assumed for inaccuracies. This manual is subject to change without notice.

Contents

6 dmSQL Commands 6-1

1   Introduction

Welcome to the DBMaker SQL Command and Function Reference manual. 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). The unique open architecture and native ODBC interface adds the freedom to build custom applications using a wide variety of programming tools, or to query a database using ODBC-compliant applications.

DBMaker is easily scalable from personal single-user databases to distributed enterprise-wide databases. Regardless of the configuration of a database, the advanced security, integrity, and reliability features of DBMaker ensure the safety of critical data. Extensive cross-platform support permits leveraging of existing hardware and allows for expansion and upgrading when required.

DBMaker provides excellent multimedia-handling capabilities to store, search, retrieve, and manipulate all types of multimedia data. Binary Large Objects (BLOBs) ensure the integrity of multimedia data by taking full advantage of the advanced security and crash recovery mechanisms included in DBMaker. File Objects (FOs) manage multimedia data while maintaining the capability to edit individual files in source applications.

1.1   Additional Resources

DBMaker provides a complete set of DBMS manuals in addition to this one. Consult one of the books listed below for more information on a particular subject.

◆ For an introduction to DBMaker's capabilities and functions, refer to the *DBMaker Tutorial*.

◆ Please refer to the *Database Administrator's Guide* for more information on designing, administering, and maintaining a DBMaker database.

◆ For more information on DBMaker management, refer to the *JServer Manager User's Guide.*

◆ For more information on DBMaker configurations, refer to the *JConfiguration Tool Reference.*

◆ For more information on DBMaker functions, refer to the *JDBA Tool User's Guide.*

◆ For more information on the dmSQL interface tool, refer to the *dmSQL User's Guide*.

◆ For more information on DCI COBOL Interfaces, refer to the *DCI User's Guide*.

◆ For more information on the ESQL/C programming, refer to the *ESQL/C User's Guide*.

◆ For more information on the native ODBC API and JDBC API, refer to the *ODBC Programmer's Guide and JDBC Programmer's Guide.*

◆ For more information on error and warning messages, refer to the *Error and Message Reference*.

◆ For more information on the SQL stored procedure, refer to the *DBMaker SQL Stored Procedure User's Guide.*

1.2   Technical Support

CASEMaker provides thirty days of complimentary support via email and phone during the evaluation period. When software is registered an additional thirty days of support is included extending the total support period for software to sixty days. However, CASEMaker continues to provide email support for bugs reported after the complimentary support or registered support has expired (free of charge).

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 name and version number

◆   Registration number

◆   Registered customer name and address

◆   Supplier/distributor where 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 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. A word in italics should not be typed, but replaced by the actual name. In addition, italics can be used to 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.
Command Line	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 Table

2   SQL Basics

This manual is intended for anyone using the SQL language with DBMaker. This includes everyone from, users performing ad-hoc queries using the dmSQL command line utility, to programmers developing custom applications using ESQL/C and the DBMaker ODBC-compliant interface.

This manual also provides a complete reference to the Structured Query Language found in DBMaker, and provides the syntax for each SQL statement. Examples and illustrations are provided throughout the manual to assist with more clarity of understanding the contents.

2.1   Syntax Diagrams

Syntax diagrams demonstrate the syntax for all SQL commands. These diagrams provide assistance when constructing a statement on the command line. To use the syntax diagram, simply follow the line(s) and arrows from start to finish. Any elements of the command that cannot be navigated around are required. Any elements that can be navigated around are optional, but provide additional options and/or flexibility.

Any words that appear in italics are placeholders for the actual names used in a database. Substitute the actual names for these placeholders. In the diagram, replace the *table_name* placeholder with the name of a table in the database. For example, in the tutorial database, you could replace the *table_name* placeholder with Customers to execute this command on the Customers table.

Sometimes it is possible to have a list of items in a command, which are shown in the syntax diagram as a circular path. The column name field can include a list of column names, separated by commas, as indicated by the circular path following the arrows.
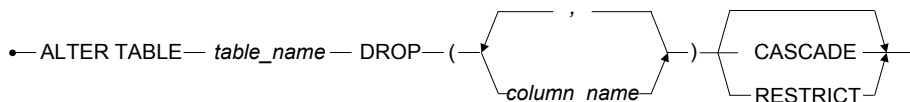
Figure 2-1: A sample syntax diagram

2.2 Data Types

When defining a column in a table, choose a data type for the field. Understand how to use each field in order to make the right choice of data type. Choosing the wrong data type can waste space in the database, or make the application program take extra steps to convert the data into a usable form.

DBMaker supports the following data types:

BIGINT, BINARY(size), BIGSERIAL, CHAR(size), NCHAR(size), DATE, DECIMAL(NUMERIC), DOUBLE, FILE, FLOAT, INTEGER, JSONCOLS, LONG VARBINARY(BLOB), LONG VARCHAR(CLOB), REAL, OID, SERIAL(start), SMALLINT, TIME, TIMESTAMP, VARCHAR(size), NVARCHAR(size) and Media types.

BIGINT

BIGINT data type is an exact signed numeric data type with a precision of nineteen and a scale of zero. The BIGINT data type uses 8 bytes of storage with a maximum value of 9,223,372,036,854,775,807 and a minimum value of −9,223,372,036,854,775,808.

Using BIGINT or INTEGER to move a value larger than the maximum allowed by those data types results in a conversion error and and the data is not moved.

➲ **Example 1**
```
37654
```

➲ **Example 2**
```
857823
```

BIGSERIAL(start)

DBMaker uses the BIGSERIAL data type when for allocating consecutive integers to uniquely identify each table contained in a database. DBMaker manages these integers internally. The value of each integer is automatically increased by one each time it is used.

When defining a BIGSERIAL column, providing an integer value for the optional START parameter specifies the first value in a number sequence. A default value of 1 is used when the START parameter is omitted. Each table may contain one column with the BIGSERIAL data type.

An integer is used to generate BIGSERIAL numbers. The integer is an exact signed numeric data type occupying 8 bytes of storage with a precision of 19 and a scale of 0. The BIGSERIAL data type has a maximum value of 9,223,372,036,854,775,806 and a minimum value of –9,223,372,036,854,775,808.

A sequential number can be inserted into a BIGSERIAL column by place a NULL or empty value in the BIGSERIAL column when inserting a new row. DBMaker inserts the sequential number for that table into the BIGSERIAL column of the new record and increases the internal value by one.

When inserting a new column, if an integer is supplied for BIGSERIAL, instead of a NULL or empty value, DBMaker uses the integer instead of the next sequential number. Additionally, the internal value is not incremented by 1. If the supplied integer value is greater than the last sequential number generated, DBMaker resets the sequence of generated sequential numbers to start with the supplied integer.

⮕ **Example 1**
```
10000, 10001, 10002, 10003, 10004, 10005, 10006, 10007
```

⮕ **Example 2**
```
10000, 10001, 5000, 10002, 10003, 11000, 11001, 11002
```

BINARY (size)

The BINARY data type is a fixed-length data type that can contain any binary value. BINARY columns maximum length can be specified as 4 KB, 8 KB, 16 KB or 32 KB, User can enter a value for the size parameter when creating a BINARY column. Any data entered in a BINARY column shorter than the column length is padded with a zero-value byte. By default, the minimum length of BINARY columns is 1 byte and the maximum length is 8056 bytes.

Enter character data by enclosing the data in single quotes (' '), the same as when entering CHAR data. However, in BINARY columns the data is stored as

hexadecimal values representing the ASCII code of the characters, not as the actual characters entered.

Alternatively, enter hexadecimal values directly by enclosing them in single quotes and appending the 'x' character (' 'x) to indicate the string contains a hexadecimal value. It requires two digits to represent all possible values for each byte in hexadecimal; use an even number of digits when entering values.

➲ **Example 1**
```
'AaBbCcDdEe'x
```

➲ **Example 2**
```
'41614262436344644565'x
```

CHAR (size)

The CHAR data type is a fixed-length data type that can contain any character from the keyboard. CHAR columns maximum length can be specified as 4 KB, 8 KB, 16 KB or 32 KB, User can enter a value for the size parameter when creating a CHAR column. By default, the minimum length of CHAR columns is 1 byte and the maximum length is 8056 bytes.

Any CHAR data in a column that is shorter than the column length is padded with spaces. When entering CHAR data, enclose it in single quotes (' '). Double-byte characters occupy two bytes. If using double-byte characters, account for this when specifying the length of the column.

➲ **Example 1**
```
'This is a CHAR string.'
```

➲ **Example 2**
```
'This is another CHAR string.'
```

DATE

There are two types of DATE data; DATE literal and DATE constant. Date literal represents the present date. DATE constant is a set point in time. The DATE data

type is fixed-length containing the calendar date (year, month and day). The DATE data type uses 4 bytes of storage. Valid values for the year are from 0001 to 9999.

The DATE data type has multiple input/output formats. If the values in the database do not appear correctly, or you are not able to enter dates you think are valid, check the date input/output formats to ensure that they are correct.

➲ **Example 1a**
```
'0001/01/01'
```

➲ **Example 1b**
```
'0001/01/01'd
```

➲ **Example 1c**
```
DATE '0001/01/01'
```

➲ **Example 2a**
```
'1999/12/31'
```

➲ **Example 2b**
```
'1999/12/31'd
```

➲ **Example 2c**
```
DATE '1999/12/31'
```

DECIMAL (NUMERIC)

The DECIMAL data type is an exact signed numeric value with a variable precision and scale. Precision refers to the total number of digits in the mantissa, both to the left and to the right of the decimal point. The default value for precision is 17 with a maximum value of 38. Scale refers to the number of digits to the right of the decimal point. The default value for scale is 6.

The amount of storage used by a DECIMAL column is based on the actual value entered, not on the default precision and scale values or the precision and scale values entered when defining the column.

To calculate the amount of storage, use the following formula:

$$\# \text{ of bytes} = \frac{p+1}{2} + 2$$

For example, the number 9283.83 would be stored as 6 bytes.

The actual calculation used is:

$$\# \text{ of bytes} = \frac{p+1}{2} + 2$$
$$= \frac{6+1}{2} + 2$$
$$= 5.5$$

If you attempt to move a value larger than the allowed maximum from a data type such as FLOAT or DOUBLE, DBMaker displays a conversion error and does not move the data. The DECIMAL data type may be abbreviated as DEC.

➲ **Example 1**
```
3452.8373645
```

➲ **Example 2**
```
736.383732652
```

DOUBLE

The DOUBLE data type is an approximate signed numeric data type with a mantissa of precision 15. Precision refers to the total number of digits in the mantissa, both to the left and to the right of the decimal point. The DOUBLE data type uses 8 bytes of storage and has a valid input range from 1.0E308 to –1.0E308.

The smallest valid input values are **1.0E-308 and –1.0E-308.**

➲ **Example 1**
```
2.89837457884451E285
```

➲ **Example 2**
```
–1.93873634847372E-174
```

FILE

The FILE data type is a structured data type that occupies 48 bytes of storage. This data type is similar to the CLOB and BLOB data types and stores the contents of any existing file as an external file that DBMaker can reference the same as any other data. DBMaker stores the data externally as a file instead of internally as an object. This allows third-party tools to access and manipulate the data in its native format, without having to re-import the data to register any changes in the database. A file object has a maximum path length of 255 characters.

The FILE column stores a reference to a record in the system catalog tables. The database uses system catalog information to find the file object. When you display a FILE column, you do not actually see what is stored in the FILE column itself. Instead, DBMaker shows one of three views of information stored in the system catalog or the file itself the filename, the file size, or the file contents.

The FILE data type can store data in two ways, as a system file object or as a user file object. A system file object copies an existing file to the file object directory of the database and gives it a unique name. The database manages this file, and deletes it when there are no references to it in the database. A user file object creates a link to an existing file, while leaving the file in the original location with the original name. Since, the user created this file; it will not be deleted when there are no references made to it in the database. DBMaker must have the read permission on a file before you can insert it into the database as a user file object.

When multiple records reference the same file, DBMaker will store only a single copy of the file and share it between records to save disk space. However, from the user's point of view, there is always a dedicated file for each record. DBMaker transparently generates a new file when updating a shared file. Other records sharing that file are not changed, and other users still see the original file. This prevents changes made to a record in one file from influencing other records.

FLOAT

The FLOAT data type is an approximate signed numeric data type having a mantissa with a precision of 15. Precision refers to the total number of digits to the left and to

the right of the decimal point. The default FLOAT data type uses 8 bytes of storage and has a valid input range from 1.0E308 to –1.0E308. The default FLOAT type can be specified as REAL or DOUBLE with the keyword **DB_FltDb**.

The smallest valid input values are **1.0E-308** and **–1.0E-308.**

➲ **Example 1**
```
2.89837457884451E285
```

➲ **Example 2**
```
–1.93873634847372E–174
```

INTEGER

The INTEGER data type is an exact signed numeric data type with a precision of 10 and a scale of 0. The INTEGER data type uses 4 bytes of storage and has a maximum value of 2,147,483,647 and a minimum value of -2,147,483,648.

If you attempt to move a value larger than the allowed maximum from a data type such as DOUBLE, DBMaker displays a conversion error and does not move the data. The INTEGER data type may be abbreviated as INT.

➲ **Example 1**
```
393848
```

➲ **Example 2**
```
–298376
```

JSONCOLS

JSONCOLS Type is a column set of 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. For details of a dynamic column, please refer to chapter *Using Dynamic Columns* in D*atabase Administrator's Guide*. For details of a JSONCOLS column, please refer to chapter *Using JSONCOLS Type* in *Database Administrator's Guide*. Dynamic columns of a table are stored as JSONCOLS type which is derived from LONG VARBINARY.

⊃ **Example 1**

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
```

⮕ **Example 2**

The following operations base on table **student**. For details of table **student**, please refer to Example 1.

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;
```

LONG VARBINARY (BLOB)

The BLOB data type is a variable-length data type that can contain any binary value. The maximum length of BLOB columns is 8 TB. Unlike the BINARY data type, which uses zero-value bytes for padding, only the bytes entered are stored in the database.

You can enter character data by enclosing the data in single quotes (' '), the same as when entering CHAR data. However, in BLOB columns the data is stored as hexadecimal values representing the ASCII code of the characters, not as the actual characters entered.

Alternately, enter hexadecimal values directly by enclosing the data in single quotes and appending the 'x' character (' 'x) to indicate a string containing a hexadecimal value. Two digits represent all possible values for each byte in hexadecimal; use an even number of digits when entering values.

➲ **Example 1**

```
'AaBbCcDdEe'x
```

➲ **Example 2**

```
'41614262436344644565'x
```

LONG VARCHAR (CLOB)

The variable-length CLOB data type can contain any character that can be entered from the keyboard. The maximum length of CLOB columns is 8 TB .

Unlike the CHAR data type, which uses spaces for padding, only the characters entered are stored in the database. When entering data in a CLOB column, enclose it

in single quotes (' '). Double-byte characters occupy two bytes each, account for this when specifying the length of the column.

➲ **Example 1**
```
'This is a varchar string.'
```

➲ **Example 2**
```
'This is another varchar string.'
```

NCHAR (size)

The NCHAR data type is a fixed-length data type that can contain any Unicode character. Each Unicode character occupies two bytes of storage in UTF16 Little-Endian (LE) encoding. The (size) parameter determines the number of 2 byte characters in the column. The (size) parameter must be entered when creating an NCHAR column, and may range from 1 to 4028 by default.

If NCHAR data is entered into a column that is shorter than the column length, the data will be padded with spaces. When entering NCHAR data, enclose the Unicode character with single quotes and prefix the quotes with 'N'.

➲ **Example 1**

The following demonstrates the syntax of a Unicode data entry:
```
N'Unicode Data'
```

If NCHAR data is input in hexadecimal format, enclose the hexadecimal string with quotes and append a 'u' character.

➲ **Example 2**

The following demonstrates the syntax of a three-character hexadecimal Unicode data entry:
```
'610a620b63f1'u
```

When a character string is input to a Unicode column but is not prefixed by 'N', then it will automatically be converted from local code to Unicode. If Unicode characters are entered into a regular CHAR type column, then the Unicode character will be

converted to the local code defined by the dmconfig.ini parameter **DB_LCode**. Characters that are not defined in the local code are represented by ☐.

Synonyms for the NCHAR data type include NATIONAL CHAR(size), and NATIONAL CHARACTER(size).

NVARCHAR (size)

The NVARCHAR data type is a variable-length data type that can contain any Unicode character. Each Unicode character occupies two bytes of storage in UTF16 Little-Endian (LE) encoding. The (size) parameter determines the number of 2 byte characters in the column. The (size) parameter must be entered when creating an NVARCHAR column, and may range from 1 to 4028 by default.

If NVARCHAR data is entered into a column that is shorter than the column length, the data is not padded with spaces. When entering NVARCHAR data, enclose the Unicode character with single quotes and prefix the quotes with 'N'.

➲ **Example 1**

The following demonstrates the syntax of a Unicode data entry:
```
N'Unicode Data'
```

If NVARCHAR data is input in hexadecimal format, enclose the hexadecimal string with quotes and append a 'u' character.

➲ **Example 2**

The following demonstrates the syntax of a three-character hexadecimal Unicode data entry:
```
'610a620b63f1'u
```

When a character string is input to a Unicode column but is not prefixed by 'N', then it will automatically be converted from local code to Unicode. If Unicode characters are entered into a regular VARCHAR type column, then the Unicode character will be converted to the local code defined by the dmconfig.ini parameter **DB_LCode**. Characters that are not defined in the local code are represented by ☐.

Synonyms for the NVARCHAR data type include NATIONAL CHAR VARYING(size), NCHAR VARYING(size), NATIONAL VARCHAR(size), and NATIONAL CHARACTER VARYING(size).

OID

The OID (object identifier) data type is a special data type that provides a unique ID for each object, record or BLOB, stored in a database. A structured data type has a precision of 10 and a scale of 0, and occupies 16 bytes of storage. DBMaker automatically generates and inserts an OID with each record. The OID is internally managed and maintained by DBMaker and cannot be used directly.

The value generated for an OID is related to the storage location of objects in the database. This means that two OIDs generated consecutively may not necessarily be sequential.

The OID values act as a hidden pseudo-column in tables, and will not appear in queries such as SELECT * FROM CUSTOMERS. Explicitly select the OID column by using 'OID' as a column name in a query.

Although it is possible to use an OID in a query to select data from a table and then use the OIDs to update the table data, this is not common practice when using the SQL language. OIDs are usually used in the internal programming interface, and not directly in the interactive dmSQL environments.

REAL

The REAL data type is an approximate signed numeric data type having a mantissa with a precision of 7. Precision refers to the total number of digits to the left and to the right of the decimal point. The REAL data type uses 4 bytes of storage and has a valid input range from 3.402823466E38 to –3.402823466E38. The smallest valid input values are 1.175494351E-38 and –1.175494351E-38. A move involving a value larger than the allowed maximum, from a data type such as DOUBLE, fails and DBMaker displays a conversion error.

➲ **Example 1**
```
3.583837E34
```

➲ **Example 2**

```
-1.873653E-21
```

SERIAL (start)

The SERIAL data type is a special data type that provides a sequence of consecutive values. DBMaker allocates an integer number for each table contained in a database and uses those numbers to generate a unique sequence for the corresponding table. DBMaker manages and maintains these integer numbers internally. The value of each integer value is automatically increased by one each time it is used.

Providing an integer value for the optional START parameter when defining a SERIAL column can specify the first value in a number sequence, or the START parameter omitted to use the default value of 1. Each table in a database can have only one column with the SERIAL data type.

The internal value used to generate a SERIAL number is actually an integer value; the SERIAL data type shares all of the properties of the INTEGER data type. It is an exact signed numeric data type with a precision of 10 and a scale of 0, which occupies 4 bytes of storage. The SERIAL data type also has the same range of values as the INTEGER data type, with a maximum value of 2,147,483,646 and a minimum value of –2,147,483,648.

Place a NULL, or empty value in the SERIAL column when inserting a new row to insert a sequential number into a SERIAL column. DBMaker will insert the sequential number for that table into the SERIAL column of the new record, and automatically increase the internal value by one.

If inserting a new column, and supplying an integer value for the SERIAL instead of a NULL or empty value, DBMaker will use the supplied integer value instead of the next sequential number; the internal value will not be incremented by 1. If the supplied integer value is greater than the last sequential number generated, DBMaker will reset the sequence of generated sequential numbers to start with the supplied integer value.

➲ **Example 1**

```
100, 101, 102, 103, 104, 105, 106, 107
```

➲ **Example 2**
```
100, 101, 50, 102, 103, 110, 111, 112
```

SMALLINT

The SMALLINT data type is an exact signed numeric data type with a precision of five and a scale of zero. The SMALLINT data type uses two bytes of storage and has a maximum value of 32,767 and a minimum value of -32,768.

If attempting to move a value larger than the permitted maximum value from a data type such as INTEGER or DOUBLE, DBMaker displays a conversion error and does not move the data.

➲ **Example 1**
```
4769
```

➲ **Example 2**
```
8376
```

TIME

There are two types of TIME data, TIME literal, and TIME constant. A TIME literal displays the present time, which is an ever-changing value. A TIME constant is a fixed moment in time. Both TIME data type settings are fixed-lengths, and use 4 bytes of storage. All time values are entered in twenty-four hour format by default unless the optional 'AM' or 'PM' values are specified.

Both TIME data types have multiple input/output formats. If the values in the database do not appear correctly or you are unable to enter perceived valid times then, check the time input/output formats for validity.

➲ **Example 1a**
```
'22:04:05'
```

➲ **Example 1b**
```
'22:04:05't
```

➲ **Example 1c**
```
TIME '22:04:05'
```

➲ **Example 2a**
```
'10:04:05 PM'
```

➲ **Example 2b**
```
10:04:05 PM't
```

➲ **Example 2c**
```
TIME 10:04:05 PM'
```

TIMESTAMP

There are two types of TIMESTAMP, TIMESTAMP literal, and TIMESTAMP constant. A TIMESTAMP literal displays the present time, which is an ever-changing value. A TIMESTAMP constant is a fixed moment in time.

Both TIMESTAMP data type settings are a fixed-length data type that contains calendar data and the time-of-day. Both TIMESTAMP data type settings use 11 bytes of storage, has a precision of 17, and a scale of 10. Valid years range from 0001 to 9999. All time values are entered in twenty-four hour format by default unless the optional 'AM' or 'PM' values are specified.

Both TIMESTAMP data type settings use the input and output formats for the TIME and DATE data types to display values and determine if input values are valid. If the values in the database do not appear correctly or you are unable to enter perceived valid times then, verify the time input and output formats.

➲ **Example 1a**
```
'1997/01/01 10:02:03'
```

➲ **Example 1b**
```
'1997/01/01 22:02:03'ts
```

➲ **Example 1c**
```
TIMESTAMP '1997/01/01 10:02:03'
```

➲ **Example 2a**
```
'01.01.1997 22:02:03'
```

➲ **Example 2b**
```
'01.01.1997 22:02:03'ts
```

➲ **Example 2c**
```
TIMESTAMP '01.01.1997 22:02:03'
```

VARCHAR (size)

The VARCHAR data type is a variable-length data type that can contain any character that can be entered from the keyboard. VARCHAR maximum columns length can be specified as 4 KB, 8 KB, 16 KB or 32 KB, User can enter a value for the size parameter when creating a VARCHAR column. By default, the minimum length of VARCHAR columns is 1 byte and the maximum length is 8056 bytes.

Only the VARCHAR characters entered are stored in the database. When entering data in a column, use single quotes (' '). If using double-byte characters, account for two bytes for each character when specifying the length of a column.

➲ **Example 1**
```
' This is a VARCHAR string.'
```

➲ **Example 2**
```
' This is another VARCHAR string.'
```

Media Types

Large object columns may also be specified as media types to aid in media process functions such as full text search for Microsoft™ Word™ documents. The following media types are available: MsWordType, HtmlType, XmlType, MsPPTType, MsExcelType, PDFType, MsWordFileType, HtmlFileType, XmlFileType, MsPPTFileType, MsExcelFileType, and PDFFileType.

Media types are domains of existing data types; MsWordType, MsPPTType, MsExcelType, PDFType, HtmlType, and XmlType are derived from LONG VARBINARY, and MsWordFileType, HtmlFileType, XmlFileType, MsPPTFileType, MsExcelFileType, and PDFFileType are derived from FILE type columns. This is important to consider if you choose to use the ALTER TABLE function to change a column from one data type to another. The characteristics of each of the media types are similar to the characteristics of the data type from which it is derived.

The features of XMLTYPE include:

- Well-formed XML checking: inserted/updated xml content must be well-formed

- XML validation: optionally specify a validation udf when creating an xmltype column and DBMaker will validate the xml content with it

- XML data is stored in the original format

- Query with XPath search: optionally specify an xpath and use extract functions to query/locate nodes in an XML data

- Update XML content specified by XPath

- Build index on XPath extract: speed up xpath querieswith indexes on frequent query xpath expression

- Altering an xmltype column or other data types to the xmltype is not allowed

➲ Example

```
dmSQL> CREATE TABLE minutes (id INT, meeting_date DATE, doc MSWORDFILETYPE);
dmSQL> INSERT INTO minutes VALUES (1, 3/3/2003, 'c:\meeting\20030303.doc');
```

2.3   Data Conversion

Data types will be converted in the following scenarios:

- When data from one object is moved to, compared with, or combined with data from another object, the data may have to be converted from the data type of one object to the data type of the other.

- When data from a SQL result column, return code, or output parameter is moved into a program variable, the data must be converted from the DBMaker system data type to the data type of the variable.

- When an expression contains data of different datatypes, there is a need for data conversion to make data compatible.

DBMaker supports both implicit and explicit conversion of data from one datatype to another.

It is recommended that users specify explicit conversions, rather than rely on implicit or automatic conversions. The reasons are as follows:

- SQL statements are easier to understand when you use explicit datatype conversion functions.

- Implicit datatype conversion can have a negative impact on performance, especially if the datatype of a column value is converted to that of a constant rather than the other way around.

- Implicit conversion depends on the context in which it occurs and may not work the same way in every case.

Explicit Data Conversion

Users can explicitly specify datatype conversions using the following SQL conversion functions: CAST, DATETOSTR, TIMETOSTR, TIMESTAMPTOSTR and TO_DATE.

CAST allows the output data to be converted to another data type. For details please refer to Chapter 3.85, *SELECT*.

The DATETOSTR function is used to convert a value in DATE type into the character string in specified format. For details please refer to Chapter 4.2.3, *DATETOSTR*.

The TIMETOSTR function is used to convert a value in TIME type into the character string with specified format. For details please refer to Chapter 4.2.4, *TIMETOSTR*.

The TIMESTAMPTOSTR function is used to convert a value in TIMESTAMP type into the character string in specified format. For details please refer to Chapter 4.2.5, *TIMESTAMPTOSTR*.

The TO_DATE function converts a selected character string to a value in DATE type. For details please refer to Chapter 4.2.6, *TO_DATE*.

Implicit Data Conversion

DBMaker automatically converts a value from one datatype to another when such a conversion makes sense. It mainly contains conversion between numeric data and character data. A numeric data and a character data both include data of multiple types. A numeric data's type can be integer (int, serial), smallint, bigint, bigserial, float, double, and decimal. A character data's type can be char, varchar, nchar and nvarchar. Before using implicit data conversion, users should open this function by using "set itcmd on" or set value of **DB_ItcMd** to 1.

The table 2-1 illustrates all valid conversions, and the direction of the conversion is from row X to column Y.

Xy	int (serial)	small-int	bigint (bigserial)	decimal	double	float	(var) char	n(var) char
int(serial)	Y	Y	Y	Y	Y	Y	Y	Y
smallint	Y	Y	Y	Y	Y	Y	Y	Y
bigint(bigserial)	Y	Y	Y	Y	Y	Y	Y	Y
decimal	Y	Y	Y	Y	Y	Y	Y	Y
double	Y	Y	Y	Y	Y	Y	Y	Y
float	Y	Y	Y	Y	Y	Y	Y	Y

(var)char	Y	Y	Y	Y	Y	Y	Y	Y
n(var)char	Y	Y	Y	Y	Y	Y	Y	Y

Table 2-1 Implicit Conversion Table

The following rules govern the direction in which DBMaker makes implicit datatype conversion:

- During INSERT operations, DBMaker converts the value to the datatype of the affected column.

- During arithmetic operations (arithmetic operators: +, -, *, /), DBMaker converts the character data to a numeric data.

  a) When only one side of an operator is character data and the other side is numeric data, DBMaker converts this character data to a numeric data of the same type with the numeric data on the other side.

  b) When the both side of an operator are character data, if meanwhile the character data both are constant character data, DBMaker converts the character data to a value of proper type, for example, in expression '123'+'123.456'+'1.23e45', '123', '123.456' and '1.23e45' will be converted to 123 (int type), 123.456 (decimal type), and 1.23e45 (double type) separately, otherwise, DBMaker converts the character data to a DOUBLE value.

- During comparison operations (comparison operators: >, >=, =, <=, <, !=, <>, IN, IS NULL), DBMaker converts the value on the right side of the operator to a value of the same type with the value on the left side.

- During concatenation operations (concatenation operators: ||, CONCAT), DBMaker converts the numeric data to a character data.

Please note that DBMaker also can implicitly convert the argument of a UDF and a default value.

⮌ Example 1

In the following statement, DBMaker implicitly converts a char(varchar) value to a int value.

```
dmSQL> set itcmd on;
```

```
dmSQL> create table t1 (c1 int);
dmSQL> insert into t1 values ('123');
dmSQL> select * from t1 where c1 = '123';
dmSQL> update t1 set c1='456'+111;
dmSQL> delete from t1 where c1 = '678'-111;
```

 ⮕ **Example 2**

In the following statement, DBMaker implicitly converts a decimal value to a
nchar(nvarchar) value.

```
dmSQL> create table t2 (c1 nchar(20), c2 nvarchar(20));
dmSQL> insert into t2 values (12345.6789, 222.222);
dmSQL> select * from t2 where c1 = 12345.6789 and c2 = 222.222;
dmSQL> update t2 set c1 = -6789.12345;
```

 ⮕ **Example 3**

In the following statement, DBMaker implicitly converts the argument of a UDF.

```
dmSQL> create table t1 (c1 int, c2 char(10), c3 nchar(10));
dmSQL> insert into t1 values(abs('-10'), -abs(-10), abs(10));
dmSQL> select * from t1;
    C1          C2                  C3
==========  ==========  ====================================
        10  -10         31003000000000000000000000000000000000
1 rows selected

dmSQL> select * from t1 where abs(c1) = abs(c2);
    C1          C2                  C3
==========  ==========  ====================================
        10  -10         31003000000000000000000000000000000000
1 rows selected

dmSQL> select * from t1 where abs(c1) = abs(c3);
    C1          C2                  C3
==========  ==========  ====================================
        10  -10         31003000000000000000000000000000000000
1 rows selected

dmSQL> select abs(c1), abs(c2), abs(c3) from t1;
    ABS(C1)                 ABS(C2)                 ABS(C3)
=====================  =====================  ====================
```

```
1.00000000000000e+001   1.00000000000000e+001   1.00000000000000e+001
1 rows selected


dmSQL> select concat(c1,c2), concat(c1,-123), concat(1.234e8, c1) from t1;
    CONCAT(C1,C2)          CONCAT(C1,-123)      CONCAT(1.234E8, C1)
===================== ==================== ======================
10-10                 10-123               12340000010
1 rows selected
```

➲ **Example 4**

In the following statement, DBMaker implicitly converts the default value.

```
dmSQL> create table t1 (c1 int default '123456');
dmSQL> insert into t1 values(default);
1 rows inserted
dmSQL> select * from t1;
     C1
===========
     123456
1 rows selected


dmSQL> create table t1 (c1 char(20) default 123456);
dmSQL> insert into t1 values(default);
1 rows inserted
dmSQL> select * from t1;
     C1
===========
      123456
1 rows selected
```

2.4 RESERVED WORDS

The following list of keywords should not be used as identifiers. DBMaker returns the ERR_RESERVED_WORD error message and does not perform the desired command when the following reserved words are used as keywords.

ABSOLUTE | ACTION | ADD | ADMIN | AFTER | AGGREGATE | ALIAS | ALLOCATE | ALTER | AND | ANY | ARE | ARRAY | AS | ASC | ASSERTION | ASENSITIVE | AT | AUTHORIZATION | BEFORE | BEGIN | BIGINT | BIGSERIAL | BINARY | BIT | BLOB | BOOLEAN | BOTH | BREADTH | BREAK | BY | CALL | CASCADE | CASCADED | CASE | CAST | CATALOG |CHAR | CHECK | CLASS | CLOB | CLOSE | COLLATE | COLLATION | COLUMN | COMMIT | COMPLETION| CONDITION | CONNECT | CONT | CONNECTION | CONSTRAINT | CONSTRAINTS | CONSTRUCTOR | CONTINUE | CORRESPONDING | CREATE | CROSS | CUBE | CURRENT | CURRENT_DATE | CURRENT_PATH | CURRENT_ROLE | CURRENT_TIME | CURRENT_TIMESTAMP | CURRENT_USER | CURSOR | CYCLE| DATE | DAY | DEALLOCATE | DEC | DECIMAL | DECLARE | DEFAULT | DEFERRABLE | DEFERRED | DELETE | DEPTH | DEREF | DESC | DESCRIBE | DESCRIPTOR | DESTROY| DESTRUCTOR | DETERMINISTIC | DICTIONARY | DIAGNOSTICS | DISCONNECT | DISTINCT |DO |DOMAIN | DOUBLE | DROP | DYNAMIC | EACH | ELSE | ELSEIF | END | END-EXEC | EQUALS | ESCAPE | EVERY | EXCEPT| EXCEPTION | EXEC | EXECUTE | EXIT | EXTERNAL | FALSE | FETCH | FIRST | FLOAT | FOR | FOREIGN | FOUND | FROM| FREE | FULL | FUNCTION | GENERAL | GET | GLOBAL | GO | GOTO | GRANT | GROUP | GROUPING | HANDLER | HAVING | HOLD | HOST | IDENTITY | IF | IGNORE | IMMEDIATE | IN | INDICATOR | INITIALIZE | INITIALLY | INNER | INOUT | INPUT | INSENSITIVE | INT | INTEGER | INTERSECT | INTO | IS | ISOLATION | ITERATE | JOIN | KEY | LANGUAGE | LANGUAGE SQL | LARGE | LAST | LATERAL | LEADING | LEAVE | LESS | LEVEL | LIKE | LIMIT | LOCAL | LOCALTIME | LOCALTIMESTAMP | LOCATOR | LOOP | MAP | MATCH | MODIFIES | MODIFY | MODULE | NAMES | NATIONAL | NATURAL | NCHAR | NCLOB | NEXT | NO | NONE | NOT | NULL | NUMERIC |

NVARCHAR | OBJECT | OF | OFF | ON | ONLY | OPEN | OPERATION |
OPTION | OR | ORDINALITY | OUT | OUTER | OUTPUT | PAD | PARTIAL |
PATH | POSTFIX | PREFIX | PREORDER | PREPARE | PRESERVE | PRIMARY |
PRIOR | PRIVILEGES | PROCEDURE | READ | READS | REAL | RECURSIVE |
REFERENCES | REFERENCING | RELATIVE | REPEAT | RESTRICT |
RESULT | RETURN | RETURNS | REVOKE | ROLE | ROLLBACK | ROLLUP |
ROUTINE | ROW | ROWS|SAVEPOINT | SCHEMA | SCROLL | SCOPE |
SEARCH | SECTION | SELECT| SENSITIVE | SEQUENCE | SERIAL | SESSION
| SESSION_USER | SET | SETS | SHORT | SIZE | SMALLINT | SOME |
SPECIFIC | SPECIFICTYPE | SQL | SQLCODE | SQLEXCEPTION |
SQLSTATE | SQLWARNING | START | STATIC | STATISTICS | STOP |
STRUCTURE | SYSTEM_USER | TABLE | TEMPORARY | TERMINATE |
THAN | THEN | TIME | TIMESTAMP | TIMEZONE_HOUR|
TIMEZONE_MINUTE | TO | TRACE | TRAILING | TRANSACTION |
TRANSLATION | TREAT | TRIGGER | TRUE | UNDER | UNION |
UNKNOWN | UNTIL | UNNEST | UPDATE | USAGE | USING | VALUE |
VALUES | VARBINARY | VARBPTR | VARCHAR | VARCPTR | VARIABLE |
VARYING | VIEW | WHEN | WHENEVER | WHERE | WHILE | WITH |
WITHOUT | WORK | WRITE | ZONE

3   SQL Commands

DBMaker provides a comprehensive SQL query language. SQL (Structured Query Language) is a query language standardized by ANSI. The current standard is ANSI-99 SQL. This chapter contains the DBMaker version of all supported ANSI-99 commands.

3.1 ABORT BACKUP

The ABORT BACKUP command cancels an online backup. Cancel a backup if errors occur during the backup operation or to perform the backup at another time. Only users with SYSADM, SYSDBA or DBA security privileges can execute the ABORT BACKUP command.

Backup mode indicates whether DBMaker will perform online incremental backups, and what data to backup. There are three backup modes NONBACKUP, BACKUP-DATA, and BACKUP-DATA-AND-BLOB. Set the backup mode in three ways using the **DB_BMode** keyword in the **dmconfig.ini** configuration file, SQL SET command at the dmSQL command prompt, or Server Manager Utility.

NONBACKUP mode provides no protection for data inserted or updated after the last full backup. A database can use the Journal to fully recover from a program failure, but a disk failure may result in loss of data. Immediately reuse Journal blocks not in use by an active transaction, after a checkpoint. Once overwritten, the database can only restore to the point in time of the last full backup.

BACKUP-DATA mode provides protection for data; excluding BLOB data inserted or updated since the last full backup. In this mode, DBMaker can perform an online incremental backup; only non-BLOB data will be stored in the backup files. A database can use the Journal to fully recover from a program failure and can partially recover from a disk failure. 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 provides protection for all data including BLOB data inserted or updated since the last full backup. In this mode, DBMaker can perform an online incremental backup; all data will be stored in the backup files. A database can use the Journal to fully recover from a program failure and fully recover from a disk failure. Use the last backup 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.

Issuing the ABORT BACKUP command does not change the backup mode of the database. The database will remain in the same backup mode it was in before the backup started.

———————————————————— ABORT BACKUP ————————————————————

Figure 3-1 ABORT BACKUP syntax

➲ **Example**

The following example illustrates aborting a backup operation.

```
BEGIN BACKUP
ABORT BACKUP
```

3.2 ABORT CONNECTION

The ABORT CONNECTION command aborts an active connection but not disconnect the connection from the database. Only users with DBA, SYSDBA or SYSADM security privileges can execute the ABORT CONNECTION command.

The ABORT CONNECTION function is similar to the KILL CONNECTION function, the only difference is that the connection cannot be disconnected, and will rollback to latest commit state. The main purpose is interrupting query if it costs a great deal of time, but not killing the connection.Executing this command will not free all lock resources held by this user.

Please note that the abort connection will be rolled back. If the connection is being carried out, it will be interrupted and rolled back. If the connection is not being carried out, it will work until the connection executes next SQL statement and then rolled back.

connection_id .... Number of the connection to abort

ABORT CONNECTION ——— *connection_id*

Figure 3-2 ABORT CONNECTION syntax

➲ **Example**

The following example aborts the connection of which ID is **12345**.

```
ABORT CONNECTION 12345
```

3.3   ADD TO GROUP

The ADD TO GROUP command adds a user to an existing group. The user gains all current and future object privileges granted to the group. Only users with SYSADM, SYSDBA or DBA security privileges can execute the ADD TO GROUP command.

Groups simplify management of object privileges in databases with a large number of users. Use a group to collect several users and even other groups. Object privileges granted to the group are automatically granted to members in the group.

Members added to a group also maintain previously assigned privileges. Members removed from a group lose object privileges to that group, but retain any other privileges granted to them directly or to another group.

Specify a group name in place of a user name, as long as the group does not already contain a reference to that group. User and group names have a maximum length of 128 characters and may contain letters, numbers, the underscore character, and the $ and # symbols. The first character may not be a number.

user_name................Name of an existing user that has at least the connect privilege.

group_name ....Name of an existing group.
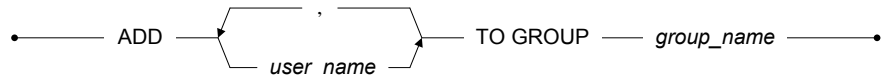
Figure 3-3 ADD TO Group syntax

➲ **Example 1**

This example illustrates adding users **Joe** and **John** to the **Manager** group.
```
dmSQL> ADD Joe, John TO GROUP Manager;
```

⮞ **Example 2**

The following example illustrates adding the groups **FullTime** and **PartTime** to the **Staff** group.

```
dmSQL> ADD FullTime, PartTime TO GROUP Staff;
```

⮞ **Example 3**

The following example illustrates adding user **Bill** and the group **FlexTime** to the **Staff** group.

```
dmSQL> ADD Bill, FlexTime TO GROUP Staff;
```

3.4 ADD TRACE

The ADD TRACE command adds trace on a single table to log the detaile OLD/NEW data. Actually, it is implemented by 3 internal triggers for insert/update/delete operation, which operation on the traced table would be logged, and the OLD/NEW data would be printed in **DBNAME_currentdate_###.TXT** as extra information. Only users with table owner, DBA, SYSDBA or SYSADM security privileges can execute the ADD TRACE command.

NOTE *DB_LgSvr need to be equal or greater then 4. Otherwise, the detail information would be skipped and nothing would be written to log files.*

table_name ....Name of an exisiting single table

```
•————— ADD TRACE ————— ON ——— table_name ———•
```

Figure 3-4 ADD TRACE Syntax

➲ **Example**

Add trace on table **tb1**, and insert, update delete record.

```
dmSQL> ADD TRACE ON tb1;
dmSQL> INSERT INTO tb1 VALUES (1, 'abc');
1 rows inserted
dmSQL> UPDATE tb1 SET c2 = 'xyz' WHERE c1=1;
1 rows updated
dmSQL> DELETE FROM tb1;
1 rows deleted
```

3.5   ALTER DATAFILE

The ALTER DATAFILE command enlarges the size of a data or BLOB file by adding a specified number of pages. Only users with SYSADM, SYSDBA or DBA security privileges can execute the ALTER DATAFILE command.

Files are physical units of storage that contain data in a database. The operating system manages files the DBMS managed data in the files. DBMaker uses Data, BLOB, and Journal type files.

Data files and BLOB *files* store user and system data. Although they have similar characteristics, DBMaker manages these two file types in different ways to improve performance. Data files store table and index data, while BLOB files store Binary Large Objects.

Journal files are special files that provide a real-time, historical record of all changes made to a database and the status of each change. This allows the database to undo changes made by a transaction that fails, or to redo changes made successfully but not written to disk after a database crash. Journal files are used only by the database management system, and are not used to store user data.

To ensure data independence of a database, operating system files cannot be referenced directly. Each database file has two names a physical file name and a logical file name. The *physical file name* is the name used by the operating system, while the *logical file name* is the name used by the database. These two file names interact via an entry in the **dmconfig.ini** file.

When using the ALTER DATAFILE command, specify the name of the logical file. Add 1 to 2,147,483,645 pages to a file, providing the total number of pages in the file does not exceed 2,147,483,647, and there is sufficient disk space. The total size of a file or all files in the same tablespace cannot exceed 8 TB.

file_name ....Name of the logical file to enlarge

number ....Number of pages to add

```
•——— ALTER DATAFILE ——— file_name ——— ADD ——— number ——— PAGES ———•
```

Figure 3-5 ALTER DATAFILE syntax

 ➲ **Example 1**

The following is an excerpt from a **dmconfig.ini** file displaying entries for four
database files with the logical and physical file names. The logical file names display
on the left and the physical file names display on the right.

```
customer_data = d:\dbmaker\tutorial\database\custdata.db 500
customer_blob = d:\dbmaker\tutorial\database\custblob.bb 1000
```

 ➲ **Example 2**

The following example adds **1000 pages** to the **customer_data** file.

```
dmSQL> ALTER DATAFILE customer_data ADD 1000 PAGES;
```

 ➲ **Example 3**

From the same **dmconfig.ini** file including the increased number of pages for the
customer_data file.

```
customer_data = d:\dbmaker\tutorial\database\custdata.db 1500
customer_blob = d:\dbmaker\tutorial\database\custblob.bb 1000
```

3.6   ALTER INDEX RENAME

The ALTER INDEX RENAME command renames an existing index on an existing table. The renaming only affects the index name in the system catalog; it will not rebuild the index in the database. Only the table owner, a DBA, or a user with the INDEX privilege may execute the ALTER INDEX RENAME command on a table.

index_name..............Index's original name

new_index_name......Index's new name

table_name...............Name of the table you are creating the index on

•—— ALTER INDEX —— *index_name* —— ON — *table_name* —— RENAME TO — *new_index_name* —•

Figure 3-6 ALTER IndexRename syntax

➲ **Example**
```
dmSQL> ALTER INDEX ix1 ON tb_tmp RENAME TO ix_new;
```

3.7 ALTER PASSWORD

The ALTER PASSWORD command changes a user password from its current value to a new value. A user can change their current password or the SYSADM may change the current password of any user.

When a user wants to change their current password, they should use the ALTER PASSWORD *old_password* TO *new_password* command. When the SYSADM changes the current password, they use the ALTER PASSWORD OF *user_name* TO *new_password* command. Only SYSADM may use the second command.

When changing a user password, the old password must match the password that is stored in the database for that user. If a user has no password, assign a password using the NULL keyword as the old password. To delete a user password use the NULL keyword as the new password.

Passwords have a maximum length of sixteen characters and may contain letters, numbers, the underscore character, and the $ and # symbols. The first character may not be a number.

user_name................Name of the user whose password is being changed

old_password............Current password for user *user_name*

new_password..........New password for user *user_name*
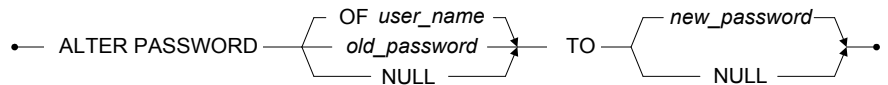
Figure 3-7 ALTER PASSWORD syntax

● **Example 1**

The following example illustrates assigning the password **abcdef** for a user with no password.

```
dmSQL> ALTER PASSWORD NULL TO abcdef;
```

➲ **Example 2**

The following example illustrates changing a password from abcdef to **a23456**.

```
dmSQL> ALTER PASSWORD abcdef TO a23456;
```

➲ **Example 3**

The following example illustrates removing a password named **a23456**.

```
dmSQL> ALTER PASSWORD a23456 TO NULL;
```

➲ **Example 4**

The following example illustrates how the **SYSADM** can change the password of user **John** to **abcedf**, regardless of the current value of the password.

```
dmSQL> ALTER PASSWORD OF John TO abcdef;
```

3.8 ALTER REPLICATION ADD REPLICATE

The ALTER REPLICATION ADD REPLICATE command adds an additional remote table to an existing table replication. Add as many additional remote tables to a replication as you wish. The table owner or a user with DBA, SYSDBA or SYSADM security privilege can execute the ALTER REPLICATION ADD REPLICATE command.

A table replication creates a full or partial copy of a table to a remote location. This allows users in remote locations to work with a local copy of data. The local copy remains synchronized with the database in another location. This way each database can service data requests immediately and efficiently, without having to go to another machine over a slower network connection. This is not the same as backing up the database to a remote location. The synchronization is done on a transaction-by-transaction basis by the *DBMS* without any intervention from users.

There are two primary types of table replication *synchronous* and *asynchronous*. *Synchronous* table replication modifies the remote table at the same time it modifies the local table. Asynchronous table replication stores changes to the local table and modifies the remote table based on a predefined schedule. The ALTER REPLICATION ADD REPLICATE command modifies both *synchronous* and *asynchronous* table replications.

Synchronous table replication in DBMaker uses a global transaction model, in which the replication of data to the remote table is treated as an integral part of the local transaction. This means that if the replication of data to the remote database fails, the transaction on the local table will also fail.

 A transaction is traditionally defined as a logical unit of work, or one or more operations on a database that must be completed together to leave the database in a consistent state. Transactions are self-contained and must either complete and change the data, or fail and leave the data unchanged.

Asynchronous table replication in DBMaker uses transaction logs to replicate data to the remote table. Modifications to the local table are stored in the transaction log, and replicated to the remote table according to a predefined schedule. Using the transaction log enables DBMaker to treat the local transaction and the remote transaction independently, allowing updates to the local tables even if the remote connection is not available. This allows *asynchronous* table replications to tolerate network and remote database failures; the replication will keep trying until any failures are corrected.

When modifying a table replication specify the replication name, local table name, and names of the additional remote tables to replicate to. The local table and the remote tables must already exist in their respective databases. DBMaker automatically drops any replications created for a table when dropping a table.

DBMaker will replicate an entire table unless a column list specifies the local table columns. Only specify a column list for the local table when creating the replication. To replicate an entire table without providing a column list, the columns in the local and remote tables must have the same names and data types.

If the column names in the local and remote tables are different, provide a column list for the remote table. Columns in the local table, from left to right, replicate to the corresponding columns in the column list for the remote table. Alternately, explicitly specify which columns in the local table correspond to columns in the remote table by providing a column list for both the local and remote tables. The number and data type of the primary key columns in both tables must match.

DBMaker does not identify replications using fully qualified names; a combination of owner and object names, but associates them with tables instead. For this reason all replication names on the same table must be unique.

Synchronous table replication operates with the same security and object privileges as the owner of the local table. If the remote table is specified using links then the replication operates with the same security and object privileges as the link.

Asynchronous table replication operates with the security privileges of the remote account specified by the IDENTIFIED BY keywords in the CREATE SCHEDULE

command. Create a schedule for an asynchronous table replication before creating the replication.

The CLEAR DATA/FLUSH DATA/CLEAR AND FLUSH DATA keywords are optional. These keywords specify the operations that take place when creating a replication. The CLEAR DATA keywords delete all data from the remote table when a replication is created. The FLUSH DATA keywords copy all data that matches a search condition into the remote table. The CLEAR AND FLUSH DATA keywords clear all data from the remote table, and then copy all data that matches a search condition into the remote table. If you do not specify an action, no action takes place.

The NO CASCADE keywords are optional. The keyword specifies a cascade replication. For example, commands flow in most organizations from the highest level to the basic level. This is similar to replicating data from point A to point B, and then to point C. This is a typical kind of Cascade replication. In the No-Cascade model A 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. If no specification exists, the default setting CASCADE will be used.

replication_name......Name of the table replication to add a remote table to.

local_table_name......Name of the local table the replication was created on.

remote_table_name ..Name of the table in the remote database.

column_name ....Name of a column in the remote table to replicate to.
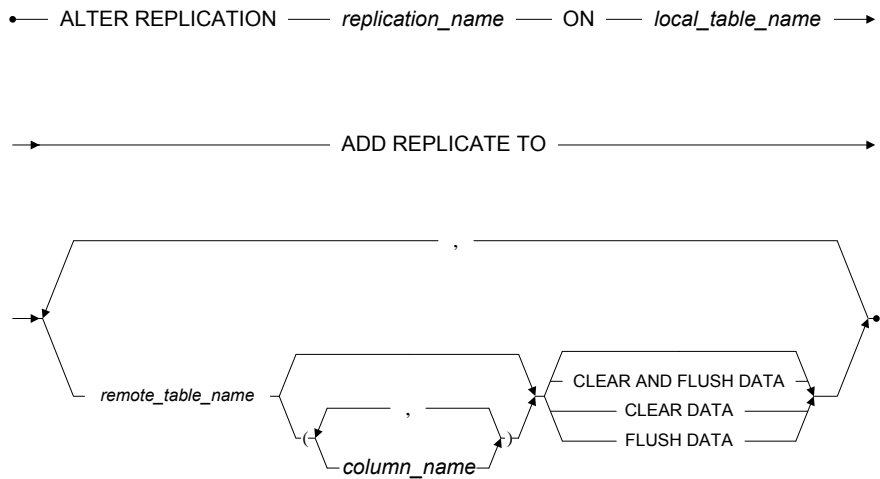
Figure 3-8 ALTER REPLICATION ADD REPLICATE syntax

⮎ **Example 1**

The following modifies a replication named **EmpRep** created on the local **Employeesinfo** table. Data replicates to the **Div1Emp** table in the remote database, which is identified by a database configuration section named **Div1Office** in the local **dmconfig.ini** file. All column names and data types in both tables are identical.

```
dmSQL> ALTER REPLICATION EmpRep ON Employeesinfo ADD REPLICATE TO
                    Div1Office:Div1Emp;
```

⮎ **Example 2**

The **CLEAR DATA** keyword causes DBMaker to delete all data in the remote table before the replication begins:

```
dmSQL> ALTER REPLICATION EmpRep ON Employeesinfo ADD REPLICATE TO
                    Div1Office:Div1Emp CLEAR DATA;
```

● **Example 3**

The **FLUSH DATA** keyword causes DBMaker to send data in the local table to the remote table before replication begins.

```
dmSQL> ALTER REPLICATION EmpRep ON Employeesinfo ADD REPLICATE TO
                       Div1Office:Div1Emp FLUSH DATA;
```

● **Example 4**

The **CLEAR AND FLUSH DATA** keyword causes DBMaker to delete all data in the remote table and then send data in the local table to the remote table.

```
dmSQL> ALTER REPLICATION EmpRep ON Employeesinfo ADD REPLICATE TO
                       Div1Office:Div1Emp CLEAR AND FLUSH DATA;
```

● **Example 5**

The following adds the replication to the **Div2Emp** table in the remote **Div2Office** database, and the **Div3Emp** table in the remote **Div3Office** database. Both remote databases have a database configuration section with the same name as the database in the local **dmconfig.ini** file.

```
dmSQL> ALTER REPLICATION EmpRep ON Employeesinfo ADD REPLICATE TO
                       Div2Office:Div2Emp CLEAR DATA,
                       Div3Office:Div3Emp FLUSH DATA;
```

3.9   ALTER REPLICATION DROP REPLICATE

The ALTER REPLICATION DROP REPLICATE command drops a remote table from an existing table replication. Drop a remote table from a table replication when you no longer want to replicate data to that table. Only the table owner or a user with DBA, SYSDBA or SYSADM security privilege can execute the ALTER REPLICATION DROP REPLICATE command.

A *table replication* creates a full or partial copy of a table in a remote 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 efficiently, without having to go to another machine over a slower network connection. This is not the same as backing up the database to a remote location. The synchronization is done on a transaction-by-transaction basis by the *DBMS*, without any user intervention.

There are two primary types of table replication, *synchronous* and *asynchronous*. Synchronous table replication modifies the remote table at the same time it modifies the local table. Synchronous table replication stores changes to the local table and modifies the remote table based on a predefined schedule. The ALTER REPLICATION DROP REPLICATE command modifies synchronous and asynchronous table replications.

Synchronous table replication in DBMaker uses a global transaction model, in which the replication of data to the remote table is treated as an integral part of the local transaction. A transaction is traditionally defined as a logical unit of work, or one or more operations on a database that must be completed together to leave the database in a consistent state. Transactions are self-contained and must either complete and change the data, or fail and leave the data unchanged. This means that if the replication of data to the remote database fails, the transaction on the local table will also fail.

Asynchronous table replication in DBMaker uses transaction logs to replicate data to the remote table. Modifications to the local table are stored in the transaction log, and

are replicated to the remote table according to a predefined schedule. Using the transaction log enables DBMaker to treat the local transaction and the remote transaction independently, updating local tables normally even if the remote connection is not available. This allows asynchronous table replications to tolerate network and remote database failures. The replication will keep trying until all failures are corrected.

To drop a remote table from a table replication, specify the replication name, the local table name, and the name of the remote table. Drop more than one remote table from a replication by listing all tables to drop. Any replications created for a table are dropped automatically when dropping the table.

replication_name......Name of the table replication to drop a remote table from.

local_table_name......Name of the local table the existing replication was created on.

remote_table_name ..Name of the table in the remote database to stop replicating to.

—— ALTER REPLICATION —— *replication_name* —— ON —— *local_table_name* ——▶

——————— DROP REPLICATE TO ———————— , ———— *remote_table_name* ————
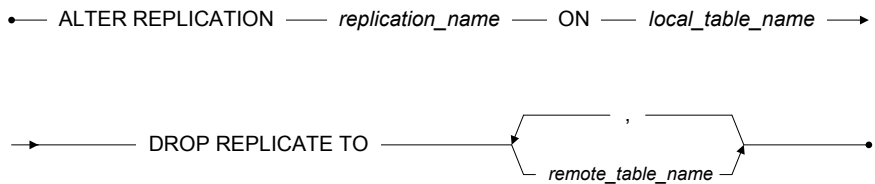
Figure 3-9 ALTER/DROP REPLICATION syntax

➲ **Example 1**

The following drops a remote table named **Div1Emp** from the replication named **EmpRep** created on the local **Employeesinfo** table.

```
dmSQL> ALTER REPLICATION EmpRep ON Employeesinfo DROP REPLICATE TO Div1Emp;
```

➲ **Example 2**

The following drops the remote tables named **Div2Emp**, **Div3Emp**, and **Div4Emp** from the replication named **EmpRep** created on the local **Employeesinfo** table.

```
dmSQL> ALTER REPLICATION EmpRep ON Employeesinfo DROP REPLICATE TO Div2Emp,
Div3Emp, Div4Emp;
```

3.10 ALTER SCHEDULE

The ALTER SCHEDULE command changes the replication schedule for an asynchronous table replication. Synchronous table replications do not use schedules, so the ALTER SCHEDULE command has no effect on a synchronous table replication. Only users with DBA, SYSDBA or SYSADM security privileges can execute the ALTER SCHEDULE command.

A *table replication* creates a full or partial copy of a table in a remote 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 efficiently, without having to go to another machine over a slower network connection. This is not the same as backing up the database to a remote location. The synchronization is done on a transaction-by-transaction basis by the *DBMS* without any intervention from users.

There are two primary types of table replication, synchronous and asynchronous. Synchronous table replication modifies the remote table at the same time it modifies the local table. Asynchronous table replication stores changes to the local table and modifies the remote table based on a predefined schedule. The ALTER SCHEDULE command affects only asynchronous table replications.

BEGIN AT specifies the date and time of the first replication for an asynchronous table replication. The date must be in *yyyy/mm/dd* format, where *yyyy* is the year in the range from 1970 to 2038, *mm* is the month in the range from 01 to 12, and *dd* is the date in the range from 01 to 31. The time must be in *hh:mm:ss* format, where *hh* is the hour in the range from 00 to 23, *mm* is the number of minutes in the range from 00 to 59, and *ss* is the number of seconds in the range from 00 to 59. The value for the year must be in the range from 1970 to 2038. Include both the date and time when using the BEGIN AT keyword. If you change the date or time of the first replication to a date in the future after a replication is already running, table data that has not yet been replicated to the remote database will wait until the new time for replication.

EVERY, defines the interval between successive replications for an asynchronous table replication. The interval may be provided as hours/minutes/seconds, days, or a

combination of both. To specify the number of hours/minutes/seconds, use EVERY *hh:mm:ss*, where *hh* is the number of hours in the range 00 to 23, *mm* is the number of minutes from 00 to 59, and *ss* is the number of seconds from 00 to 59. EVERY *d* DAYS, specifies the number of days, where *d* is the number of days in the range from 1 to 365. To specify a combination of both, use EVERY *d* DAYS AND *hh:mm:ss*.

RETRY, indicates how many times DBMaker tries replicating table data if there is an error while trying to process a single SQL statement, such as a lock time-out error, or rollback to *savepoint* due to a full Journal. To specify the number of times to try, use RETRY *n* TIMES, where *n* is the number of times to try in the range from 0 to 2,147,483,647. The default value is *0*.

If DBMaker encounters a network error or remote database error that prevents it from connecting to the remote server, DBMaker waits until the next scheduled replication to send any table data that was not successfully replicated. It will retry once if it encounters a transaction, which requires a rollback, but waits until the next scheduled replication if this fails.

The AFTER keyword is optional. This keyword is used together with the RETRY keyword to specify the interval between successive retries in the event of an error. Use AFTER *s* SECONDS to specify the interval, where *s* is the number of seconds in the range from 0 to 2,147,483,647. The default value is *5*.

The ON ERROR keyword specifies the action DBMaker takes when data in the remote database has been updated in such a way that the replication cannot take place. This includes situations where DBMaker tries to delete a record from the remote table, which has already been deleted, or tries to insert a record into a remote table that already exists. DBMaker provides two options when encountering this type of error, STOP ON ERROR and IGNORE ON ERROR. STOP ON ERROR indicates DBMaker stops replicating data when an error of this type occurs. IGNORE ON ERROR indicates that DBMaker ignores the data that caused the error and continues replicating the remaining data. The default behavior is IGNORE.

The IDENTIFIED BY keywords specify the user name and password to use when connecting to the remote database. The user name provided must be an existing user in the remote database with sufficient privileges on the remote tables to perform

INSERT, DELETE, and UPDATE operations. Security and object privileges granted to that user determine the operations that can be performed

Specify the remote database name to alter the schedule. The remote database name cannot be a database link. All asynchronous table replications on this database will use the new schedule.

yyyy/mm/dd .... Date to begin the replication

hh:mm:ss .... Time to begin the replication and time interval to replicate

d .... Day interval to replicate to the remote table

n .... Number of times to retry in the event of a failure

s .... Number of seconds to wait before retrying replication in the event of a failure

user_name .... User name of the account in the remote database

password .... Password of the account in the remote database
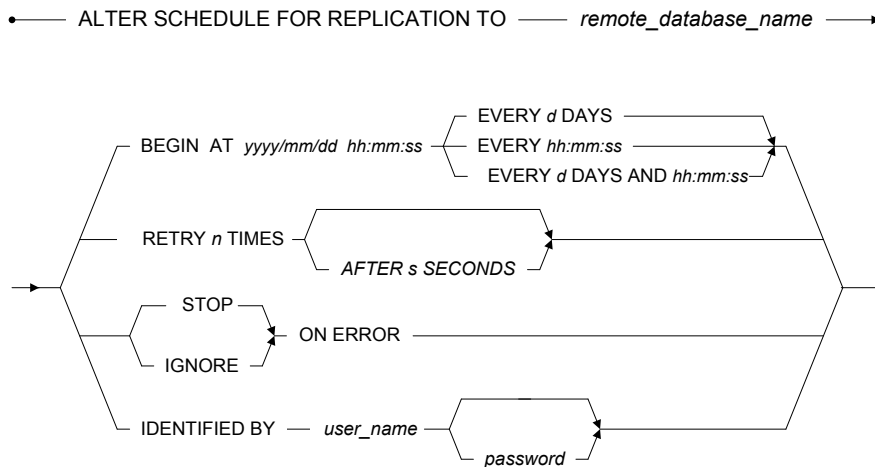
Figure 3-10 ALTER SCHEDULE syntax

⭢ **Example 1**

The following alters the replication schedule for the asynchronous replication named **EmpRep**. The number of times to retry after an error **lock time-out**, or a **rollback** to save point due to a full Journal, is set to **3**, with an interval of **5 seconds** between successive retries.

```
dmSQL> ALTER SCHEDULE FOR REPLICATION TO EmpRep
            RETRY 3 TIMES AFTER 5 SECONDS;
```

⭢ **Example 2**

The following alters the replication schedule for the asynchronous replication named **EmpRep**. The action DBMaker should take when data in the remote database has been updated in such a way that the replication couldn't take place is set to **STOP**:

```
dmSQL> ALTER SCHEDULE FOR REPLICATION TO EmpRep
            STOP ON ERROR;
```

⮱ **Example 3**

The following alters the replication schedule for the asynchronous replication named **EmpRep**. The **username** and **password** used for connecting to the remote database is set to a new value.

```
dmSQL> ALTER SCHEDULE FOR REPLICATION TO EmpRep
               IDENTIFIED BY RepUser rdejpe88;
```

3.11 ALTER TABLE ADD COLUMN

The ALTER TABLE ADD COLUMN command modifies the definition of an existing table and adds new columns. Only the table owner, a DBA, or a user with the ALTER privilege for that table may execute the command.

Specify a column definition by providing a column name and a data type or domain. Optionally, add multiple columns in a single command, however, the total number of columns in the table, after executing the command, must not exceed the maximum number of columns permitted in a table. The maximum number of columns allowed in a table is 2000.

table_name...............Name of the table to add columns

column_definition.....New definition for the column to alter
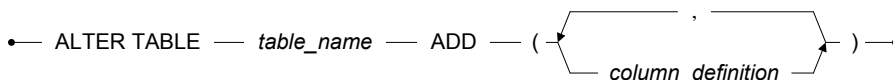
Figure 3-11 ALTER TABLE ADD COLUMN syntax

Column Definition

Specify a data type for each column. DBMaker supports the following data types: BIGINT, BIGSERIAL, BINARY, CHAR, DATE, DECIMAL, DOUBLE, FILE, FLOAT, INTEGER, BLOB, CLOB, OID, SERIAL, SMALLINT, TIME, TIMESTAMP and VARCHAR.

Optionally, specify a user-defined domain for the column instead of a data type. Domains are a combination of data type, default value, and constraints that are applied to a column when it is defined using the domain data type. See the

DEFAULT and CHECK keywords below for a description of default values and constraints. Default values and constraints provided in the column definition will override those of the domain. Column definitions can also provide constraints in addition to those of the domain.

The NULL/NOT NULL keywords are optional. These keywords specify whether a column can contain a NULL value; can be left empty, when inserting a new row. The NULL keyword specifies that a column may contain an undefined value when a new row is inserted. The NOT NULL keyword specifies that a value must be provided when a new row is inserted. The NOT NULL keyword cannot be used unless a table is empty, since the NOT NULL rule will be violated causing existing rows not to contain a value for the column. As a result, the column will not be created.

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 DEFAULT keyword is optional. This keyword is used to specify a default value that will be inserted into a column if no value is provided when inserting a new row. Constants, results from built-in functions, or the NULL keyword may be used as the default value. Use built-in functions that have no argument, such as PI( ), NOW( ), or USER( ), when defining a column. When using the NULL keyword as the DEFAULT value, the column cannot be defined with the NOT NULL keyword. The DEFAULT keyword is not normally required when using user-defined domains instead of the standard DBMaker data types, since domains normally include their own DEFAULT clause.

The ON UPDATE keyword 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 CHECK keyword is optional. This keyword is used to specify a range of acceptable values; constraints, that may be entered in a column. The expression that specifies the range of acceptable values may be any expression that evaluates a true or

false statement. The VALUE keyword may be used in the expression in conjunction with the CHECK keyword to represent the value of the column. If an SQL statement does not satisfy the CHECK condition, it is not processed. The CHECK keyword is not normally required when using user-defined domains in place of the standard DBMaker data types, since domains normally include their own CHECK clause.

The GIVE keyword is optional. This keyword is used to specify the value inserted into the new column for any rows that already exist in the table. If you do not provide a value using the GIVE keyword, DBMaker inserts a NULL value into the new column for any existing rows; columns using the SERIAL data type cannot contain NULL values, use the GIVE keyword when adding a SERIAL column. Constants, results from built-in functions, or the NULL keyword may be used as the GIVE value. Use the NULL keyword as the GIVE value; the column cannot be defined with the NOT NULL keyword. Also, use the SEQUENTIAL/SEQ keywords with the GIVE keyword when you insert a SERIAL column. These keywords specify that DBMaker will insert serial values into existing rows, starting with the value specified by the definition of the SERIAL data type in the column definition. The serial values continue to increment as new rows are inserted.

The BEFORE/AFTER keywords are optional. These keywords specify the location to insert the new column in relation to an existing column. The BEFORE keyword specifies DBMaker should insert the new column before, to the immediate left of, the specified column. The AFTER keyword specifies DBMaker should insert the new column after, to the immediate right of, the specified column. If you do not specify a relative location using the BEFORE/AFTER keywords, DBMaker simply appends the column to the right side of the table.

Adding a new column to a table has no effect on any views or synonyms based on that table. Column names have a maximum length of 128 characters and may contain letters, numbers, the underscore character, and the $ and # symbols. The first character must not be a number.

column_name...........Name of the new column

data_type ....Data type to use for the new column

domain_name ....Name of the domain to use for the new column

literal......................Literal value to be used if no value is inserted

constant ....Constant value to be used if no value is inserted

function_name ....Built-in function to be used if no value is inserted

constraint_name.......Name of constraint to be put on column

boolean_expression....Expression that evaluates to true or false

column_name_a.......The new column is positioned after the existing column with name *column_name_a*

column_name_b.......The new column is positioned before the existing column with name *column_name_b*
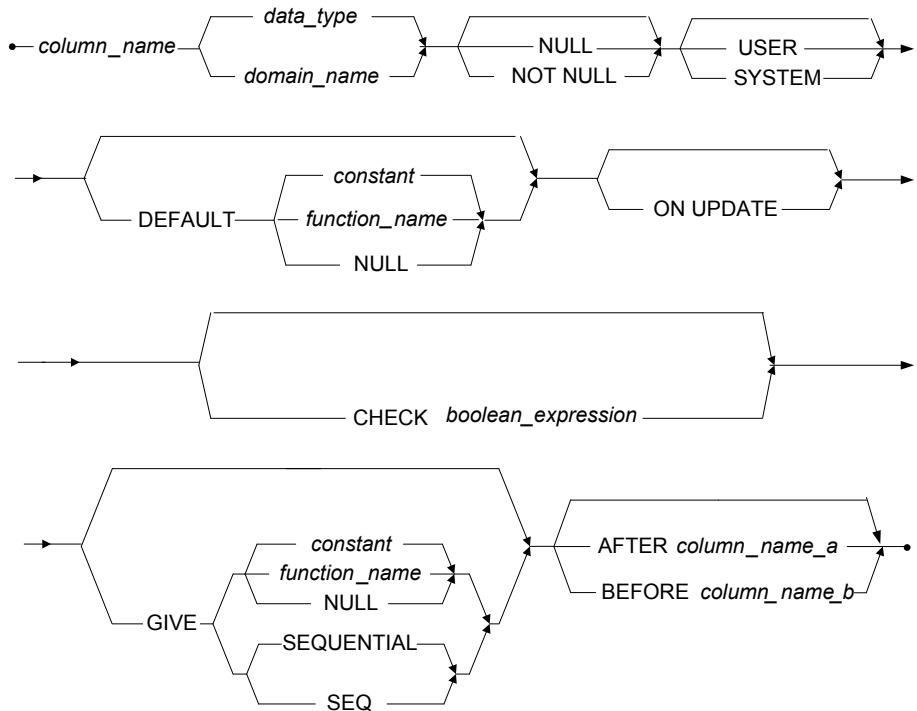
Figure 3-12 COLUMN DEFINITION syntax

➲ **Example 1**

The following example adds the **HireDate** column with the **DATE** data type to the **Employeesinfo** table.

```
dmSQL> ALTER TABLE Employeesinfo ADD (HireDate DATE);
```

➲ **Example 2**

The following adds the same **HireDate** column from the previous example, but adds the **NOT NULL** keyword to require a value is entered for this column when inserting a new row.

```
dmSQL> ALTER TABLE Employeesinfo ADD (HireDate DATE NOT NULL);
```

➲ **Example 3**

The following adds the same **HireDate** column from the previous example, but adds the **DEFAULT** keyword to insert a default value if no value is entered. This is the only case when you may omit a value for a column defined with the **NOT NULL** keyword. In this example, the built-in function **NOW( )** is used to insert the current date if no value is specified for this column.

```
dmSQL> ALTER TABLE Employeesinfo ADD (HireDate DATE NOT NULL DEFAULT NOW());
```

➲ **Example 4**

The following adds the same **HireDate** column from the previous example, but adds the **ON UPDATE** keyword to auto update a default value if other columns's value is changed.

```
dmSQL> ALTER TABLE Employeesinfo ADD (HireDate DATE NOT NULL DEFAULT NOW() ON
UPDATE);
```

➲ **Example 5**

The following adds the same **HireDate** column from the previous example, but adds the **CHECK** keyword to specify a range of acceptable values that may be entered in the **HireDate** column. The **VALUE** keyword represents the value to enter in the column.

```
dmSQL> ALTER TABLE Employeesinfo ADD (HireDate DATE NOT NULL DEFAULT NOW() CHECK
VALUE > '01/01/1995');
```

➲ **Example 6**

The following adds the same **HireDate** column from the previous example, but uses the user-defined **D_ValidDates** domain instead of the **DATE** data type. The **DEFAULT** and **CHECK** keywords are usually not required when using domains, since domains normally include their own **DEFAULT** and **CHECK** clauses.

```
dmSQL> ALTER TABLE Employeesinfo ADD (HireDate D_ValidDates NOT NULL);
```

3.12 ALTER TABLE ADD DYNAMIC COLUMN

The ALTER TABLE ADD DYNAMIC COLUMN command adds description information for a dynamic column. Only the table owner, a DBA, a SYSDBA, a SYSADM, or a user with the ALTER privilege for that table may execute the ALTER TABLE ADD DYNAMIC COLUMN command.

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.

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.

For details of a dynamic column, please refer to chapter *Using Dynamic Column* in *Database Administrator's Guide*. For details of a JSONCOLS column, please refer to chapter *Using JSONCOLS Type* in *Database Administrator's Guide*.

table_name...............Name of the table that has a JSONCOLS column

column_name...........Name of the dynamic column for which description information is added

data_type ....Data type to use for the dynamic column/added description information
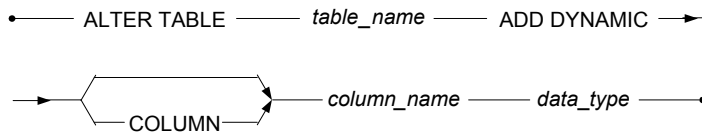
Figure 3-13 ALTER TABLE ADD DYNAMIC COLUMN syntax

➲ **Example**

The following example illustrates adding description information for a dynamic column.

```
dmSQL> CREATE TABLE books(name CHAR(50),info JSONCOLS);
dmSQL> INSERT INTO books(name,id,price) VALUES('C language','abc','19');
1 rows inserted
dmSQL> INSERT INTO books(name,id,price) VALUES('College english','2','32');
1 rows inserted
dmSQL> ALTER TABLE books ADD DYNAMIC COLUMN id INT;
dmSQL> ALTER TABLE books ADD DYNAMIC COLUMN price FLOAT;
dmSQL> SELECT name,id,price FROM books;
              NAME                   ID        PRICE
================================= ========== ====================
C language                          NULL    1.900000000000e+001
College english                        2    3.200000000000e+001
2 rows selected
```

3.13 ALTER TABLE DROP COLUMN

The ALTER TABLE DROP COLUMN command modifies the definition of an existing table and drops a column that was previously defined. To execute the ALTER TABLE DROP COLUMN command on a table, only the table owner, a DBA, a SYSDBA, a SYSADM, or user with ALTER privilege for that table.

Use this command to drop a column from a table when it is no longer necessary. You cannot drop a column if a primary or foreign key has been defined on that column, unless you drop the primary or foreign key first. If you drop a column with a defined view, the view will become invalid and DBMaker returns an error if you try to use it. This command should be used with caution since the data in a column cannot be recovered once dropped.

The CASCADE/RESTRICT keywords are optional. These keywords denote whether to remove or check dependent objects refered to the dropped column. When the CASCADE keyword is specified, it will remove all the dependent objects with the column. When the RESTRICT keyword is specified, it will not drop column that is referenced by any view definition, foreign key, or constraint. The RESTRICT keyword ensures that only columns with no dependent objects can be deleted.

table_name...............Name of the table dropping the column

column_name...........Name of the column to be dropped
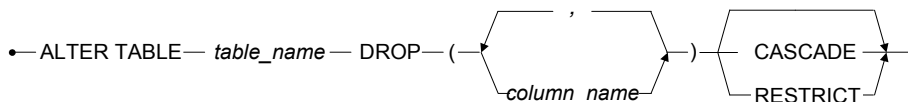
Figure 3-14 ALTER TABLE DROP COLUMN syntax

⮕ **Example 1**

This command drops the **BirthDate** column from the **Employeesinfo** table.

```
dmSQL> ALTER TABLE Employeesinfo DROP (BirthDate);
```

➲ **Example 2**

The following command drops the **BirthDate** and **HireDate** columns from the **Employeesinfo** table.

```
dmSQL> ALTER TABLE Employeesinfo DROP (BirthDate, HireDate);
```

➲ **Example 3**

The following command drops the column **BirthDate** from the **Employeesinfo** table and the dependent view **EmpView**.

```
dmSQL> CREATE VIEW EmpView AS SELECT BirthDate FROM Employeesinfo;
dmSQL> ALTER TABLE Employeesinfo DROP (BirthDate) CASCADE;
```

3.14 ALTER TABLE DROP DYNAMIC COLUMN

The ALTER TABLE DROP DYNAMIC COLUMN command drops the description information of a dynamic column, but doesn't drop data of this dynamic column. To execute the ALTER TABLE DROP DYNAMIC COLUMN command on a table, only the table owner, a DBA, a SYSDBA, a SYSADM, or user with ALTER privilege for that table.

Please note that, if a user drops a JSONCOLS column or the table which contains this JSONCOLS column, the description information of the dynamic columns contained in this JSONCOLS column will be automatically dropped by system.

For details of a dynamic column, please refer to chapter Using Dynamic Column in *Database Administrator's Guide*. For details of a JSONCOLS column, please refer to chapter *Using JSONCOLS Type* in *Database Administrator's Guide*.

table_name...............Name of the table containing the dynamic column to drop descripton information

column_name...........Name of the dynamic column to drop description information
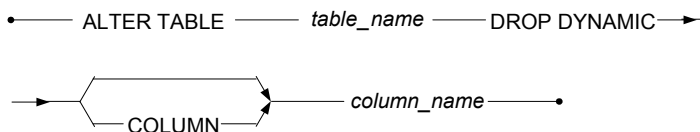
Figure 3-15 ALTER TABLE DROP DYNAMIC COLUMN syntax

➲ **Example**

The following example illustrates dropping description information of the dynamic column **id**. For details of table **books**, please refer to *ALTER TABLE ADD DYNAMIC COLUMN*.

```
dmSQL> ALTER TABLE books DROP DYNAMIC COLUMN id;
dmSQL> SELECT name,id,price FROM books;
```

```
    NAME                ID              PRICE
============ ======================= ==============
C language    abc                         *9e+001
College engl* 2                           *3e+001
2 rows selected
```

3.15 ALTER TABLE DROP FOREIGN KEY

The ALTER TABLE DROP FOREIGN KEY command modifies the definition of an existing table and drops a foreign key that was previously defined. Only the table owner, a DBA, or a user with the ALTER privilege for the table may execute the command.

A *key* is a column or combination of columns that help identify specific rows in a table. The columns that make up a key are known as *key columns*. A *unique key* is a key in which no two records have the same value for the key field.

A *primary key* is a key that uniquely identifies each row in a table. Without a primary key, it is impossible to distinguish between specific rows in a table because rows may contain duplicate values. The *DBMS* does not allow defining of a primary key on columns that contain duplicate values or entering a duplicate value in a primary key that already exists.

A *foreign key* is a key that corresponds to the primary key or a unique index of another table. This establishes a parent-child relationship between two tables that are represented by common data values. The parent table contains the primary key or unique index, and the child table contains the foreign key.

Referential integrity ensures that every value in a child key; the foreign key of the child table, has a corresponding value in the parent key; the primary key or unique index of the parent table. Referential integrity is enforced between tables using the parent-child relationship established with foreign keys. DBMaker has automatic support for referential integrity constraints between tables through the definition of foreign keys. When adding a record to a child table, the value in the child key must also exist in the parent key. Similarly, when deleting a record from the parent table, all records in the child key with the same value must be deleted first.

Referential actions provide a means to update or delete a parent key when referential integrity would not normally allow it, for example, when a child key references a parent key. The referential actions define the operation DBMaker should perform on

all matching child keys when you update or delete a parent key. DBMaker supports four referential actions for both updates and deletes: CASCADE, SET NULL, SET DEFAULT, and NO ACTION. CASCADE performs the update or delete on matching child keys as well as the parent key. SET NULL sets the value of matching child keys to NULL. SET DEFAULT sets the value of matching child keys to the default value of the column. NO ACTION enforces normal referential integrity rules. When no referential action is defined when a foreign key is created then, DBMaker uses NO ACTION by default.

Use the ALTER TABLE DROP FOREIGN KEY command to drop a foreign key on a table when it is no longer necessary. After dropping a foreign key, DBMaker no longer enforces referential integrity or performs referential actions on the child table. Without the foreign key it is possible to enter values in the child table that do not exist in the parent table and to update or delete values in the parent table. This command should be used with caution.

table_name .... Name of the table dropping the foreign key

key_name .... Name of the foreign key to be dropped

●── ALTER TABLE ── *table_name* ── DROP FOREIGN KEY ── *key_name* ──●

Figure 3-16 ALTER TABLE DROP FOREIGN KEY syntax

➲ **Example**

The following drops foreign key **fkey** from the **Salary** table.
```
dmSQL> ALTER TABLE Salary DROP FOREIGN KEY fkey;
```

3.16 ALTER TABLE DROP PRIMARY KEY

The ALTER TABLE DROP PRIMARY KEY command modifies the definition of an existing table and drops the primary key that was previously defined. Only the table owner, a DBA, or a user with both the ALTER and INDEX privileges for that table may execute the command.

A *key* is a column or combination of columns that help identify specific rows in a table. The columns that make up keys are *key columns*. A *unique key* is a key in which no two records have the same value for the key field.

A *primary key* is a key that uniquely identifies each row in a table. Without a primary key, it is impossible to distinguish between specific rows in a table because rows may contain duplicate values. The *DBMS* does not allow defining of a primary key on columns that contain duplicate values, and does not allow a duplicate value in a primary key.

A *foreign key* is a key that corresponds to the primary key or a unique index of another table. This establishes a parent-child relationship between two tables that are represented by common data values. The parent table contains the primary key or unique index, and the child table contains the foreign key columns corresponding to columns in the parent table.

Referential integrity ensures that every value in a child key; the foreign key of the child table, has a corresponding value in the parent key; the primary key or unique index of the parent table. Referential integrity is enforced between tables using the parent-child relationship established with foreign keys. DBMaker has automatic support for referential integrity constraints between tables through the definition of foreign keys. When adding a record to a child table, the value in the child key must also exist in the parent key. Similarly, when deleting a record from the parent table, all records in the child key with the same value must be deleted first.

Use the ALTER TABLE DROP PRIMARY KEY command to drop the primary key on a table when it is no longer necessary. DBMaker enforces referential integrity when

a foreign key is defined. Drop all foreign keys that refer to a primary key before you drop the primary key. After dropping a primary key, DBMaker no longer requires a unique key value for each record; it will be possible to enter values that may make two records indistinguishable from each other possibly causing database inconsistency. Use this command with caution.

table_name .... The name of the table from which the primary key is dropped

•———— ALTER TABLE ——— *table_name* ——— DROP PRIMARY KEY ————•

Figure 3-17 ALTER TABLE DROP PRIMARY KEY syntax

➔ **Example**

The following command drops the **Primary Key** from the **Employeesinfo** table.
```
dmSQL> ALTER TABLE Employeesinfo DROP PRIMARY KEY;
```

3.17   ALTER TABLE FOREIGN KEY

The ALTER TABLE FOREIGN KEY command modifies the definition of an existing table and adds a new foreign key. To execute the ALTER TABLE FOREIGN KEY command on a table, you must have the DBA security privilege, ALTER privilege on the table, *and* be the owner of the table, or have the REFERENCE privilege on the columns or table containing the primary key.

A *key* is a column or combination of columns that help identify specific rows in a table. The columns that make up a key are known as *key columns*. A *unique key* is a key in which no two records have the same value for the key field.

A *primary key* is a key that uniquely identifies each row in a table. Without a primary key, it is impossible to distinguish between specific rows in a table because rows may contain duplicate values. The *DBMS* does not allow you to define a primary key on columns that contain duplicate values, and does not allow entering a duplicate value in a primary key that already exists.

A *foreign key* is a key that corresponds to the primary key or a unique index of another table. This establishes a parent-child relationship between two tables that is represented by common data values stored in the tables. The parent table contains the primary key or unique index, and the child table contains the foreign key columns corresponding to columns in the parent table.

Referential actions provide a means to update or delete a parent key when referential integrity would not normally allow it such as when a parent key is referenced by a child key. The referential actions define the operation DBMaker should perform on all matching rows in the child key when updating or deleting a parent key. DBMaker supports four referential actions for both updates and deletes: CASCADE, SET NULL, SET DEFAULT, and NO ACTION.

The ON UPDATE/ON DELETE keywords are optional. These keywords specify the referential action DBMaker should perform when updating or deleting a value in a parent key. The referential actions for these keywords are CASCADE, SET NULL, SET DEFAULT, and NO ACTION.

CASCADE performs an update or delete on all matching values in the child key when updating or deleting the parent key. This will set the value of the child key to the same value as the parent key when a row in the parent key updates, or will delete all matching values in the child key with the same value as the parent key when deleting a row in the parent key.

SET NULL sets all matching values in the child key to NULL when you update or delete a row in the parent key. You cannot use the SET NULL action when the child key was defined with the NOT NULL constraint.

SET DEFAULT sets all matching values in the child key to the default value of the column when you update or delete a row in the parent key. You cannot use the SET DEFAULT action when the default value is NULL and the child key was defined with the NOT NULL constraint.

NO ACTION enforces normal referential integrity rules. DBMaker will use NO ACTION by default.

No limit exists for the number of foreign keys on a table. The parent key may be the primary key or any other unique index of a table, but create the parent key before adding the child key. The number of columns and column type or length must be the same in the parent key and the child key. The column order of corresponding keys may be different in each table, provided they are listed in corresponding order in the ALTER TABLE FOREIGN KEY command. The primary key of the parent table is used by default.

Columns in a foreign key may contain null values. If a foreign key contains a null value, it automatically satisfies referential integrity. You may not create a foreign key on a view, but may create one on a synonym. Foreign key names have a maximum length of 128 characters, and may contain numbers, letters, underscore characters, and $ and # symbols. The first character may not be a number.

table_name .... Name of the table adding the foreign key to

key_name .... Name of the new foreign key

column_name .... 1. Name of the column the foreign key is created on

.... 2. Name of the column referenced by the foreign key

parent_table_name ...Name of the table the foreign key references
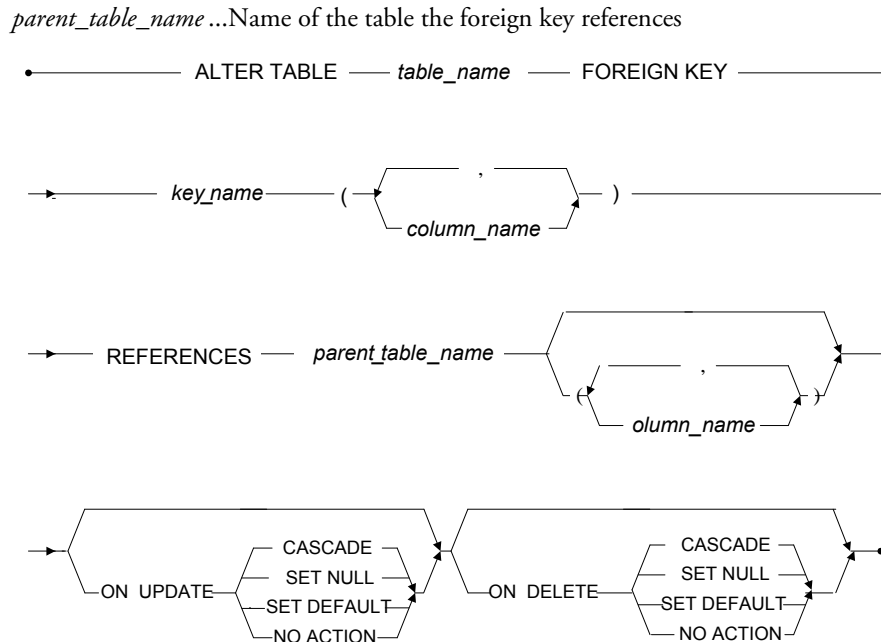
Figure 3-18 ALTER TABLE FOREIGN KEY syntax

⦿ **Example 1**

The following creates a foreign key named **fkey_CNo** on column **CustNo** of table **Accounts** that references the **Customers** table. In the example, no column name is specified for the parent key, DBMaker will use the primary key of the Customers table as the parent key. The primary key of the Customers table must be defined before executing the command.

```
dmSQL> ALTER TABLE Accounts FOREIGN KEY fkey_CNo (CustNo)
                    REFERENCES Customers;
```

⦿ **Example 2**

The following creates the same foreign key **fkey_CNo** from the previous example, but specifies the **CustNo** column as the parent key. The **CustNo** column can be the primary key of the **Accounts** table or any other unique index. The primary key or

other unique index of the Customers table must be defined before executing this command.

```
dmSQL> ALTER TABLE Accounts FOREIGN KEY fkey_CNo (CustNo)
                    REFERENCES Customers (CustNo);
```

➲ **Example 3**

The following creates a foreign key named **fkey_No** on columns **PartNo** and **StockNo** of table **Invoice** that references the **Stock** table. Column order in the **Invoice** table (**PartNo**, **SuppNo**) is different from the corresponding columns in the **Stock** table (**SuppNo**, **PartNo**). This is acceptable provided corresponding columns from each table are listed in the same order in the command.

```
dmSQL> ALTER TABLE Invoice FOREIGN KEY fkey_No (SuppNo, PartNo)
                    REFERENCES Stock (SuppNo, PartNo);
```

➲ **Example 4**

The following creates the same foreign key **fkey_No** from the previous example, but defines the referential actions DBMaker should perform. The **ON UPDATE SET DEFAULT** keywords specify DBMaker to set all matching values in the child key to the default column value when updating a row in the parent key. The **ON DELETE SET NULL** keywords specify DBMaker to set all matching values in the child key to **NULL** when deleting a row in the parent key.

```
dmSQL> ALTER TABLE Invoice FOREIGN KEY fkey_No (SuppNo, PartNo)
                    REFERENCES Stock (SuppNo, PartNo)
                    ON UPDATE SET DEFAULT
                    ON DELETE SET NULL;
```

3.18   ALTER TABLE MODIFY COLUMN

The ALTER TABLE MODIFY COLUMN command modifies the definition of existing columns in a table. Only the table owner, a DBA, a SYSDBA, a SYSADM, or a user with the ALTER privilege for that table may execute the ALTER TABLE MODIFY COLUMN command.

table_name...............Name of the table you are modifying the column on

column_name...........Name of the column you are modifying

column_definition.....New definition for the column
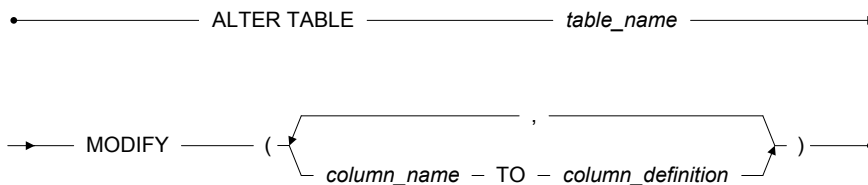
Figure 3-19 ALTER TABLE MODIFY COLUMN syntax

Column Definitions

To specify a column definition, provide a column name and a data type or domain. Modify multiple columns in a single command, up to the maximum number of 252 columns permitted in a table.

Specify a data type for each column modified. DBMaker supports the following data types: BINARY, CHAR, DATE, DECIMAL, DOUBLE, FILE, FLOAT, INTEGER, BLOB, CLOB, OID, SERIAL, SMALLINT, TIME, TIMESTAMP and VARCHAR.

Optionally, specify a user-defined domain for the column instead of a data type. Domains are a combination of data type, default value, and constraint that are applied to a column when it is defined using a domain data type. (See the DEFAULT and CHECK keywords below for a description of default values and constraints). Default values and constraints provided in the column definition will override those of the domain. Column definitions can also provide constraints in addition to those of the domain.

The NULL/NOT NULL keywords are optional. These keywords specify whether a column can contain a NULL value, left empty, when inserting a new row. The NULL keyword specifies that a column may contain an undefined value when inserting a new row. The NOT NULL keyword specifies that a value must be provided when a new row is inserted. The NOT NULL keyword cannot be used when modifying a column that was previously defined with NULL, unless the table is empty, or by using the GIVE keyword.

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 DEFAULT keyword is optional. This keyword is used to specify a default value that will be inserted into a column if no value is provided. Constants, results from built-in functions, or the NULL keyword may be used as the default value. Only use built-in functions that have no argument PI( ), NOW( ), or USER( ), when defining a column. Use the NULL keyword as the DEFAULT value; the column cannot be defined with the NOT NULL keyword. The DEFAULT keyword is not normally required when using user-defined domains instead of the standard DBMaker data types, since domains normally include their own DEFAULT clause.

The ON UPDATE keyword 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 CHECK keyword is optional. This keyword is used to specify a range of acceptable values that may be entered in a column. The expression that specifies the range of acceptable values may be any expression that evaluates a true or false

statement. The VALUE keyword may be used in the expression in conjunction with the CHECK keyword to represent the value of the column. If an SQL statement does not satisfy the CHECK conditions, it is not processed. The CHECK keyword is not normally required when using user-defined domains instead of the standard DBMaker data types.

The GIVE keyword is optional. This keyword is used to specify the value inserted into the modified column for any existing rows that contain NULL values. If you modify a column from NULL to NOT NULL and do not provide a value using the GIVE keyword, DBMaker will not modify the column. Constants, results from built-in functions, or the NULL keyword may be used as the GIVE value. Use the NULL keyword as the GIVE value; the column cannot be defined with the NOT NULL keyword. Alternately, use the SEQUENTIAL/SEQ keywords with the GIVE keyword when modifying a column to a SERIAL column. These keywords specify that DBMaker will insert serial values into existing rows, starting with the value specified by the definition of the SERIAL data type in the column definition. The serial values will continue to increment as you insert new rows.

The BEFORE/AFTER keywords are optional. These keywords specify the location to position the modified column in relation to another column. The BEFORE keyword specifies DBMaker to position the modified column before; to the immediate left of, the specified column. The AFTER keyword specifies DBMaker to position the modified column after; to the immediate right of, the specified column. If you do not specify a relative location using the BEFORE/AFTER keywords, DBMaker leaves the column in the original position.

Modifying a column in a table makes all views and stored commands defined on the table invalid, but has no effect on any synonyms based on that table. Column names have a maximum length of 128 characters, and may contain letters, numbers, the underscore character, and the $ and # symbols. The first character may not be a number.

column_name...........Name of the modified column

data_type ....Data type to use for the modified column

domain_name ....Name of the domain to use for the modified column

literal...................... Literal value to be used if no value is inserted

constant .... Constant value to be used if no value is inserted

function_name .... Built-in function to be used if no value is inserted

constraint_name....... Constraint to be applied to the column

boolean_expression.... Expression that evaluates to true or false

column_name_a....... The modified column will be positioned after *column_name_a*

column_name_b....... The modified column will be positioned before *column_name_b*
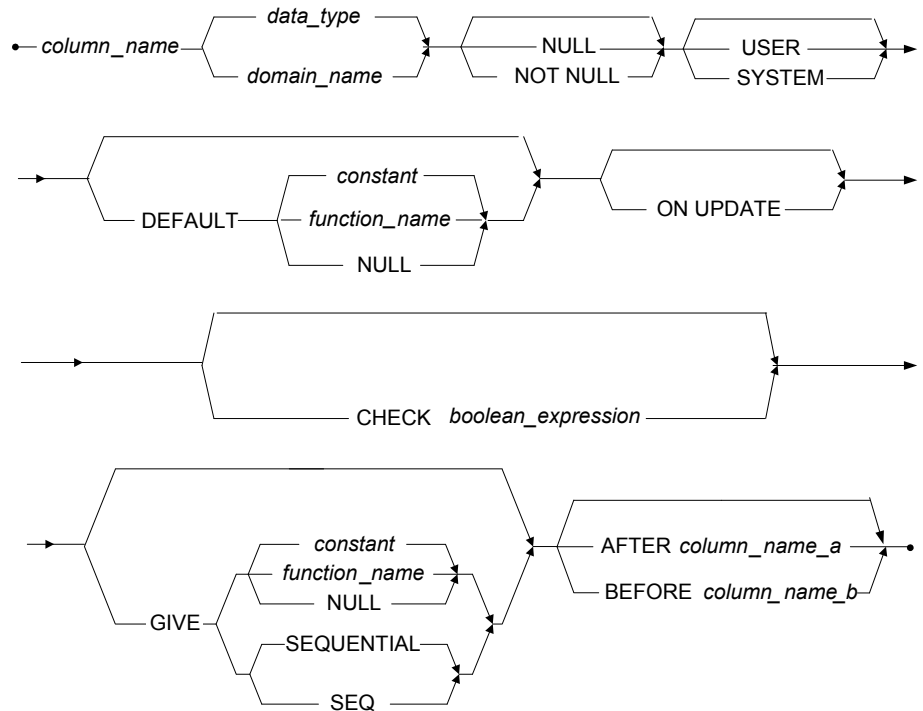
Figure 3-20 The Column Definitions syntax

➲ **Example 1**

The following modifies the length of the **Phone** column in the **Employeesinfo** table by changing the data type from **CHAR(15)** to **CHAR(20)**.

```
dmSQL> ALTER TABLE Employeesinfo MODIFY (Phone TO Phone CHAR(20));
```

➲ **Example 2**

The following modifies the length of the **Phone** column in the **Employeesinfo** table by changing the data type from **CHAR(15)** to **CHAR(20)**. Adds the **NOT NULL** keyword and requires a value to be entered for this column, when inserting a new row. Any rows that previously contained **NULL** values are assigned a new value using the **GIVE** keyword.

```
dmSQL> ALTER TABLE Employeesinfo MODIFY (Phone TO Phone CHAR(20)
                                        NOT NULL
                                        GIVE '000-0000');
```

➲ **Example 3**

The following modifies the data type of the **Quantity** and **Amount** columns in the **LineItems** table by changing the data type of both columns from **SMALLINT** to **INT**.

```
dmSQL> ALTER TABLE LineItems MODIFY (Quantity TO Quantity INT,
                                    Amount TO Amount INT);
```

3.19 ALTER TABLE MODIFY DYNAMIC COLUMN

The ALTER TABLE MODIFY DYNAMIC COLUMN command modifies the existing description information of a dynamic column. To execute the ALTER TABLE MODIFY DYNAMIC COLUMN command on a table, only the table owner, a DBA, a SYSDBA, a SYSADM, or user with ALTER privilege for that table.

Dynamic columns only support modifying data type.

For details of dynamic columns, please refer to chapter *Using Dynamic Column* in *Database Administrator's Guide*.

table_name .... Name of the table containing the dynamic column whose description information will be modified

column_name .... Name of the dynamic column whose description information will be modified

data_type .... Data type to use for the modified dynamic column

Figure 3-21 ALTER TABLE MODIFY DYNAMIC COLUMN syntax

➲ **Example**

The following example illustrates modifying description information of the dynamic column **price**. For details of table **books**, please refer to *ALTER TABLE ADD DYNAMIC COLUMN*.

```
dmSQL> ALTER TABLE books MODIFY DYNAMIC COLUMN price TYPE TO INT;
dmSQL> SELECT name,id,price FROM books;
    NAME                 ID                     PRICE
============ ======================== ==============
```

```
C language    abc                              19
College engl* 2                                32
2 rows selected
```

3.20   ALTER TABLE PRIMARY KEY

The ALTER TABLE PRIMARY KEY command modifies the definition of an existing table and adds a primary key. Only the table owner, a DBA, or a user with both the ALTER and INDEX privileges for the table may execute the command.

A *key* is a column or combination of columns that help identify specific rows in a table. A *unique key* is a key in which no two records have the same value or the key field.

A *primary key* is a key that uniquely identifies each row in a table. Without a primary key, it is impossible to distinguish between specific rows in a table because rows may contain duplicate values. The *DBMS* will not define a primary key on columns that contain duplicate values, or enter a duplicate value in a primary key that already exists.

A *foreign key* is a key that corresponds to the primary key or a unique index of another table. This establishes a parent-child relationship between two tables that is represented by common data values stored in the tables. The parent table contains the primary key or unique index, and the child table contains the foreign key columns corresponding to columns in the parent table.

Referential integrity ensures that every value in a child key; the foreign key of the child table, has a corresponding value in the parent key; the primary key or unique index of the parent table. Referential integrity is enforced between tables using the parent-child relationship established with foreign keys. DBMaker has automatic support for referential integrity constraints between tables through the definition of foreign keys. When adding a record to a child table, the value in the child key must also exist in the parent key. Similarly, when deleting a record from the parent table, all records in the child key with the same value must be deleted first.

Primary keys ensure data integrity in a table by requiring unique key values in each record of the primary key. Since this means columns in a primary key may not contain duplicate or null values, define the key columns with the NOT NULL constraint.

Each table may only have one primary key. You cannot name a primary key for this reason. Instead, DBMaker will automatically create and maintain a unique, internally

managed index named PrimaryKey for the primary key in each table. Since DBMaker builds an index on the primary key, it is not necessary to build another index on the columns in the primary key to increase the performance of query operations.

Primary keys may be built on up to 32 columns, providing the size of the columns does not exceed 4000 bytes. Primary keys cannot be created on views, but may be created on synonyms. When creating a primary key on a synonym, the primary key is created on the base table.

table_name...............Name of the table adding the primary key to

column_name...........Name of the column the primary key is created on

Figure 3-22 ALTER TABLE PRIMARY KEY syntax

➲ **Example**

The following example creates a primary key on column **CustNo** in the **Customers** table. The **CustNo** column must be defined with the **NOT NULL** constraint, and all values in the **CustNo** column must be unique, or the table must be empty.

```
dmSQL> ALTER TABLE Customers PRIMARY KEY (CustNo);
```

3.21 ALTER TABLE RENAME

The ALTER TABLE RENAME command changes the name of an existing table. Only the table owner, a DBA, or a user with the ALTER privilege for that table can execute the ALTER TABLE RENAME command on a table.

A table name can be renamed when it only contains an index and/or text index. Dependent objects like stored command, stored procedure, trigger, and foreign key are not supported with the RENAME command.

table_name .... The table's name to alter

new_table_name ...... The table's new name

```
•—— ALTER TABLE — table_name ——— RENAME TO ——— new_table_name —•
```

Figure 3-23 ALTER TABLE RENAME Syntax

3.22   ALTER TABLE SET OPTIONS

The ALTER TABLE SET OPTIONS command modifies the definition of an existing table and changes its options. Only the table owner, a DBA, or a user with the ALTER privilege for that table can execute the ALTER TABLE SET OPTIONS command on a table.

LOCK MODE specifies the lock mode (lock level) DBMaker uses when accessing data in a table. DBMaker has three lock modes; table, page, and row. Page lock mode is set by default. To determine the lock mode of a table, examine the LOCKMODE column of the SYSTABLE.

LOCK MODE TABLE locks an entire table. This mode decreases concurrency by preventing simultaneous user access to the locked table. It also uses fewer lock resources and requires less memory in the *System Control Area (SCA).*

LOCK MODE PAGE locks a single data page. This mode is a trade-off between concurrency and lock resources. It provides moderate concurrency since other users may access data in other pages, but not in the locked page.

LOCK MODE ROW locks a single row. This mode increases concurrency by allowing additional users to access any data except the locked row. It also uses more lock resources and requires more memory in the *SCA*.

FILLFACTOR specifies the maximum percentage of a data page that can be filled. This allows the database to optimize the use of data pages by reserving space for future updates to existing records. The *number* parameter can have a value from 50 to 100, which represent a fillfactor from 50% to 100%. To determine the fillfactor of a table, examine the FILLFACTOR column of the SYSTABLE system table.

NOCACHE limits the number of page buffers used to cache data during a table scan. DBMaker stores page buffers in a buffer chain with the most recently used page at the beginning. When the NOCACHE option is turned on, data pages read during a table scan are placed at end of the buffer chain. The end of the buffer chain will be flushed before the beginning and subsequent data pages read during the table scan will overwrite the previous pages. This effectively limits the page buffers used during a

table scan to one page buffer. To determine the cache mode of a table, examine the CACHEMODE column of the SYSTABLE system table.

The SERIAL option resets the counter for a serial column. This allows starting a new sequence in a serial column without having to modify the table.

Using the ALTER TABLE SET OPTIONS command has no effect on any views or synonyms based on that table.

table_name ....Name of the table to change options on

number....................Value to use for the fillfactor

n ....Time interval in days to wait between statistics updates
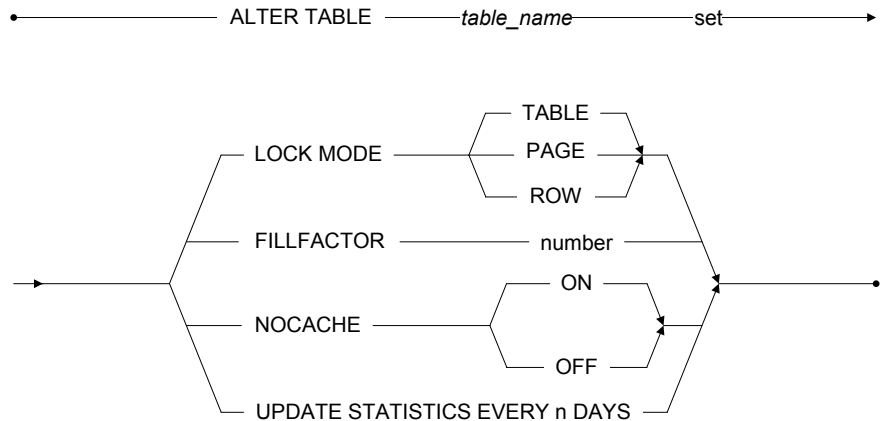
Figure 3-24 ALTER TABLE SET OPTIONS syntax

➲ **Example 1**

The following sets the **LOCK MODE** to **TABLE** on the **Customers** table.
```
dmSQL> ALTER TABLE Customers SET LOCK MODE TABLE;
```

⮕ **Example 2**

The following sets the **LOCK MODE** to **PAGE** on the **Customers** table.
```
dmSQL> ALTER TABLE Customers SET LOCK MODE PAGE;
```

⮕ **Example 3**

The following sets the **LOCK MODE** to **ROW** on the **Customers** table.
```
dmSQL> ALTER TABLE Customers SET LOCK MODE ROW;
```

⮕ **Example 4**

The following sets the **FILLFACTOR** to **90%** on the **Customers** table.
```
dmSQL> ALTER TABLE Customers SET FILLFACTOR 90;
```

⮕ **Example 5**

The following turns on the **NOCACHE** option on the **Customers** table.
```
dmSQL> ALTER TABLE Customers SET NOCACHE ON;
```

⮕ **Example 6**

The following turns off the **NOCACHE** option on the **Customers** table.
```
dmSQL> ALTER TABLE Customers SET NOCACHE OFF;
```

⮕ **Example 7**

The following alters the **SERIAL** counter value of table **tb_tmp** from its current value to **100**.
```
dmSQL> ALTER TABLE tb_tmp SET SERIAL 100;
```

3.23   ALTER TABLE TO ANOTHER TABLESPACE

The ALTER TABLE TO ANOTHER TABLESPACE command moves a table to another tablespace, 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. Only the table owner, a DBA, or a user with both the ALTER and INDEX privileges for the table may execute the command.

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 a 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 has 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.

table_name .... Name of the table to be moved

tablespace_name .... Name of the tablespace to move to

●── ALTER TABLE ── *table_name* ── MOVE TABLESPACE ── *tablespace_name* ──●

Figure 3-25 ALTER TABLE TO ANOTHER TABLESPACE syntax

➲ **Example**

The following moves the table **Employeesinfo** in **ts_mode** to another tablespace **ts_new.**

```
dmSQL> ALTER TABLE Employeesinfo MOVE TABLESPACE ts_new;
```

3.24 ALTER TABLESPACE

The ALTER TABLESPACE command adds a file to an existing tablespace or changes the tablespace type from autoextend to regular or from regular to autoextend or changes the tablespace type from read-write to read-only or from read-only to read-write. Only users with DBA, SYSDBA or SYSADM security privileges can execute the ALTER TABLESPACE command.

The way data is physically stored on computers has little or no significance to most users. DBMaker uses the relational data model to hide the details of the physical storage model and present data using a logical storage model instead.

In the DBMaker physical storage model, files are physical storage structures that contain the data in the database. Files are managed by the operating system, with the exception of raw UNIX devices, while data in the files are managed by the *DBMS*. DBMaker uses three types of files during normal operation Data, BLOB, and Journal.

Journal *files* are special files that provide a real-time, historical record of all changes made to a database and the status of each change. This allows the database to undo changes made by a transaction that fails or to redo changes made successfully but not written to disk after a database crash. Journal *files* are used only by the database management system not to store user data.

Data *files* and BLOB *files* are used to store user and system data. Although they have similar characteristics, DBMaker manages these two file types in different ways to improve performance. Data *files* store table and index data, while BLOB *files* store only Binary Large OBjects (BLOBs).

In the DBMaker logical storage model, tablespaces are the logical storage structures used to partition information in a database into manageable areas. Each tablespace may contain several tables and indexes. Data in the tablespace is managed by the *DBMS*, but is physically stored in files. There are three types of tablespaces: regular, autoextend, and system.

Regular *tablespaces* have a fixed size and contain one or more data or BLOB *files*. They may be extended manually by enlarging existing files or adding new files in the tablespace. When adding a new file, first make an entry in the **dmconfig.ini**,

specifying the logical file name, the physical file name, and the initial file size in the appropriate database section. A regular tablespace may contain a maximum of 32767 *files*, with a maximum cumulative file size of 8 TB. On *UNIX* platforms, regular tablespaces may be placed on raw devices.

NOTE   *For more information on raw devices, see your UNIX system documentation.*

Autoextend *tablespaces* automatically increase in size to hold additional data as required. They must contain at least one or more data files, and may contain BLOB *files*. The difference between regular and autoextend tablespaces is, an autoextend tablespace automatically extends. A DBA can arrange tables for each type of tablespace. When adding a file to a regular tablespace, first make an entry in the **dmconfig.ini**, specifying the logical file name, physical file name, and initial file size in the appropriate database section. Autoextend tablespaces do not support raw devices.

DBMaker generates system tablespaces while creating a database. Each database has one system tablespace, which contains the system catalog tables used to store schema, security, and status information. The system tablespace is created as an autoextend tablespace, unless creating a database on a *UNIX raw device*. System tablespaces automatically contain one DATA and one BLOB *file*. System tablespaces may be converted to regular tablespaces.

Use the SET AUTOEXTEND OFF *keywords* to change any autoextend tablespace to a regular tablespace. To restrict the amount of disk space a tablespace will occupy, change a tablespace from autoextend to regular.

NOTE   *A file in an autoextend tablespace will grow to fill all available space on a disk to a maximum of 8 TB.*

Use the SET AUTOEXTEND ON *keywords* to change any regular tablespace to an autoextend. Change a tablespace from regular to autoextend when the tablespace is exhausted.

Read-only tablespaces do not allow users to perform any modifications on the tablespace. However, the read-only tablespace has many advantages:

• Eliminates the need to perform backups. Read-only tablespaces just need a single backup after being made read only.

- Recovery becomes easier. When the instance is started, DBMaker will take advantage of the fact that read-only tablespace does not need any media recovery.

- A read only tablespace requires few system resources than an updateable tablespace (no lock).

Use the SET READ ONLY keywords to change any read-write tablespace to a read-only tablespace.

Use the SET READ WRITE keywords to change any read-only tablespace to a read-write tablespace.

Use the ADD DATAFILE *keywords* to add a new Data or BLOB *file* to a tablespace. Files added to a tablespace do not have to be located on the same physical disk. In UNIX, file can be stored on raw devices. DBMaker writes to raw device files directly instead of relying on operating system calls, allowing faster access, and performance improvements over normal files.

As mentioned earlier, files that make up a tablespace are referenced within the database using logical file names to maintain physical data independence. The logical file names are mapped to the physical file names in the; **dmconfig.ini** configuration file, as shown in the examples. DBMaker will create a new file in the default database directory specified by the **DB_DbDir** keyword in the **dmconfig.ini** unless a different directory or path is specified.

Logical file names have a maximum length of 128 characters, and may contain numbers, letters, the underscore character, and the $ and # symbols .The first character may not be a number. Physical file names have a maximum length, including drive and path names, of 256 characters. Include any characters and symbols permitted by the operating system, except spaces.

When adding a new file, specify the file type with the TYPE = DATA and TYPE=BLOB *keywords*. The default file type is data.

Also, indicate the file size; in data pages, for a Data *file* or BLOB frames for a BLOB *file*. Data pages can be: 4 KB, 8 KB, 16 KB or 32 KB, while BLOB pages are variable in size and can range from 8 KB to 256 KB. DBMaker increases the initial size of

autoextend tablespaces as required. To determine the size of a BLOB frame, check the **DB_BfrSz** keyword for a database in the **dmconfig.ini** file.

tablespace_name ....Name of the tablespace to modify

file_name ....Name of the file to add to the tablespace
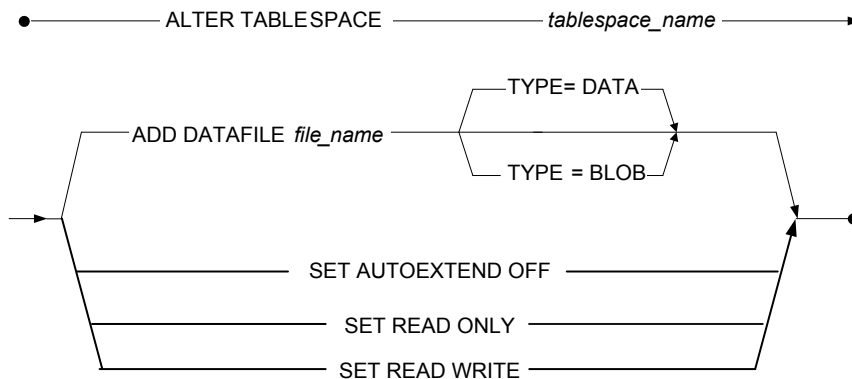
Figure 3-26 ALTER TABLESPACE syntax

➲ **Mapping 1**

Before executing example 1, add a line to the **dmconfig.ini** file to map the logical file name to the physical file name and indicate the initial file size as 4 KB pages if you had set the page size to 4 KB. In this example, the file size will be 400 KB.

```
file1=c:\dbmaker\databases\f1.db 100
```

➲ **Example 1**

The following adds the file **f1.db** to the **ts_new** tablespace file **f1.db** has the logical file name of **file1**.

```
dmSQL> ALTER TABLESPACE ts_new ADD DATAFILE file1 TYPE=DATA;
```

➲ **Mapping 2**

Before executing the commands in example 2, add a line to the **dmconfig.ini** file to map the logical file name to the physical file name and indicate the initial file size in

frames. In this example, the file size will be 4000 KB if the default BLOB frame size of 8 KB is used.

```
file2=c:\dbmaker\databases\f2.bb 500
```

➲ **Example 2**

The following example changes the tablespace mode from **autoextend** to **regular** and adds file **f2.bb** to the **ts_mode** tablespace; file **f2.db** has the logical file name of **file2**.

```
dmSQL> ALTER TABLESPACE ts mode SET AUTOEXTEND OFF;
dmSQL> ALTER TABLESPACE ts_mode ADD DATAFILE file2 TYPE=BLOB;
```

➲ **Example 3**

The following example changes the tablespace mode from read-write to read-only.

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

3.25 ALTER TABLESPACE DROP DATAFILE

The ALTER TABLESPACE DROP DATAFILE command drops an empty datafile from a tablespace. Only users with DBA, SYSDBA or SYSADM security privileges can execute the ALTER TABLESPACE DROP DATAFILE command.

When dropping a datafile from a tablespace it is imperative that the datafile is empty. If the datafile contains data then the command will abort and an error message will be returned to the user. Users are not able to drop a datafile if the datafile is the only one in the tablespace. It is also important to note that users cannot remove the system datafile from the system tablespace or the default datafile from the default tablespace.

This command only drops the logical file, so after committing this command, user need to drop the physical datafiles and remove the information in the dmconfig.ini manually.

tablespace_name ......Name of the datafile's tablespace

file_name ....Name of the datafile to be dropped
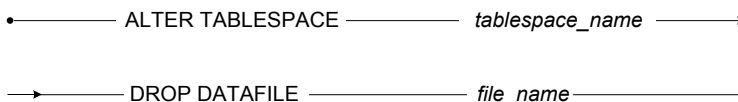
Figure 3-27 ALTER TABLESPACE DROP DATAFILE syntax

➲ **Example**

A user wants to drop datafile **tsfile1** from tablespace **ts_new**.

```
dmSQL> ALTER TABLESPACE ts_new DROP DATAFILE tsfile1;
```

3.26   ALTER TRIGGER ENABLE

The ALTER TRIGGER ENABLE command enables or disables an existing trigger on a table. Only the table owner, a DBA, a SYSDBA or a SYSADM can execute the ALTER TRIGGER ENABLE command.

A *trigger* is a database server mechanism that automatically executes predefined commands in response to specific events. This allows a database to perform complex or unconventional operations that might not be possible using standard SQL commands. Since triggers are under the control of the database server, they can ensure data consistency, regardless of the source. DBMaker will transparently fire the trigger every time a user or application program generates a trigger event.

A trigger automatically enables when created. To suspend a trigger when testing database operations that may cause the trigger to fire, use the DISABLE keyword. Disabling a trigger does not remove it from the database and you can enable it again with the ENABLE keyword.

trigger_name .... Name of the trigger to enable or disable

table_name .... Name of the table associated with the trigger
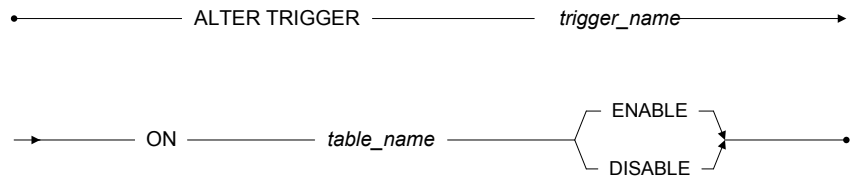
Figure 3-28 ALTER TRIGGER ENABLE syntax

➲ **Example 1**

The following disables the trigger **Trig_emp** on the **Employeesinfo** table.
```
dmSQL> ALTER TRIGGER Trig_emp ON Employeesinfo DISABLE;
```

➲ **Example 2**

The following enables the trigger **Trig_emp** on the **Employeesinfo** table.
```
dmSQL> ALTER TRIGGER Trig_emp ON Employeesinfo ENABLE;
```

3.27 ALTER TRIGGER REPLACE

The ALTER TRIGGER REPLACE command replaces a trigger. Only the table owner, a DBA, a SYSDBA or a SYSADM can execute the ALTER TRIGGER REPLACE command.

A *trigger* is a database server mechanism that automatically executes predefined commands in response to specific events. This allows a database to perform complex or unconventional operations that might not be possible using standard SQL commands. Since triggers are under the control of the database server, they can ensure data consistency, regardless of the source. DBMaker will transparently fire the trigger every time a user or application program generates a trigger event.

Specify the name of the trigger when altering or replacing it. Also specify the new trigger action, action time, event, table, and type.

NOTE *The ALTER TRIGGER REPLACE command, only functions on the original trigger table.*

Unlike most database objects, DBMaker does not identify triggers using fully qualified names, but associates them with tables instead. For this reason all trigger names on the same table must be unique. The trigger action operates with the same security and object privileges as the owner of the trigger table, not with the privileges of the user executing the trigger event.

The BEFORE and AFTER keywords specify when the database server should perform the trigger action relative to the trigger event and the trigger action time. The BEFORE keyword instructs the database server to perform the trigger action before the trigger event. The AFTER keyword instructs the database server to perform the trigger action after the trigger event.

The INSERT, DELETE, and UPDATE keywords specify the event that fires a trigger. There are some differences in the use of the INSERT and DELETE keywords, and the UPDATE keyword. The INSERT keyword instructs a trigger to fire whenever a row is inserted into a table. The DELETE keyword instructs a trigger to fire whenever deleting a row from a table. The UPDATE keyword specifies a trigger to fire

after updating any column in a table. Also, use UPDATE OF to specify a column list to fire a trigger after updating specific columns.

NOTE *A unique column name can only be used in one UPDATE trigger in a table.*

The ON keyword specifies the name of the table to replace the trigger with on the trigger table. The trigger table must be a permanent table in the database. A trigger cannot be created on a temporary table, view, or synonym.

trigger_name ....Name of the trigger to replace

column_name...........Name of the column to create the new trigger on

table_name...............Name of the table to create the new trigger on

sql_statement............Statement to execute when the trigger fires
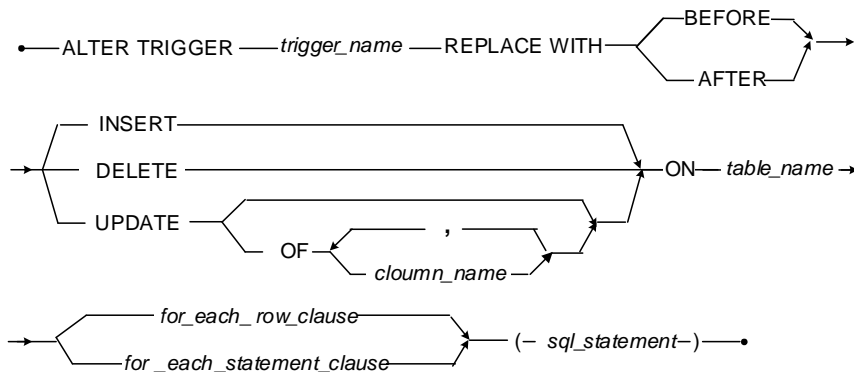
Figure 3-29 ALTER TRIGGER REPLACE syntax

For Each Row Clause

The REFERENCING keyword specifies an alias for the OLD and NEW keywords. When replacing a row trigger, indicate in the trigger action whether referencing a value of a column, before or after the trigger fires. Use the REFERENCING keyword in place of the OLD and NEW keywords when tables named OLD and NEW already exist.

The FOR EACH ROW keywords instructs a trigger to fire once for each row the trigger event modifies. Triggers defined using the FOR EACH ROW keyword do not fire if the statement firing the trigger does not process rows.

The WHEN keyword specifies rows, which satisfy the search condition, to fire a trigger. The WHEN clause is evaluated for each row the trigger event modifies. If the search condition is true, the trigger fires for that row. If the search condition is false, the trigger does not fire. The result of the WHEN condition only affects the execution of the triggered action, it has no effect on the statement that fires the trigger.

old_name................Alias for referencing the values as they existed in the trigger table before the trigger action fires

new_name ....Alias for referencing the values as they exist in the trigger table after the trigger action fires

search_condition.......Conditions a row must meet for a trigger to fire
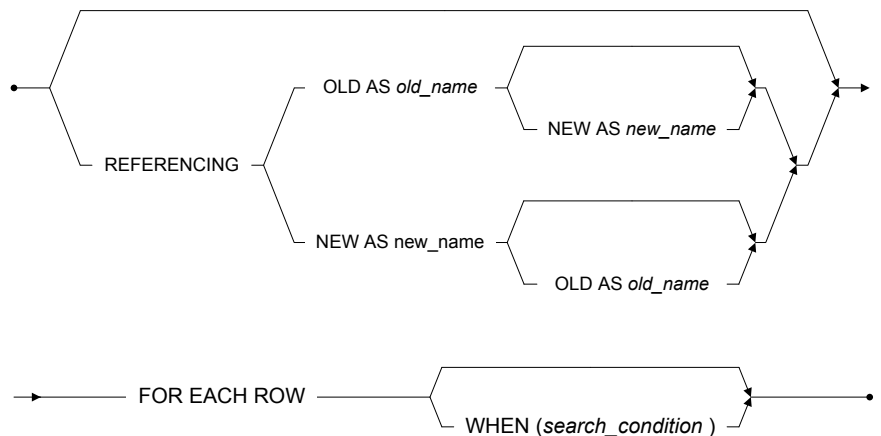
Figure 3-30 For Each Row Clause syntax

For Each Statement Clause

The FOR EACH STATEMENT *keywords* specify a trigger fire only once for each statement that fires the trigger. Triggers defined using the FOR EACH STATEMENT *keywords* fire even if the statement that fires the trigger does not process any rows.

The statement that the trigger executes when it fires is known as the *trigger action*. The trigger action may be an INSERT, UPDATE, DELETE, or EXECUTE PROCEDURE statement. Only built-in functions that have no argument PI( ), NOW( ), or USER( ) can be used when specifying the trigger action. Stored procedures executed by a trigger cannot contain any COMMIT, ROLLBACK, or SAVEPOINT transaction control statements.

Create multiple triggers for each trigger event on the trigger table by using the trigger action time; BEFORE and AFTER *keywords*, in combination with the trigger type; FOR EACH ROW and FOR EACH STATEMENT *keywords*. For example, you can combine the trigger action time and the trigger type to create four triggers for the INSERT trigger event: BEFORE/FOR EACH STATEMENT, BEFORE/FOR EACH ROW, AFTER/FOR EACH ROW, and AFTER/FOR EACH STATEMENT.

NOTE *Also supported by the UPDATE and DELETE trigger events.*

When using UPDATE OF instead of UPDATE, one trigger for each column in the table for each *trigger action time/trigger type combination* can be created. A table with four columns can have four UPDATE OF triggers for each: BEFORE/FOR EACH STATEMENT, BEFORE/FOR EACH ROW, AFTER/FOR EACH ROW, and AFTER/FOR EACH STATEMENT combination. When using UPDATE OF to specify a trigger, UPDATE cannot be used to create a trigger on that table. When you replace a trigger with a new one, no column already used in another UPDATE OF trigger may be specified.

FOR EACH STATEMENT

Figure 3-31 For Each Statement Clause syntax

➲ **Example 1**

Originally defined as a **FOR EACH ROW** trigger, this command will replace it with a **FOR EACH STATEMENT** trigger by altering the **Trig_emp** trigger on the **Employeesinfo** table.

```
dmSQL> ALTER TRIGGER Trig_emp REPLACE WITH
                BEFORE UPDATE ON Employeesinfo
                FOR EACH ROW
        (INSERT INTO NameChange VALUES (OLD.FName, OLD.LName,
                                        NEW.FName, NEW.LName));
```

➲ **Example 2**

This command will replace the **UPDATE** trigger event with an **INSERT** trigger event by altering the **Trig_emp** trigger on the **Employeesinfo** table from example 1.

```
dmSQL> ALTER TRIGGER Trig_emp REPLACE WITH
                AFTER INSERT ON Employeesinfo
                FOR EACH ROW
        (INSERT INTO NameChange VALUES (OLD.FName, OLD.LName,
                                        NEW.FName, NEW.LName));
```

➲ **Example 3**

This command will replace the **INSERT** statement with an **EXECUTE PROCEDURE** statement by altering the **Trig_emp** trigger on the **Employeesinfo** table from example 2.

```
dmSQL> ALTER TRIGGER Trig_emp REPLACE WITH
                AFTER INSERT ON Employeesinfo
                FOR EACH ROW
        (EXECUTE PROCEDURE LogTime);
```

3.28   BEGIN BACKUP

The BEGIN BACKUP command places a database in a special state that allows backing up of all files without requiring other users to disconnect or shut down the database. Only users with DBA, SYSDBA or SYSADM security privileges can execute the BEGIN BACKUP command.

Media failure is the failure of the online secondary or auxiliary storage of a computer system. The most common secondary and auxiliary storage devices are hard disks. Media failures are usually caused by physical trauma to the disk itself: head crash, fire, earthquake, exposure to high vibration, or g-forces outside its physical operating limits.

When a media failure occurs, one or more files can be physically damaged. Provide archiving or backup to successfully restore a database. Create backups of database files periodically, to restore the database in the event of a media failure. There are several different types of backups.

An online backup is can be 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. A DBMS must provide the capability to back up a database online.

An offline backup is 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 before the shut down. Offline backups can be inconvenient for users, since they must remember to complete all active transactions and disconnect from the database. A DBMS does not need to provide the capability to back up a database offline.

A full backup creates a copy of all data and Journal files, providing a copy of the entire database system at one point in time. Full backups archive the entire database and require a large amount of storage space, but can restore the database quickly.

A differential backup is based on the latest full backup of the data. This is known as the base of the differential, or the differential base. A differential backup contains only

the data that has changed since the differential base was created. A differential base is typically used for several successive differential backups. During a restore operation, the full backup and its corresponding differential backup combine to produce a fully restored database.

An incremental backup creates a copy of only the Journal files that have changed since the last full backup. These files provide a copy of the changes made to the database since the last full backup. Incremental backups archive only Journal files and require only a small amount of storage space, but need more time to restore the database.

DBMaker supports five types of backups: offline full backups, online full backups, online differential backups, online incremental backups and online incremental to current backups. Before performing an incremental backup, perform either an offline full backup or an online full backup. If full backup is not performed first, you may be unable to restore the database in the event of a media failure.

To perform an offline full backup, make sure all users are disconnected and shut down the database. If an error occurs while the database is shutting down, completing the backup operation or restoring the database may be impossible. Backup all Data, BLOB, and Journal files. Using an offline full backup can restore a database up to the point in time of shutting down.

To perform an online full backup, start the database in NON-BACKUP, BACKUP-DATA, or BACKUP-DATA-AND-BLOB mode. To begin the backup, issue the BEGIN BACKUP command. Back up all Data and BLOB files. After these files have been backed up, issue the END BACKUP DATAFILE command. Then back up all Journal files. Next, issue the END BACKUP JOURNAL command to complete the backup and return the database to normal operation. Using an online full backup can restore a database from, the point in time the END BACKUP DATAFILE command was executed to and the point in time the currently active Journal file was copied.

To perform a differential backup, start the database in NON-BACKUP, BACKUP-DATA, or BACKUP-DATA-AND-BLOB mode. A full backup must first exist before a differential backup is created. A differential base must exist before a differential backup is created. DBMaker's differential backup only includes data files (e.g., *.DB

and *.BB), not journal files. This is because journal files changed too frequently. So, when doing a differential backup, only useful journal blocks are copied.

To perform an online incremental backup or an online incremental backup to current, start the database in either BACKUP-DATA or BACKUP-DATA-AND-BLOB mode.

Only users that have read permissions on the database files from the operating system can perform an offline full backup, and only users with DBA, SYSDBA or SYSADM security privileges can perform online backups. In addition, only one user at a time can perform an online backup.

Abort an online backup at any time by issuing the ABORT BACKUP command. After this command executes, you will not be able to use the files from this backup to restore the database.

Perform an online full backup and an online differential backup at any time with the database in any backup mode, including NON-BACKUP mode. Incremental online backups may only be performed when the database is running in BACKUP-DATA or BACKUP-DATA-AND-BLOB mode.

The backup mode indicates the type of information DBMaker backs up during an online incremental backup. Change the backup mode online or offline, using one of three different methods: offline with the **DB_BMode** keyword in the **dmconfig.ini** configuration file, online with the SQL SET command at the dmSQL command prompt, or online with the Server Manager utility provided with DBMaker.

NON-BACKUP mode provides no protection for data inserted or updated since the last full backup. In this mode, a database cannot perform online incremental backups. A database can use the Journal to fully recover from an 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 can only be restored to the point in time of the last full backup.

To set the backup mode to NON-BACKUP using the **DB_BMode** keyword, open the **dmconfig.ini** file using any text editor and change the value of **DB_BMode** to 0. You may use the SET BACKUP OFF command during an online full backup to set the backup mode to NON-BACKUP. This command must be executed after the

BEGIN BACKUP command, but before the END BACKUP JOURNAL command, and only during an online full backup.

BACKUP-DATA mode provides protection for data, excluding BLOB data that was added or changed since the last full backup. In this mode, DBMaker can perform an online incremental backup, but since changes to BLOB data are not recorded in the Journal, they are not stored in the backup Journal files. Any records containing BLOB data added or changed since the last full backup will have the BLOB data replaced with a NULL value. After restoring the database, manually update all records with the new BLOB data. A database can use the Journal to fully recover from an instance failure and partially recover from media failure.

To set the backup mode to BACKUP-DATA using the **DB_BMode** keyword, open the **dmconfig.ini** file using any text editor and change the value of **DB_BMode** to 1. Use the SET DATA BACKUP ON command during an online full backup to set the backup mode to BACKUP-DATA. This command must be executed after the BEGIN BACKUP command, before the END BACKUP JOURNAL command, and during an online full backup.

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, DBMaker can perform an online incremental backup, and all data will be stored in the backup Journal files. A database can use the Journal to fully recover from an instance failure, and can fully recover from a disk failure. Use the last backup 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.

To set the backup mode to BACKUP-DATA-AND-BLOB using the **DB_BMode** keyword, open the **dmconfig.ini** file using a text editor and change the value of **DB_BMode** to 2. Use the SET BLOB BACKUP ON command during an online full backup to set the backup mode to BACKUP-DATA-AND-BLOB. This command must be executed after the BEGIN BACKUP command, before the END BACKUP JOURNAL command, and only during an online full backup.
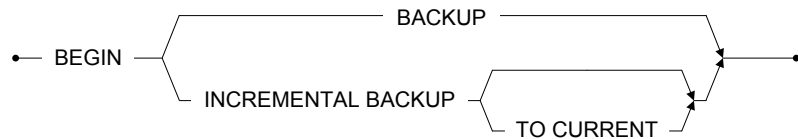
Figure 3-32 BEGIN BACKUP syntax

➲ **Example**

The following shows the steps involved in a full online backup. To begin, issue the **BEGIN BACKUP** command to notify DBMaker that a full backup is in progress. Then, copy all data and BLOB files to the backup location using operating system commands. Next, issue the **END BACKUP DATAFILE** command. Then, use operating commands to copy all Journal files to the backup location. Finally, issue the **END BACKUP JOURNAL** command. On completion, this command returns the database to normal operation.

```
BEGIN BACKUP
    Copy data and BLOB files to backup location using OS commands
    Change backup mode if desired
    Abort the backup if desired
END BACKUP DATAFILE
    Copy Journal files to backup location using OS commands
    Change the backup mode if desired
    Abort the backup if desired
END BACKUP JOURNAL
```

3.29 BEGIN WORK

The BEGIN WORK command is an optional command used in a script file to document the beginning of a transaction; DBMaker ignores this command.

————————————————— BEGIN WORK —————————————————

Figure 3-33 BEGIN WORK syntax

➲ **Example**

The following illustrates how the **BEGIN WORK** command can be used in a script file to document the beginning of a transaction; the text may be located anywhere within the script file.

```
BEGIN WORK
    ...
    SQL Command
    SQL Command
    ...
COMMIT WORK
```

3.30 CHECK

The CHECK command checks the database objects specified for data consistency. You may want to check database consistency if queries are returning inconsistent or erroneous results, or receiving frequent or unusual error messages. Only the owner of the object, a DBA, a SYSDBA or SYSADM may execute the CHECK command.

DBMaker checks the consistency of a database, indexes, tables, files, tablespaces, and the system catalog. Checking the consistency of database objects can be time and resource consuming. Use the CHECK command only when necessary, and try to schedule its use for off-peak times when inconveniences to users are minimized.

When checking a database object, DBMaker first checks the system catalog tables to ensure all catalog information is valid and correct. If any errors are found in the system catalogs, checking stops immediately. If the system catalog has errors, the database may have serious consistency errors. Then DBMaker checks the physical structure and data integrity of the object and any related objects. When checking an object, DBMaker also checks, all objects contained in or related to the original object. Also checks the indexes, data pages, files, and tables.

Some types of errors can be repaired. Dropping the index and rebuilding it can usually correct most problems. It is also possible to correct a corrupted table by unloading all records in the table, dropping the table, then recreating the table, and reloading all data.

If a database does have consistency errors, immediately back up the database, including all data and Journal files. DBMaker can fix some types of consistency errors after recovering from a crash. To engage DBMaker crash recovery routines, shut down and restart the database. After the database restarts, execute the CHECK command again to see if the error has been corrected.

If any inconsistency still exists, contact the CASEMaker customer service. CASEMaker customer support representatives will assist you with repairing the database.

NOTE   *For information on how to contact a CASEMaker customer service*
            representative in your area, see your license agreement.

tablespace_name ....Name of the tablespace to check

file_name ....Name of the file to check

table_name...............Name of the table to check

index_name..............Name of the index to check
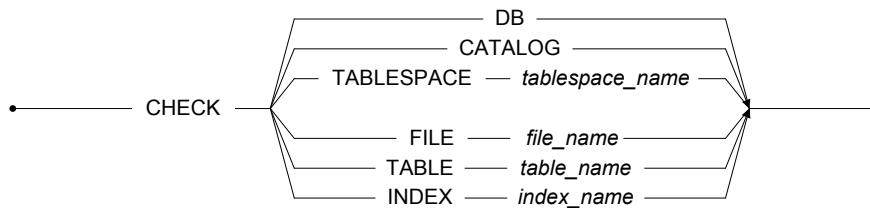
Figure 3-34 CHECK syntax

➲ **Example 1**

The following command checks the consistency of data in the **Customers** table.

```
dmSQL> CHECK TABLE Customers;
```

➲ **Example 2**

The following command checks the consistency of data in index **idxCustNum** of the
Customers table; when specifying an index name, specify the table name.

```
dmSQL> CHECK INDEX Customers.idxCustNum;
```

➲ **Example 3**

The following command checks the consistency of Data pages or frames in a BLOB
file in the **customer_data** file.

```
dmSQL> CHECK FILE customer_data;
```

 Example 4

The following command checks the consistency of database objects in the specified tablespace and may include files, tables, data pages, and data in all tables in the **ts_new** tablespace.

```
dmSQL> CHECK TABLESPACE ts_new;
```

 Example 5

The following command checks the consistency of the database system catalogs.

```
dmSQL> CHECK CATALOG;
```

 Example 6

The following command checks the consistency of all database objects.

```
dmSQL> CHECK DB;
```

3.31 CHECKPOINT

The CHECKPOINT command forces DBMaker to take a checkpoint. Take a checkpoint if database activity is very high or you infrequently back up or restart the database. Only users with DBA, SYSDBA or SYSADM security privileges can execute the CHECKPOINT command.

A *checkpoint* event brings the database to a clean state. DBMaker writes all Journal *records* and all *dirty data pages* in 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 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, redone, and ignored.

DBMaker automatically takes a checkpoint when a database starts or terminates when performing an *online backup*, or when the Journal is full. This 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 an automatic checkpoint occurs must wait until the checkpoint operation completes. DBMaker will also abort the current transaction if the Journal is full and issuing a checkpoint cannot reclaim enough Journal space to complete the transaction. In this situation, redo all commands in the aborted transaction.

To avoid any unnecessary delays in transaction processing, periodically take manual checkpoints using the CHECKPOINT command. Periodic manual checkpoints reduce the amount of time required to start, terminate, and back up a database, time transactions wait for checkpoint operations to complete, and the possibility of a full Journal. The optimal time interval between manual checkpoints depends on the activity frequency in the database.

———•—————————————— CHECKPOINT ————————————————•———

Figure 3-35 CHECKPOINT syntax

➲ **Example**

The following example forces the system to take a checkpoint.

```
CHECKPOINT
```

3.32 CLOSE DATABASE LINK

The CLOSE DATABASE LINK command closes links to a remote database. Use this command to close a single link, or multiple links at the same time. Any user with an active link to a remote database can execute the CLOSE DATABASE LINK command.

A database link creates a connection to a remote database, providing access to remote data from the local database. Links provide additional security information. Links enable a user to connect to a remote database with a different user name. Alternately, use the public link to connect to a remote database without an account.

When executing the CLOSE DATABASE LINK command and specifying a link name, DBMaker closes the link to the remote database if it no active transactions exist. When executing the CLOSE DATABASE LINK command and specifying a remote database, DBMaker closes all links that connect to the remote database. If a link has an active transaction, it remains open and DBMaker returns an error. Wait until the transaction has finished and retry closing the link.

The NONACTIVE keyword closes all links to a remote database that are not being used by an active transaction. If a transaction is using a link when you execute the CLOSE DATABASE LINK command using the NONACTIVE keyword, the link remains open. To close this link, wait until the transaction is finished and try closing it again.

The ALL keyword closes all links to a remote database. If a transaction is using a link when you execute the CLOSE DATABASE LINK command using the ALL keyword, the link remains open and DBMaker returns an error. To close this link, wait until the transaction is finished.

link_name................Name of the link to a remote database to close

remote_database_name…close all links to the remote database *remote_database_name*

```
                                                    link_name
                                              remote_database_name
  •——————— CLOSE DATABASE LINK ——————        NONACTIVE         ——————•
                                                   ALL
```
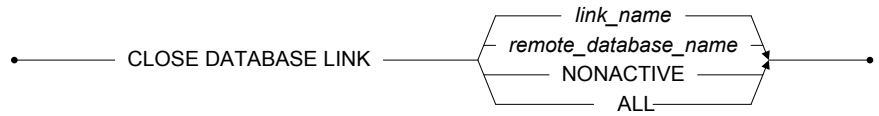
Figure 3-36 CLOSE DATABASE LINK syntax

➲ **Example 1**

The following closes the **FieldLink**.
```
dmSQL> CLOSE DATABASE LINK FieldLink;
```

➲ **Example 2**

The following closes all links to the remote database identified in the local
dmconfig.ini file as **FieldOffice**.
```
dmSQL> CLOSE DATABASE LINK FieldOffice;
```

➲ **Example 3**

The following closes all links to not being used by an active transaction.
```
dmSQL> CLOSE DATABASE LINK NONACTIVE;
```

➲ **Example 4**

The following closes all links unless a link is being used by an active transaction,
DBMaker will return an error and the link will remain open.
```
dmSQL> CLOSE DATABASE LINK ALL;
```

3.33   COMMIT WORK

The COMMIT WORK command commits the current transaction. DBMaker automatically starts a new transaction after execution of the COMMIT WORK command. Any user with CONNECT or higher security privileges can execute the COMMIT WORK command.

A transaction, traditionally defined as a logical unit of work, or one or more operations on a database that need to complete together in order to leave the database in a consistent state. Transactions are self-contained and must either complete successfully, change the data, or fail and leave the data unchanged.

For example, suppose you store two different kinds of information in the database records of shipments sent to customers and records of items currently in stock, including quantity of items. When an item ships to a customer, the item and the quantity shipped are added to the shipment list. The quantity shipped must also be subtracted from the items currently in stock. If both of these operations are not completed together as a logical unit of work, the database will be in an inconsistent state. The quantity of items in stock will be too high; items shipped and not subtracted from items in stock, or too low; items subtracted from items in stock and not shipped. Both of these operations together make up a single transaction, and must complete successfully or both will fail.

If a transaction completes successfully and changes the data, it has been committed. If a transaction fails and leaves the data unchanged, it has been rolled back.

When executing the COMMIT WORK command, DBMaker will write all changes made by commands in the current transaction to the database. The COMMIT WORK command only writes changes for the current transaction. The COMMIT WORK command is not required if the connection to a database is running in AUTOCOMMIT mode.

AUTOCOMMIT mode controls when DBMaker will commit a transaction. When AUTOCOMMIT mode is on, each command is treated as a separate transaction. Pressing the Enter key to execute a command automatically commits the command if

it completes successfully, or rolls it back if an error occurs during execution. When AUTOCOMMIT mode is OFF, all commands between successive COMMIT WORK commands form a single transaction. Executing the COMMIT WORK command commits any changes made in the transaction, and executing the ROLLBACK WORK command rolls back all changes.

In the event of a database crash, DBMaker will automatically roll back any transactions that have not been committed. If the changes made in the rolled back transactions reflected in the database, redo all commands in these transactions when the database restarts.
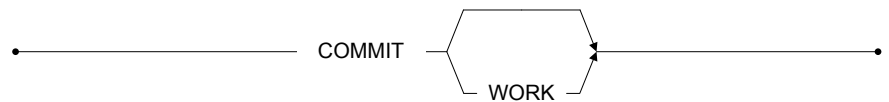
Figure 3-37 COMMIT WORK syntax

➲ **Example**

The following example commits the changes made by all commands executed between the first and second **COMMIT WORK** commands with **AUTOCOMMIT** mode turned off.

```
COMMIT WORK
    ...
    SQL Command
    SQL Command
    ...
COMMIT WORK
```

3.34 CREATE COMMAND

The CREATE COMMAND creates a new stored command. Use stored commands to quickly and conveniently execute frequently used SQL data-manipulation statements. To execute the CREATE COMMAND, only users with the RESOURCE or higher security privileges, and all security and object privileges necessary to execute the SQL statement may use this command.

A stored command is a compiled SQL data-manipulation statement permanently stored in the database in executable format. Repeatedly execute the stored command without waiting for DBMaker to compile and optimize the command. Stored commands are similar to stored procedures except; they can only contain a single command and cannot contain program logic.

When creating a stored command, specify the command name and a valid SQL data-manipulation statement of SELECT, INSERT, UPDATE, or DELETE. Use host variables as placeholders for column values in the SQL statement. This permits assigning actual values to the column when executing he command. To use host variables in a stored command, replace any data or column value with a question mark (?).

When executing a stored command that has host variables, use result constants from built-in functions, the NULL keyword, the DEFAULT keyword, or another host variable. Only use built-in functions that have no argument, such as RAND( ), PI( ), CURDATE( ), and NOW( ), when providing a value for a host variable. To use NULL value for the host variable, the value represented by the host variable must be capable of accepting the NULL values. The number of parameters provided when executing a stored command must equal the number of host variables in the command definition.

When dropping a table or a column that is referenced by a stored command or altering a table and modify the column definition using the BEFORE and AFTER keywords, the stored command becomes invalid and cannot be used again. Altering a table and adding a column without using the BEFORE and AFTER keywords has no

impact on a stored command. Drop an invalid stored command to remove it from the database.

Stored command names must be unique in the database. Stored command names have a maximum length of 128 characters and may contain numbers, letters, underscore characters and $ and # symbols. The first character may not be a number.

OR REPLACE: specify OR REPLACE to re-create the stored command that already exists, that is to say, you can use this clause to change the definition of an existing stored command.

command_name.......Name of the new stored command to create

select_statement ....A valid SELECT statement

insert_statement ....A valid INSERT statement

update_statement......A valid UPDATE statement

delete_statement ....A valid DELETE statement
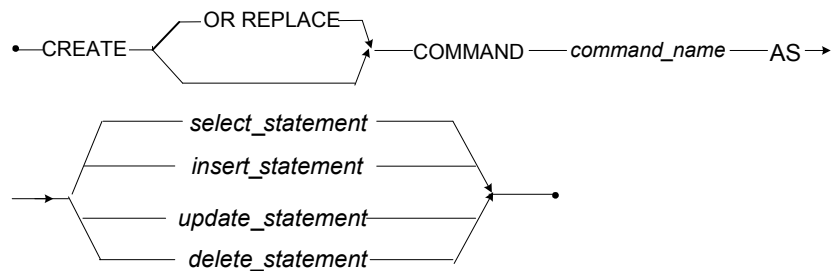
Figure 3-38 CREATE COMMAND syntax

➲ **Example 1**

The following creates a stored command named **sc_select** and selects all **employees** in the **Employeesinfo** table whose last name begins with the letter '**A**'.

```
dmSQL> CREATE COMMAND sc_select AS SELECT * FROM Employeesinfo WHERE LastName
LIKE 'A%';
```

➲ **Example 2**

To create a stored command named **sc_update** that uses host variables to update the **Manager** column in the **Employeesinfo** table, you can use the following syntax:

```
dmSQL> CREATE COMMAND sc_update AS UPDATE Employeesinfo SET Manager = ? WHERE
Manager = ?;
```

or

```
dmSQL> CREATE COMMAND OR REPLACE sc update AS UPDATE Employeesinfo SET Manager
= ? WHERE Manager = ?;
```

　　　　　　　　　©Copyright 1995-2017 CASEMaker Inc.

3.35   CREATE DATABASE LINK

The CREATE DATABASE LINK command creates a new public or private link to a remote database. Database links permits a user to access objects in remote databases the same way as objects a local database. Only a DBA, a SYSDBA or a SYSADM may execute the CREATE DATABASE LINK command to create a public link to a database. Only users with CONNECT or higher security privileges may execute the CREATE DATABASE LINK command to create a private link to a database.

A database link creates a connection to a remote database, providing access to remote data a local database. Although you can directly identify remote databases, links provide additional benefits since they also contain security information. This permits connecting to a remote database with a different user name or an account using a public link.

Provide the link name and the remote database name when creating a database link. The **dmconfig.ini** file for both the local and remote database must contain a database configuration section for the opposite database. This database configuration section must contain the IP address and the port number of the opposite database server. Enter the IP address using the **DB_SvAdr** keyword and the port number using the **DB_PtNum** keyword.

The PUBLIC/PRIVATE keywords are optional. These keywords specify the type of database link to create, public or private. Public links are available to all users in a database. Private links are available only to the user that creates them. Only a DBA, a SYSDBA or a SYSADM can create a public database link, while any user can create a private database link. If both a public and private link exists with the same name, DBMaker uses the private link instead of the public link. DBMaker creates a private link by default.

The IDENTIFIED BY keywords are optional. This keyword specifies the user name and password to use when connecting to the remote database. The user name provided must be an existing user in the remote database with the CONNECT or higher security privileges. When the link is used to connect to the remote database, the operations a user can perform depend on the security and object privileges granted to.

If a user name is not specified when connecting to the remote database, DBMaker uses the current user name in the local database.

Link names have a maximum length of 128 characters, and may contain numbers, letters, the underscore character, and the symbols $ and #. The first character may not be a number.

link_name................Name of the link to create to a remote database

remote_db_name ......Name of the remote database to connect to

user_name................Name of a user in the remote database with CONNECT or higher security privileges

password..................Password of the user in the remote database
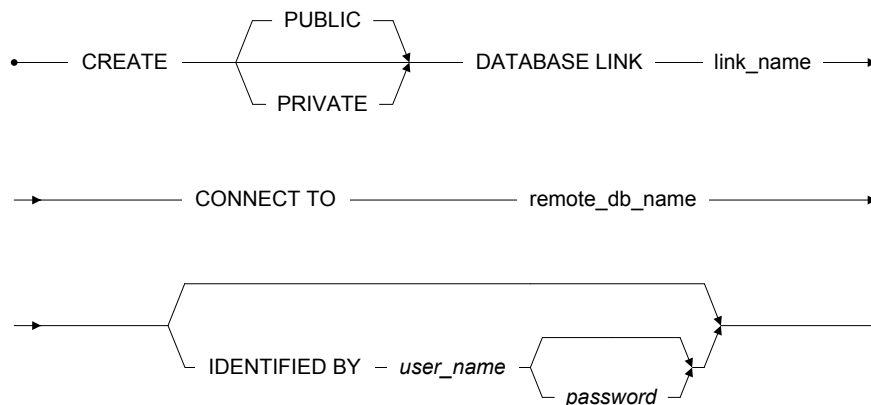
Figure 3-39 CREATE DATABASE LINK syntax

⮞ **Example 1**

The following example creates a public database link named **FieldLink** to the remote **FieldOffice** database. The user creating the link must have DBA, SYSDBA or

SYSADM security privileges in the local database and must have the same user name in both the local and remote databases. Using this link automatically connects the user to the remote database with the same user name as the link creator. It provides the security and object privileges granted to this user in the remote database.

```
dmSQL> CREATE PUBLIC Database LINK FieldLink CONNECT TO FieldOffice;
```

➲ **Example 2**

The following example creates a public database link named **FieldLink** to the remote **FieldOffice** database. The user creating the link must have DBA, SYSDBA or SYSADM security privileges in the local database. Using this link automatically connects the user to the remote database with the user name **LinkUser** and password **dil3ryx9**. It provides the security and object privileges granted to this user.

```
dmSQL> CREATE PUBLIC Database LINK FieldLink CONNECT TO FieldOffice
                        IDENTIFIED BY LinkUser dil3ryx9;
```

➲ **Example 3**

The following creates a private database link named **FieldLink** to the remote **FieldOffice** database. The user creating the link must the same user name in both the local and remote databases. Using this link automatically connects the user to the remote database with the same user name as the local database. It uses the security and object privileges granted to the user account in the remote database. If there is a public link with the same name, the private link is used instead.

```
dmSQL> CREATE PRIVATE Database LINK FieldLink CONNECT TO FieldOffice;
```

➲ **Example 4**

The following creates a private database link named **FieldLink** to the remote **FieldOffice** database. Using this link automatically connects a user to the remote database with the user name **Vivian** and password **a23456**. It provides the security and object privileges granted to this user. This is useful if you have a different user name in the local and remote databases. If there is a public link with the same name, the private link is used instead.

```
dmSQL> CREATE PRIVATE Database LINK FieldLink CONNECT TO FieldOffice
                        IDENTIFIED BY Vivian a23456;
```

3.36   CREATE DOMAIN

The CREATE DOMAIN command creates a new domain with an optional default value and optional integrity constraints. Any user with RESOURCE or higher security privileges can execute the CREATE DOMAIN command.

A domain is a user-defined data type that brings together a data type, a default value, and a value constraint. Use a domain in the column definition of CREATE TABLE or ALTER TABLE ADD COLUMN statements in place of a data type to define the set of valid values entered in the column.

For example, create a domain based on the DATE data type with a default value of NOW( ) that only accepts dates between January 1st, 1900 and today. Any column created using this domain will inherit these characteristics, allowing consistent definitions for columns that contain the same data type without specifying default values and value constraints each time.

When creating a domain, specify the data type and optionally specify a default value and a value constraint. Any data type may be used that DBMaker supports when creating a domain, except the SERIAL data type. Specifies default values and value constraints using the DEFAULT and CHECK keywords.

Domains can be created with the TEXT CONVERTER syntax in the CREATE DOMAIN clause. DBMaker uses the TEXT CONVERTER function to convert the CLOB, NCLOB, BLOB, and FILE data to pure text for creating text indexes and PURETEXT( ) UDF when the TEXT CONVERTER syntax on the domain is specified. The TEXT CONVERTER function-name should contain one argument of a BLOB related type. The return type must be CLOB or NCLOB data types or an error is returned. At most **32767** domains can be created using the TEXT CONVERTER syntax.

The DEFAULT keyword is optional. This keyword specifies a default value inserted into a column if no value is provided when inserting a new row. Constants, results from built-in functions, or the NULL keyword may be used as the default value. Only use built-in functions that have no argument like PI( ), NOW( ), or USER( ), when

creating a domain. If using the NULL keyword as the DEFAULT value, the column cannot be defined with the NOT NULL keyword.

The CHECK keyword is optional. This keyword is used to specify a range of acceptable values (constraints) that may be entered in a column. The expression that specifies the range of acceptable values may be any expression that evaluates to true or false. The VALUE keyword may be used in the expression in conjunction with the CHECK keyword to represent the value of the column. If an SQL statement does not satisfy the CHECK conditions, it will not be processed.

Specifying the default values and value constraints using domains gives the same results as specifying them in a standard column definition. However, default values provided in the column definition will override the default value of the domain and the column definition can add value constraints in addition to those of the domain.

Ensure the value constraints specified in a column definition do not conflict with the value constraints provided by the domain. DBMaker does not check for conflicting constraints when creating a column based on a domain. The conflicting constraints may prevent inserting or updating some or all of the data.

Domain names have a maximum length of 128characters, and may contain numbers, letters, the underscore character, and the symbols $ and #. The first character may not be a number.

NOTE   *Only functions that do not take an argument may be used when creating domains.*

domain_name .... Name of the domain that to create

data_type .... Data type to use for the domain

constant .... Constant value to be used if no value is inserted

function_name .... Built-in function to be used if no value if inserted

constraint_name .... Name of constraint to be applied to domain

boolean_expression .... Any expression that evaluates to true or false
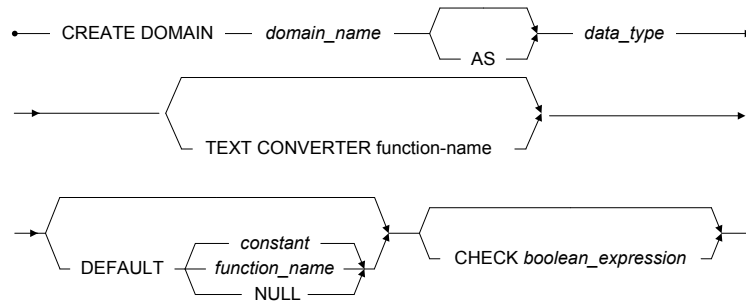
Figure 3-40 CREATE DOMAIN syntax

➲ **Example 1**

The following creates a domain named **AllNum** based on the **INTEGER** data type.

```
dmSQL> CREATE DOMAIN AllNum AS INTEGER;
```

➲ **Example 2**

The following creates a domain named **AllNum** based on the **INTEGER** data type that has a default value of *0*.

```
dmSQL> CREATE DOMAIN AllNum AS INTEGER DEFAULT 0;
```

➲ **Example 3**

The following creates a domain named **AllNum** based on the **INTEGER** data *type*, which does not allow **NULL** values.

```
dmSQL> CREATE DOMAIN AllNum AS INTEGER CHECK VALUE IS NOT NULL;
```

➲ **Example 4**

The following creates a domain named **PosNum** based on the **INTEGER** data type, which only allows values from 0 to 100, and has a default value of *0*.

```
dmSQL> CREATE DOMAIN PosNum AS INTEGER DEFAULT 0 CHECK VALUE >= 0 AND VALUE <=
100;
```

➲ **Example 5**

The following creates a domain named **ValidDate** based on the **DATE** data type, which uses the **NOW( )** function as both the default value and one of the value constraints.

```
dmSQL> CREATE DOMAIN ValidDate AS DATE
                     DEFAULT NOW()
                     CHECK VALUE > '01/01/1900' AND VALUE <= NOW();
```

3.37   CREATE GROUP

The CREATE GROUP command creates a new user group. Users in this group gain all object privileges granted to the group. Only users with SYSADM, SYSDBA or DBA security privileges can execute the CREATE GROUP command.

Groups simplify the management of object privileges in a database with a large number of users. Use a group to collect all users that require the same object privileges. Any object privileges granted for the group are automatically granted to all members in the group. After creating a new group, add users to the group using the ADD TO GROUP command.

DBMaker also provides support for nested groups. Add a group as a member in another group, provided there are no circular references from the group being added. For example, you cannot add group1 as a member of group2 if group2 is already a member of group1, and cannot add group 1 as a member of itself. Add a group, as a member in another group is the same as adding a user.

The group name cannot be SYSTEM, PUBLIC, or GROUP, or the same as any existing user or group names. Group names have a maximum length of 128 characters and may contain letters, numbers, underscore characters, and symbols $ and #. The first character may not be a number.

group_name ....Name of the new group to create

```
•————— CREATE GROUP ——— group_name ——————•
```

Figure 3-41 CREATE GROUP syntax

⮑ Example

The following creates a new group named **Manager**.

```
dmSQL> CREATE GROUP Manager;
```

3.38 CREATE HASH INDEX

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. Bucket n sets the array size for the hash table being created.

index_name .... Name of the new hash index to create

table_name .... Name of the memory table you are creating the index on

column_name .... Name of the column(s) created on the hash index

bucket n .... sets the array size
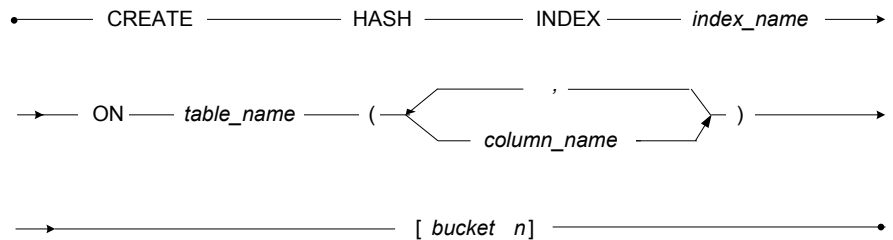
Figure 3-42 CREATE HASH INDEX syntax

⮕ **Example**

With the memory table created, a hash index **idx1**, can be made on memory table **tb_mem**, using columns **c01_int** and **c02_char** with an array size of **31**.

```
dmSQL> CREATE HASH INDEX idx1 ON tb_mem (c01_int, c02_char) BUCKET 31;
```

3.39 CREATE INDEX

The CREATE INDEX command creates a new index on an existing table. Use indexes to increase the performance of queries by quickly locating specific rows in a table without examining the entire table. Only the table owner, a DBA, or a user with the INDEX privilege may execute the CREATE INDEX command on a table.

An index is a mechanism that provides fast access to specific rows in a table based on the values of one or more columns from the table (known as the key). Indexes contain the same data as the key columns, but the data is structured and sorted to make retrieval much faster. Once an index is created on a table, its operation is transparent to users of the database. The DBMS uses the index to improve query performance whenever possible.

When creating an index specify the index name, the name of the table creating the index on and the name of the key columns in the table. Create an index on one or more columns, up to a maximum of 32 columns. Any column in a table can be used in an index. DBMaker limits indexes to a maximum record size of 4,000 bytes.

Creating indexes for frequently used expressions will improve query performance. For XML columns, create the index on XML UDF: **extract**() and **extractvalue**() to speed up xpath queries. Please note the primary differences between extract() and extractvalue(). **Extract**() allows multi-value, one value, or zero value results, however, asc/desc and unique index are not allowed. **Extractvalue**() only allows UDF results having one value or zero values. If the UDF result is multi-value, then the create index fails for the existing tuple and the insert data fails for the newly inserted tuple, however, asc/desc and unique index are allowed with **extractvalue**().

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.

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 can NOT allow the following statements:

- sub-query

- host variable

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

The following XPath rules help build useful indexes. The XPath:

- should not include a predicate

- should not include a function

- should include an absolute location path.

- may only allow 'child' axis

- should have a result nodeset containing only leaf nodes (simple type element node or attribute node)

- the qname must be identical for all element nodes

- the name of each attribute must be identical for all attribute nodes

- must be base on an attribute node or an element leaf node

- cannot be a complex non-leaf node or a comment node, for example, '/order/items/item/@product' or '/order/date'

- should not allow position '/order/items/item[1]/@product'

- may allow the function 'count(/order/items/item)'

- should not allow expressions

The UNIQUE keyword is optional. This keyword specifies whether an index is unique. In a unique index, no more than one row can have the same key value and cannot contain duplicate values. Each NULL value in an index is treated, as a unique value making it possible to have multiple rows with NULL values in a unique index. When creating an index on a non-empty table, DBMaker checks whether all existing keys are distinct. If duplicate keys exist, DBMaker returns an error message and does not create the index. Whenever you insert or update a record in a table that has a unique index, DBMaker checks to ensure there is no existing record that already has the same key values as the new or updated record. DBMaker does not create unique indexes by default. When creating a unique index, specify using the UNIQUE keyword.

The AUTO keyword is optional. This keyword specifies whether an index can be auto performed by auto index daemon. It's behavior is similar to non-unique index, but it can be automatically created or dropped by the auto index daemon. If the option AUTOCOMMIT is set ON, DBMaker only requires *Update(U)* lock when creating an auto index, which means DBMaker allows other users to query the table simultaneously.When creating an auto index, users need to specify the AUTO keyword.

The ASC/DESC keywords are optional. These keywords specify whether the sort order of the index is ascending or descending. You can specify the sort order on a column-by-column basis, so it is possible to have some index columns in ascending order while others are in descending order. The sort order of an index may affect the order of query output in some cases. If an index is in descending order, it is possible

the output will appear in descending order even though you did not specify this in the query. If have a specific sort order for a query, specify it using the ORDER BY clause. The default sort order for columns in an index is ASC.

The FILLFACTOR keyword is optional. This keyword specifies the percentage of an index page that can be filled. This allows the database to optimize the use of index pages by reserving space for updates for existing records. The number parameter can have a value from 1 to 100, which represents a fillfactor of 1% to 100%. For frequently updated tables, after indexing the table set a low fillfactor value (e.g., 50) to reserve free space for inserting new key values. If you plan to update the table infrequently, leave the fillfactor at the default value of 100.

When you load data into a table, DBMaker updates all indexes on that table each time a new record is inserted. For this reason, try to load all data before creating an index on a table. It is much more efficient to create an index after loading a large amount of data than to create an index before loading the data.

Index names must be unique for each table. Index names have a maximum length of 128 characters and may contain numbers, letters, underscore characters, and symbols $ and #. The first character may not be a number.

Indexes can also be created in tablespaces different from where their master tables reside.

index_name .... Name of the new index to create

table_name .... Name of the table you are creating the index on

column_name .... Name of the column(s) created on the index

expression .... Expression created on the index

number .... Value to use for the fillfactor

tablespace_name .... Name of the tablespace where the index is created
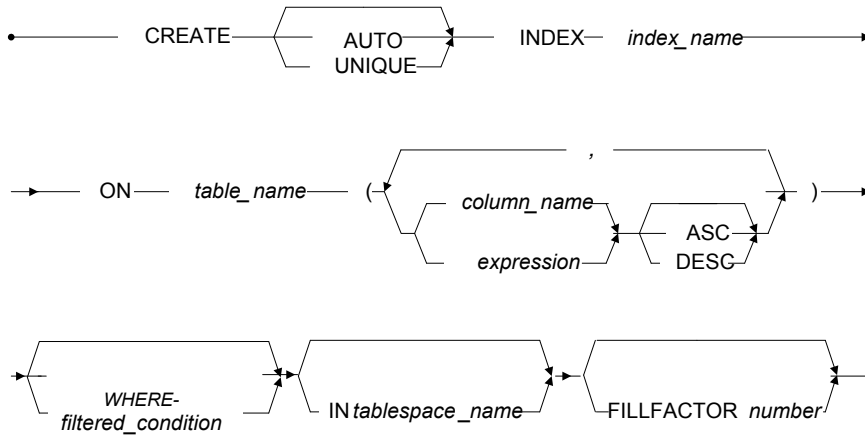
Figure 3-43 CREATE INDEX syntax

➲ **Example 1**

The following creates an index named **NameIndex** on the **FName** and **LName** columns of the **Employeesinfo** table; the index is not unique and may contain duplicate values.

```
dmSQL> CREATE INDEX NameIndex ON Employeesinfo (FName, LName);
```

➲ **Example 2**

The following creates an index named **NameIndex** on the **FName** and **LName** columns of the **Employeesinfo** table, both sorted in descending order.

```
dmSQL> CREATE INDEX NameIndex ON Employeesinfo (FName DESC, LName DESC);
```

➲ **Example 3**

The following example creates a unique index named **ClassIndex** on the **Course** and **Section** columns of the **Classes** table; index may not contain duplicate values.

```
dmSQL> CREATE UNIQUE INDEX ClassIndex ON Classes (Course, Section);
```

➲ **Example 4**

The following creates a unique index named **ClassIndex** on the **Course** and **Section** columns of the **Classes** table; the index may not contain duplicate values and has a fillfactor of **80**.

```
dmSQL> CREATE UNIQUE INDEX ClassIndex ON Classes (Course, Section) FILLFACTOR 80;
```

➲ **Example 5**

The following creates a unique index named **ExprIndex** on the **concat**(**Course, Section**) columns of the **Classes** table; the index may not contain duplicate values and has a fillfactor of **80**.

```
dmSQL> CREATE UNIQUE INDEX ExprIndex ON Classes (concat(Course,Section))
FILLFACTOR 80;
```

➲ **Example 6**

The following creates an auto index named **AUTO_1D_2** on the column **ID** and **NAME** of the table **tb_staff** (sorted in descending order).

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

➲ **Example 7**

The following creates an auto index named **AUTO_1DX_expr** on the expression **basepay+bonus** of the table **tb_salary** (sorted in descending order).

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

➲ **Example 8**

The following creates a filtered index named **FILIDX_ income** using the **where** clause for table **tb_salary**.

```
dmSQL> CREATE INDEX FILIDX_income ON tb_salary(basepay+bonus,tax)WHERE ID>30;
```

3.40 CREATE PROCEDURE

The CREATE PROCEDURE command generates a new stored procedure. Using stored procedures allows the database engine to bypass repeatedly compiling and optimizing SQL commands. This provides increased performance of frequently repeated tasks. Users with security privileges of RESOURCE level or higher and also having security and object privileges necessary to execute the SQL statement may use the CREATE PROCEDURE command.

A stored procedure is a compiled SQL data-manipulation statement permanently stored in a database in executable format. It is executed as a command in interactive SQL, or invoked from application programs, trigger actions or by other stored procedures.

When creating a stored procedure, specify the procedure's name and a valid SQL data-manipulation statement of SELECT, INSERT, UPDATE, or DELETE. Use host variables as placeholders for column values in the SQL statement. Later when executing the command actual values are assigned to the column. To use host variables in a stored command, replace any data or column value with a question mark (?).

FROM FILE

OR REPLACE: specify OR REPLACE 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.

File_-name...............File name of procedure to create
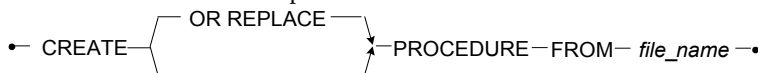
Figure 3-44 CREATE PROCEDURE FROM FILE syntax

➲ **Example: create or replace procedure**

```
dmSQL> CREATE PROCEDURE FROM 'file-name';
dmSQL> CREATE OR REPLACE PROCEDURE FROM 'file-name';
```

ESQL SP

OR REPLACE: specify OR REPLACE 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.

module_name.................. The module name of procedure to create

procedure_name .... Name of procedure to create

procedure_paramter......... Parameters of procedure to create

procedure_return_result… Rreturn a result set from the procedure to create

NOTE *Not support create or replace command syntax in execute procedure or set autocommit off, it will throw error while call sp to replace the command.*
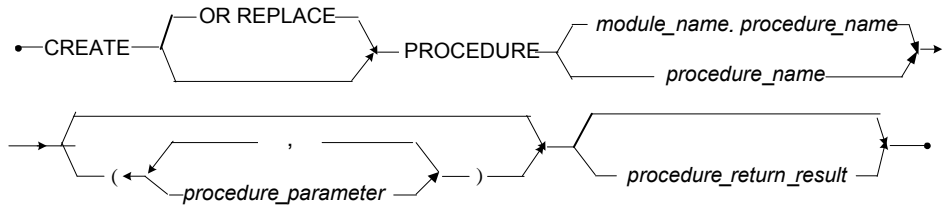
Figure 3-45 CREATE PROCEDURE syntax
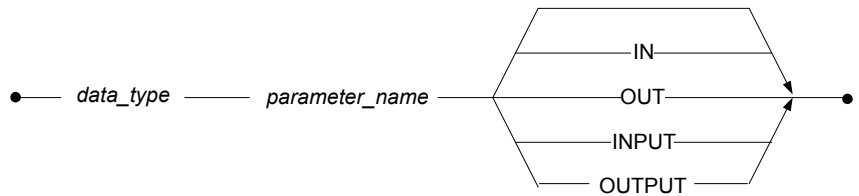
Figure 3-46 CREATE PROCEDURE: procedure_parameter syntax
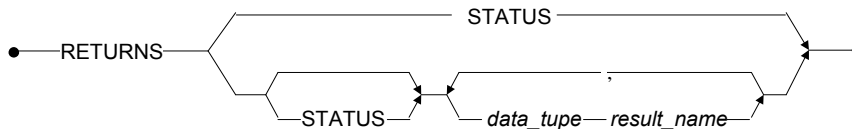
Figure 3-47 CREATE PROCEDURE: procedure_return_result syntax

➲ **Example: create or replace procedure.**

- CREATE ESQL SP FROM FILE:

```
dmSQL> CREATE OR REPLACE PROCEDURE FROM 'proc1.ec';
```

- Write the ec file:

```
EXEC SQL CREATE OR REPLACE PROCEDURE proc1 (char(10) i1, char(10) i2 output)
 returns char(10) o1, char(10) o2;
{
            EXEC SQL BEGIN CODE SECTION;
            EXEC SQL select FName from tb_staff where LName =:i1 into:i2;
            EXEC SQL returns select * from tb_staff into :o1,:o2;
            EXEC SQL END CODE SECTION;
}
```

JAVA SP

OR REPLACE: specify OR REPLACE 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.

module_name ....The module name of procedure to create

procedure_name ....Name of procedure to create

procedure_parameter ....Parameters of procedure to create

date_type ....Date type of return variable

variable_name ....Name of return variable

package.class.method ....The java method of procedure to create

argtype............................The argument type of java method

java_sourcecode_jar_file… The physical jar files of java source code

related_jar_file................Logical jar file
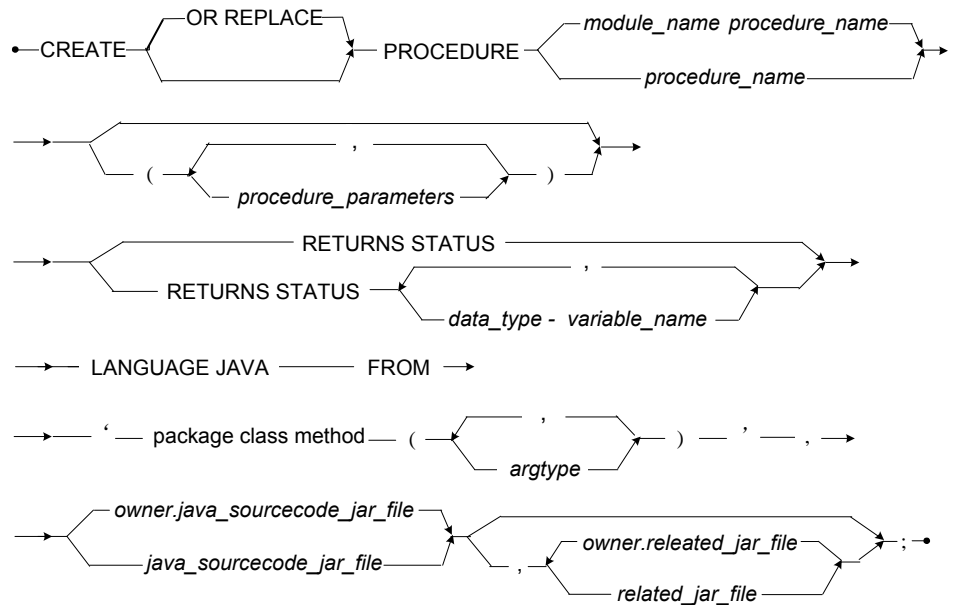
Figure 3-48 CREATE JavaSP syntax

➲ **Example: create or replace procedure.**

- Write a java file AddStaff.java

```
package staff;
import java.sql.*;


public class AddStaff
{
                // Add an row into the tb staff table
                public static void addStaff(String fName, String lName)
                throws Exception
                {
```

```
                                        // Register DBMaker JDBC Driver
                                        Class.forName("dbmaker.sql.JdbcOdbcDriver");
                                        // Connect to database
                                         Connection conn =
DriverManager.getConnection("jdbc:default:connection");

                                         // Prepare SQL statement
                                        PreparedStatement pstmt =
conn.prepareStatement("insert into tb staff values(?,?)");

                                        // Set values of the dynamic SQL argument
                                        pstmt.setString(1, fName);
                                        pstmt.setString(2, lName);

                                        // Execute the dynamic SQL statement
                                        pstmt.execute();

                                        // Close the dynamic SQL statement
                                        pstmt.close();

                                        // Close the connection
                                           conn.close();
            }
}
```

- Compile the AddStaff.java file in DOS command line, then it will create the AddStaff.class file in the current directory.

```
javac AddStaff.java
```

- Copy AddStaff.class to current_dir\staff.

- Zip the class, It will create the addStaff.jar file in the current directory.

```
jar cvf addStaff.jar    staff\AddStaff.class
```

- Make directory *jar\SYSADM* in <**DB_SpDir**>, and then move the addStaff.jar file into <**DB_SpDir**>*jar\SYSADM*.

- Add jarfile

```
dmSQL> ADD JARFILE addStaff addStaff.jar;
```

- Execute to create or replace the Java SP:

```
dmSQL> CREATE OR REPLACE PROCEDURE addStaff(char(12) fname,char(12) lname)
RETURNS STATUS LANGUAGE JAVA FROM
 'staff.AddStaff.addStaff(String,String)',addStaff;
```

SQL SP

OR REPLACE: specify OR REPLACE 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.

module_name .... The module name of procedure to create

procedure_name .... Name of procedure to create

procedure_parameter Parameters of procedure to create

date_type .... Date type of return variable

variable_name .... Name of return variable

sp_declare_main .... The main declare variable section of procedure to create

sp_statement_main ... The main statement section of procedure to create

NOTE   *Not support create or replace command syntax in execute procedure or set autocommit off, it will throw error while call sp to replace the command.*
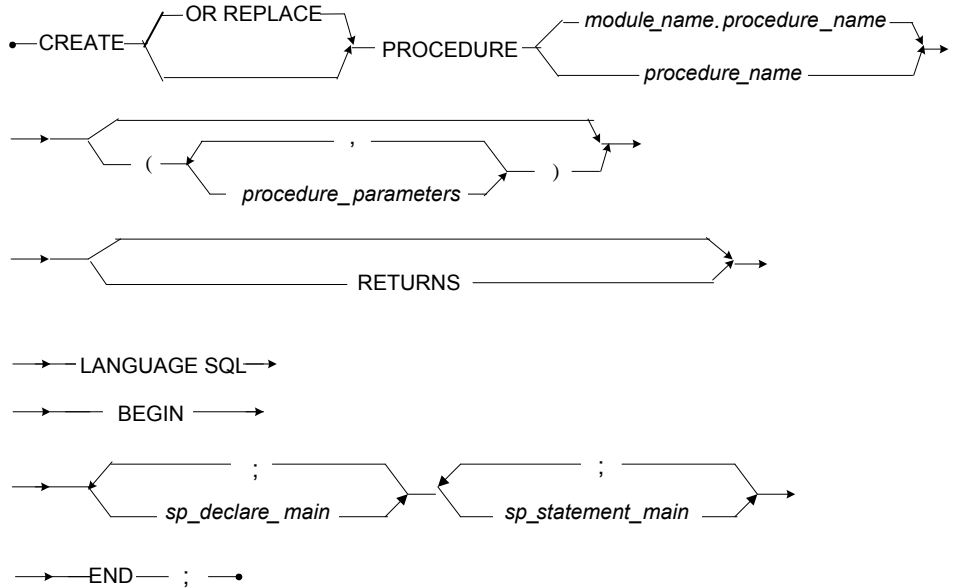
Figure 3-49 CREATE SQL SP syntax

⊃ **Example: create or replace procedure:**

- CREATE SQL SP FROM FILE:

```
dmSQL> CREATE OR REPLACE PROCEDURE FROM 'proc1.sp';
```

- Write the sp file:

```
dmSQL> CREATE OR REPLACE PROCEDURE proc1
LANGUAGE SQL
BEGIN
    DECLARE cur CURSOR WITH RETURN FOR select * from tb_staff;
    OPEN cur;
END;
```

3.41   CREATE REPLICATION

The CREATE REPLICATION command generates a new table replication for a table. Replications, synonyms, or views may not be created on a temporary table. Only the table owner, a DBA, a SYSDBA or a SYSADM can execute the CREATE REPLICATION command.

A table replication creates a full or partial copy of a table in a remote 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 efficiently, without having to go to another machine over a slower network connection. This is not the same as backing up the database to a remote location, since the synchronization is done on a transaction-by-transaction basis by the DBMS itself, without any intervention from users.

There are two primary types of table replication, synchronous and asynchronous. Synchronous table replication modifies the remote table at the same time it modifies the local table. Asynchronous table replication stores changes to the local table and modifies the remote table based on a schedule. Use the CREATE REPLICATION command to create synchronous and asynchronous table replications.

Synchronous table replication in DBMaker uses a global transaction model, in which the replication of data to the remote table is treated as an integral part of the local transaction. This means that if the replication of data to the remote database fails, the transaction on the local table will also fail.

Asynchronous table replication in DBMaker uses transaction logs to replicate data to the remote table. Modifications to the local table are stored in the transaction log, and are replicated to the remote table according to a predefined schedule. Using the transaction log enables DBMaker to treat the local transaction and the remote transaction independently, permitting updates to local tables normally even if the remote connection is not available. This allows asynchronous table replications to tolerate network and remote database failures, since the replication will keep trying until any failures are corrected.

When creating a table replication specify the replication name, the local table name, and the names of the remote destination tables. Both the local table and the remote tables must already exist in their respective databases. DBMaker will automatically drop any replications when dropping a table.

DBMaker will replicate the entire table unless using a column list. When replicating an entire table without a column list, the columns in the local table and corresponding columns in the remote table must have the same names and data types. Columns in the local table (from left to right) will replicate to the corresponding columns named in the column list for the remote table. Specify which columns in the local table correspond to columns in the remote table by providing a column list for both the local and remote tables. In all cases, include the primary key columns in the replication and the number and data types of primary key columns in both tables must match.

DBMaker does not identify replications using fully qualified names, but associates them with tables instead. All replication names on the same table must be unique. Synchronous table replications operate with the same security and object privileges as the creator, unless the remote table is specified using links. In this case, the replication operates with the same security and object privileges as the link. Asynchronous replications operate with the same security and object privileges as the user specified in the IDENTIFIED BY clause of the CREATE SCHEDULE command that is associated with the database containing the remote table.

The ASYNC keyword is optional. This keyword specifies that the replication being created is an asynchronous table replication. Before creating an asynchronous table replication, create a replication schedule for the remote database that contains the remote table. If this keyword is not used, DBMaker creates a synchronous table replication by default.

The optional keyword "WHERE" clause specifies the search condition used when replicating data to a remote table. DBMaker only replicates rows that satisfy the search condition. See the WHERE clause in the description of the SELECT command for more information.

The CLEAR DATA/FLUSH DATA/CLEAR AND FLUSH DATA keywords are optional. These keywords specify the operations that take place when creating a replication. The CLEAR DATA keywords delete all data from the remote table when generating the replication. The FLUSH DATA keywords copy all data that matches a search condition into the remote table. The CLEAR AND FLUSH DATA keywords clear all data from the remote table, and then copy all data that matches a search condition into the remote table.

The NO CASCADE keywords are optional. It takes action only when the replication's type is asynchronous. The keyword specifies if it is a cascade replication. Let us use an example to describe cascade replications. Commands flow in most organizations, from the highest level to the basic level. This is similar to replicating data from A to B, and then to C. This is a typical kind of cascade replication. The 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 specification is CASCADE.

If you drop a table or a column that is referenced by an asynchronous table replication, alter a table and modify the column definition, or alter a table and add a column using the BEFORE and AFTER keywords, the synchronous replication becomes invalid and cannot be used again. Altering a table and adding a column without using the BEFORE and AFTER keywords has no impact on a synchronous replication. Asynchronous table replications are not affected when you alter a table. Drop an invalid replication to remove it from the database. Any replications created on a table are dropped automatically when dropping a table.

Replication names have a maximum length of 128 characters, and may contain numbers, letters, the underscore character, and the symbols $ and #. The first character may not be a number.

replication_name......Name of the table replication to create

local_table_name......Name of the local table to replicate

column_name ....1. Name of a column in the local table

...............................2. Name of a column in the remote table

search_condition ....Conditions a row must meet to be replicated

remote_table_name ...Name of the table in the remote database
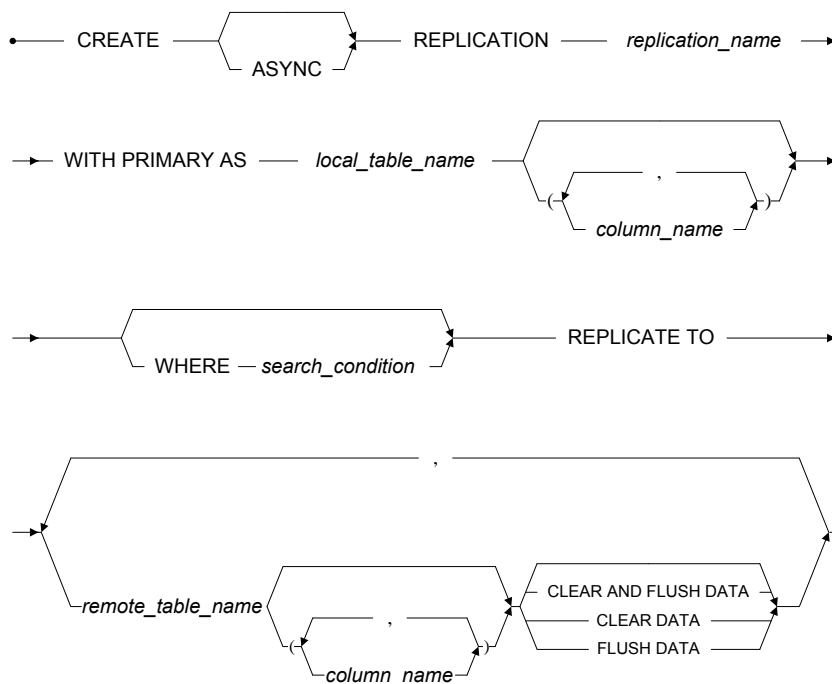
Figure 3-50 CREATE REPLICATION syntax

➲ **Example 1**

The following creates a replication named **EmpRep** for the local table named
Employeesinfo. The remote database is identified in the database configuration
section named **FieldOffice** in the local **dmconfig.ini** file. The remote table is also

named **Employeesinfo** and all column names and data types in both tables are the same.

```
dmSQL> CREATE REPLICATION EmpRep WITH PRIMARY AS Employeesinfo
                     REPLICATE TO FieldOffice:Employeesinfo;
```

➲ **Example 2**

The following is similar to the above example, but all data in the remote table is deleted and any data in the local table is replicated to the remote table.

```
dmSQL> CREATE REPLICATION EmpRep WITH PRIMARY AS Employeesinfo
                     REPLICATE TO FieldOffice:Employeesinfo
                     CLEAR AND FLUSH DATA;
```

3.42 CREATE SCHEDULE

The CREATE SCHEDULE command creates a replication schedule for asynchronous table replications. Synchronous table replications do not use schedules, so the CREATE SCHEDULE command has no effect on a synchronous table replication. Only users with DBA, SYSDBA or SYSADM security privileges can execute the CREATE SCHEDULE command.

A table replication creates a full or partial copy of a table in a remote 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 efficiently, without having to go to another machine over a slower network connection. This is not the same as backing up the database to a remote location, since the synchronization is done on a transaction-by-transaction basis by the DBMS itself, without any intervention from users.

The NO CASCADE keywords are optional. It takes action only when the replication type is asynchronous. The keyword specifies cascade replication. Let us use an example to describe cascade replications. Commands flow in most organizations from the highest level to the basic level. This is similar to replicating data from A to B, and then to C. This is typical cascade replication. The 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 specification is CASCADE.

DBMaker not only allows asynchronous table replication to other DBMaker databases, but also to Oracle, SYBASE, INFORMIX, and Microsoft SQL Server databases. This type of replication is known as heterogeneous table replication. Heterogeneous table replication allows DBMaker to coexist with other databases in a heterogeneous environment. Since DBMaker needs to preprocess the replicated data before sending it to a third-party remote database, specify the type of DBMS replicating to when creating a schedule in a heterogeneous environment. Do this with the ORACLE, SYBASE, INFORMIX, and MICROSOFT keywords, where ORACLE indicates a remote Oracle database, SYBASE indicated a remote SYBASE

database, INFORMIX indicated a remote INFORMIX database, and MICROSOFT represents a remote Microsoft SQL Server database.

When creating a heterogeneous table replication, the CLEAR DATA, FLUSH DATA, or CLEAR AND FLUSH DATA keywords cannot be used. Manually delete or insert data in the third-party remote database to put the table in its initial state before the replication begins. In addition, performing schema checking on the third-party remote database cannot be done. Check schema to ensure that columns and data types in the remote table are compatible with the columns and data types in the local table. When creating a schedule for a heterogeneous table replication, use the WITH NO CHECK keywords to prevent DBMaker from performing schema checking (See the description for the WITH NO CHECK keyword later in this section). DBMaker makes use of the ODBC Driver Manager to perform heterogeneous table replication; the DBMaker server must be located on Windows platforms. The third-party remote databases may be located on either Windows or UNIX platforms.

BEGIN AT specifies the date and time of the first replication for an asynchronous table replication. The date must be in yyyy/mm/dd format, where yyyy is the year in the range 1970 to 2038, mm is the month in the range from 01 to 12, and dd is the date in the range 01 to 31. The time must be in *hh:mm:ss* format, where *hh* is the hour in the range from 00 to 23, *mm* is the number of minutes in the range from 00 to 59, and *ss* is the number of seconds in the range from 00 to 59. The value for the year must be in the range from 1970 to 2038. Include the date and time when using the BEGIN AT keyword. If you change the date or time of the first replication to a date in the future after a replication is already running, any table data that has not been replicated to the remote database will wait until the new time for replication.

The EVERY command defines the interval between successive replications for an asynchronous table replication. The interval may be provided as hours/minutes/seconds, days, or a combination of both. To specify the number of hours/minutes/seconds, use EVERY *hh:mm:ss*. Specify the number of days with EVERY *d* DAYS, where *d* is the number of days in the range from 1 to 365. To specify a combination of both, use EVERY *d* DAYS AND *hh:mm:ss*.

RETRY indicates how many times DBMaker should try replicating table data if there is an error while trying to process a single SQL statement, such as a lock time-out

error, or rollback to save point due to a full Journal. To specify the number of times to try, use RETRY *n* TIMES, where *n* is the number of times to try in the range from 0 to 2,147,483,647. The default value is *0*. DBMaker waits until the next scheduled replication to send any table data that was not replicated successfully when not using the RETRY keyword and an error occurs while processing a statement, encounters a network error, remote database error, or any error, which requires a transaction rollback.

The AFTER keyword is optional. This keyword is used together with the RETRY keyword to specify the interval between successive retries in the event of an error. To specify the interval use the AFTER *s* SECONDS, where *s* is the number of seconds in the range from 0 to 2,147,483,647. The default value is *5*.

The STOP ON ERROR keywords are optional. These keywords specify the action DBMaker should take when data in the remote database has been updated in such a way that the replication could not take place. This could include situations where DBMaker tries to delete a previously deleted record from the remote table or tries to insert a record into the remote table that already exists. DBMaker provides two options when encountering this type of error, STOP ON ERROR and IGNORE ON ERROR. STOP ON ERROR indicates DBMaker will stop replicating data when an error of this type occurs, and IGNORE ON ERROR indicates that DBMaker will ignore the data that caused the error and continue replicating the remaining data. The default behavior is IGNORE.

The WITH NO CHECK keywords are optional. Since DBMaker cannot currently perform schema checking on a third-party database, use this keyword when creating a heterogeneous table replication. When using the WITH NO CHECK keywords, users must take responsibility for schema checking, and ensure that columns and data types in the remote table are compatible with the columns and data types in the local table. The WITH NO CHECK keywords are not necessary if performing a homogeneous table replication (e.g., from one DBMaker database to another DBMaker database).

The IDENTIFIED BY keywords specify the user name and password to use when connecting to the remote database. The user name provided must be an existing user in the remote database with sufficient privileges on the remote table to perform INSERT, DELETE, and UPDATE operations. When replicating table data to the

remote database, the operations you can perform on the remote table depend on the security and object privileges granted to that user.

remote_database_name…Name of the table in the remote database to create the replication schedule for; cannot be a database link.

yyyy/mm/dd..............Date to begin replication

hh:mm:ss..................1. Time to begin replication

..............................2. Replication time interval

d ....Day interval for replication to the remote table

n ....times to retry in the event of a failure

s ....seconds to wait before retrying in the event of a failure

user_name................remote database account User name

password ....remote database account Password

CREATE SCHEDULE FOR REPLICATION TO — *remote_database_name*

(ORACLE / MICROSOFT)

BEGIN AT *yyyy/mm/dd hh:mm:ss*
EVERY *d* DAYS
EVERY *hh:mm:ss*
EVERY *d* DAYS AND *hh:mm:ss*

RETRY *n* TIMES
AFTER *s* SECONDS

STOP ON ERROR   WITH NO CHECK

IDENTIFIED BY — *user_name*   *password*
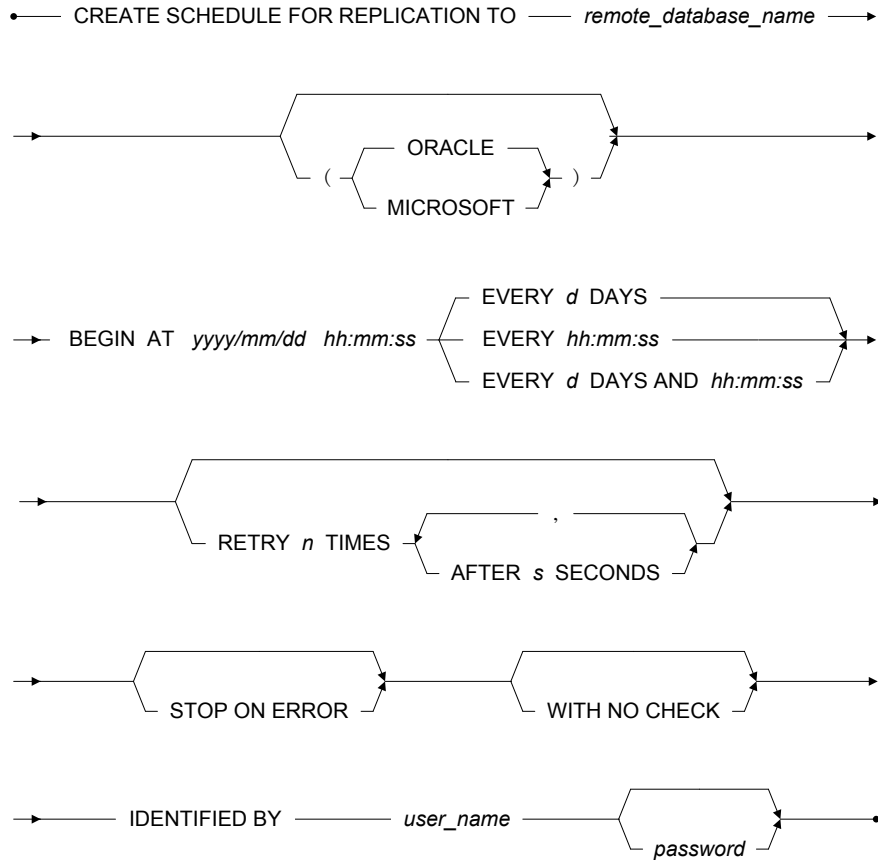
Figure 3-51 CREATE SCHEDULE syntax

➲ **Example 1**

The following creates a replication schedule for the asynchronous replication named **EmpRep**. The date and time of the first replication is set to a new date in the future,

with a replication interval of **7 days** and **12 hours**, the date is in the future; any table data that has not been replicated will wait until the new date before it is replicated.

```
dmSQL> CREATE SCHEDULE FOR REPLICATION TO EmpRep
            BEGIN AT 2001/10/10 00:00:00 EVERY 7 DAYS AND 12:00:00;
```

⮭ **Example 2**

The following example creates the same schedule as the previous example but also sets the times to retry after an error, lock time-out and a rollback to save point due to a full Journal to **3 times** with an interval of **5 seconds** between successive tries.

```
dmSQL> CREATE SCHEDULE FOR REPLICATION TO EmpRep
            BEGIN AT 2001/10/10 00:00:00 EVERY 7 DAYS AND 12:00:00
            RETRY 3 TIMES AFTER 5 SECONDS;
```

⮭ **Example 3**

The following creates the same schedule as the example above and sets the action DBMaker should take when data in the remote database has been updated in such a way that the replication cannot take place to **STOP**:

```
dmSQL> CREATE SCHEDULE FOR REPLICATION TO EmpRep
            BEGIN AT 2001/10/10 00:00:00 EVERY 7 DAYS AND 12:00:00
            RETRY 3 TIMES AFTER 5 SECONDS
            STOP ON ERROR;
```

⮭ **Example 4**

The following creates the same schedule as the example above and sets the user name and password to use when connecting to the remote database to **RepUser** and **rdejpe88**.

```
dmSQL> CREATE SCHEDULE FOR REPLICATION TO EmpRep
            BEGIN AT 2001/10/10 00:00:00 EVERY 7 DAYS AND 12:00:00
            RETRY 3 TIMES AFTER 5 SECONDS
            STOP ON ERROR
            IDENTIFIED BY RepUser rdejpe88;
```

⮭ **Example 5**

This is a heterogeneous table replication; specify the **WITH NO CHECK** keywords to prevent DBMaker from performing schema checking on the remote database. Ensure that columns and data types in the remote table are compatible with the

columns and data types in the local table the following creates the same schedule as the example above and uses the **ORACLE** keyword to indicate that the remote table is in an *Oracle 8.0* database.

```
dmSQL> CREATE SCHEDULE FOR REPLICATION TO EmpRep (ORACLE)
                BEGIN AT 2001/10/10 00:00:00 EVERY 7 DAYS AND 12:00:00
                RETRY 3 TIMES AFTER 5 SECONDS
                STOP ON ERROR
                WITH NO CHECK
                IDENTIFIED BY RepUser rdejpe88;
```

3.43 CREATE SCHEMA

The CREATE SCHEMA command creates and enters a new schema into the current database system. A schema is essentially a namespace: it contains named objects, also known as schema objects, (tables, view, index, synonym, trigger, domain, command, procedure) whose names may duplicate those of other objects existing in other schemas. Schema objects are accessed by qualifying their names with the schema name as a prefix.

Only users with RESOUCE privileges or above can create a schema. If the *user_name* is omitted when creating a schema, the schema creator becomes the default user. Only users with DBA authority may create schemas owned by users other than themselves.

When a user is granted connect privileges to DBMaker, DBMaker will create a default schema for the user. The schema name will be the user's name. The schema name must be unique. If a schema in the database, with the same name, already exists an error will be returned.

The owner of the schema is determined as follows:

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

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

schema_name ....Name of the new schema to create

user_name ....Name of the owner of the newly created schema

CREATE SCHEMA ——————— *schema_name* ——————————→

AUTHORIZATION ——— *user_name*
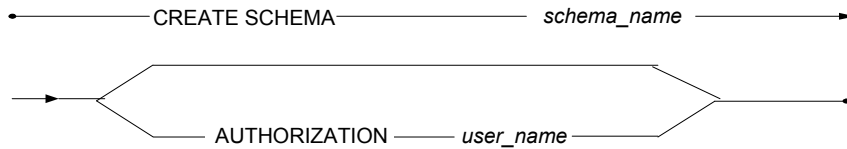
Figure 3-52 CREATE SCHEMA syntax

➲ **Example 1**

A user YUBIN, with RESOURCE authority, creates schema **schm_def**. YUBIN is the default owner of the schema.

```
dmSQL> CREATE SCHEMA schm_def;
```

➲ **Example 2**

A user, with DBA authority, creates a schema with the user YUBIN as the owner. YUBIN becomes the default schema name because no schema name was specified when the schema was created.

```
dmSQL> CREATE SCHEMA AUTHORIZATION YUBIN;
```

NOTE *It is import to remember that when a user is granted connection status DBMaker automatically creates a schema for the user with the schema name being the user's name. If a schema already exists in the database with the same name an error message will be returned.*

➲ **Example 3**

A user, with DBA authority, creates schema **schm_auth** with the user YUBIN as the owner.

```
dmSQL> CREATE SCHEMA schm_auth AUTHORIZATION YUBIN;
```

➲ **Example 4**

A user, with DBA authority, creates schema **inventory**. The user then creates the schema objects **inventory.part** and **partind** for the schema. The user then grants full user authority to the user YUBIN on the table created. The user YUBIN does not have any privileges on the schema **inventory**.

```
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 YUBIN;
```

3.44   CREATE SYNONYM

The CREATE SYNONYM command creates a new synonym on an existing table or view. You cannot create a synonym on a temporary table or on another synonym. Only the table or view owner, a DBA, a SYSDBA or a SYSADM have the privileges to execute the CREATE SYNONYM command on a table or view.

DBMaker normally identifies tables and views with fully qualified names that are a composite of the owner name and object name. To help simplify statements that use fully qualified table and view names, DBMaker provides synonyms.

A *synonym* is an alias that can be used for a table or view. It requires no storage space other than its definition in the system catalog. Using synonyms, users can access a table or view through the corresponding synonym without having to use the fully qualified name.

Create more than one synonym for a table or view using unique synonym names. This allows users to refer to synonym names without prefixing an owner name. If a user owns a table with the same name as a synonym, DBMaker always uses the table and ignores the synonym with the same name. To use the table referenced by the synonym, provide the fully qualified name for that table. All synonyms on a table or view are dropped automatically when dropping the referenced table or view.

Synonym names have a maximum of 128 characters, and may contain numbers, letters, underscore characters, and symbols $ and #. The first character may not be a number.

OR REPLACE: specify OR REPLACE to re-create the synonym that already exists, that is to say, you can use this clause to change the definition of an existing synonym..

synonym_name ....Name of the new synonym to create

table_name...............Name of the table to create the synonym on

view_name ....Name of the view to create the synonym on
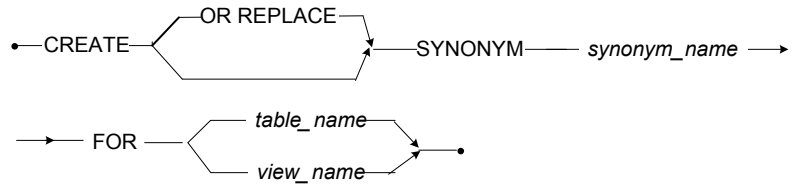
Figure 3-53 CREATE SYNONYM syntax

➲ **Example 1**

The following creates a synonym named **AllEmp** for the **AllEmployees** table owned by **User1**; use the synonym **AllEmp** in place of the fully qualified table name **User1.AllEmployees** in subsequent SQL statements.

```
dmSQL> CREATE SYNONYM AllEmp FOR User1.AllEmployees;
```

➲ **Example 2**

The following creates a synonym named **SalesEmp** for the **SalesEmployees** view owned by **User2**. Use the synonym **SalesEmp** in place of the fully qualified view name **User2.SalesEmployees** in subsequent SQL statements.

```
dmSQL> CREATE SYNONYM SalesEmp FOR User2.SalesEmployees;
```

3.45   CREATE TABLE

The CREATE TABLE command creates a new table. You should specify a tablespace when creating the table. DBMaker will create a table in the system tablespace by default. Any user with RESOURCE or higher security privileges can execute the CREATE TABLE command.

Tables are the primary unit of data storage in a relational database, and any information you enter in a database is stored in tables. Each table represents a single type of real-world object and contains information on individual objects of that type. These can be real objects, customers or products, and abstract objects, orders or transactions. Each table in a database is given a unique name and this name normally identifies the type of object stored in the table. Tables store the information about the objects they represent in rows and columns.

Rows, also called records or tuples, contain information that defines a single type of entity having common characteristics. Each row represents an individual occurrence of that type of entity. The rows are identified using one or more of the characteristics of the entity. They do not have any particular order and there is no guarantee that the rows will be listed in the same order twice.

Columns, also called fields or attributes, contain information that defines the characteristics of an entity. Each column represents one characteristic or item of data that is stored for each individual occurrence of an entity. They are identified using a descriptive name and a data type. Each column is referenced using a unique column name. Columns in a table can be rearranged without affecting SQL queries.

Ensure data integrity by applying constraints or rules. When creating a table, apply domain and column integrity constraints on individual columns, and table integrity constraints.

Domain constraints are defined as part of the domain definition and are applied to all columns based on the domain. When inserting a new row or updating an existing row, each domain constraint is evaluated. Domain constraints can include NULL/ NOT NULL constraints, default values, and CHECK constraints.

Column constraints are defined on a specific column and do not affect other columns in the same table. Whenever inserting a new row or updating an existing row, each column constraint is evaluated. Column constraints can include NULL or NOT NULL constraints, default values, and CHECK constraints.

Table constraints are defined on a set of columns. Whenever inserting a new row or updating an existing row, each table constraint is evaluated after, all domain and column constraints are evaluated as true. Only after the table constraint is also evaluated as true will the statement be processed. Table constraints can include UNIQUE and CHECK constraints, primary keys, and foreign keys.

To create a table, provide at least the table name and column definitions. Tables must have at least one column and can have as many as 2,000 columns. Please note, the maximum number of table columns also depends on the page size.

DBMaker identifies each table by a unique combination of schema name and table name, known as the fully qualified name. Table names have a maximum length of 128 characters, and may contain numbers, letters, the underscore character, and the symbols $ and #. The first character may not be a number. Table names must be unique among all tables in a database. Only users with DBA privileges can create a table with another user's table schema name. The specified table schema name must exist in the database. The default schema name is the creator of the table. Table names are case-insensitive.

To specify a column definition, provide at least a column name and a data type or domain. The syntax and usage of keywords used in column definitions are shown on the following pages.

table_name ....……Name of the new table to create

column_definition ....……Definition for a column

primary_key_defintion…...Definition for a primary key

foreign_key_definition…...Definition for a foreign key

constraint_name......……Name of the constraint to be applied to the table

tablespace_name......……Name of the tablespace to create the table in

boolean_expression ....……..Expression that evaluates true or false conditions

number ....The fillfactor value
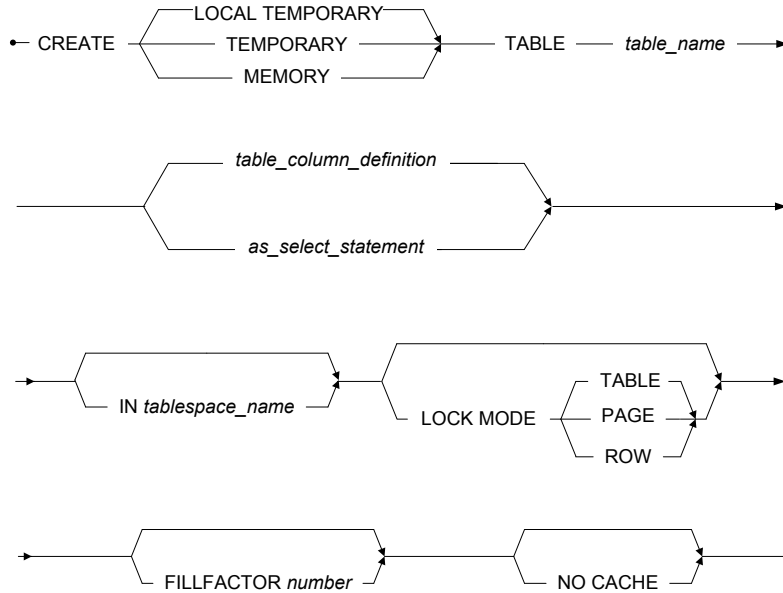
Figure 3-54 CREATE TABLE syntax
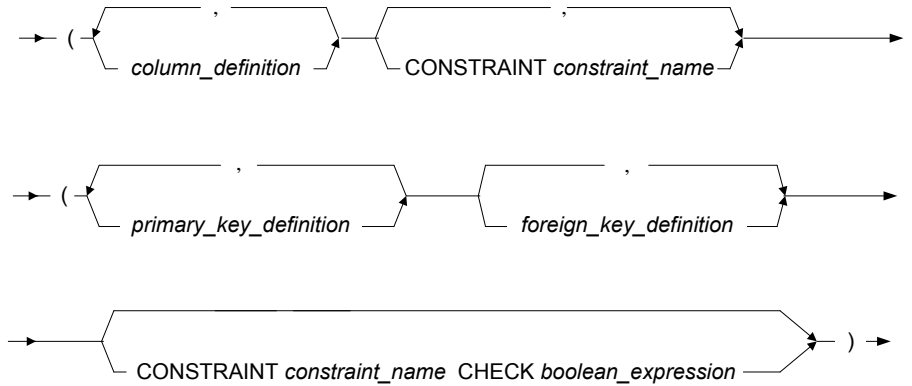
Figure 3-55 CREATE TABLE: table_column_definition syntax
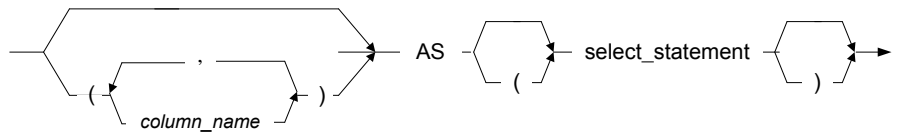
Figure 3-56 CREATE TABLE: as_select_statement syntax

Column Definitions

DBMaker identifies columns in a table by a unique combination of owner name, table name, and column name, known as the fully qualified name. Column names have a

maximum length of 128 characters, and may contain numbers, letters, the underscore character, and the symbols $ and #. The first character may not be a number. Column names must be unique among all columns in the same table. Column names are case insensitive.

DBMaker supports the following data types: BIGINT, BIGSETIAL, BINARY, CHAR, DATE, DECIMAL, DOUBLE, FLOAT, FILE, INTEGER, BLOB, CLOB, OID, SERIAL, SMALLLINT, TIME, TIMESTAMP and VARCHAR.

Optionally, use a domain for a column instead of a data type. Domains are a combination of data type, default value, and constraints that are applied to a column when it is defined using a domain as the data type. See the column definition DEFAULT and CHECK keywords below for a description of default values and constraints. Default values and constraints provided in the column definition will override those of the domain. Column definitions can also provide constraints in addition to those of the domain.

The NULL/NOT NULL keywords are optional. These keywords specify whether a column can contain a NULL value when inserting a new row. The NULL *keyword* specifies that a column may contain an undefined value when a new row is inserted, while the NOT NULL keyword specifies that a value must be provided when a new row is inserted. The NULL/NOT NULL keyword, **NULL is** used by default.

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 DEFAULT keyword is optional. This keyword is used to specify a default value that will be inserted into a column if no value is provided when inserting a new row. Constants, results from built-in functions, or the NULL keyword may be used as the default value. You can only use built-in functions that have no argument like PI( ), NOW( ), or USER( ), when defining a column. If using the NULL keyword as the DEFAULT value, the column cannot be defined with the NOT NULL keyword.

The ON UPDATE keyword 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 CHECK keyword, in the column definition, is optional. This keyword is used to specify a range of acceptable values that may be entered in a column. The expression that specifies the range of acceptable values may be any expression that evaluates to true or false. The VALUE keyword may be used in the expression in conjunction with the CHECK keyword to represent the value of the column. If an SQL statement does not satisfy the CHECK conditions, it will not be processed.

column_name ....Name of the column to create

data_type ....Name of the data type to use for the column

domain_name ....Name of the domain to use in place of a data type

literal.......................A literal value to use if no value is inserted

constant ....Constant value to use if no value is inserted

function_name ....Built-in function to use if no value is inserted

constraint_name.......Name of the constraint to be created

boolean_expression....Expression that evaluates true or false conditions
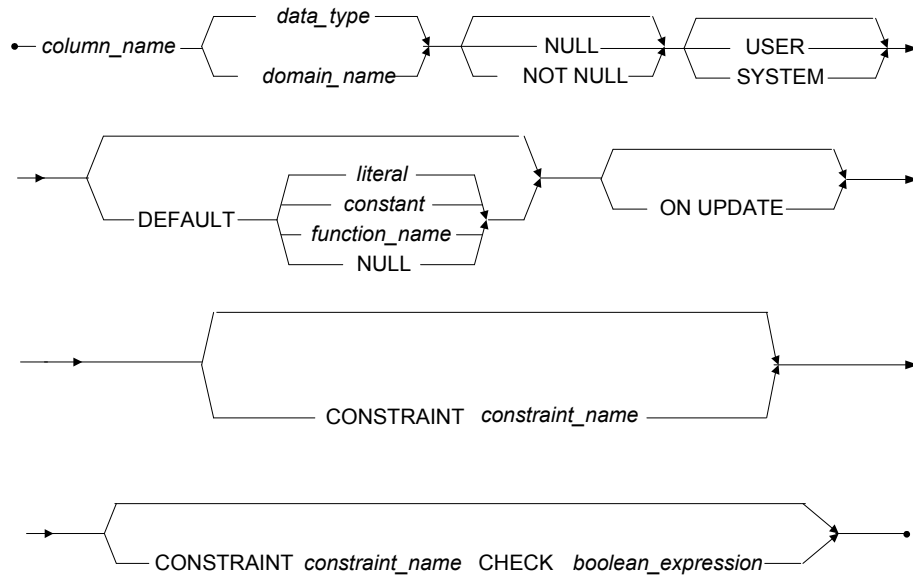
Figure 3-57 Column Definitions syntax

Primary Key and Unique Definitions

A key is a column or combination of columns that help identify specific rows in a table. The columns that make up a key are known as key columns. A unique key is a key in which no two records have the same value or the key field.

A primary key is a key that uniquely identifies each row in a table. Without a primary key, it is impossible to distinguish between specific rows in a table because rows may contain duplicate values. The DBMS does not permit defining a primary key on columns that contain duplicate values or to enter a duplicate value in a primary key that already exists.

Primary keys ensure data integrity in a table by requiring unique key values in each record of the primary key. This means columns in a primary key may not contain duplicate or null values, define the key columns with the NOT NULL constraint.

Primary keys may be built on up to 32 columns, providing the size of the columns does not exceed 4,000 *bytes*.

Each table may only have one primary or unique key. A primary key cannot be renamed. Instead, DBMaker automatically creates and maintains a unique, internally managed index named PrimaryKey for the primary key in each table. Since DBMaker builds an index on the primary key, it is not necessary to build another index on the columns in the primary key to increase the performance of query operations.

constraint_name.......Name of the constraint to be created

column_name ....Name of the column to create the primary key on
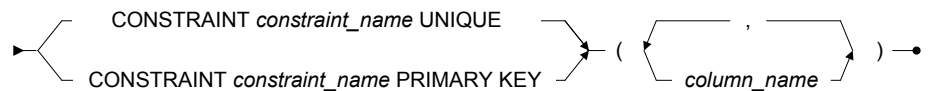
Figure 3-58 Primary Key and Unique Definitions syntax

Foreign Key Definitions

A foreign key is a key that corresponds to the primary key or a unique index of another table. This establishes a parent-child relationship between two tables that is represented by common data values stored in the tables. The parent table contains the primary key or unique index, and the child table contains the foreign key whose columns correspond to columns in the parent table.

Referential integrity ensures that every value in a child key has a corresponding value in the parent key. Referential integrity is enforced between tables using the parent-child relationship established with foreign keys. DBMaker has automatic support for referential integrity constraints between tables through the definition of foreign keys. When adding a record to a child table, the value in the child key must also exist in the parent key. Similarly, when deleting a record from the parent table, all records in the child key with the same value must be deleted first.

Referential actions provide a means to update or delete a parent key when referential integrity would not normally allow it. The referential actions define the operation DBMaker should perform on all matching rows in the child key when you update or delete a parent key. DBMaker supports four referential actions for both updates and deletes: CASCADE, SET NULL, SET DEFAULT, and NO ACTION.

The ON UPDATE/ON DELETE keywords are optional. These keywords specify the referential action DBMaker should perform when you update or delete a value in a parent key that is referenced by a child key. The referential actions for these keywords are: CASCADE, SET NULL, SET DEFAULT, and NO ACTION.

CASCADE performs an update or delete on all matching values in the child key when updating or deleting the parent key. This will set the value of the child key to the same value as the parent key when update or delete a row in the parent key.

SET NULL sets all matching values in the child key to NULL when updating or deleting a row in the parent key. The SET NULL action cannot be used when the child key was defined with the NOT NULL constraint.

SET DEFAULT sets all matching values in the child key to the default value of the column when updating or deleting a row in the parent key. You cannot use the SET DEFAULT action when the default value is NULL and the child key was defined with the NOT NULL constraint.

NO ACTION enforces normal referential integrity rules. DBMaker uses NO ACTION by default.

There is no practical limit to the number of foreign keys in a table. The parent key may be the primary key or any other unique index of a table, but a parent key must be created before adding the child key. The number of columns and column type or length must be the same in the parent key and the child key. The column order of corresponding keys may be different in each table, provided they are listed in corresponding order in the foreign key definition. The primary key of the parent table is used by default.

Columns in a foreign key may contain null values. If a foreign key contains a null value, it satisfies referential integrity automatically. A foreign key may not be created on a view, but may be created on a synonym. Foreign key names have a maximum

length of 128characters, and may contain numbers, letters, the underscore character, and the symbols $ and #. The first character may not be a number.

constraint_name.......Name of the constraint to be created

key_name.................Name of the foreign key to be created

column_name ....1. Name of the column the foreign key is created on

..............................2. Name of the column referenced by the foreign key

parent_table_name...Name of the table the foreign key references
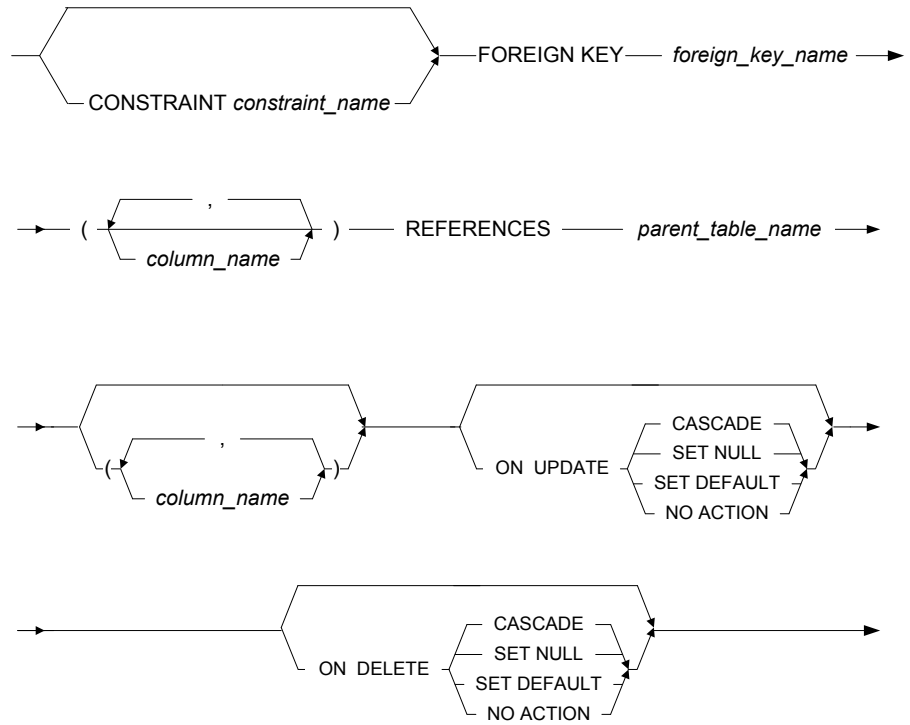
Figure 3-59 Foreign Key Definitions syntax

Table Options

DBMaker provides a number of optional features that can be used when creating a table. Specify the behavior of these options using: TEMPORARY/TEMP/MEMORY, IN, CHECK, LOCK MODE, NOCACHE, and FILLFACTOR keywords.

The TEMPORARY/TEMP keywords are optional. These keywords specify that a table should be created as a temporary table instead of a permanent table. Data access is faster in temporary tables since no locks are used and no Journal records are written for temporary tables. However, temporary tables can only be used by the table owner, and are automatically deleted when you disconnect from the database. Also, drop a temporary table at any time while still connected to the database using the DROP TABLE command.

The MEMROY keywords are optional. Memory tables, for almost all intents and purposes, function in the same manner as a regular table in DBMaker. The differences lie in the fact that memory tables are temporary tables, their life cycle being connection based. This means that when user create a memory table, it ill be dropped when the user drop it or when user disconnected from the database. Unlike a regular table, memory table are only stored in the memory of the connection that created them. They cannot be used by other connection and they can only have data selected or inserted, their data cannot be updated or deleted. Memory tables do support the transaction controls: commit, rollback, define save point and rollback to save point.

These keywords specify that a table should be created as a temporary table instead of a permanent table. Data access is faster in temporary tables since no locks are used and no Journal records are written for temporary tables. However, temporary tables can only be used by the table owner, and are automatically deleted when you disconnect from the database. Also, drop a temporary table at any time while still connected to the database using the DROP TABLE command.

The IN keyword is optional. This keyword specifies the name of the tablespace the table will be created in. Tablespaces are the logical areas of storage used to partition information in a database into manageable areas. Permits separate tables according to

logical groupings, or to place frequently used tables in different storage locations .The table is created in the system tablespace by default.

The CHECK keyword, in the table definition, is optional. This keyword behaves in a manner similar to the CHECK keyword used in the column definition. It normally is used to ensure data from multiple columns falls into an acceptable range of values. The expression of acceptable values may be any expression that evaluates to true or false. Column names may be used in the expression in conjunction with the CHECK keyword to represent the value of a column. If an SQL statement does not satisfy the CHECK conditions, it is not processed.

The LOCK MODE keyword is optional. This keyword specifies the lock level DBMaker uses when accessing data in a table. DBMaker includes the table, page, and rowlock modes. Page lock mode is used by default. To determine the lock mode of a table, examine the LOCKMODE column of the SYSTABLE system table.

LOCK MODE TABLE locks an entire table. This mode decreases concurrency by preventing other users from accessing the locked table at the same time. It also uses fewer lock resources and requires less memory in the System Control Area (SCA).

LOCK MODE PAGE locks a single data page. This mode is a trade-off between concurrency and lock resources. It provides moderate concurrency since other users may access data in other pages, but not access any data on the same page.

LOCK MODE ROW locks a single row. This mode increases concurrency by allowing other users to access any data except the locked row at the same time. It also uses more lock resources and requires more memory in the SCA.

FILLFACTOR specifies the percentage of a data page that can be filled. This allows the database to optimize the use of data pages, reserving space for updates to records. The number parameter can have a value from 50 to 100, which represents a fillfactor of 50% to 100%. To determine the fillfactor of a table, examine the FILLFACTOR column of the SYSTABLE system table.

NOCACHE limits the number of page buffers used to cache data during a table scan. DBMaker stores page buffers in a buffer chain with the most recently used page at the beginning and the least recently used page end. When the NOCACHE option is turned on, data pages read during a table scan are placed at the end of the buffer

chain. Since the end of the buffer chain will be flushed before the beginning, subsequent data pages read during the table scan, will replace the previous page. This effectively limits the page buffers used during a table scan to one page buffer. To determine the cache mode of a table, examine the CACHEMODE column of the SYSTABLE system table.

When creating a table, you are the table owner. You have all object privileges on the table, and may assign object privileges for that table to other users. As the table owner, you retain all object privileges on the table even if your security privilege is reduced to CONNECT.

NOTE *Both forms of the CHECK and CHECK VALUE syntaxes have been updated in DBMaker to be SQL 99 compliant.*

⮕ **Example 1**

The following creates a table named **Scores** in the system tablespace with **StudentNo**, **Math**, **English**, **Science**, and **History** columns, defined with the **INTEGER** data type.

```
dmSQL> CREATE TABLE Scores (StudentNo INTEGER,
                      Math INTEGER,
                   English INTEGER,
                   Science INTEGER,
                   History INTEGER);
```

⮕ **Example 2**

The following creates the same table from the example above in the **StudentRecords** tablespace, columns may not contain NULL values, and a default value of *zero* is assigned to the **Math**, **English**, **Science**, and **History** columns with the table owner name **Madison**.

```
dmSQL> CREATE TABLE Madison.Scores
                  (StudentNo INTEGER NOT NULL,
                      Math INTEGER NOT NULL DEFAULT 0,
                   English INTEGER NOT NULL DEFAULT 0,
                   Science INTEGER NOT NULL DEFAULT 0,
                   History INTEGER NOT NULL DEFAULT 0)
          IN StudentRecords;
```

➲ **Example 3**

The following creates the same table from the example above and the **Math**, **English**, **Science**, and **History** columns must contain values from 0 to 100.

```
dmSQL> CREATE TABLE Scores (StudentNo INTEGER NOT NULL,
                         Math INTEGER NOT NULL DEFAULT 0
                                     CHECK VALUE >= 0 AND VALUE <= 100,
                      English INTEGER NOT NULL DEFAULT 0
                                     CHECK VALUE >= 0 AND VALUE <= 100,
                      Science INTEGER NOT NULL DEFAULT 0
                                     CHECK VALUE >= 0 AND VALUE <= 100,
                      History INTEGER NOT NULL DEFAULT 0
                                     CHECK VALUE >= 0 AND VALUE <= 100)
           IN StudentRecords;
```

➲ **Example 4**

The following creates the same table from the example above and defines a table constraint to ensure: the sum of the **Math**, **English**, **Science** and **History** columns is less than **400**, the lock mode is set to **PAGE**, specifies a **FILLFACTOR** of **90**, and turns on the **NOCACHE** option.

```
dmSQL> CREATE TABLE Scores (StudentNo INTEGER NOT NULL,
                         Math INTEGER NOT NULL DEFAULT 0
                                     CHECK VALUE >= 0 AND VALUE <= 100,
                      English INTEGER NOT NULL DEFAULT 0
                                     CHECK VALUE >= 0 AND VALUE <= 100,
                      Science INTEGER NOT NULL DEFAULT 0
                                     CHECK VALUE >= 0 AND VALUE <= 100,
                      History INTEGER NOT NULL DEFAULT 0
                                     CHECK VALUE >= 0 AND VALUE <= 100)
           IN StudentRecords
           CHECK Math + English + Science + History <= 400;
```

➲ **Example 5**

The following creates the same table from the example above, but sets the lock mode to **PAGE**, specifies a **FILLFACTOR** of **90**, and turns on the **NOCACHE** option.

```
dmSQL> CREATE TABLE Scores (StudentNo INTEGER NOT NULL,
                         Math INTEGER NOT NULL DEFAULT = 0
                                     CHECK VALUE >= 0 AND VALUE <= 100,
```

```
                         English INTEGER NOT NULL DEFAULT = 0
                                        CHECK VALUE >= 0 AND VALUE <= 100,
                         Science INTEGER NOT NULL DEFAULT = 0
                                        CHECK VALUE >= 0 AND VALUE <= 100,
                         History INTEGER NOT NULL DEFAULT = 0
                                        CHECK VALUE >= 0 AND VALUE <= 100)
              IN StudentRecords
              CHECK Math + English + Science + History <= 400
              LOCK MODE PAGE
              FILLFACTOR 90
              NOCACHE;
```

➲ Example 6a

```
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 6b

```
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 6c**

```
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 6d**

```
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
```

CREATE TABLE AS SELECT

Use the CREATE TABLE AS SELECT syntax to create a table and the column definition and data derived from the select_statement. It will create the table's column definition like CREATE VIEW and insert data like SELECT INTO.

➲ **Example**

The following creates the table from the query that selects the **Math** score > 70's **StudentNo** from the Scores table.

```
dmSQL> CREATE TABLE Scores70 AS SELECT StudentNo, Math FROM Score
```

```
      WHERE Math > 70 IN tablespace1;
```

3.46 CREATE TABLESPACE

The CREATE TABLESPACE command generates a new tablespace. A new tablespace permits increasing the physical storage available to the database. Only users with DBA, SYSDBA or SYSADM security privileges can execute the CREATE TABLESPACE command.

DBMaker uses the relational data model to hide the details of the physical storage model and present data using a logical storage model. In the DBMaker physical storage model, files are physical storage structures that contain the data in the database. Files are managed by the operating system, with the exception of raw *UNIX* devices, while data in the files is managed by the DBMS. DBMaker uses three types of files during normal operation Data, BLOB, and Journal.

Data files and BLOB files store user and system data. Although they have similar characteristics, DBMaker manages these two file types in different ways to improve performance. Data files store table and index data, while BLOB files store only Binary Large OBjects (BLOBs).

Journal files are special files that provide a real-time, historical record of all changes made to a database and the status of each change. This allows the database to undo changes made by a transaction that fails, or redo changes made successfully but not written to disk after a database crashes. Journal files are used only by the database management system, and are not used to store user data.

In the DBMaker logical storage model, tablespaces are the logical storage structures used to partition information in a database into manageable areas. Each tablespace may contain several tables and indexes. Data in the tablespace is managed by the DBMS, but is physically stored in data and BLOB files. The three types of tablespaces included are regular, autoextend, and system.

Regular tablespaces are tablespaces that have a fixed size and contain one or more data or BLOB files. Manually extend a regular tablespace by enlarging existing files or adding new files. A regular tablespace may contain a maximum of 32,767 files, with a

maximum cumulative size of 8 TB. On UNIX platforms, regular tablespaces may be placed on raw devices.

NOTE   *For more information on raw devices, see your UNIX system documentation.*

Autoextend tablespaces are tablespaces that automatically increase in size to hold additional data as required. Regular and autoextend tablespaces may contain one or many data files, and *BLOB files*. It is possible for an autoextend tablespace to run out of space. The maximum file size is 8 TB and or the disk may be full. Add files to autoextend tablespaces manually to extend an autoextend tablespace by enlarging existing files. Do this to pre-allocate space for improved performance when inserting a large amount of data into an autoextend tablespace. Autoextend tablespaces cannot be used with raw devices.

System tablespaces are tablespaces generated by DBMaker when creating a database. Each database has one system tablespace, which contains the system catalog tables used to store schema, security, and status information about the entire database. The system tablespace is a special type of autoextend tablespace. System tablespaces contain one data and one BLOB file created automatically with the tablespace and not used to store user data. System tablespaces may be converted to regular tablespaces and may not be used with raw devices.

The AUTOEXTEND keyword is optional. This keyword specifies whether a tablespace is created as an autoextend tablespace. An autoextend tablespace can extend its size automatically as when requiring additional space. An autoextend tablespace may be changed to a regular tablespace at any time. It may also be changed back to an autoextend tablespace at any time.

The BACKUP BLOB keyword is optional. This keyword specifies whether DBMaker will back up BLOB data in this tablespace when the database is in BACKUP_DATA_AND_BLOB mode. DBMaker backs up all BLOB data in the tablespace when the database is in BACKUP_DATA_AND_BLOB mode and BACKUP BLOB is ON. When BACKUP BLOB is set to OFF, DBMaker does not back up any BLOB data in the tablespace, regardless of the backup mode.

To ensure data independence within the database, operating system files cannot be referenced directly within a database. To work around this, each database file has two

names, a physical file name and a logical file name. The physical file name is the name used by the operating system, while the logical file name is the name used by the database. These two names are related by an entry in the **dmconfig.ini** file. Before executing the CREATE TABLESPACE command, make an entry in the **dmconfig.ini** specifying the logical file name, the physical file name, and the initial size of each physical file in the appropriate database configuration section. Please see the following examples.

The DATAFILE keyword specifies the logical file name and the type of files to create when creating the tablespace. Specify multiple files up to a maximum of 32,767; providing the type of tablespace permits it and there is sufficient disk space. Tablespaces must contain at least one data file. Add more files to a tablespace using the ALTER TABLESPACE command.

The TYPE keyword specifies whether DBMaker will create a new file as a data file or a BLOB file. Use TYPE = DATA to create a new data file, and TYPE = BLOB to create a new BLOB file. When not specifying the type of file using the TYPE keyword, the default file will be created as a data file.

DBMaker creates all physical files in the default database directory specified by the **DB_DbDir** keyword in **dmconfig.ini**, unless a directory or path for the file is specified. The initial file size is specified as a number of data pages for data files, or a number of BLOB frames for BLOB files.

Specify an initial file size for data files by specifying a value from 2 to 2,147,483,647 pages. To calculate the actual size of the file in kilobytes, multiply this value by the value of **DB_PgSiz** as specified in **dmconfig.ini**. Specify an initial file size for BLOB files by specifying a value from 2 to 524,287 frames. To calculate the actual size of the file in kilobytes, multiply this value by the value of **DB_BfrSz** from the **dmconfig.ini** file.

The files in a tablespace do not have to be located on the same disk; you may specify a different disk or different path on the same disk for each file in the tablespace. If using *UNIX*, also allocate files in a regular tablespace on raw devices. Using raw devices allows faster access and performance improvements over regular operating system files.

DBMaker writes to raw device files directly instead of relying on operating system calls.

Tablespace names and logical file names have a maximum of 128characters and may contain numbers, letters, underscore characters and symbols $ and #. The first character may not be a number. Tablespace names are case-sensitive.

Physical file names have a maximum length, including drive and path names, of 255 characters, and may contain any characters and symbols permitted by the operating system, except spaces. The case-sensitivity of physical file names is dependent on the operating system.

tablespace_name ....Name of the new tablespace to create

file_name ....Logical name of the physical tablespace files
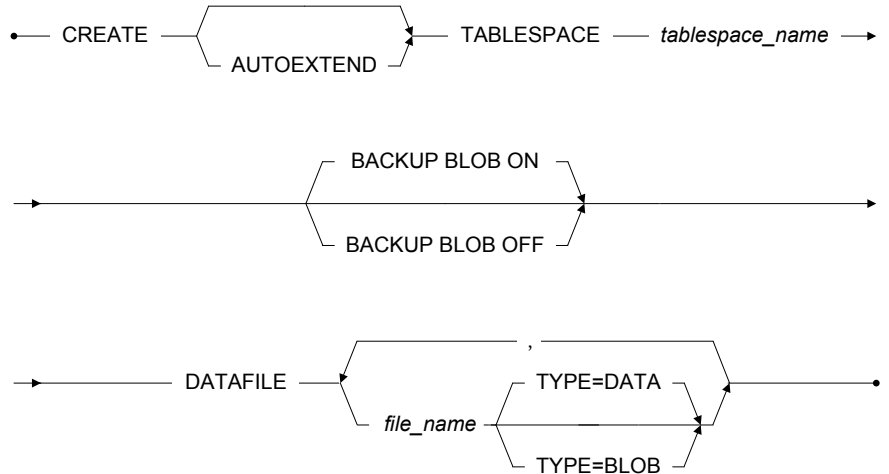
Figure 3-60 CREATE TABLESPACE syntax

➲ **Mapping 1**

Before executing example 1, add a line to the **dmconfig.ini** file to map the logical file names to the physical file names, and indicate the initial physical file size in pages for data files or frames for **BLOB** files. The size of the data file will be **800 KB** using the default Page Size of **8 KB** and the size of the **BLOB** file will be **3200 KB**, using the default **BLOB** frame size of **32 KB**.

```
datafile = c:\dbmaker\database\ts_reg_df.db 100
blobfile = c:\dbmaker\database\ts_reg_bf.bb 100
```

➲ **Example 1**

The following creates a regular tablespace named **ts_reg** with one logical data file named **datafile** and one logical BLOB file named **blobfile** and permits adding additional data or BLOB files to the tablespace, up to a maximum of 32767 files.

```
dmSQL> CREATE TABLESPACE ts_reg DATAFILE datafile TYPE=DATA, blobfile TYPE=BLOB;
```

➲ **Mapping 2**

Before executing example 2, add a line to the **dmconfig.ini** file to map the logical file names to the physical file names, and indicate the initial physical file size in pages for data files or frames for **BLOB** files. The size of the data file will be **800 KB** using the default Page Size of **8 KB** and the size of the **BLOB** file will be **3200 KB** using the default **BLOB** frame size of **32 KB**.

```
datafile = c:\dbmaker\database\ts_ext_df.db 100
blobfile = c:\dbmaker\database\ts_ext_bf.bb 100
```

➲ **Example 2**

The following creates an autoextend tablespace named **ts_ext** with one logical data file named **datafile**, and one logical BLOB file named **blobfile**; additional data or **BLOB** files may not be added to this tablespace.

```
dmSQL> CREATE AUTOEXTEND TABLESPACE ts_ext DATAFILE datafile TYPE=DATA,
                                    blobfile TYPE=BLOB;
```

3.47 CREATE TEXT INDEX

Two types of index may be created with DBMaker, a signature text index or an inverted file (IVF) text index. Signature text indexes are built in the same tablespace as the column for which the index is being built. IVF indexes are built in a separate file and exhibit better performance for larger indexes.

The CREATE TEXT INDEX command creates a new text index on a column or columns. Use text indexes to increase the performance of full-text queries by quickly locating specific words in columns containing text without examining the entire table. Only the table owner, a DBA, a SYDBA, a SYSADM, or a user with the INDEX privilege on that table may execute the CREATE TEXT INDEX command.

A text index is a mechanism that provides fast access to rows that contain one or more words or phrases in columns containing text. Text indexes contain a representation of all the text found in the text columns they are based on. The data is encoded and structured to make retrieval much faster than directly from the table. An index's operation is transparent to users and the DBMS uses it to improve full-text query performance.

When creating a text index, specify an index name, the name of the table, and the name of the column or columns. Text indexes may be created on columns defined with the CHAR, VARCHAR, CLOB, NCHAR, NVARCHAR, NCLOB, or FILE data types. Text indexes may not be created on system tables, temporary tables, or views.

The Order By clause supports a search for a word or words in a column and ranks the results in another column. After creating a text index with Order By Column, the result will be output ranked by the Order By Column automatically while DBMaker processes a query on the text index, speeding up the query. For example, to search the *content* column and order by *post time* column, add an Order By Post Time clause at the end of select statement. DBMaker must have a sorting on the result for the order by clause. The sorting will take a lot of time. If you have created the text index with Order By *Post Time* column, you can get a sorted result without adding the Order By Clause. Specify the ASC or DESC keyword to denote the ranking as ascending or

descending. The default order is ascending. The Order By Column attribute can also take affect on the increment part of the rebuild index command. However, it cannot re-order the records across old data or increment data.

When loading data into a table, DBMaker does not update any text indexes on that table. Load all data before creating a text index on a table, when possible. Rows containing matching text entered into a table after the text index was created will not be returned with the full-text search results. To include these rows in the search results, rebuild the text index using the REBUILD TEXT INDEX command.

Text index names must be unique for the each table. Text index names have a maximum length of 128 characters, and may contain numbers, letters, the underscore character, and the symbols $ and #. The first character may not be a number.

Signature Text Index

Signature text indexes can be built on all character type columns, including CHAR, VARCHAR, LONG VARCHAR, NCHAR, NVARCHAR, NCLOB, and FILE types. A table can have multiple text indexes, and text indexes can be built on multiple columns.

TOTAL TEXT SIZE is the estimated total size of all documents in the columns on which the text index will be built in MB. The range is from 1 to 200, and the default value is 32. This value is used for estimation and performance optimization by DBMaker and does not actually place a constraint on the number of documents allowed in a column. If the estimated total size exceeds 200 MB, use 200 MB or create an inverted file (IVF) index for significantly improved query performance.

SCALE is the expected ratio of index size to total column size. If you set the TOTAL TEXT SIZE to 20 and expect the index to use approximately 10 MB of storage, then you should set the scale to 50 (50%). Search performance increases as the scale increases. You can enter a range is from 10 to 200. The default value is *40*.

text_index_name ......Name of the text index to create

table_name...............Name of the table to create the text index on

column_name...........Name of the column to create the index on

order_column_name. Name of the column to start with

number...................value used with parameters SCALE and TOTAL TEXT SIZE
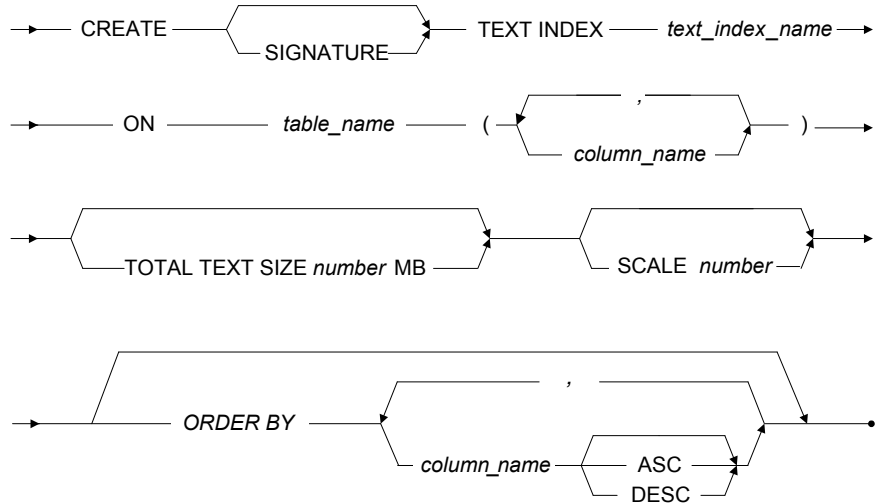
Figure 3-61 CREATE SIGNATURE TEXT INDEX syntax

➲ **Example 1**

The following creates a signature text index named **TxtIdx** on the **FName** column of the **Employeesinfo** table, using the default values for all parameters, and order by **Emp_ID** column.

```
dmSQL> CREATE SIGNATURE TEXT INDEX TxtIdx ON Employeesinfo(FName) ORDER BY
Emp_ID;
```

➲ **Example 2**

The following command creates a signature text index named **TxtIdx** on the **FName** column of the **Employeesinfo** table, estimating the total size of the column at 20 MB, and creating an index that scales to 50% of the size of the actual text index.

```
dmSQL> CREATE SIGNATURE TEXT INDEX TxtIdx ON Employeesinfo(FName) TOTAL TEXT SIZE
20 MB SCALE 50;
```

Inverted File Text Index

The CREATE IVF TEXT INDEX command creates a new inverted file (IVF) text index on a specified column. An IVF text index can be used in place of a standard index to increase the performance of queries, particularly on columns that contain more than 200 MB of data.

A table owner or a user with DBA, SYSDBA or SYSADM security privilege can create an IVF text index.

IVF indexes are sorted in the operating system's file system, and are administered through the database. The location where the IVF index should be stored is specified when the index is created. DBMaker manages the creation of sub-directories within the IVF index root directory.

text_index_name ......Name of the text index to create

table_name...............Name of the table to create the text index on

column_name...........Name of the column to create the index on

path ....Full directory path for storing the index

order_column_name .Name of the column to start with

number ....Value used with parameters SCALE and TOTAL TEXT SIZE
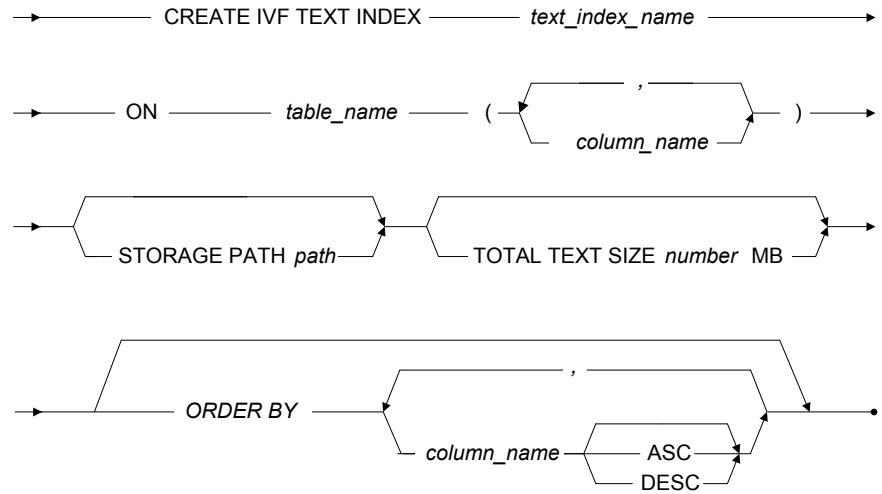
Figure 3-62 CREATE IVF TEXT INDEX syntax

◉ **Example 1**

The following creates an IVF text index named **TxtIdx** on the **LName** column of the **Employeesinfo** table, and using the default values for all parameters.

```
dmSQL> CREATE IVF TEXT INDEX TxtIdx ON Employeesinfo(LName);
```

◉ **Example 2**

The following command creates an IVF text index named **TxtIdx** on the **LName** column of the **Employeesinfo** table, and stores the IVF text index in the logical file **DB_IvfDir**, while estimating the total size of the column at 100 MB.

```
dmSQL> CREATE IVF TEXT INDEX TxtIdx ON Employeesinfo(LName) STORAGE PATH
DB_IVFDIR TOTAL TEXT SIZE 100 MB ORDER BY Emp_ID ASC;
```

3.48 CREATE TRIGGER

The CREATE TRIGGER command creates a new trigger on a table. Use triggers to customize a database in ways that would not be possible with standard SQL commands. Only the table owner, a DBA, a SYSDBA, or a SYSADM with all security and object privileges necessary to execute the SQL statement that defines the trigger action may execute the command.

A *trigger* is a database server mechanism that automatically executes predefined commands in response to specific events. This allows a database to perform complex or unconventional operations. Triggers are under the control of the database server and ensure that data is handled consistently, regardless of the source. A trigger on a table is transparent to users.

When creating a trigger, specify a name, trigger action time (when a trigger should fire relative to the trigger event), the trigger event (the event that causes the trigger to fire), a trigger table (the table the trigger is being created for), trigger type (type of trigger to be fired), and the trigger action (the action the database should perform when the trigger fires). Any triggers created on a table are dropped automatically when dropping the table.

DBMaker associates triggers using tables instead of fully qualified names. All trigger names on the same table must be unique. The trigger action operates with the same security and object privileges as the owner of the trigger table, and not with the privileges of the user executing the trigger event.

The BEFORE/AFTER keywords specify when the database server should perform the trigger action relative to the trigger event. This is known as the trigger action time. The BEFORE keyword specifies the database server to perform the trigger action before the trigger event. The AFTER keyword specifies that the database server should perform the trigger action after the trigger event.

The INSERT/DELETE/UPDATE keywords specify the event that fires a trigger. This is known as the trigger event. The INSERT keyword specifies that a trigger fires whenever inserting a row into a table, and the DELETE keyword specify that a trigger

fire whenever deleting a row from a table. The UPDATE keyword specifies that a trigger fire after updating any column in a table. Use UPDATE OF to instruct a column list when to fire a trigger after updating specific columns. Using UPDATE OF to specify a column list limits the use of each column name to on instance on all UPDATE triggers for that table.

The ON keyword specifies the name of the table to create the trigger on, known as the trigger table. The trigger table must be a permanent table in the database, not a temporary table, a view, or a synonym. Only specify a single trigger table for each trigger.

OR REPLACE: specify OR REPLACE to re-create the trigger that already exists, that is to say, you can use this clause to change the definition of an existing trigger.

trigger_name............Name of the trigger to create

column_name..........Name of the column to create the trigger on

table_name..............Name of the table to create the trigger on

sql_statement............Statement to execute when the trigger fires
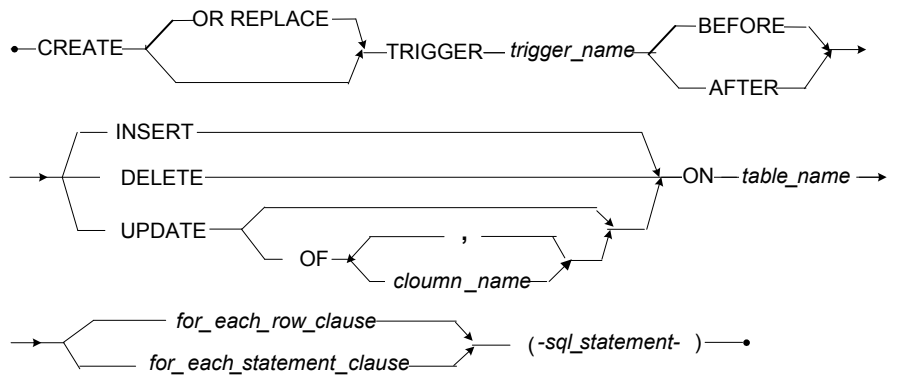
Figure 3-63 CREATE TRIGGER syntax

For Each Row Clause

The REFERENCING keyword specifies an alias for the OLD and NEW keywords. You usually need to indicate in the action, when creating a row trigger, to reference the value of a column before or after the trigger fires. Use the OLD and NEW keywords to refer to values from the trigger table, in cases where tables named OLD and NEW already exist in a database, use the alias specified by the REFERENCING keyword.

The FOR EACH ROW keyword specifies a trigger to fire once for each row the trigger event modifies. Triggers defined using the FOR EACH ROW keyword do not fire if the statement firing the trigger does not process rows.

The WHEN keyword specifies that only rows satisfying the search condition will cause the trigger to fire. The WHEN clause is evaluated for each row the trigger event modifies. If the search condition is true, the trigger fires for that row. If the search condition is false, the trigger does not fire. The result of the WHEN condition only affects the execution of the triggered action, it has no effect on the statement that fires the trigger.

old_name ....Alias for referencing the values, as they existed in the trigger table before the trigger action fires

new_name................ Alias for referencing the values, as they exist in the trigger table after the trigger action fires

search_condition ......Conditions a row must meet for a trigger to fire
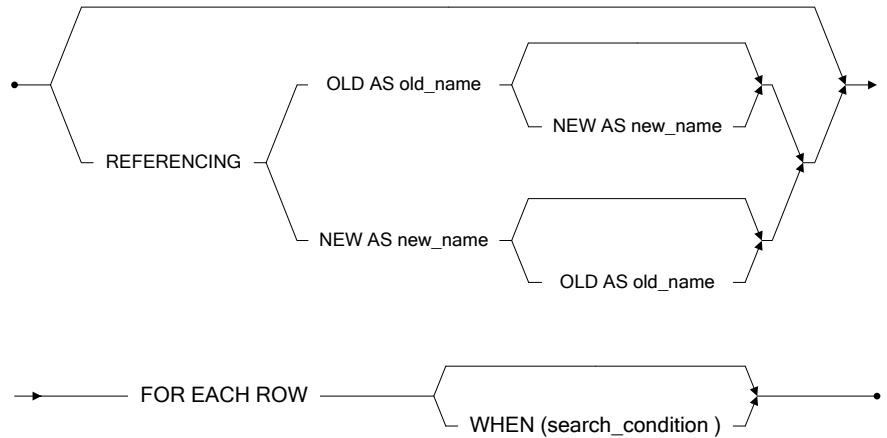
Figure 3-64 For Each Row Clause syntax

For Each Statement Clause

The FOR EACH STATEMENT keyword specifies that a trigger will fire once for each statement firing it. Triggers defined using the FOR EACH STATEMENT keyword will fire even if the statement firing it does not process rows.

The statement that the trigger executes when it fires is known as the trigger action. The trigger action may be an INSERT, UPDATE, DELETE, or EXECUTE PROCEDURE statement. If you want to use built-in functions when specifying the trigger action, only use functions that have no argument, such as PI( ), NOW( ), or USER( ). Stored procedures executed by a trigger cannot contain any transaction control statements COMMIT, ROLLBACK, or SAVEPOINT.

It is possible to create multiple triggers for each trigger event on the trigger table using the trigger action time, BEFORE and AFTER keywords, in combination with the trigger type, FOR EACH ROW and FOR EACH STATEMENT keywords. For example, combine the trigger action time and the trigger type to create four triggers

for the INSERT trigger event BEFORE/FOR EACH STATEMENT, BEFORE/FOR EACH ROW, AFTER/FOR EACH ROW, AFTER/FOR EACH STATEMENT. The same combinations for the UPDATE and DELETE trigger events may be performed.

Using the UPDATE OF instead of UPDATE will create at most, one trigger for each column in the table for each time/trigger type combination. This means that a table with four columns can have four UPDATE OF triggers for each combination BEFORE/FOR EACH STATEMENT, BEFORE/FOR EACH ROW, AFTER/FOR EACH ROW, and AFTER/FOR EACH STATEMENT. When using UPDATE OF to specify a trigger, the use of UPDATE is not permitted.

Trigger names must be unique for each table, have a maximum of 128 characters, and may contain numbers, letters, the underscore character and symbols $ and #. The first character may not be a number.

⟶————————————————— FOR EACH STATEMENT —————————————————⟶

Figure 3-65 For Each Statement Clause syntax

➲ **Example 1**

The following creates an **UPDATE** trigger named **Trig_update** on the **Employeesinfo** table that places the values before and after the update, into another table called **NameChange**. The trigger fires before the trigger action for each row updated in the table and fires regardless of the sequence of columns updated.

```
dmSQL> CREATE TRIGGER Trig_update BEFORE UPDATE ON Employeesinfo
                   FOR EACH ROW
                 (INSERT INTO NameChange
                      VALUES (OLD.FName, OLD.LName,
                              NEW.FName, NEW.LName));
```

➲ **Example 2**

The following creates an **INSERT** trigger named **Trig_insert** on the **Employeesinfo** table that executes the stored procedure called **SendMail** when inserting a new row in the **Employeesinfo** table and uses the **REFERENCING** keyword to provide an alias for the **OLD** and **NEW** keywords. The trigger will fire after the trigger action for each row inserted into the table.

```
dmSQL> CREATE TRIGGER Trig_insert AFTER INSERT ON Employeesinfo
                     REFERENCING OLD AS pre NEW AS post
                     FOR EACH ROW
                     (EXECUTE PROCEDURE SendMail(pre.FName,
                                                 pre.LName,
                                                 WelcomeMessage));
```

➲ **Example 3**

The following creates an **UPDATE** trigger named **Trig_update** on the **Orders** table that executes the stored procedure called **LogTime** when updating the **Orders** table, and will fire before the trigger action only once, regardless of how many rows the trigger action updates.

```
dmSQL> CREATE TRIGGER Trig_update BEFORE UPDATE ON Orders
                     FOR EACH STATEMENT
                     (EXECUTE PROCEDURE LogTime);
```

➲ **Example 4**

Suppose that the database have two table **tb_staff** and **tb_change** as follow:

```
dmSQL> CREATE TABLE tb_staff (FName char(10), LName char(10));
dmSQL> CREATE TABLE tb_change (new_FName char(10), new_LName char(10), old_FName
                     char(10), old_LName char(10));
```

Create or replace trigger **trig_update**.

```
dmSQL> CREATE OR REPLACE TRIGGER trig_update BEFORE UPDATE ON tb_staff
       FOR EACH ROW   (INSERT INTO tb_change VALUES (NEW.FName, NEW.LName,
       OLD.FName, OLD.LName));
```

3.49   CREATE VIEW

The CREATE VIEW command creates a new view based on existing tables or views. Only the owner of the base table with the RESOURCE privilege or users with, view, or SELECT privilege for the table may execute the command.

A view is a virtual table based on existing tables or views. Views appear to users like a real table with named columns and rows of data. Unlike a real table, the view is not stored permanently in the database. The data visible through a view is not physically stored in the database, but is instead stored in the original tables. Views are stored in the database as a definition and a user-defined view name. The view definition is an SQL query that DBMaker uses to access data from the original tables whenever using a view.

Use a view to tailor the appearance of a database to provide each user with a personalized view of a database. Provide security and restricted access to data by allowing users to see only the data they are authorized to see. Views also isolate users from changes to the underlying structure of the database. They present a consistent image of the database even if the underlying tables have changed.

Views can simplify the organization of a database by joining or grouping related data from several tables and presenting it as a single table. Use views to provide a subset of rows stored in the base table by having a condition on the returned results.

There are two disadvantages to using views instead of a real table, the performance, and the restrictions on updates. Performance is not as good for queries on a view as it is for queries directly on the source tables. The database must first retrieve the view definition, build it into the original query, perform the query, and then display the results. There are also update restrictions imposed by using views, since the database may not be able to manage updates on complicated views.

The SELECT statement that defines the view cannot contain INTO clauses. Currently DBMaker can update a view if that view is based on a single table.

Specify a list of column names for a view. The number of column names that are specified must match the number of columns in the SELECT statement. If not

specifying a list of column names, the view inherits the column names from the underlying tables.

View names and column names have a maximum of 128 characters and may contain numbers, letters, underscore characters, and symbols $ and #. The first character may not be a number.

OR REPLACE: specify OR REPLACE to re-create the view that already exists, that is to say, you can use this clause to change the definition of an existing view.

view_name...............Name of the new view to create

column_name ....Name of a column in the view

select_statement ....Select statement that specifies view contents
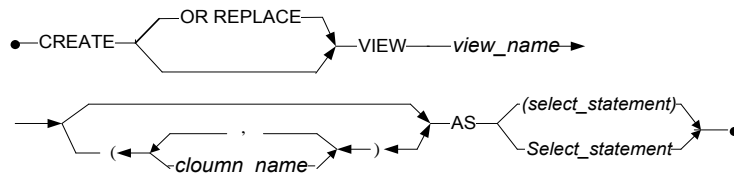
Figure 3-66 CREATE VIEW syntax

➲ **Example**

To create a view named **View_Emp** on the **Employeesinfo** table, you can use the following syntax:

```
dmSQL> CREATE VIEW View_Emp AS SELECT Name, Salary from Employeesinfo WHERE
Salary > 50000;
```

or:

```
dmSQL> CREATE OR REPLACE VIEW View_Emp AS SELECT Name, Salary from Employeesinfo
WHERE Salary >= 100000;
```

3.50   DECLARE SET

The DECLARE SET command defines a connection variable used in local connection. Only the user in this local connection can execute the command and the variable is enabled only in local connection.

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. Once connection to the database disconnects, all connection variables will be automatically freed.

Executing this command can store a value defined by type and value. The CV can replace the expression value in every SQL command.

To use the CV, users must add the symbol @ before the variable name, otherwise the dmsql will recognized the variable name as a column name or other identifier. CV name is not case sensitive.

data_type ....Data type to use for the CV

@variable_name.......Variable name

expression ....The result of the expression is a value of the variable name

The expression includes not only the assignment value in simple expression, but also C and Lua functions, such as build-in functions and user-defined functions.

⊷ DECLARE SET → *data_type*→ @ *variable_name*→ = → *expression* ─•

Figure 3-67 DECLARE SET syntax

➲ **Example 1**

To create a connection variable named **aa** and set the value to 1, the type to int.

```
dmSQL> DECLARE SET INT @aa = 1;
dmSQL> SELECT @aa;
          @AA
=======================
                      1
```

➲ **Example 2**

To create a connection variable named **bb** and set the value to '**syscom**', the type to char(20)**.**

```
dmSQL> DECLARE SET CHAR(20) @bb = 'syscom';
dmSQL> SELECT @bb;
          @BB
=======================
SYSCOM
```

➲ **Example 3**

To create a connection variable named **cc** and set the value to an expression.

```
dmSQL> DECLARE SET INT int @cc1 = 100+200;
dmSQL> DECLARE SET INT @cc2 = @cc1+300;
dmSQL> SELECT @cc1;
          @CC1
=======================
                    300
dmSQL> SELECT @cc2;
          @CC2
=======================
                    600
```

➲ **Example 4**

To create a connection variable named **dd** and set the value to an expression.

```
dmSQL> DECLARE SET CHAR(40) @dd = CONCAT('abcd','efgh');
dmSQL> SELECT @dd;
          @DD
=======================
abcdefgh
```

➲ **Example 5**

To create a connection variable named **ee** and set the value to an expression.

```
dmSQL> DECLARE SET DOUBLE @ee = 9.999999*100;
dmSQL> SELECT @EE;
        @EE
=======================
 9.99999900000000e+002
```

3.51   DELETE

The DELETE command deletes all rows matching the search condition from a table. Only rows from a single table may be deleted. Rows from the system tables may not be deleted. Only the table owner, a DBA, a SYSDBA, a SYSADM, or a user with the delete privilege on the table may execute the command. DBMaker only deletes rows that satisfy the search condition. Cursors are only available within ODBC programs.

See the WHERE clause in the SELECT command for more information on the search condition.

table_name .... Name of the table you want to delete rows from

search_condition .... Conditions a row must meet to be deleted

cursor_name .... Name of the cursor to use for a positioned delete
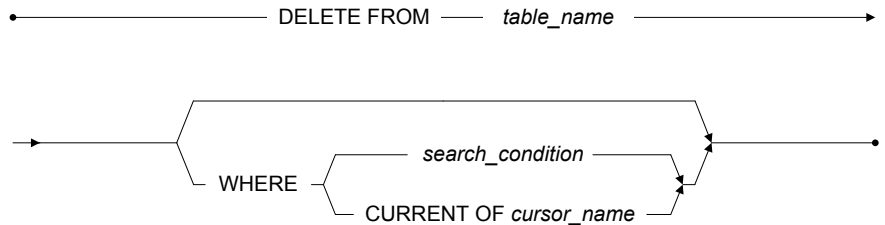
Figure 3-68 DELETE syntax

⮞   **Example 1**

The following deletes the employee number **1234** from the **Employeesinfo** table.

```
dmSQL> DELETE FROM Employeesinfo WHERE Emp_ID = '1234';
```

➲ **Example 2**

The following deletes all employee names that begin with "**John**" from the **Employeesinfo** table.

```
dmSQL> DELETE FROM Employeesinfo WHERE FName LIKE 'John%';
```

3.52 DROP COMMAND

The DROP COMMAND removes an existing stored command from the database. Only the stored command owner or a user with DBA, SYSDBA or SYSADM security privilege may execute the DROP COMMAND command.

A stored command is an SQL data-manipulation statement that is compiled and permanently stored in the database in executable format. This permits repeat execution of the stored command without waiting for DBMaker to compile and optimize the command each time. Stored commands are similar to stored procedures, except they can only contain a single command and cannot contain program logic.

The stored command becomes invalid and cannot be used again when dropping a table or a column that is referenced by a stored command, alter a table and modify the column definition, or alter a table and add a column using the BEFORE and AFTER keywords. Altering a table and adding a column without using the BEFORE and AFTER keywords has no impact on a stored command. Drop an invalid stored command to remove it from the database.

IF EXISTS: It will ensure that not throw an error while the stored command does not exist.

command_name.......Name of the stored command to remove from the database
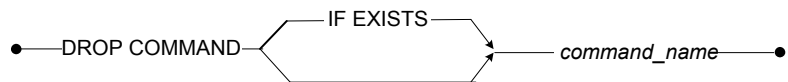
Figure 3-69 DROP COMMAND syntax

⊃ **Example**

Drop the stored command named **sc_select** with the following syntax:
```
dmSQL> DROP COMMAND sc_select;
```

or:

```
dmSQL> DROP COMMAND IF EXISTS sc_select;
```

3.53   DROP DATABASE LINK

The DROP DATABASE LINK command removes an existing public or private database link from the database. Only the owner of a private link may drop his or her own private link and only a user with DBA, SYSDBA or SYSADM security privilege can drop a Public link.

A database link creates a connection to a remote database to provide access to remote data. Links provide the benefit of security information, allowing connections to a remote database with a user name different from a local one, or connect to a remote database using a public link with no account.

The PUBLIC/PRIVATE keywords are optional. These keywords specify the type of database link to drop, public or private. Public links are available to all users in a database. Private links are available only to the user that creates them. When no specific type of link is specified, DBMaker tries to drop a private link by default.

link_name ....Name of the link to remove from the database

Figure 3-70 DROP DATABASE LINK syntax

➛   **Example 1**

The following drops the private link named **FieldLink**.

```
dmSQL> DROP PRIVATE DATABASE LINK FieldLink;
```

➲ **Example 2**

The following drops the public link named **FieldLink**.

```
dmSQL> DROP PUBLIC DATABASE LINK FieldLink;
```

3.54   DROP DOMAIN

The DROP DOMAIN command removes an existing domain from the database. Only the domain owner or a user with DBA, SYSDBA or SYSADM security privilege can execute the DROP DOMAIN command.

A domain is a user-defined data type that brings together a data type, default value, and value constraint. Use a domain in the column definition of CREATE TABLE or ALTER TABLE ADD COLUMN statements in place of a data type to define the set of valid values that can be entered into the column.

A domain cannot be dropped if there are existing columns in a table that were defined using the domain. To drop a domain that is referenced by existing columns, first drop all columns that reference the domain. Do this by dropping the entire table and then recreating the table without the domain, or by dropping a single column using the ALTER TABLE DROP COLUMN command.

The CASCADE/RESTRICT keywords are optional. These keywords denote whether to remove or check dependent objects refered to in the the dropped domain. When the CASCADE keyword is specified, it will remove all the dependent objects with the domain and replace the column definition with the domain definition. When the RESTRICT keyword is specified, it will not drop a domain that is referred in any table definition. The RESTRICT keyword ensures that only a domain with no dependent objects can be deleted.

domain_name .... Name of the domain to remove from the database
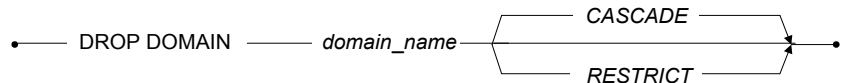
Figure 3-71 DROP DOMAIN syntax

‌➲   **Example**

The following example removes the domain named **ValidDate.**

```
dmSQL> DROP DOMAIN ValidDate;
```

3.55 DROP GROUP

The DROP GROUP command removes an existing group from the database. Only users with DBA, SYSDBA or SYSADM security privileges can execute the DROP GROUP command.

Groups simplify the management of object privileges in a database with a large number of users. Use a group to collect users that require the same object privileges. Any object privileges granted to the group are automatically granted to all members in the group. DBMaker also provides support for nested groups, a group as a member of another group, provided there are no circular references from the member group to the other group.

When a group is removed from a database, all members lose privileges granted to that group. Members retain all other privileges granted to them directly or to other groups they are members of. The PUBLIC group cannot be removed; DBMaker manages this group internally.

group_name............. Name of the group to remove from the database

DROP GROUP ——— *group_name*

Figure 3-72 DROP GROUP syntax

➲ **Example**

The following removes the group named **Manager** from the database.
```
dmSQL> DROP GROUP Manager;
```

3.56   DROP INDEX

The DROP INDEX command removes an existing index on a table from the database. Only the table owner, a DBA, a SYSDBA, a SYSADM, or a user with the INDEX privilege for that table may execute the DROP INDEX command.

An index is a mechanism that provides fast access to specific rows in a table based on the values of one or more columns from the table, known as the key. Indexes contain the same data as the key columns from the table they are based on, but the data is structured and sorted to make retrieval much faster than the table. Once creating an index, its operation is transparent to users; the DBMS uses the index to improve query performance whenever possible.

Drop an index from any table in the database except the system tables. If an index has foreign keys that refer to it, drop those foreign keys before dropping the index. Drop an index if it becomes fragmented, which reduces its efficiency. Rebuilding the index creates a denser, unfragmented index.

index_name...............Name of the index to remove

table_name...............Name of the table to remove the index from

•————— DROP INDEX ——— *index_name* ——— FROM ——— *table_name* ———•

Figure 3-73 DROP INDEX syntax

➲ **Example**

The following drops the index named **NameIndex** from the **Employeesinfo** table; if there are any foreign keys, which refer to **NameIndex**, drop them before dropping **NameIndex**.

```
dmSQL> DROP INDEX NameIndex FROM Employeesinfo;
```

3.57   DROP PROCEDURE

The DROP PROCEDURE command removes an existing procedure from the database. Only the table owner, a DBA, a SYSDBA, a SYSADM, or a user with the PROCEDURE privilege for that table may execute the DROP PROCEDURE command.

IF EXISTS: It will ensure that no error is thrown if the store procedure does not exist.

procedure_name .... Name of the procedure to remove from the database
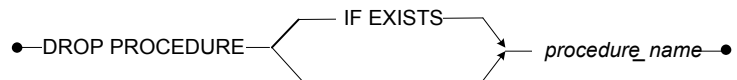
Figure 3-74 DROP PROCEDURE syntax

➲ **Example**

Drop the stored procedure **sp_proc1** with the following syntax:
```
dmSQL> DROP PROCEDURE sp_proc1;
```

or:
```
dmSQL> DROP PROCEDURE IF EXISTS sp_proc1;
```

3.58 DROP REPLICATION

The DROP REPLICATION command removes an existing table replication from the database. Only the table owner or a user with DBA, SYSDBA or SYSADM security privilege can execute the DROP REPLICATION command.

A table replication creates a full or partial copy of a table in a remote 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 efficiently, without having to go to another machine over a slower network connection. This is not the same as backing up the database to a remote location, since the synchronization is done on a transaction-by-transaction basis by the DBMS itself, without any intervention from users.

There are two primary types of table replication, synchronous and asynchronous. Synchronous table replication modifies the remote table at the same time it modifies the local table, while asynchronous table replication stores changes to the local table and modifies the remote table based on a schedule. Use the DROP REPLICATION command to drop both synchronous and asynchronous table replications.

replication_name ......Name of the table replication to remove

table_name...............Name of the table to remove the replication from

•⎯ DROP REPLICATION ⎯ *replication_name* ⎯ FROM ⎯ *table_name* ⎯•

Figure 3-75 DROP REPLICATION syntax

➲ **Example**

The following example drops the replication named **EmpRep** from the **Employeesinfo** table.

```
dmSQL> DROP REPLICATION EmpRep FROM Employeesinfo;
```

3.59 DROP SCHEDULE

The DROP SCHEDULE command removes an existing replication schedule to a remote database. Drop all associated asynchronous table replications before dropping a replication schedule. Only the local table owner or a user with DBA, SYSDBA or SYSADM security privilege can execute the DROP SCHEDULE command.

Use the DROP SCHEDULE command to drop a replication schedule for asynchronous table replications. Drop all associated asynchronous table replications before dropping a replication schedule. This would include any asynchronous table replication that replicates data to the remote database specified in the schedule.

remote_database_name….Name of the remote database to remove the replication schedule from

```
•——— DROP SCHEDULE FOR REPLICATION TO ——— remote_database_name ———•
```

Figure 3-76 DROP SCHEDULE syntax

➲ **Example**

The following drops the replication schedule for the remote database named **DivOneDb**.

```
dmSQL> DROP SCHEDULE FOR REPLICATION TO DivOneDb;
```

3.60   DROP SCHEMA

The DROP SCHEMA command removes a schema from the current database system. A schema is essentially a namespace: it contains named objects, also known as schema objects, (tables, view, index, synonym, trigger, domain, command, procedure) whose names may duplicate those of other objects existing in other schemas. Schema objects are accessed by qualifying their names with the schema name as a prefix.

Only users who created the schema or users with DBA authority can drop a schema from the database.

The schema to be removed must be empty. A schema containing schema objects cannot be dropped. Before attempting to drop a schema, drop all schema objects contained in the schema.

The CASCADE/RESTRICT keywords are optional. These keywords denote whether to remove or check dependent objects refered to in the schema to be dropped. When the CASCADE keyword is specified, it will remove all the dependent objects with the schema. When the RESTRICT keyword is specified, it will ensure that only a schema with no dependent objects can be deleted.

schema_name .... The name of the schema to be removed
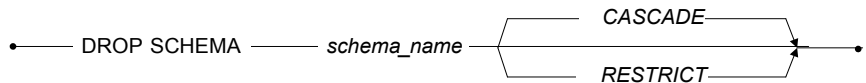
Figure 3-77 DROP SCHEMA syntax

3.61 DROP SYNONYM

A synonym is an alias that can be used for a table or view. A synonym requires no storage space, other than its definition in the system catalog. More than one synonym can be created for a table or view, but all synonym names must be unique. The DROP SYNONYM command removes a synonym from a table or view. Only the synonym owner or a user with DBA, SYSDBA or SYSADM security privilege can execute the DROP SYNONYM command.

DBMaker normally identifies tables and views with fully qualified names that are a composite of the owner name and object name. To help simplify statements that use fully qualified table and view names, DBMaker provides the usage of synonyms.

This allows users to refer to synonym names without prefixing an owner name. DBMaker will always use the table name and ignore a synonym with the same name. To use the table referenced by a synonym, provide the fully qualified name. All synonyms on a table or view are automatically dropped when a referenced table or view are dropped.

A synonym from any table in the database may be dropped, except for system tables. DBMaker internally manages all synonyms on the system tables, and does not permit dropping them.

IF EXISTS: It will ensure that not throw an error while the synonym does not exist.

synonym_name ....Name of the synonym to remove from the database
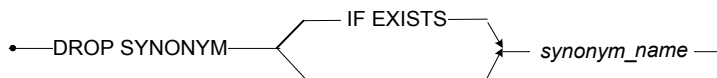
Figure 3-78 DROP SYNONYM syntax

➲ Example

Drops the synonym named **Staff** created on the **Employeesinfo** table with the following syntax:

```
dmSQL> DROP SYNONYM Staff;
```

or:

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

3.62 DROP TABLE

The DROP TABLE command removes a table. Only the table owner or a user with DBA, SYSDBA or SYSADM security privilege can execute the DROP TABLE command.

When dropping a table, DBMaker also drops all indexes and primary keys on the table. If the table has a primary key that is referenced by one or more foreign keys, drop all foreign keys that reference the primary key before dropping the table.

The CASCADE/RESTRICT keywords are optional. These keywords denote whether to remove or check dependent objects refered to in the table to be dropped. When the CASCADE keyword is specified, it will remove all the dependent objects with the table. When the RESTRICT keyword is specified, it will ensure that only a table with no dependent objects can be deleted.

IF EXISTS: It will ensure that no error is thrown if the table does not exist.

table_name...............Name of the table to drop from the database

CASCADE...............It will remove the dependent objects as index, foreign key, synonym, view and trigger with the table

RESTRICT..............It will ensure that only a table with no dependent objects as index, foreign key, synonym, view and trigger can be deleted
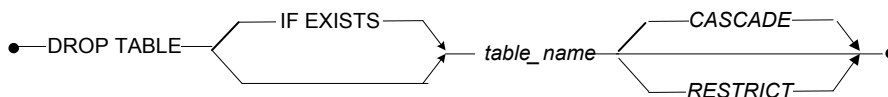
Figure 3-79 DROP TABLE syntax

⊃ **Example**

Drops the **Employeesinfo** table with the following syntax:

```
dmSQL> DROP TABLE Employeesinfo;
```

or:

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

3.63   DROP TABLESPACE

The DROP TABLESPACE command removes a tablespace. Only users with DBA, SYSDBA or SYSADM security privilege can execute the DROP TABLESPACE command.

When dropping a tablespace, DBMaker automatically drops all logical files in the tablespace. Use operating system commands to manually remove the physical files that correspond to logical files and free the disk space. If a tablespace contains tables, drop all tables in the tablespace before dropping the tablespace.

tablespace_name ....Name of the tablespace to drop from the database

DROP TABLESPACE————— *tablespace_ name* ——————

Figure 3-80 DROP TABLESPACE syntax

➲ **Example**

The following drops the **ts_emp** tablespace, drop all tables in the tablespace before dropping the tablespace.
```
dmSQL> DROP TABLESPACE ts_emp;
```

3.64 DROP TEXT INDEX

The DROP TEXT INDEX command removes an existing signature or IVF text index on a column in a table from the database. Only the table owner, a DBA, a SYSDBA, a SYSADM, or a user with the INDEX privilege for the table may execute the DROP TEXT INDEX command.

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 containing text. Text indexes contain a representation of all the text found in the text columns they are based on, but the data is encoded and structured to make retrieval much faster than directly from the table. Once a text index is created for a table, its operation is transparent to users of the database; the DBMS uses the index to improve full-text query performance whenever possible.

text_index_name ......Name of the text index to remove

table_name ....Name of the table to remove the text index from

Figure 3-81 DROP TEXT INDEX syntax

➲ **Example**

The following drops the text index named **TxtIdx** from the **Employeesinfo** table.

```
dmSQL> DROP TEXT INDEX TxtIdx FROM Employeesinfo;
```

3.65 DROP TRIGGER

The DROP TRIGGER command removes a trigger. Only the table owner or a user with DBA, SYSDBA or SYSADM security privilege can execute the DROP TRIGGER command.

A *trigger* is a database server mechanism that automatically executes predefined commands in response to specific events. This allows a database to perform complex or unconventional operations that might not be possible using standard SQL commands. Since triggers are under the control of the database server, they can ensure data is handled consistently regardless of the source. A trigger operation is transparent to users of the database DBMaker fires the trigger every time a user or application program generates a trigger event.

When dropping a table or a column that is referenced by a trigger, altering a table and modify the column definition, or altering a table and adding a column using the BEFORE and AFTER keywords, the trigger becomes invalid and cannot be used again. Altering a table and adding a column without using the BEFORE and AFTER keywords has no impact on a trigger. Drop an invalid trigger to remove it from the database. Any triggers created on a table are dropped automatically when a table is dropped.

IF EXISTS: It will ensure that no error is thrown if the trigger does not exist.

trigger_name ....Name of the trigger to remove

table_name...............Name of the table to remove the trigger from
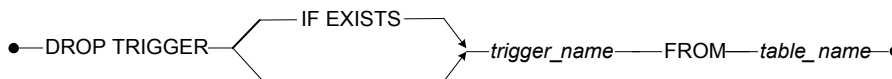
Figure 3-82 DROP TRIGGER syntax

➲ **Example**

Drops the trigger named **Trig_emp** from the **Employeesinfo** table with the following syntax:

```
dmSQL> DROP TRIGGER Trig_emp FROM Employeesinfo;
```

or:

```
dmSQL> DROP TRIGGER IF EXISTS Trig_emp FROM Employeesinfo;
```

3.66   DROP VIEW

The DROP VIEW command removes a view. Only the view owner or a user with DBA, SYSDBA or SYSADM security privilege can execute the DROP VIEW command.

When a view is dropped, DBMaker will invalid all views based on that view. System views may not be dropped.

The CASCADE/RESTRICT keywords are optional. These keywords denote whether to remove or check dependent objects refered to in the view to be dropped. When the CASCADE keyword is specified, it will remove all the dependent objects with the view. When the RESTRICT keyword is specified, it will not drop view that is referred in any view definition or synonym. The RESTRICT ensures that only a view with no dependent objects can be deleted.

IF EXISTS: It will ensure that no error is throwed if the view does not exist.

view_name...............Name of the view to remove from the database
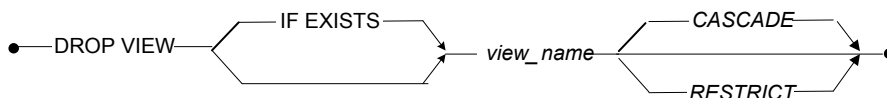
Figure 3-83 DROP VIEW syntax

⊃   **Example 1**

Drops the view named **SalesStaff with** the following syntax:
```
dmSQL> DROP VIEW SalesStaff;
```

or:
```
dmSQL> DROP VIEW IF EXISTS SalesStaff;
```

➲ **Example 2**

The following will not drop the view named **SalesStaff** when any synonym or view references it.

```
dmSQL> DROP VIEW SalesStaff RESTRICT;
```

3.67 END BACKUP

The END BACKUP command ends the backup state DBMaker places the database in during an online backup. Only users with DBA, SYSDBA or SYSADM security privileges can execute the END BACKUP command.

To perform an online full backup, start the database in NON-BACKUP, BACKUP-DATA, or BACKUP-DATA-AND-BLOB mode. To begin the backup, issue the BEGIN BACKUP command. Use operating system commands or backup utilities to back up all data and BLOB files to the backup device. After these files have been backed up, issue the END BACKUP DATAFILE command. Then use operating system commands or backup utilities to back up all Journal files. After these files have been backed up, issue the END BACKUP JOURNAL command to complete the backup and return the database to normal operation. Using an online full backup, can restore a database from the point in time the END BACKUP DATAFILE command was executed to the point in time the currently active Journal file was copied.

```
BEGIN BACKUP; //copy all data files by manually
END BACKUP DATAFILE; //copy all journal files by manually
END BACKUP JOURNAL; //thus, a full backup completed
```

To perform an online differential backup, start the database in NON-BACKUP, BACKUP-DATA, or BACKUP-DATA-AND-BLOB mode. Users can do the differential backup without the manual backup method. Please note, a differential backup is based on the most recent full backup and contains only the data that has changed since the time the differential base was created.

To perform an online incremental backup or an online incremental backup to current, the database must have been started in BACKUP-DATA or BACKUP-DATA-AND-BLOB mode.

Abort an online backup at any time by issuing the ABORT BACKUP command; for more information, see the ABORT BACKUP command. After executing the ABORT BACKUP command, the files from this backup may not be used to restore the database. Delete these backup files so they will not be confused with files from valid backups when you are restoring your database.

```
                                        ┌──── DATAFILE ────┐
●─────────── END BACKUP ───┤                  ├───────────●
                                        └──── JOURNAL ─────┘
```
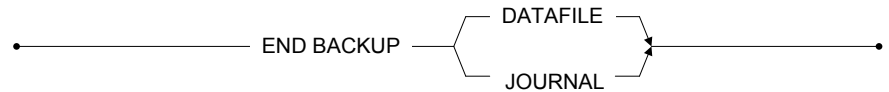
Figure 3-84 END BACKUP syntax

➲ **Example**

The following shows the steps involved in a full online backup. To begin, issue the **BEGIN BACKUP** command to notify DBMaker that a full backup is in progress, and then copy all data and BLOB files to the backup location. Once the files are copied, issue the **END BACKUP DATAFILE** command. Then copy all Journal files to the backup location. Once the files are copied, issue the **END BACKUP JOURNAL** command. Following this command the database will return to normal operation.

```
BEGIN BACKUP
    Copy data and BLOB files to backup location using OS commands
    Change backup mode if desired
    Abort the backup if desired
END BACKUP DATAFILE
    Copy Journal files to backup location using OS commands
    Change the backup mode if desired
    Abort the backup if desired
END BACKUP JOURNAL
```

3.68 EXECUTE COMMAND

The EXECUTE COMMAND executes a stored command. Use stored commands to quickly execute frequently used SQL data-manipulation statements without. Only a DBA, a SYSDBA, a SYSADM, or a user with the EXECUTE privilege may execute the EXECUTE COMMAND command.

A *stored command* is an SQL data-manipulation statement that is compiled and permanently stored in the database in an executable format. This permits repeated execution of the stored command without waiting for DBMaker to compile and optimize it. Stored commands are similar to stored procedures, except they can only contain a single command and cannot contain program logic.

Use host variables as placeholders for column values in the SQL statement of a stored command. This permits assigning actual values to the column executing the command, instead of when creating it. To use host variables in a stored command, replace any data or column value with a question mark symbol (?).

To execute a stored command that has host variables use constants: results from built-in functions, the NULL keyword, the DEFAULT keyword, or another host variable. Only use built-in functions that have no argument, RAND( ), PI( ), CURDATE( ), or NOW( ), when providing a value for a host variable. Use a NULL value for the host variable. The value represented by the host variable must be capable of accepting NULL values. The number of parameters provided when executing a stored command must equal the number of host variables in the command definition.

command_name ....Name of the stored command to execute

value ....Input parameter that corresponds to a host variable in the stored command
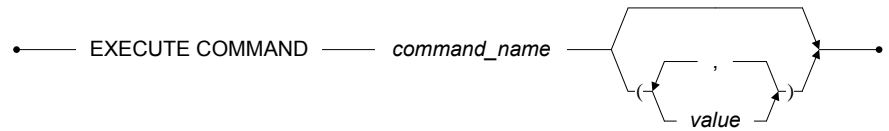
Figure 3-85 EXECUTE COMMAND syntax

➲ **Example 1**

The following executes the stored command named **sc_select**. This stored command has no input parameters.

```
dmSQL> EXECUTE COMMAND sc_select;
```

➲ **Example 2**

The following executes the stored command named **sc_input**; the command has two input parameters that provide a value.

```
dmSQL> EXECUTE COMMAND sc_input(10002, 10006);
```

3.69 GRANT (Execute Privileges)

The GRANT command grants execute privileges on executable database objects to individual users. Only the object owner or a user with DBA, SYSDBA or SYSADM security privilege can execute the command.

EXECUTE privileges control which executable database objects a user can use. DBMaker has three types of executable objects: stored commands, stored procedures, and projects.

The COMMAND keyword specifies the object as a stored command. Only users with all security and object privileges necessary to execute the SQL statement that makes up the stored command and the EXECUTE privilege may use this command.

The PROCEDURE keyword specifies an object being granted the EXECUTE privilege as a stored procedure. Only the EXECUTE privilege on the stored procedure is required.

The PROJECT keyword specifies an object being granted the EXECUTE privilege as a project containing one or more stored procedures. Granting EXECUTE privilege on a project automatically grants EXECUTE privileges on all procedures in that project.

The user who creates an executable database object is the owner of that object. The owner and any DBA, SYSDBA or SYSADM automatically have EXECUTE privileges on that object. To grant the EXECUTE privilege to all users grant the privilege to PUBLIC. All current and future users will then have the EXECUTE privileges on the executable database object.

executable_name ....Name of the executable object to grant execute privileges on

user_name ....Grant execute privileges to user *user_name*

group_name ....Grant execute privileges to group *group_name*
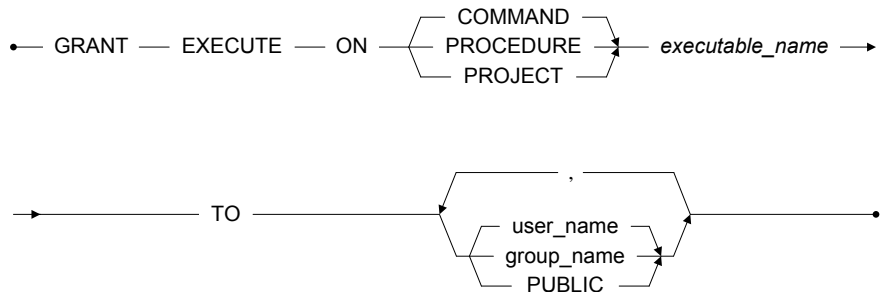
Figure 3-86 GRANT (Execute Privileges) syntax

➲ **Example 1**

The following grants the **EXECUTE** privilege on the stored command named **ListUserTables** to the user named **Vivian**.

```
dmSQL> GRANT EXECUTE ON COMMAND ListUserTables TO Vivian;
```

➲ **Example 2**

The following grants the **EXECUTE** privilege on the stored procedure named **ShowUsers** to the users named **Jenny** and **John**, and the group **Managers**.

```
dmSQL> GRANT EXECUTE ON PROCEDURE ShowUsers TO Jenny, John, Managers;
```

➲ **Example 3**

The following grants the **EXECUTE** privilege on all stored procedures in the **InternetFunc** to all users using the **PUBLIC** keyword.

```
dmSQL> GRANT EXECUTE ON PROJECT InternetFunc TO PUBLIC;
```

3.70   GRANT (Object Privileges)

The GRANT command grants access privileges on database objects to individual users. Only the object owner or a user with DBA, SYSDBA or SYSADM security privilege can execute the command.

Object privileges control which database objects a user can access and the actions they can perform. There are seven object privileges: SELECT, INSERT, DELETE, UPDATE, INDEX, ALTER, and REFERENCE. The keywords ALL and ALL PRIVILEGES can also be used to simultaneously grant privileges on an object.

- *SELECT* privilege selects data in a database object. It applies to the entire object and cannot be granted to specific columns.

- *INSERT* privilege inserts new data into a database object. It can be restricted to specific columns.

- *DELETE* privilege deletes data from a database object. It applies to the entire object and cannot be granted on specific columns.

- *UPDATE* privilege updates data in a database object. It can be restricted to specific columns.

- *INDEX* privilege creates an index on a database object. It applies to the entire object and cannot be granted on specific columns.

- *ALTER* privilege alters the schema of a database object. It applies to the entire object and cannot be granted on specific columns.

- *REFERENCE* privilege creates referential constraints, such as foreign keys, on a database object. It can be restricted to specific columns.

The user who creates a schema object is the owner of that object. The owner and any DBA, SYSDBA or SYSADM is automatically granted all object privileges. System catalog tables belong to a special virtual user called SYSTEM. All users including the SYSADM have only SELECT privilege on system catalog tables. Additional object privileges on the system catalog tables may not be added.

Privileges on specific columns and on the entire database object cannot be granted at the same time. Use the command twice, once to grant privileges on specific columns, and once to grant privileges on the entire table. It is possible to grant object privileges to all users simultaneously by granting the privileges to PUBLIC. All current and future users will then have those privileges for the database object.

column_name .... Name of the column to grant object privileges on

table_name .... Name of the table to grant object privileges on

user_name .... Name of the user to grant object privileges to

group_name .... Name of the group to grant object privileges to
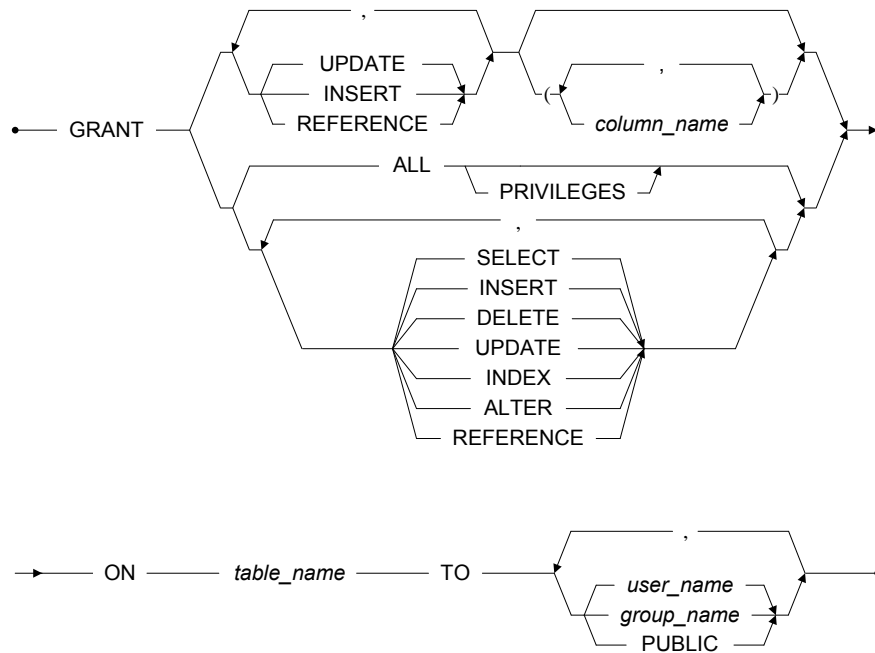
Figure 3-87 GRANT (Object Privileges) syntax

‌‍ **Example 1**

The following grants **SELECT**, **INSERT**, and **UPDATE** object privileges on the
Checks table to the user named **Vivian**.

```
dmSQL> GRANT SELECT, INSERT, UPDATE ON Checks TO Vivian;
```

‌‍ **Example 2**

The following grants **INSERT**, **UPDATE**, and **REFERENCE** privilege on the
Amount, **PayDate** columns of the **Checks** table to the user named **Jenny**.

```
dmSQL> GRANT INSERT, UPDATE, REFERENCE (Amount, PayDate) ON Checks TO Jenny;
```

➲ Example 3

The following grants all object privileges on the table **Checks** to the user named **John**.

```
dmSQL> GRANT ALL ON Checks TO John;
```

3.71 GRANT (Security Privileges)

The GRANT command creates new users or changes the security privileges of existing users. Only users with SYSADM or SYSDBA security privileges may execute the command. When creating a database DBMaker will create the SYSADM default user with no password. Change the SYSADM password immediately after creating the database to prevent unauthorized access. The SYSADM user is the only authorized user in the database until security privileges are granted to other users.

The SYSADM can grant CONNECT, RESOURCE, DBA, SYSDBA and ACCESS security privileges to a user. Granting CONNECT security privilege effectively adds a new user name to the database. Once a user name exists, the SYSADM may grant higher security to that user. Users with higher authority have all privileges of users with lower authority. Only users with SYSADM or SYSDBA security privilege can grant security privileges to other users. The SYSADM has all privileges of the SYSDBA authority level, and the SYSDBA authority must be granted by users with SYSADM authority.

CONNECT security privilege is necessary before a user can connect to a database. Once a user is granted the CONNECT security privilege they have been added to the database as a user. All users must be granted CONNECT security privilege before they can be granted any other security privileges. A user with CONNECT security privilege may create temporary tables in a database, or perform queries on any data they have been granted permission.

RESOURCE security privilege allows a user to create, alter, and drop tables, domains, and indexes. As the owner of any objects they create, users with RESOURCE privilege may grant and revoke object privileges to other users and create synonyms and views for any objects they own.

The DBA privilege has the same capabilities as the RESOURCE privilege, but may also create tablespaces and files. Users with the DBA privilege can also grant or revoke object privileges for schema objects owned by other users, except system schema objects.

The SYSDBA privilege has the same capabilities as the DBA privileges, but can also grant CONNECT, RESOURCE, DBA and ACCESS security privileges to a user, grant, change, or revoke privileges of objects owned by users with DBA authority, and change other users' passwords except SYSADM and other SYSDBA.

User names have a maximum length of 128 characters and passwords have a maximum length of 16 characters. Passwords can contain letters, numbers, the underscore character, and the symbols $ and #, but the first character cannot be a number.

The ACCESS/ALLOW privilege allows a user to connect to database from certain IPs. This can protect your database and avoid malicious connections. The IP is a standard Internet Protocol format. It only contains numbers and '*'.

The BLOCK privilege forbids a user to connect to database from certain IPs. This can protect your database and avoid malicious connections. The IP is a standard Internet Protocol format. It only contains numbers and '*'.

user_name................Name of the user to grant security privileges to

password ....Password of the user when connecting to the database

ip_address ....Address of the user to grant security privileges from
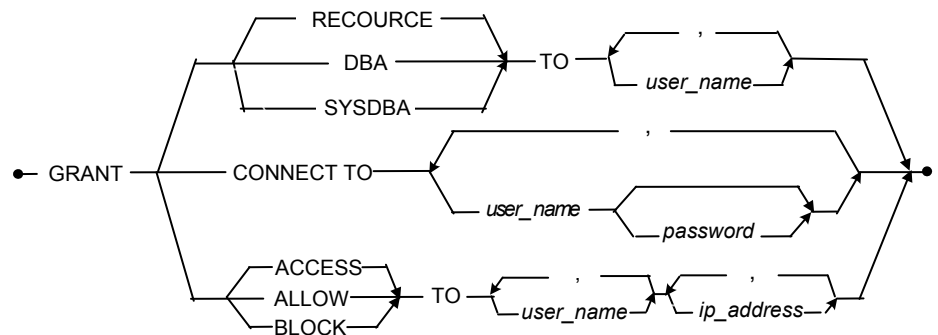
Figure 3-88 GRANT (Security Privileges) syntax

➲ **Example 1**

The following grants the **CONNECT** privileges to users named **vivian** and **jenny** with no password.
```
dmSQL> GRANT CONNECT TO vivian, jenny;
```

➲ **Example 2**

The following grants the **CONNECT** privilege to a user named **vivian** with the password **shuka828** and a user named **jenny** with the password **grala833**.
```
dmSQL> GRANT CONNECT TO vivian shuka828, jenny grala833;
```

➲ **Example 3**

The following grants the **RESOURCE** privilege to users **vivian** and **jenny**.
```
dmSQL> GRANT RESOURCE TO vivian, jenny;
```

➲ **Example 4**

The following grants the **DBA** privilege to users **vivian** and **jenny**.
```
dmSQL> GRANT DBA TO vivian, jenny;
```

➲ **Example 5**

The following grants the **ACCESS** privilege to the users **vivian** and **jenny** with the addresses **192.4.55.3** and **219.3.44.***.
```
dmSQL> GRANT ACCESS TO vivian,jenny '192.4.55.3','219.3.44.*';
```

3.72 INSERT

The INSERT command inserts new rows in a table. Rows may not be inserted into the system catalog tables. Only the table owner, a DBA, a SYSDBA, a SYSADM, or a user with the INSERT privilege for the entire table or for the specific column may execute the INSERT command.

Use this command to insert a single row by providing values using the VALUES keyword. The values provided may be constants, the results of built-in functions, or bound variables in a program using the ODBC API. Also, use this command to insert a set of rows using data selected from other tables using a SELECT statement. The rows selected must have columns with data types compatible the table.

When specifying columns to provide values for, name the columns in any order when executing the INSERT command. Omitting the column list specifies to use all columns, in the order created. In this case, provide a value for each column in the table, even if the value is empty. If the values provided do not match the data type of the column, DBMaker converts the values to the proper data type. The default value for a column is used when a value is not provided.

Use the referential integrity rules when inserting data into a child table that has a foreign key linking it to a parent table. Do not try to insert a value into a child key that does not exist in the parent key, unless it is a NULL value. Insert a new row in the parent key first.

To insert a string that contains a single quote, replace the single quote in the string with two consecutive single quotes. Have an even number of single quotes in a value, or DBMaker will wait for another single quote to close the string value. To insert the default value in a row, leave the value empty or specify the default value using the DEFAULT keyword.

OR REPLACE: The OR REPLACE option is optional. This option is used to ensure that DBMaker will replace the old row with the new row if the two rows have the same value for some columns. That is to say, if the row that users will insert into a table already exists in the table (judged with the primary key or the unique index), the

old row will be deleted from the table and then the new row will be inserted into the table, if not, the new row will be inserted directly into the table.

To use the option OR REPLACE, users must own privilege both INSERT and DELETE at the same time.

The INSERT statement with the option OR REPLACE would return a count to indicate the number of affected rows. This is the sum of the deleted rows and the inserted rows.

Please note that the option OR REPLACE makes sense only when the table owns the primary key or the unique index. If the primary key or the unique index both of which are used to confirm whether the new row duplicates the old row, does not exist, the INSERT statement with the option OR REPLACE becomes equivalent to that without the option OR REPLACE, and the new row will be inserted directly into the table when it is executed, which would result in duplicate records in the table. We do not recommend this option if the inserted rows are the query result of other tables and contain a large number of data, because this would reduce insertion efficiency.

In addition, we also do not recommend that users create the unique index on column containing a large number of data, because it not only does not work in quick query, but also may returns an error message 8332: 'expression or predicate needs too large memory´ when users execute the INSERT OR REPLACE statement to insert data.

table_name...............Name of the table to insert a new row into

column_name...........Name of the column to insert a value for

literal.......................Literal value to be inserted

constant...................Constant value to insert

bind_variable...........Name of the bound variable to insert, with ODBC only

select_statement.........Statement to be selected
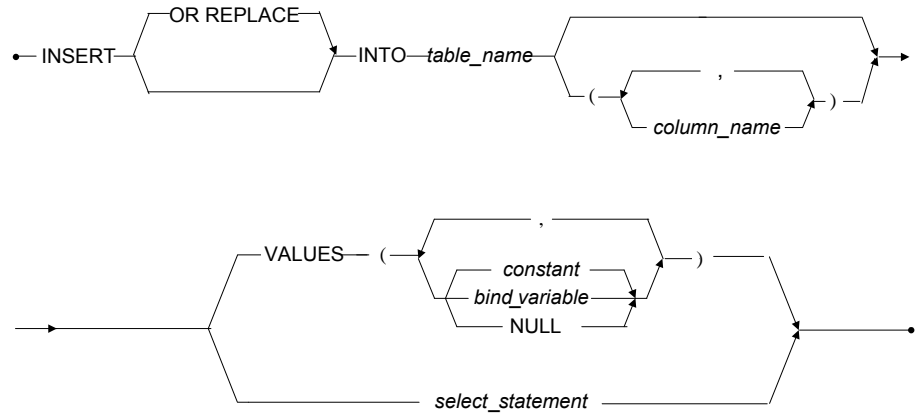
Figure 3-89 INSERT syntax

⊃ **Example 1**

The following inserts a row into the **Employeesinfo** table.

```
dmSQL> INSERT INTO Employeesinfo VALUES (1234, 'John', '01/01/1998', 2500);
```

⊃ **Example 2**

The following inserts values into **Emp_ID**, **FName**, and **HireDate** columns.

```
dmSQL> INSERT INTO Employeesinfo (Emp_ID, FName, HireDate)
    VALUES (1234, 'John', '01/01/1998');
```

⊃ **Example 3**

The following inserts rows into the **Employeesinfo** table that were selected from the **TempStaff** table where the **Emp_ID** column has values greater than **10567**.

```
dmSQL> INSERT INTO Employeesinfo (Emp_ID, FName, HireDate)
    SELECT Emp_ID, FName, HireDate FROM TempStaff WHERE Emp_ID > 10567;
```

⊃ **Example 4**

The following inserts a row into a **CHAR** column containing a single quote with the values inserted into all other columns set to the default value using the **DEFAULT** keyword.

```
dmSQL> INSERT INTO TB_TMP VALUES ('Joe''s Diner', DEFAULT, DEFAULT);
```

➲ **Example 5**

The following inserts a row into the **Employeesinfo** table that owns a primary key **Emp_ID** and a unique index **idx2** on column **FName**.

```
dmSQL> INSERT INTO Employeesinfo VALUES (1,'BB', '01/01/1986');
```

or
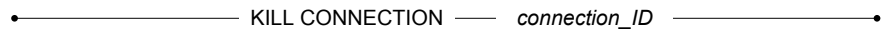
```
dmSQL> INSERT OR REPLACE INTO Employeesinfo VALUES (1,'BB', '01/01/1986');
```

3.73 KILL CONNECTION

The KILL CONNECTION command terminates a user connection to a database. Only a user with DBA, SYSDBA or SYSADM security privilege can execute the KILL CONNECTION command.

Executing this command frees all lock resources held by this user. Use this command when a user is holding resources needed by other users for high priority operations, or when the database administrator must shut down the database and not all users have logged off.

connection_ID.......... Connection number to kill

KILL CONNECTION ——— *connection_ID*

Figure 3-90 KILL CONNECTION syntax

➲ **Example**

The following kills the connection for the user connection ID **12345**.
```
dmSQL> KILL CONNECTION 12345;
```

3.74 LOAD STATISTICS

The LOAD STATISTICS command loads statistics from a text file containing statistical data for a DBMaker database. Create a statistics file for a database using the UNLOAD STATISTICS command. This file may be edited using any ASCII text editor and can be modified to provide any statistical data for testing or other purposes. Only users with DBA, SYSDBA or SYSADM security privileges can execute the LOAD STATISTICS command.

file_name ....Name of the file containing the statistical data to load

```
•————————— LOAD STATISTICS FROM ——— file_name ——————————•
```

Figure 3-91 LOAD STATISTICS syntax

● **Example**

The following example loads the statistics file **stat.dat** into the database.
```
dmSQL> LOAD STATISTICS FROM stat.dat;
```

3.75   LOCK TABLE

The LOCK TABLE command controls access to a table by other users. Only the table owner, a DBA, a SYSDBA, a SYSADM, or a user with the SELECT privileges (to lock the table in SHARE mode) or the UPDATE or DELETE privileges (to lock the table in EXCLUSIVE mode may execute this command.

This command locks a table in SHARE or EXCLUSIVE mode to control access to a table. SHARE mode allows other users read access to the table but denies write access; other users cannot insert, update, or delete rows if the table is locked in SHARE mode. EXCLUSIVE mode denies other users both read and write access. Other users cannot select, insert, update, or delete rows if the table is locked in EXCLUSIVE mode.

Use this command to reduce the number of locks acquired in a database operation. If the default lock level on a table is *page* or *row*, use this command to get a table level lock in order to avoid getting many lower level locks. In general, there is no need to do this since DBMaker automatically upgrades the lock level on a table if too many locks are acquired.

The WAIT/NO WAIT keywords are optional. These keywords specify whether DBMaker should wait to acquire a lock if the lock is not available immediately. If specifying the NO WAIT option, DBMaker does not wait to acquire a lock and returns an error message stating the lock could not be acquired. The amount of time DBMaker wait is determined by the **DB_LTimO** keyword in the **dmconfig.ini** file. The default value is *WAIT*.

table_name ....Name of the table to change the lock settings for
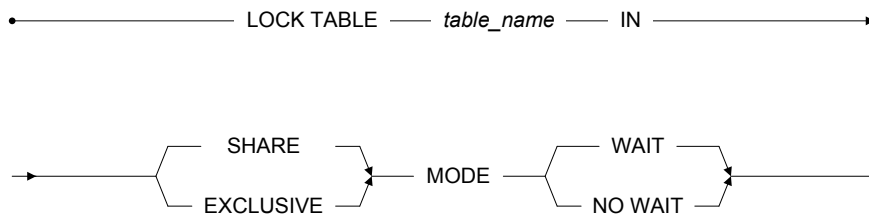
Figure 3-92 LOCK TABLE syntax

➔ **Example 1**

The following locks the **Employeesinfo** table in **SHARE** mode with the **WAIT** option.

```
dmSQL> LOCK TABLE Employeesinfo IN SHARE MODE WAIT;
```

➔ **Example 2**

The following locks the **Employeesinfo** table in **EXCLUSIVE** mode with the **NO WAIT** option.

```
dmSQL> LOCK TABLE Employeesinfo IN EXCLUSIVE MODE NO WAIT;
```

3.76   REBUILD COMMAND

The REBUILD COMMAND command rebuilds a stored command. Only users with DBA, SYSDBA or SYSADM security privileges can execute the REBUILD COMMAND command.

Rebuilding stored command can avoid stored command execution efficiency turn bad. For example, if users created a stored command on a table with few records, with the table records growing, the stored command execution efficiency will become worse.

Rebuild Stored Command function support Rebuild Stored Command Syntax and auto rebuild while update statistics.

command_name .... Name of the stored command to rebuild

Figure 3-93 REBUILD COMMAND syntax

➡ **Example**

The following example rebuilds the stored command named **sc_select**.
```
dmSQL> REBUILD COMMAND sc_select;
```

3.77   REBUILD INDEX

The REBUILD INDEX command rebuilds an existing index on a table. Only the table owner, a DBA, a SYSDBA, a SYSADM, or a user with the INDEX privilege for that table may execute the REBUILD INDEX command.

An *index* is a mechanism that provides fast access to specific rows in a table based on the values of one or more columns, known as the key. Indexes contain the same data as the key columns from the table they are based on, but the data is structured and sorted to make retrieval much faster than the table. Its' operation is transparent to users of the database. The DBMS uses the index to improve query performance whenever possible.

Rebuild an index for any table creating a denser unfragmented index and increasing efficiency.

index_name..............The name of index which need to rebuild

table_name...............The name of table which need to rebuild the index

Figure 3-94 REBUILD INDEX syntax

⮕ **Example**

The following example rebuilds the index named **NameIndex** from the **Employeesinfo** table.

```
dmSQL> REBUILD INDEX NameIndex FOR Employeesinfo;
```

3.78 REBUILD INDEX IN ANOTHER TABLESPACE

REBUILD INDEX IN ANOTHER TABLESPACE command rebuilds an index on a table in another tablespace, and the original index will automatically be deleted. Only the table owner, a DBA, a SYSDBA or a user with both ALTER and INDEX privileges for the table may execute the REBUILD INDEX IN ANOTHER TABLESPACE command.

NOTE   *Users cann't rebuild indexs for permanent tables in TMPTABLESPACE.*

NOTE   *Indexs on temporary tables only can be rebuilded in TEMTABLESPACE.*

NOTE   *Indexs on system tables only can be rebuilded in SYSTABLESPACE.*

index_name .... Name of the index to be rebuilt

table_name .... Name of the table which rebuilt index belongs to

tablespace_name .... Name of the tablespace which rebuilt index belongs to

●———— REBUILD INDEX ——— *index_name* —— FOR —— *table_name* —— IN —— *tablespace_name* ————●

Figure 3-95 REBUILD INDEX IN ANOTHER TABLESPACE syntax

➲ **Example**

The following example rebuild the index **NameIndex** on the table **Employeesinfo** stored in the tablespace **ts_mode** in another tablespace **ts_new**.

```
dmSQL> REBUILD INDEX NameIndex FOR Employeesinfo IN ts_new;
```

3.79 REBUILD TEXT INDEX

The REBUILD TEXT INDEX command rebuilds an IVF or signature text index for a table. This updates the text index to include new data. Only the table owner, a DBA, a SYSDBA, a SYSADM, or a user with the INDEX privilege for that table may execute the REBUILD TEXT INDEX command.

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 containing text. Text indexes contain a representation of all the text found in the text columns they are based on, but the data is encoded and structured to make retrieval much faster than directly from the table. An index operation is transparent to users. The DBMS uses the index to improve full-text query performance.

When loading data into a table, DBMaker does not update any text indexes on that table, thus loading all data before creating a text index. Rows containing matching text entered into a table after the text index was created will not be returned with the full-text query results. To include these rows in the search results, rebuild the text index using the REBUILD TEXT INDEX command.

The incremental option is the default setting for the REBUILD TEXT INDEX syntax. Incremental appends text entered into a table after the text index was created, thus making the text available to be returned with full-text query results. The full option rebuilds an entire text index by dropping and rebuilding the index based on a new full-text query.

text_index_name ......Name of the text index to rebuild

table_name...............Name of the table to rebuild the text index on

incremental ....creates a partial index and appends it to the current index

full...........................drops the current index and creates a new index

```
      •———  REBUILD TEXT INDEX —  text_index_name  ———  FOR ——————▶


      ————▶——  table_name  ———————————  INCREMENTAL  ——————
                                    ┌——————————————————————┐——————•
                                    └——————————  FULL  ————————————┘
```
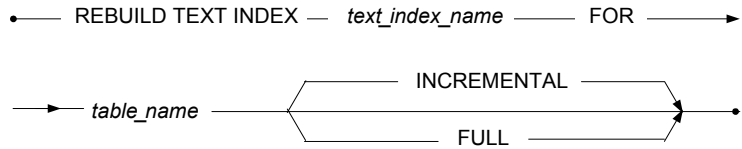
Figure 3-96 REBUILD TEXT INDEX syntax

➲ **Example**

The following rebuilds the text index named **TxtIdx** on the **Employeesinfo** table.

```
dmSQL> REBUILD TEXT INDEX TxtIdx FOR Employeesinfo;
```

3.80 REMOVE FROM GROUP

The REMOVE FROM GROUP command removes a user from an existing group. The user will lose all object privileges that have been granted to the group, but retain any privileges that have been granted to them directly. Only users with SYSADM, SYSDBA or DBA security privileges may execute the REMOVE FROM GROUP command.

Groups simplify the management of object privileges in a database with a large number of users. Use a group to organize users and/or groups. Any object privileges granted to the group are automatically granted to all members in the group.

Members added to a group after object privileges have been granted gain those object privileges in addition to the object privileges that have been granted to them directly.

Specify a group name in place of the user name, as long as the group you are trying to remove is not a part of the group that you are currently using. User and group names have a maximum length of 128 characters, and may contain letters, numbers, the underscore character, and the symbols $ and #. The first character may not be a number.

user_name................Name of the user to remove from the group

group_name ....Name of the group to remove the user from
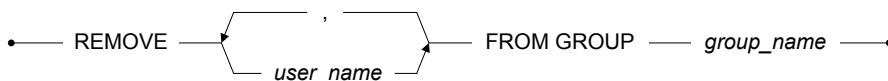
Figure 3-97 REMOVE FROM GROUP syntax

➲ **Example 1**

The following removes the user named **Vivian** from the group **SalesStaff**.

©Copyright 1995-2017 CASEMaker Inc.

```
dmSQL> REMOVE Vivian FROM GROUP SalesStaff;
```

➲ Example 2

The following removes the group named **NYSalesStaff** from the group named **SalesStaff**.

```
dmSQL> REMOVE NYSalesStaff FROM GROUP SalesStaff;
```

3.81 REMOVE TRACE

The REMOVE TRACE command removes trace from a single table that log the detaile OLD/NEW data. Only the user with table owner, a DBA, a SYSDBA, or a SYSADM security privileges can execute the REMOVE TRACE command.

table_name...............Name of an exisiting single table

```
•——————— REMOVE TRACE ——————— ON ——————————— table_name ———————•
```

Figure 3-98 REMOVE TRACE syntax

3.82 RESUME SCHEDULE

The RESUME SCHEDULE command resumes a suspended replication schedule for an asynchronous table. Only the local table owner, a DBA, a SYSDBA, or a SYSADM may execute the RESUME SCHEDULE command.

remote_database_name…. The name of remote database which need to resume the replication schedule

•—— RESUME SCHEDULE FOR REPLICATION TO —— *remote_database_name* ——•

Figure 3-99 RESUME SCHEDULE syntax

⊃ **Example**

The following resumes the replication schedule for the remote database named **DivOneDb**.

```
dmSQL> RESUME SCHEDULE FOR REPLICATION TO DivOneDb;
```

3.83 REVOKE (Execute Privileges)

The REVOKE command revokes execute privileges on executable database objects from individual users or groups. Only the object owner, a DBA, a SYSDBA or a SYSADM may execute the command.

Execute privileges control which executable database objects a user can use. DBMaker includes the stored command, stored procedure, and project executable objects.

The COMMAND keyword specifies revoking of the EXECUTE privilege on a stored command. Only users with all security and object privileges necessary to execute the SQL statement that makes up the stored command in addition to having EXECUTE privilege on the command may execute a stored command.

The PROCEDURE keyword specifies revoking of the EXECUTE privilege on a stored procedure. Only the EXECUTE privilege on the stored procedure is required to execute this command.

The PROJECT keyword specifies revoking of the EXECUTE privilege on a project containing one or more stored procedures. Revoking EXECUTE privilege on a project automatically revokes EXECUTE privileges on all procedures in that project.

Only the owner, a DBA, a SYSDBA or a SYSADM automatically have the EXECUTE privilege. It is possible to revoke EXECUTE privileges from all users simultaneously by revoking the privilege from PUBLIC. All current users will lose EXECUTE privileges on the executable database object.

executable_name.......Name of the executable object to revoke execute privileges on

user_name................Name of the user to revoke execute privileges from

group_name ....Name of the group to revoke execute privileges from
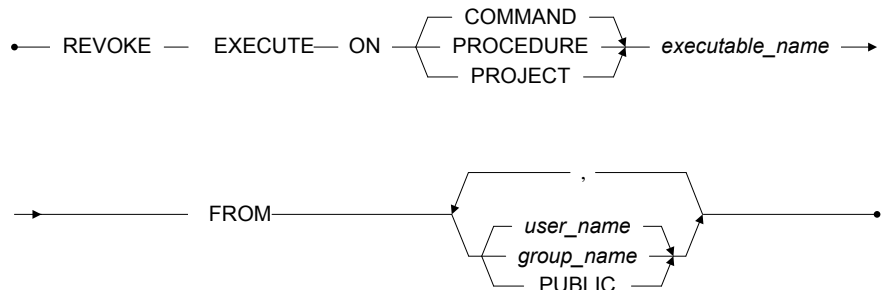
Figure 3-100 REVOKE (Execute Privileges) syntax

➲ **Example 1**

The following revokes **EXECUTE** privilege on the stored command named **ListUserTables** from the user named **Vivian**.

```
dmSQL> REVOKE EXECUTE ON COMMAND ListUserTables FROM Vivian;
```

➲ **Example 2**

The following revokes the **EXECUTE** privilege on the stored procedure named **ShowUsers** from the users named **Jenny** and **John**, and the group **Managers**.

```
dmSQL> REVOKE EXECUTE ON PROCEDURE ShowUsers FROM Jenny, John, Managers;
```

➲ **Example 3**

The following revokes the **EXECUTE** privilege on all stored procedures in the **InternetFunc** from all present and future users using the **PUBLIC** keyword.

```
dmSQL> REVOKE EXECUTE ON PROJECT InternetFunc FROM PUBLIC;
```

3.84 REVOKE (Object Privileges)

The REVOKE command revokes access privileges on database objects from individual users or groups. Only the object owner, a DBA, a SYSDBA or a SYSADM can execute the command.

Object privileges control which database objects a user can access and the actions they can perform. There are seven object privileges SELECT, INSERT, DELETE, UPDATE, INDEX, ALTER, and REFERENCE. The keywords ALL and ALL PRIVILEGES can also be used to simultaneously revoke all privileges on an object.

- *SELECT privilege-* permits selection of data in a database object, applies to the entire object and cannot be granted on specific columns.

- *INSERT privilege-* permits insertion of new data into a database object. The privilege can also be restricted to specific columns.

- *DELETE privilege-* permits the deletion of data from a database object, applies to an entire database object, and cannot be granted on specific columns.

- *UPDATE privilege-* permits updates of data in a database object. The privilege can also be restricted to specific columns.

- *INDEX privilege-* permits creation of an index for a database object, which cannot be granted on specific columns.

- *ALTER privilege-* permits altering the schema of a database object, applies to the entire object and cannot be granted on specific columns.

- *REFERENCE privilege-* permits creation of referential constraints, foreign keys, on a database object. The privilege can also be restricted to specific columns.

System catalog tables belong to a special virtual user called SYSTEM. All users including the SYSADM have only SELECT privilege on system catalog tables. Object privileges on the system catalog tables may not be revoked.

To privileges on specific columns and on the entire database object, use the command twice, once to revoke privileges on specific columns, and once to revoke privileges on

the entire table. It is possible to revoke object privileges to all users simultaneously by revoking the privileges from PUBLIC. All current users will then lose those privileges on the database object.

column_name ....Name of the column to revoke object privileges on

table_name ....Name of the table to revoke object privileges on

user_name...............Name of the user to revoke object privileges from

group_name ....Name of the group to revoke object privileges from
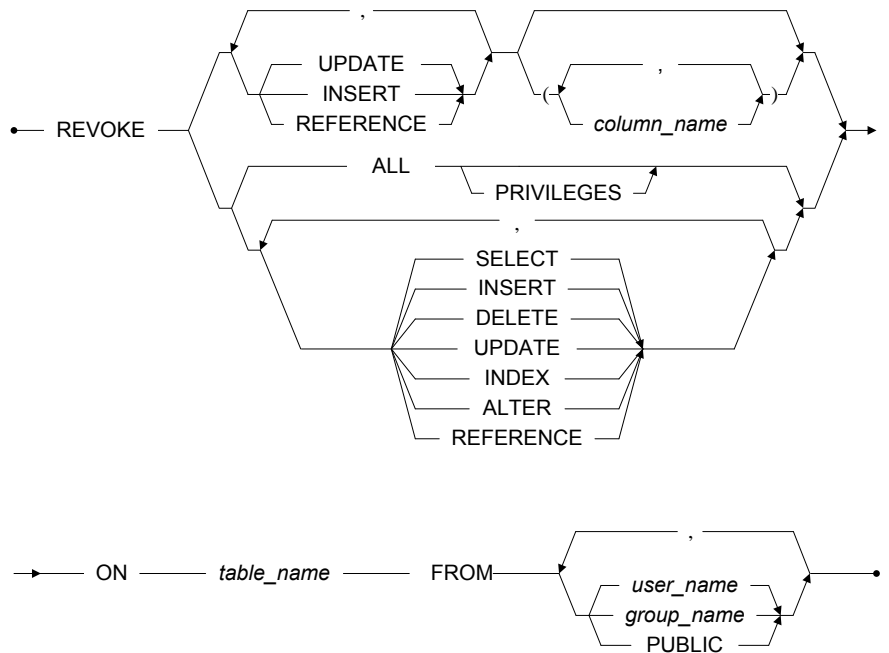
Figure 3-101 REVOKE (Object Privileges) syntax

⮑ **Example 1**

The following revokes the **SELECT**, **INSERT**, and **UPDATE** object privileges on the **Checks** table from the user named **Vivian**.

```
dmSQL> REVOKE SELECT, INSERT, UPDATE ON Checks FROM Vivian;
```

⮑ **Example 2**

The following revokes the **INSERT**, **UPDATE**, and **REFERENCE** object privileges on the Amount and **PayDate** columns of the **Checks** table from the user named **Jenny**.

```
dmSQL> REVOKE INSERT, UPDATE, REFERENCE (Amount, PayDate) ON Checks FROM Jenny;
```

⮑ **Example 3**

The following revokes all object privileges on the table **Checks** from the user named **John**.

```
dmSQL> REVOKE ALL ON Checks FROM John;
```

3.85   REVOKE (Security Privileges)

The REVOKE command removes a user from a database or changes the security privileges of a user. Only users with SYSADM or SYSDBA security privileges can execute the command

The SYSADM can revoke SYSDBA, DBA, RESOURCE, CONNECT and ACCESS privileges from a user. Revoking the CONNECT privilege effectively removes a user ID from the database. Once a user ID is removed, that user can no longer connect to the database. Revoking lower security privileges does not revoke higher ones, with the exception of the CONNECT security privilege. Revoking the CONNECT security privilege revokes all higher security privileges.

The SYSDBA privilege has all of the same capabilities as the DBA privilege, but it can execute the REVOKE command to revoke DBA, RESOURCE, CONNECT and ACCESS privileges from a user, except SYSADM and SYSDBA privilege. If revoking SYSDBA privilege from a user, it will retain the DBA privilege.

The DBA privilege has all of the same capabilities as the RESOURCE privilege, but may additionally create tablespaces and files. Users with DBA privileges can also grant or revoke object privileges for schema objects owned by other users, except for system schema objects.

The RESOURCE privilege allows a user to create, alter, and drop all tables, domains, and indexes. As the owner of any objects they create, users with RESOURCE security privilege may grant and revoke object privileges to other users and create synonyms and views for any objects they own.

The CONNECT privilege is necessary before a user can connect to a database. Once a user is granted a CONNECT privilege, they have been added to the database as a user. All users must be granted the CONNECT security privilege before they can be granted any other security privileges. A user with the privilege may create temporary tables in a database, or perform queries on any data to which they have been granted permission.

The ACCESS/ALLOW privilege allows a user to connect to database from certain IPs. This can protect your database and avoid malicious connections. The IP is a standard Internet Protocol format. It only contains numbers and '*'.

The BLOCK privilege forbids a user to connect to database from certain IPs. This can protect your database and avoid malicious connections. The IP is a standard Internet Protocol format. It only contains numbers and '*'.

To revoke all constraints of a user for the specified IP checking rule, use the "REVOKE ALLOW/BLOCK FROM user_name ALL" statement. ALL indicates all IP addresses.

If the REVOKE command is used to revoke RESOURCE, DBA or SYSDBA authority from a user, it will not take effect until the next time the user connects to the database.

user_name ....Name of the user to revoke security privileges from

ip_address ....Address of the user to revoke security privileges from
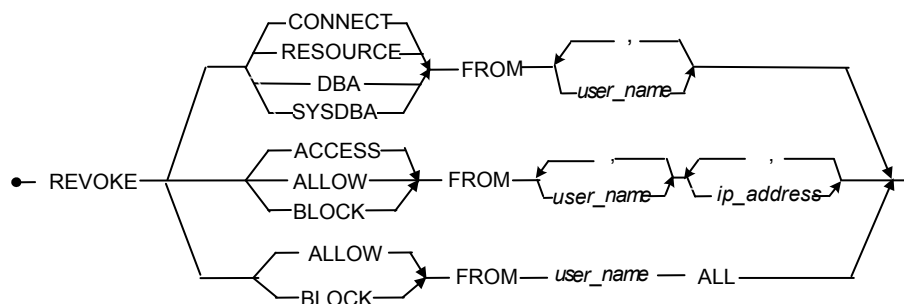
Figure 3-102 REVOKE (Security Privileges) syntax

● **Example 1**

The following revokes the **DBA** privilege from the users named **vivian** and **jenny**.

```
dmSQL> REVOKE DBA FROM vivian, jenny;
```

➲ **Example 2**

The following revokes the **RESOURCE** privilege from the users named **vivian** and **jenny**.

```
dmSQL> REVOKE RESOURCE FROM vivian, jenny;
```

➲ **Example 3**

The following revokes the **CONNECT** privilege from the users named **vivian** and **jenny**, revoking all privileges and removing the users from the database.

```
dmSQL> REVOKE CONNECT FROM vivian, jenny;
```

➲ **Example 4**

The following revokes the **ACCESS** privilege from the user named **vivian** and **jenny** with the addresses **192.55.3.4** and **219.5.3.***.

```
dmSQL> REVOKE ACCESS FROM Vivian,jenny '192.55.3.4','219.5.3.*';
```

3.86   ROLLBACK

The ROLLBACK command rolls back the current transaction to the beginning of the transaction or to a predefined savepoint. Any user with CONNECT or higher privileges can execute the command.

Use the ROLLBACK command to roll back all changes made by commands in a current transaction. Using the ROLLBACK command releases all locks acquired by a transaction. This command does not function while a database is running in the AUTOCOMMIT mode.

Also, use the ROLLBACK command to roll back a portion of the changes made by commands in a current transaction. Commands executed after the savepoint are rolled back, but no commands before the savepoint are. The transaction remains active and no locks are released.

savepoint_name ....Name of the savepoint to roll back to
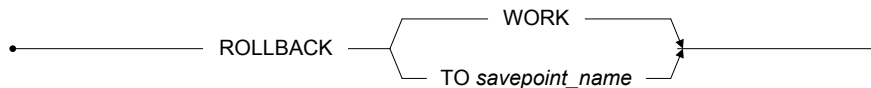
Figure 3-103 ROLLBACK syntax

⮞ **Example 1**

The following rolls back the entire active transaction, effectively aborting the transaction. All locks acquired by the transaction are released.

```
dmSQL> ROLLBACK WORK;
```

➲ **Example 2**

The following rolls back all commands executed after the savepoint, **SavePoint1**, but retains commands executed before the savepoint; the transaction remains active and locks are not released.

```
dmSQL> ROLLBACK TO SavePoint1;
```

3.87   SAVEPOINT

The SAVEPOINT command sets a savepoint in the current transaction and assigns a name. Only users with CONNECT or higher privileges can execute the SAVEPOINT command.

The SAVEPOINT command can be used in conjunction with the ROLLBACK command, to roll back a portion of the commands in a transaction. Specify a savepoint name in the ROLLBACK command and DBMaker rolls back all commands that were executed after the savepoint. The transaction remains active and locks acquired by the transaction are not released.

When specifying a savepoint name that does not exist, DBMaker rolls back the entire transaction and returns an error. The transaction is aborted and all locks acquired by the transaction are released. If trying to assign the same savepoint name twice in the same transaction, the first savepoint is canceled and the name is assigned to the second savepoint.

savepoint_name ....Name to assign to the savepoint

```
├──────── SAVEPOINT ──── savepoint_name ────────┤
```

Figure 3-104 SAVEPOINT syntax

➲ **Example**

The following sets a savepoint named **SavePoint1** in the active transaction.

```
dmSQL> SAVEPOINT SavePoint1;
```

3.88   SELECT

The SELECT command allows you to find, retrieve, and display data. Only the table owner, a DBA, a SYSDBA, a SYSADM, or a user with the SELECT privilege for that table can execute the SELECT command on a table.

The result of the SELECT command is a set of rows known as the result set, which meets the conditions specified. Specify the tables or views in a database to query; the condition data must meet to be returned in the result set, and the sequence in which the data in the result set is output. A SELECT statement can be a UNION of several single commands.

select .... SELECT clause lists the columns to retrieve data from

from .... FROM clause lists the tables the columns are located in

where .... WHERE clause specifies criteria return values must match

group by .... GROUP BY clause specifies groups for summary results

having .... HAVING clause specifies filter conditions for summary results

order by .... ORDER BY clause specifies the sort order

for browse .... FOR BROWSE clause specifies only shared locks should be acquired on the data in the query

into .... INTO clause specifies the table where the result will be inserted

limit .... LIMIT clause specifies the number of return records from offset *n* for the entire return set
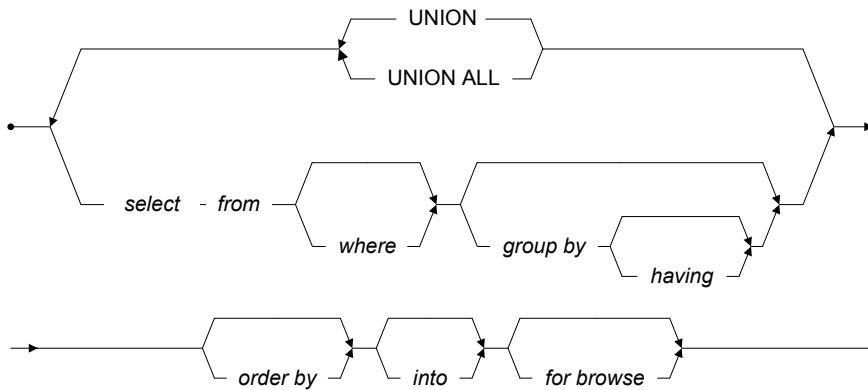
Figure 3-105 SELECT (using FROM) syntax

SELECT WITHOUT FROM

The SELECT without the use of the FROM syntax is used to get UDF or expression results. It does not require the user to use the FROM table clause in the query. Thus, the user cannot specify a column or table name in the SELECT without the use of the FROM query.

The following syntax cannot be used in conjunction with the SELECT without the use of the FROM syntax: WHERE, GROUP BY, HAVING, ORDER BY, DISTINCT, and UNION.

Figure 3-106 SELECT without the use of the FROM syntax

Example
```
dmSQL> SELECT ABS(100), COS(100.0);
```

SELECT Clause

The SELECT clause contains the SELECT keyword and the list of database objects or expressions to include in the result set. Use the ALL or DISTINCT keywords to indicate whether duplicate values should be returned. DBMaker returns all rows by default when either the ALL or DISTINCT keywords are not specified.

The value in the result list may be a column name, an expression, a constant, or an asterisk (*). An asterisk represents all columns from the source table. Optionally prefix a source name in front of the column name or asterisk.

Use any of the four basic types of expressions column, constant, function, and aggregate functions, in the select item list. If including a constant in the select list, the same value is returned for every row. An aggregate function returns one value for a set of rows. Aggregate functions are usually used in the GROUP BY clause.

Use the OID associated with each row in a table as a column name by using the name "OID" in the column list. The OID is essentially a hidden column whose value uniquely identifies each row in a database. The OID values are not necessarily sequential.

Use a display label to assign a temporary name to a column in the result set or to values generated by an expression that do not come from a column. Use the AS keyword to assign a display label to a column in the result set.

expression .... Expression that returns a value to include in the result set.

column_name .... Name of a column to retrieve data values from.

label .... Name for the result set column that is different from the original name for the source column.
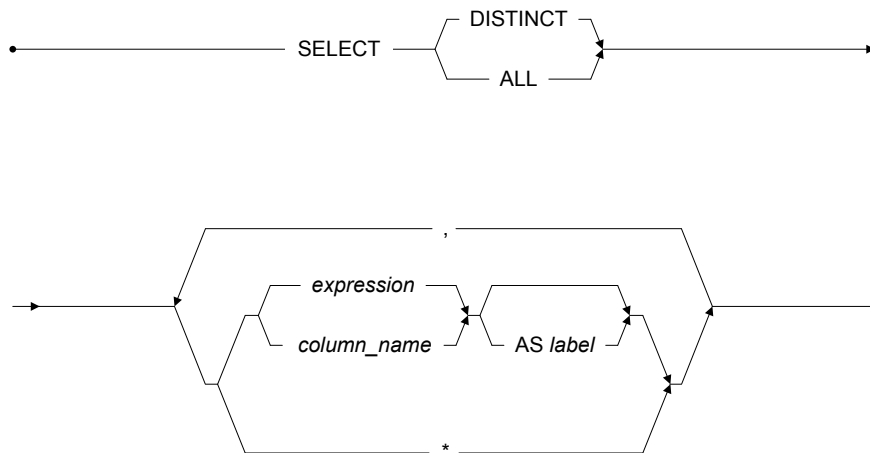
Figure 3-107 SELECT Clause syntax

FROM Clause

The FROM clause lists table sources and views used to select the data from. This identifies where the column name comes from if there are ambiguities. The source may be a table name, a view name a query result, or a synonym name. A source may be a single source, or an outer source which has the keyword OUTER followed by one or more single sources.

Supply a correlation name for a table name to refer to the table in other clauses of the SELECT statement. This may help make the statement more readable. Correlation names are especially useful with self-joins.

➲ **Example:**

The following query selects values from t2 that correspond to the maximum value from column c1 and groups them by values from c2. Finally, the result set is given the correlation name t3.

```
dmSQL> SELECT * FROM (SELECT MAX(c1) FROM t2 GROUP BY c2) AS t3 (c1);
```

Use the OUTER JOIN keyword OUTER, LEFT OUTER, JOIN, or LEFT JOIN to form outer joins. There can be more than one OUTER JOIN keyword in a SELECT statement. All sources before the OUTER keyword must be dominant sources. All of the sources after the OUTER JOIN keyword must be subservient sources. Specify all of the outer join table sequences in the FROM clause and specify the outer join factor in the WHERE clause. The entire join factor in the WHERE clause will be treated as the Outer Join factors. The other factors will be evaluated before the Outer Join factors.

DBMaker also support ANSI and ODBC outer join syntax to specify the outer join factors in the ON clause. The other factors in the WHERE clause will be evaluated after the outer join factors.

A CROSS JOIN specifies the cross product of two tables and returns the same rows as if no WHERE clause was specified in an old-style, non-SQL-92-style join. The result is same as if a user specified ',' in the FROM table_list.

➲ **Example**

```
dmSQL> SELECT * FROM t1 CROSS JOIN t2 CROSS JOIN t3 WHERE t1.c1 = t2.c1 AND t2.c2
= t3.c3;
```

The result is same as the following query:

➲ **Result**

```
dmSQL> SELECT * FROM t1,t2,t3 WHERE t1.c1 = t2.c1 AND t2.c2 = t3.c3;
```

In DBMaker 3.5 and later version, manually specify the type of scan to use in a query, and which index to use in a scan. In addition, the DBMaker query optimizer now automatically determines the most efficient type of scan to use, even if you have not recently updated database statistics.

source......................Name of the table to retrieve data from or query result.

alias ....Alternate name for the source used in other clauses
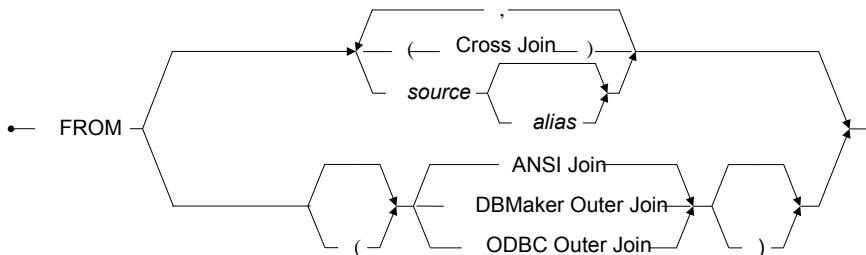
Figure 3-108 FROM Clause syntax

FORCED INDEX SCANS

Force an index scan with the following syntax.

```
table_name (INDEX [=] index_name [ASC|DESC])
```

The value of 0 can be used to force a table scan or the value 1 can be used to force a primary key index scan, may also be used.
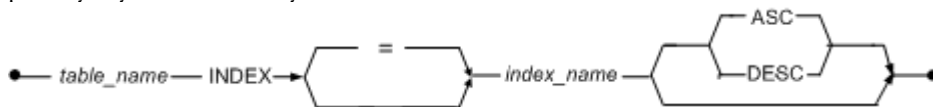
Figure 3-109 Force Index Scans syntax

➲ **Example 1**

To force a table scan specify the value **0**:

```
dmSQL> SELECT * FROM tb_tmp (INDEX=0);
```

➲ **Example 2**

To force an index scan on a primary key specify the value **1**:

```
dmSQL> SELECT * FROM tb_tmp (INDEX=1);
```

➲ **Example 3**

To force an index scan on the index **idx1**:

```
dmSQL> SELECT * FROM tb_tmp (INDEX idx1);
```

➲ **Example 4**

Allows the query optimizer to decide what type of scan to use on table **t1**, but forces an index scan on the **idx1** index for table **t2**:

```
dmSQL> SELECT * FROM t1, t2 (INDEX idx1);
```

FORCED INDEX SCAN AND "ALIAS"

General syntax used to force an index scan and provide an alias for the table:

```
table_name (INDEX [=] index_name) alias_name
```
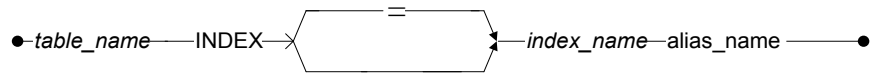

Figure 3-110 Force Index Scans and 'Alias' syntax

➲ **Example**

To force an index scan on the **idx1** index, and provides an alias for the table:

```
dmSQL> SELECT * FROM t1 (INDEX idx1) a, t1 b WHERE a.c1 = b.c1;
```

FORCED INDEX SCAN AND "SYNONYM"

General syntax used to force an index scan using a synonym:

```
synonym_name (INDEX [=] index_name)
```
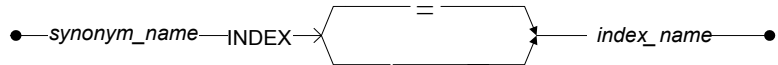

Figure 3-111 Force Index Scans and 'Synonym' syntax

➲ **Example**

To force an index scan on the **idx1** index using synonym **s1**:

```
dmSQL> SELECT * FROM s1 (INDEX idx1);
```

FORCED INDEX SCAN AND "VIEW"

General syntax used to force an index scan when creating a view:

```
view_name (INDEX [=] index_name)
```
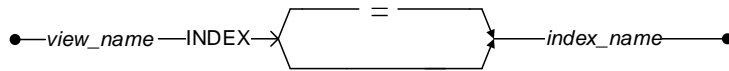

Figure 3-112 Force Index Scans and 'View' syntax

➲ **Example 1**

To force an index scan on the **idx1** index when creating view **v1**:

```
dmSQL> CREATE VIEW v1 as SELECT * FROM t1 (INDEX idx1);
```

You cannot force an index when selecting a view.

➲ **Example 2**

A wrong usage that will return errors:

```
dmSQL> SELECT * FROM v1 (INDEX idx1);
```

FORCED TEXT INDEX SCANS

General syntax used to force a text index scan:

```
table_name (TEXT INDEX [=] index_name)
```
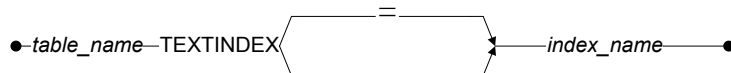

Figure 3-113 Force Text Index Scans syntax

➲ **Example**

To force a text index scan on the **tidx1**index:

```
dmSQL> SELECT * FROM t1 (TEXT INDEX tidx1);
```

SOURCE SUBCLAUSE

The *source* subclause used in the FROM clause may be either a table name or a result set from a query. To use the result set from a query, use the syntax provided in Figure 3-115.

Correlation_name .... Represents the result set of a subquery.
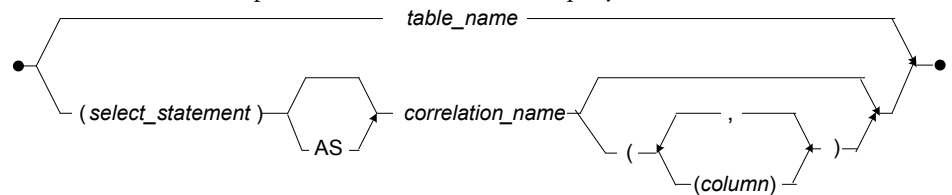
Figure 3-114 Source subclause syntax

WHERE Clause

Use the WHERE clause to specify the search condition and join criteria on the data being selected. If a row satisfies the search conditions, it is returned as part of the result set. Refer to the sub query topic to see how to use a SELECT statement, sub query, within a WHERE clause.

Use the percent symbol (%) and the underscore symbol (_) as wildcards in the quoted strings. The percent symbol matches zero or more characters, and the underscore symbol matches exactly one character. The ESCAPE clause is optional and permits the defining of an escape character in order to include the percentage sign and underscore characters in a quoted string without having them interpreted as wildcards. Use two consecutive single-quotes to include a single-quote character in a quoted string.

The predicate used in the WHERE clause may be a simple comparison using the following:

- *Relational Operators* — these may be one of the following: >, >=, <=, <, =, and <>. The relational operator condition is satisfied when the expression on either side of the relational operator fulfills the relation set up by the operator.

- *BETWEEN* — this comparison takes the form: *x BETWEEN y AND z*; the *BETWEEN* condition is satisfied when the value or expression to the left of the *BETWEEN* keyword lies in the inclusive range, denoted by the *AND* keyword, of the two expressions on the right of the keyword.

- *IN* — this comparison takes the form: *x IN (y, z, ...)*; the *IN* condition is satisfied when the value or expression to the left of the *IN* keyword is included in the list of values to the right of the keyword.

- *IS NULL* — this takes the form: *x IS NULL*; the *IS NULL* condition is satisfied when the value or expression to the left of the IS *NULL* keywords is a *NULL* value.

- *IS NOT NULL* — this takes the form: *x IS NOT NULL*; the *IS NOT NULL* condition is satisfied when the value or expression to the left of the *IS NOT NULL* keywords contains a value other than a *NULL* value.

- *LIKE* — this takes the form: *x LIKE 'y' ESCAPE 'z'*; the *LIKE* condition is satisfied when the string value or expression to the left of the *LIKE* keyword meets the criteria specified in the case-sensitive quoted string to the right of the keyword.

- *MATCH* — this takes the form: *x NOT CASE MATCH 'y'*; the *MATCH* condition is satisfied when the quoted string to the right of the *MATCH* keyword matches the entire string value or expression to the left of the keyword. The *NOT* keyword inverts the search results and *CASE* keywords keyword makes the search case-sensitive, both are optional.

- *CONTAIN* — this takes the form *x NOT CASE CONTAIN 'y'*; the *CONTAIN* condition is satisfied when the quoted string to the right of the *CONTAIN* keyword matches any part of the string value or expression to the left of the keyword. The *NOT* keyword inverts the search results and the *CASE* keywords makes the search case-sensitive, both are optional.

- *CONTAINS* – the contains operator's condition is satisfied when the concatenated string from concatenate columns matches the string pattern.

  Can use the syntax: [NOT] CONTAINS (column || column [|| column]…, 'string pattern'[, option string])

â **Example**

  The following select statement will select the record from c4 where both c1 and c4 contain the string '**Mail Server**'. The option **CASE** makes the search case-sensitive.

```
dmSQL> SELECT c4 FROM mcol WHERE CONTAINS(c1 || c4, 'Mail Server',
'CASE');
```
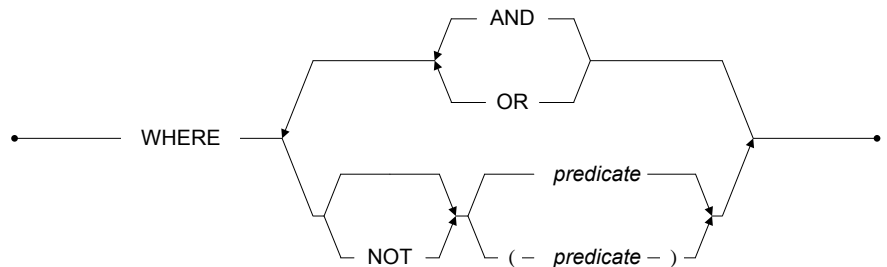

Figure 3-115 WHERE Clause syntax

CAST

CAST allows the output data to be converted to another data type. The chart below illustrates valid conversions. The table denotes the behavior of data types that are converted from row **X** to column **Y**.

The Numeric, Character, and Date/Time data types include multiple data types. Numeric data types include; integer (int, serial), smallint, float, double, and decimal. Character data types include char and varchar. Date/Time data types include; date, time, timestamp.

Xy	int (serial)	smallint	decimal	double	float	(var) char	(var) binary	date	time	timestamp	file	blob	clob
int(serial)	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N
smallint	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N
decimal	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N
double	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N
float	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N
(var)char	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N
(var)binary	N	N	N	N	N	Y	N	N	N	N	N	N	N
date	N	N	N	N	N	Y	N	Y	N	Y	N	N	N
time	N	N	N	N	N	Y	N	N	Y	N	N	N	N
timestamp	N	N	N	N	N	Y	N	Y	Y	Y	N	N	N
file	N	N	N	N	N	Y	Y	N	N	N	Y	N	N
blob	N	N	N	N	N	Y	Y	N	N	N	N	Y	Y
clob	N	N	N	N	N	Y	Y	N	N	N	N	Y	Y

Table 3-1 CAST Conversion Table

➲ **Example 1**

Use CAST( ) in a WHERE predicate.

```
dmSQL> SELECT * FROM t1 WHERE CAST(c1 AS CHAR(20)) LIKE '2001%';
```

➲ **Example 2**

Use CAST( ) in an expression.

```
dmSQL> SELECT CAST(c1+c2 AS CHAR(10)) FROM t1;
```

➲ **Example 3**

Use a nested CAST( ) statement.

```
dmSQL> SELECT CAST(CAST(123 AS CHAR(10)) || CAST(45 AS CHAR(10)) AS INT) FROM t1;
```

CASE

CASE is an SQL 99 function.
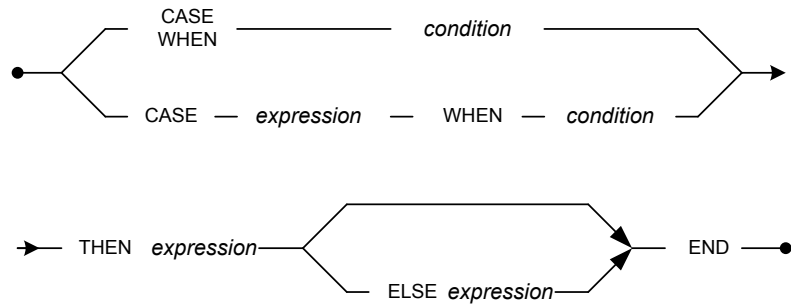
Figure 3-116 CASE Syntax

➲ **Example 1**

CASE WHEN p1 THEN v1 ELSE CASE WHEN p2 THEN v2 ELSE… ELSE v*n*
END…END. This means that if p1 is true then v1 else if p2 is true then v2 else…else
v*n*. This statement can be performed with the following:

```
dmSQL> SELECT CASE WHEN c1=3 THEN c2 ELSE CASE WHEN c1=5 THEN c3 ELSE c4 END END
FROM t1;
```

➲ **Example 2**

CASE c1 WHEN d1 THEN v1 ELSE CASE c1 WHEN d2 THEN v2 ELSE…ELSE
v*n* END…END. This means that if c1=d1 then v1 else if c1=d2 then v2 else…else v*n*.
This statement can be performed with the following:

```
dmSQL> SELECT CASE c1 WHEN 3 THEN c2 ELSE CASE c1 WHEN 5 THEN c3 ELSE c4 END END
FROM t1;
```

➲ **Example 3**

CASE WHEN p1 THEN v1 WHEN p2 THEN v2 WHEN…ELSE v*n* END. This
means that if p1 is true then v1 else if p2 is true then v2 else…else v*n*. This statement
can be performed with the following:

```
dmSQL> SELECT CASE WHEN c1=3 THEN c2 WHEN c1=5 THEN c3 ELSE c4 END FROM t1;
```

COALESCE

COALESCE is an SQL 99 function. COALESCE (v1, v2, v3, …, vn) is equivalent to "if v1 IS NOT NULL then v1 else if v2 IS NOT NULL then v3 else…………else vn".
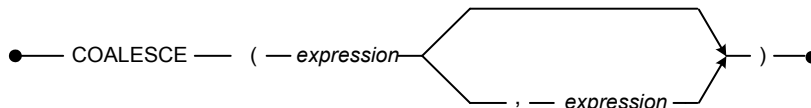
Figure 3-117 COALESCE Syntax

➲ **Example 1**
```
dmSQL> SELECT COALESCE(c1, 7) FROM t1;
```

➲ **Example 2**
```
dmSQL> SELECT COALESCE(c1, c2, c3, 7) FROM t1;
```

NULLIF

NULLIF is an SQL 99 function. NULLIF(v1, v2) is the equivalent to "if v1 = v2 then NULL else v1".

Figure 3-118 NULLIF Syntax

➲ **Example 1**
```
dmSQL> SELECT NULLIF(c1, 7) FROM t1;
```

➲ **Example 2**
```
dmSQL> SELECT NULLIF(t1.c1, t2.c1) FROM t1, t2;
```

IFNULL

IFNULL is an ODBC function. IFNULL (v1, v2) is the equivalent to coalesce(v1,v2) and it's equivalent to "if v1 is not null, then v1 else v2".

```
●──── IFNULL ─────── ( ─────── expression , expression ──── ) ───────●
```

Figure 3-119 IFNULL Syntax

➲ **Example 1**

```
dmSQL> SELECT IFNULL(c1, 7) FROM t1;
```

➲ **Example 2**

```
dmSQL> SELECT IFNULL(t1.c1, t2.c1) FROM t1, t2;
```

Compound Comparisons

Combine simple conditions with the logical operators AND, OR, and NOT to form compound conditions. Use the AND keyword to combine two search conditions which must be both true. Use the OR keyword to combine two search conditions when one or the other (or both) must be true. Finally, use the NOT keyword to select rows where a search condition is false.

➲ **Example 1**

```
dmSQL> SELECT * FROM Customer
       WHERE City NOT IN ('LA', 'NY') AND Age > 40;
```

➲ **Example 2**

```
dmSQL> SELECT * FROM Orders
       WHERE Price > 10,000 OR Ship_Date = TODAY;
```

Join Conditions

A *join condition* is a relational operators comparison on two columns where each column is from a different table (like: **Orders.CusNum = Customer.CusNum**).

Join two tables when creating a relationship with a join condition in the WHERE clause between columns from two tables. The effect of the join is to create a temporary composite table in which each pair of rows, one from each table, satisfying the join condition is linked to form a single row. There are four table join types, two-table-joins, multiple table-joins, self-joins, and outer-joins.

ON <SEARCH_CONDITION>

The ON <search_condition> specifies the condition on which the join is based. The condition can specify any predicate, although columns and comparison operators are often used.

➲ **Example**

```
dmSQL> SELECT ProductID, Suppliers.SupplierID
       FROM Suppliers JOIN Products
       ON (Suppliers.SupplierID = Products.SupplierID);
```

ANSI OUTER-JOIN

An outer join is a join of two or more tables with outer-join conditions for pairs of tables. An outer-join condition is a comparison, relational operators, on two columns from each table. All records of the left most table, will be returned and the result of the right table will be NULL if the outer-join condition is FALSE.

The following graph shows the ANSI JOIN and optimizer hint syntax:
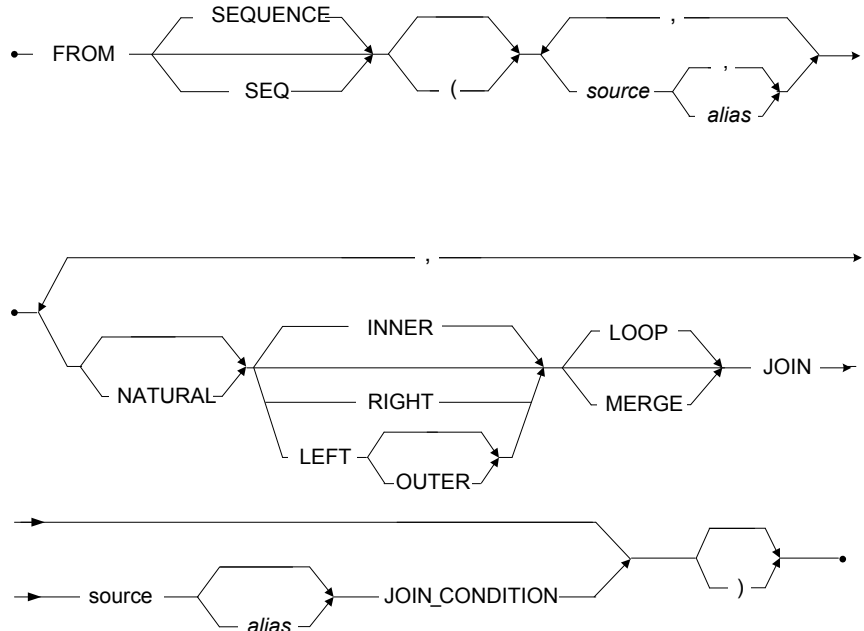
Figure 3-120 ANSI Join syntax
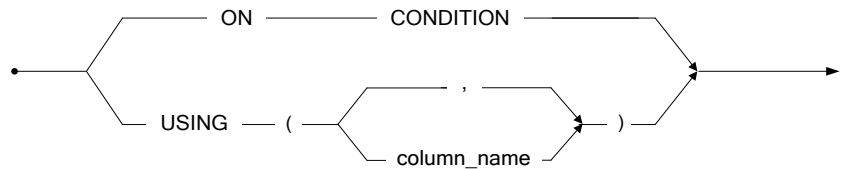
Figure 3-121 ANSI Join Condition syntax

The SEQUENCE, SEQ, LOOP and MERGE keywords are used as optimizer hints, it is not ANSI syntax. The optimizer will choose the execution plan if the specified

keyword could be used in the join execution. If it has no effect, the optimizer will not return any error message.

When the SEQUENCE/SEQ keywords are specified they force the join sequence like the table join order in the SQL command. The join table execution sequence will not be changed by the optimizer. This keyword will have no effect when used with an outer join.

➲ Example 1

```
dmSQL> SELECT * FROM SEQ t1 INNER JOIN t2 ON t1.c1=t2.c1 INNER JOIN t3 ON
t1.c2=t3.c2;
```

The LOOP/MERGE keywords specify the join execution method of the inner or outer join. The join execution order of the joined table will not be changed when specifying the join execution method. When the LOOP keyword is specified, the optimizer will use a nested join for the inner or outer join. When the MERGE keyword is specified, the optimizer will use the merge join for the inner and the outer join with equal join.

➲ Example 2

```
dmSQL> SELECT * FROM t1 INNER MERGE JOIN t2 ON t1.c1=t2.c1;
```

DBMAKER OUTER-JOIN

The following syntax is old DBMaker syntax. The difference with the ANSI outer-join syntax is the outer join factor is decided by the DBMaker optimizer. The RIGHT-JOIN is not supported with the following syntax and users cannot mix the following syntax with the ANSI outer-join syntax.
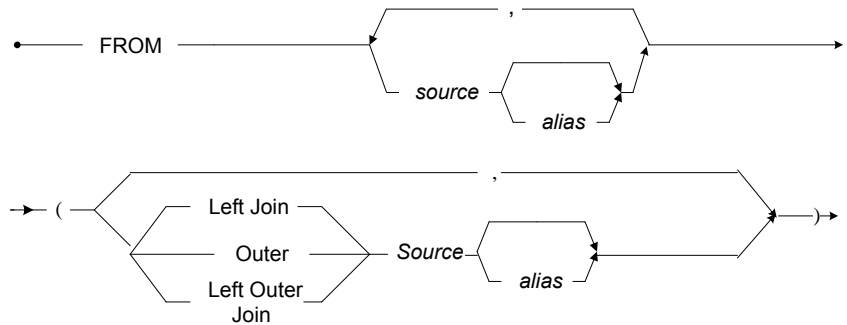
Figure 3-122 DBMaker Outer-Join Syntax

ODBC OUTER-JOIN

The ODBC Outer-Join uses the same syntax as the ANSI Outer-Join with the exception that all of the options must be used.
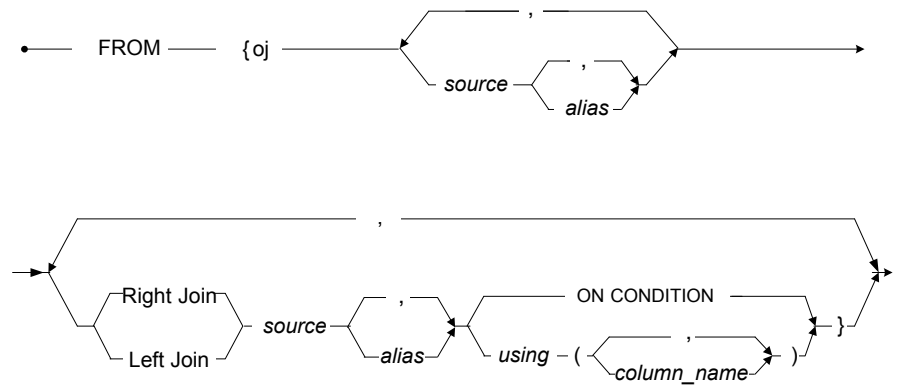
Figure 3-123 ODBC Outer-Join Syntax

SELF-JOIN

To join a table to itself, list the table name twice in the FROM clause and assign it two different aliases. Use the aliases to refer to each of the "two" tables in the **WHERE** clause. Suppose in the **Employeesinfo** table that there is a **Manager_ID** field, which is an employee ID for managers.

➲ **Example**

To list all of the employee's names together with their manager's name, join the **Employeesinfo** table with itself

```
dmSQL> SELECT e.FName AS Emp, m.Fname AS Manager
                        FROM Employeesinfo e, Employeesinfo m
                        WHERE e.Manager_Id = m.Emp_Id;
```

RIGHT-JOIN

Right-Join specifies that all rows from the right table not meeting the join condition to be included in the result set, and output columns that correspond to the other table are set to NULL, in addition to all rows returned by the inner-join.

➲ **Example**

```
dmSQL> SELECT * FROM t1 RIGHT JOIN t2 ON t1.c1 = t2.c1;
```

INNER-JOIN

The usage of INNER JOIN specifies that all matching pairs of rows be returned. It will discard unmatched rows from both tables. This is the default join type if only the JOIN keyword is specified in a query.

➲ **Example 1**

```
dmSQL> SELECT * FROM t1 INNER JOIN t2 ON t1.c1 = t2.c1 ;
```

➲ **Example 2**

```
dmSQL> SELECT * FROM t1 JOIN t2 ON t1.c1 = t2.c1;
```

➲ **Result**

```
dmSQL> SELECT * FROM t1, t2 WHERE t1.c1 = t2.c1;
```

NATURAL JOIN

When the NATURAL keyword is specified before the JOIN type, you cannot use the ON condition or USING column list to specify the join condition or the join column list. A NATURAL JOIN will perform an equal join on the common column name of the joined table. The result of the NATURAL JOIN is same as specifying all the common column names in the USING column list. The projection list for "select *" will be the joined column followed by the rest of the joined table's columns.

➲ **Example 1**
```
dmSQL> SELECT * FROM t1 NATURAL INNER JOIN t2;
```

➲ **Example 2**
```
dmSQL> SELECT * FROM t1 NATURAL LEFT JOIN t2;
```

ON CONDITION

The ON condition specifies the join condition for the joined table.

➲ **Example 1**
```
dmSQL> SELECT * FROM t1 INNER JOIN t2 ON t1.c1 = t2.c1;
```

➲ **Example 2**
```
dmSQL> SELECT * FROM t1 LEFT JOIN t2 ON t1.c1 = t2.c1;
```

USING COLUMN LIST

The USING column list is used to specify the joined column list of the joined table. When USING is specified, every column name specified in the USING column list should exist and be comparable in the joined table. The result will be the same as specifying an equal join with the columns in the ON clause. The projection list for "select *" will be the joined column followed by the rest of the joined table's columns.

➲ **Example 1**
```
dmSQL> SELECT * FROM t1 INNER JOIN t2 USING (c1, c2);
```

➲ **Example 2**
```
dmSQL> SELECT * FROM t1 LEFT JOIN t2 USING (c1) LEFT JOIN t3 USING (c1);
```

TWO TABLE-JOIN

A two-table join combines two tables with join conditions.

➲ **Example1**

The following is a two table-join, which combines the **Emp_Name** with the **Dept_Name** using **Dept_id**.

```
dmSQL> SELECT FName, Dept_Name FROM Employeesinfo, Department
                    WHERE Employeesinfo.Dept_ID = Department.Dept_Id;
```

➲ **Example2**

The following is a two table outer join which selects all records of the **Department** table and produce **NULL** for the project that does not belong to this department

```
dmSQL> SELECT Dept_id, Dept_Name, Proj_Name FROM Department d outer Project p
                              WHERE d.Dept_id = e.Dept_Id;
```

MULTIPLE TABLE-JOIN

A multiple table-join is a join of more than two tables with join conditions for pairs of tables. A join condition is a comparison, relational operators, on two columns from each table.

➲ **Example**

The following is a three table-join, which selects all the projects engaged by the **employeesinfo** in the **Engineering department**.

```
dmSQL> SELECT Dept_Name, Proj_Name FROM Department d, Project p, Employeesinfo e
                        WHERE d.Dept_id = e.Dept_Id AND
                              p.Emp_Id = e.Emp_Id AND
                              Dept_Name = 'Engineering';
```

FORCED LOOP JOIN (NESTED JOIN)

General syntax used to force a Nested Join between two tables:

```
table_name { INNER | OUTER } LOOP JOIN table_name
```
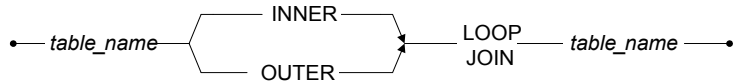
Figure 3-124 Force Loop Join Syntax

A forced join of this type must use INNER JOIN or OUTER JOIN syntax.

➔ **Example 1**

```
dmSQL> SELECT * FROM t1 INNER LOOP JOIN t2 ON t1.c1=t2.c1;
```

➔ **Example 2**

```
dmSQL> SELECT * FROM t1 OUTER LOOP JOIN t2 ON t1.c1=t2.c1;
```

FORCED MERGE JOIN

General syntax used to force a Merge Join between two tables:

```
table_name { INNER | OUTER } MERGE JOIN table_name
```
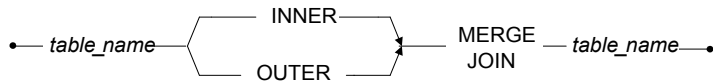

Figure 3-125 Force Merge Join Syntax

When join cannot use Merge Join then a Force Merge Join is useless, however, an error message is not returned.

➔ **Example 1**

```
dmSQL> SELECT * FROM t1 INNER MERGE JOIN t2 ON t1.c1=t2.c1;
```

➔ **Example 2**

```
dmSQL> SELECT * FROM t1 OUTER MERGE JOIN t2 ON t1.c1=t2.c1;
```

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] table_name_list
```
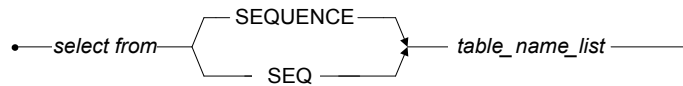
Figure 3-126 Force Join Sequence Syntax

➲ **Example 1**
```
dmSQL> SELECT * FROM SEQUENCE t1, t2, t3 WHERE t1.c1=t2.c1 AND t2.c2=t3.c2;
```

➲ **Example 2**
```
dmSQL> SELECT * FROM SEQ t1 INNER JOIN t2 ON t1.c1=t2.c1 INNER JOIN t3 ON
t1.c2=t3.c2;
```

GROUP BY Clause

Use the **GROUP BY** clause to produce summary data within a group. A group is a set of rows that have the same values of group by columns. A single row of aggregate results is produced for each group. The column to group results by is identified by column name or display label.

Using the GROUP BY clause restricts can be entered in the SELECT clause. A select item in a group by query must be one of the following:

• An aggregate function used to produce a single value to summarize the rows contained in a group

• A grouping column, which is listed in the GROUP BY clause

• A constant

• An expression involving an above combination

In practice, a GROUP BY query always includes both a grouping column and an aggregate function. Each row that contains a null value in a column, specified by the GROUP BY clause, belongs to a single group; all null values are grouped into one group.

The USING HASH/SORT clause is used as optimizer hint syntax. When the USING HASH is specified, the optimizer will choose the hash method for GROUP BY execution. The optimizer will not choose the hash method when there are too many groups for GROUP BY when specifying USING HASH. When USING SORT is specified, the optimizer will try to use an index scan if there are any indexes with the same column as the GROUP BY clause or it will choose the execution plan to sort by the GROUP BY column when executing GROUP BY.
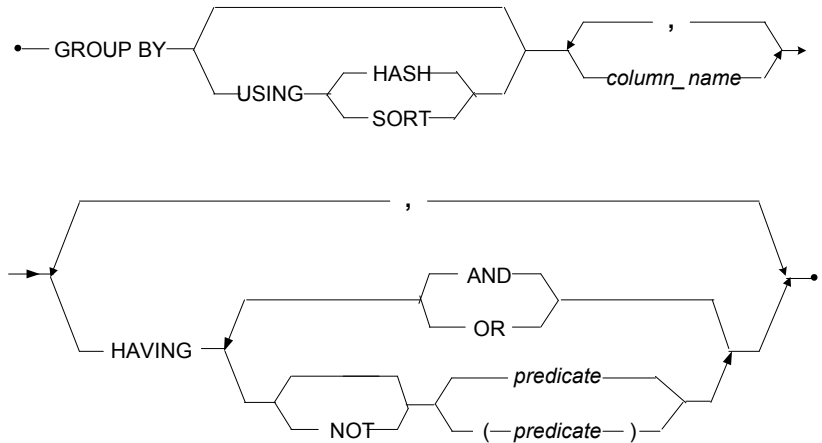
Figure 3-127 GROUP BY Clause syntax

➲ **Example 1**

The following uses **SELECT** to retrieve **Dept_Id** and **AVG**(**salary**) for each employee and then adds the employees **AVG**(**salary**) to **ID 1** to get an average salary for the entire group.

```
dmSQL> SELECT Dept_Id, AVG(Salary) FROM Employeesinfo
                    GROUP BY Dept_Id;
dmSQL> SELECT Dept_Id AS ID1, AVG(Salary) FROM Employeesinfo
                    GROUP BY ID1;
```

⮞ **Example 2**

The following uses **SELECT** to retrieve **Dept_Id** and **AVG(salary)** for each department in the **Employeesinfo** table by HASH method.

```
dmSQL> SELECT Dept_Id, AVG(Salary) FROM Employeesinfo
                    GROUP BY Dept_Id USING HASH;
```

FORCED GROUP BY METHOD

General syntax used to force a Join Sequence:

```
GROUP BY column_name_list [USING SORT | USING HASH] having .....
```
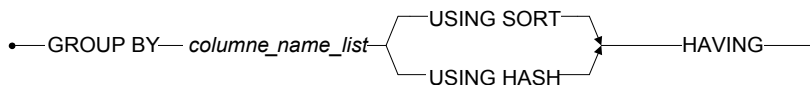

Figure 3-128 Force Group by Method Syntax

⮞ **Example 1**

```
dmSQL> SELECT c1,c2,COUNT(*) FROM tb_test GROUP BY c1,c2 USING HASH;
```

⮞ **Example 2**

```
dmSQL> SELECT c1,c2,COUNT(*) FROM tb_test GROUP BY c1,c2 USING SORT HAVING
SUM(c3)>0;
```

HAVING Clause

The **HAVING** clause is used to select or reject a group. A sub-query can appear in the having clause. Refer to the *SUBQUERY* section for more information.

⮞ **Example**

The following example shows the average sales amount for departments with total sales exceeding one million dollars.

```
dmSQL> SELECT Dept_Name, AVG(Amount) FROM Sales
                        GROUP BY Dept_Name
                        HAVING SUM(Amount) > 1000000;
```

ORDER BY Clause

The result rows of a query are not arranged in any particular order. Use the ORDER BY clause to sort query results by the values contained in one or more columns.

The ASC/DESC keywords specify the sort order of the results as ascending, smallest value first, or descending order. The default order is ascending. NULL values are treated as larger that non-null values for sorting purposes. Using the ASC keyword to specify sort order, NULL values would come after any non-null values.

column_name .... Name of the column or display label in the SELECT list to sort

the query results by

column_number .... Integer that represents the placement of a column or expression

in the SELECT list

expression .... To sort the result query by a specified expression
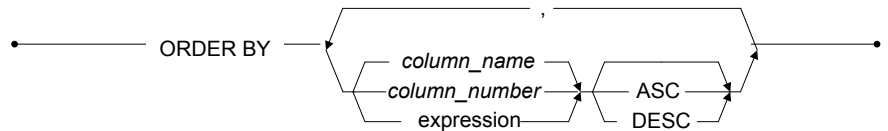
Figure 3-129 ORDER BY Clause syntax

➲ **Example 1**

The following sorts the results by name in ascending order by default, and age in descending order.

```
dmSQL> SELECT Name, Address, Age FROM Customer
                  ORDER BY Name, Age DESC;
```

➲ **Example 2**

The following uses a column number and display label in the ORDER BY clause.

```
dmSQL> SELECT Dept_Id, Salary + Bounce AS Total_Com, FName
                    FROM Employeesinfo
                    ORDER BY 1, Total_Com;
```

UNION OPERATOR

Use the UNION operator to combine the results of two or more queries into one result. Duplicate rows are removed from the combined results when using the UNION operator and the combined results have distinct values for each row. If certain that no duplicate rows exist in individual results, or to keep duplicate rows, use the UNION ALL keywords. UNION ALL keeps the rows from individual result sets and is faster than the UNION operator.

There are restrictions on results that can be combined by a UNION operator:

- The two results need to contain the same number of columns.

- The corresponding items in each result must have compatible data types, not the same column names. The column name of the first result becomes the column name of the combined result.

- Use an *ORDER BY* clause following the last *SELECT* clause and refer to the ordered column by its position in the *SELECT* list column number.

➲ **Example 1**

The following shows the use of the **UNION** clause in a **SELECT** statement.

```
dmSQL> SELECT C1, C2 FROM T1
        UNION
        SELECT C3, C4 FROM T2
        ORDER BY 2;
```

➲ **Example 2**

The following example shows the use of the **UNION ALL** clause in a **SELECT** statement.

```
dmSQL> SELECT 'MOVIE', Event FROM Entertainment WHERE Type = 'MOVIE'
         UNION ALL
```

```
SELECT 'BOOK', Name FROM MyBook;
```

SUB-QUERIES

A sub-query is a query that appears within the WHERE or HAVING clause of another SQL statement. A sub query is always enclosed in parentheses, but otherwise it has the same form of a SELECT statement.

A sub-query must produce a single column of data as its query result. In addition, when the query result is used in a simple relational operator comparison, the sub query must only create a single row value.

➲ **Example**

The following is a sub query selects employees whose salary is greater than the average.

```
dmSQL> SELECT Name FROM Employeesinfo
       WHERE Salary > (SELECT AVG(Salary) FROM Employeesinfo);
```

IN SUB-QUERY

The IN sub-query is a membership test. It is true if the value of the expression matches one or more of the values selected by the sub query. In the IN, membership test the sub query may return more than one row of one column data.

➲ **Example**

The following selects all the **employees** whose department is located in **NY**.

```
dmSQL> SELECT FName FROM Employeesinfo
       WHERE Dept_Id
          IN (SELECT Dept_Id FROM Department WHERE City = 'NY');
```

EXISTS SUB-QUERY

The existence test checks whether a sub query produces any rows. In a sub-query, sometimes it is necessary to refer to the value of a column in the "current" row of the main query. This is called an *outer reference*. The **d.Dept_id** column in the example is an outer-reference. There can be multiple levels of sub-queries, and the outer reference can refer to the columns of tables in any outer-level sub-query.

➲ **Example**

The following lists all departments with at least one **EMPLOYEE** in that **Department** whose salary exceeds $**500,000**.

```
dmSQL> SELECT Dept_Name FROM Department d
       WHERE EXISTS
             (SELECT Dept_Id FROM EMPLOYEESINFO e
                   WHERE e.Salary > 500000 AND d.Dept_Id = e.Dept_Id);
```

ANY/ALL/SOME Sub-query

Use the ALL keyword in a sub query. The search condition is true if the comparison is true for every value returned. If the sub query returns no value, an empty set, the condition is true. If there is a NULL in the returning set, the condition is false.

Use the ANY keyword in a sub query. The search condition is true if the comparison is true for at least one of the value returned. If the sub query returns no value, the condition is false.

➲ **Example**

The following example selects non-manager employees with a **Salary** greater than at least one **Manager**.

```
dmSQL> SELECT FName FROM Employeesinfo
       WHERE Manager = 'N' AND Salary > ANY
             (SELECT Salary FROM EMPLOYEESINFO WHERE Manager = 'Y');
```

FOR BROWSE Clause

The FOR BROWSE keywords designate the browse mode to be used in the selection. In browse mode, no locks are acquired so other users do not block the selection. Since no locks are acquired, the read is not guaranteed to be repeatable. Browse mode is useful for browsing data or producing reports.
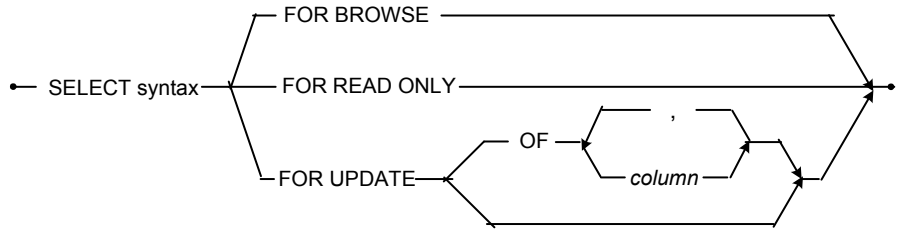
Figure 3-130 FOR BROWSE Clause syntax

LIMIT

LIMIT specifies the number of returned records from offset *n* for the entire return set.

offset .... Offset from the first returned records in the result set

rows .... The number of returned rows
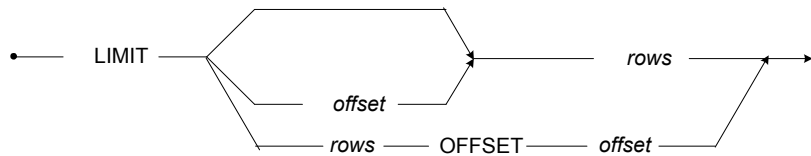
Figure 3-131 LIMIT syntax

● **Example**

```
dmSQL> SELECT * FROM tb_test ORDER BY c1 LIMIT 10;
```

Aggregate Functions

Aggregate functions compute a single result from a set of input values. DBMaker supports the following built-in aggregate functions:

- MIN

- MAX

- AVG

- COUNT

- SUM

- XMLAGG

The MIN function returns the minimum value of all input values.

The MAX function returns the maximum value of all input values.

The AVG function returns the average (arithmetic mean) of all input values.

The COUNT function returns the number of records which meet the set standards.

The SUM function returns the sum of all input values.

The XMLAGG function returns the concatenation of XML values.

The syntax is as follows:

```
{AVG|MAX|MIN|SUM|XMLAGG} ([ALL|DISTINCT] expression
[,comparsion_predicate])
|COUNT (* [,comparsion_predicate])
|COUNT ([ALL|DISTINCT] expression [,comparsion_predicate])
```

Comparison_ predicate.......expression with comparison_operator.
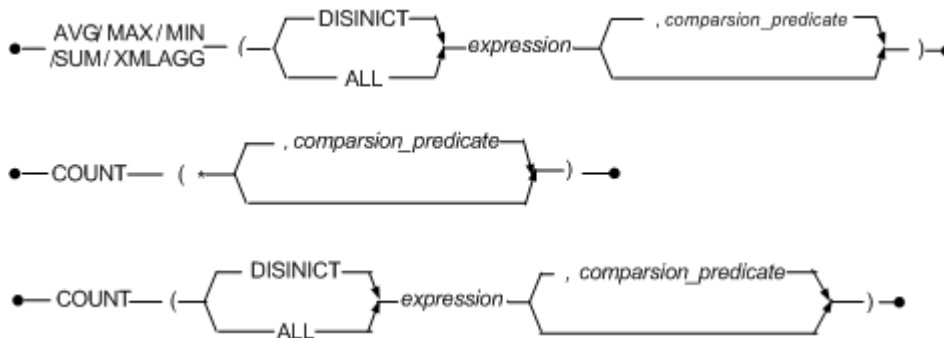
Figure 3-132 AGGREGATE FUNCTION syntax

➲ **Example**

```
dmSQL> SELECT COUNT(*) FROM tb_test;
```

An aggregate function can only be used as an aexpression in the result list, HAVING clause, GROUP BY clause and ORDER BY clause of a SELECT command. It is forbidden in other clauses, such as WHERE clause.

WINDOW Functions

A window function performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. But unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row.

The syntax is as follows:

```
func_name() OVER ([PARTITION_BY_CLAUSE] ORDER_BY_CLAUSE)
```

PARTITION_BY_CLAUSE……. specify the columns used to divide the result set into partitions. The window function is applied to each partition separately and computation restarts for each partition.

ORDER_BY_CLAUSE…….. specify the columns used to specify the order to apply the window function.
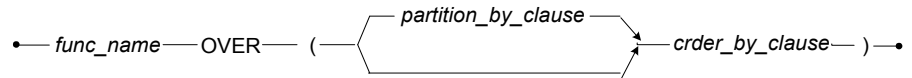
Figure 3-133 WINDOW FUNCTION syntax

DBMaker supports the following window functions:

- ROW_NUMBER

- RANK

- DENSE_RANK

The row_number function returns the sequential number of a row within a partition of a result set, starting at 1 for the first row in each partition. The returned type is BIGINT.

The rank function returns the rank of a value in a group of values. It is very similar to the dense_rank function. However, the rank function can cause non-consecutive rankings if the tested values are the same. Whereas, the dense_rank function always result in consecutive rankings. The two functions returned type is BIGINT.

⊃ **Example**

The following example shows ROW NUMBER,RANK and DENSE_RANK of the sale quantity for the book category.

```
dmSQL> SELECT TITLE, BOOK_CATEGORY, SALE_QTY,
       ROW_NUMBER() OVER (PARTITION BY BOOK_CATEGORY ORDER BY SALE_QTY) AS
       ROW_NUMBER,
       RANK() OVER (PARTITION BY BOOK_CATEGORY ORDER BY SALE_QTY) AS RANK,
       DENSE_RANK() OVER (PARTITION BY BOOK_CATEGORY ORDER BY SALE_QTY) AS
       DENSE_RANK
       FROM BOOK_STORE;
```

The result as following:

TITLE	CATEGORY	SALE_QTY	ROW_NUMBER	RANK	DENSE_RANK
book3	business	20	1	1	1
book2	business	30	2	2	2
book1	business	40	3	3	3
book1	computer	10	1	1	1
book2	computer	20	2	2	2
book3	computer	20	3	2	2
book4	computer	30	4	4	3

There are some restrictions on using the three WINDOW functions:

• ORDER BY CLAUSE cannot use order by constant.

⊃ **Example**
```
dmSQL> SELECT row_number() OVER (ORDER BY 1) FROM t1;
```

• OVER clause must be the same for all window function used in the query.

➲ **Example**

```
dmSQL> SELECT row_number() OVER (ORDER BY c1), row_number() OVER (ORDER
BY c2) FROM t1;
```

• GROUP BY or aggregate function is not supported with window function.

➲ **Example**

```
dmSQL> SELECT max(c1), row_number() OVER (GROUP BY c1) FROM t1;
```

• Window function are not supported in the INSERT, DELETE or UPDATE.

• Window function are not supported in the WHERE clause or subquery.

XML Functions

The xml function is a set of functions which produce xml content from SQL data.

DBMaker support XML functions, which are parts of the SQL statement. Users can use these functions via dmsql, odbc or jdbc interface.

DBMaker supports the following XML functions:

• xmlelement

• xmlforest

• xmlagg(xml)

• xmlcomment(text)

The syntax for xmlelement is as follows:

```
xmlelement(name name [, xmlattributes(value AS attname [, ... ])] [,
content, ...])
```

The xmlelement expression produces an xml element with the given name, attributes and content.

name .... xml element tag name. If the name contains invalid name character, it will use hex format to replace it. For example, if name is 'phone number'(there is a space between phone and number), the tag name would be replaced as phone_x20_number.

attname ....attribute names.

content ....can be plain text, sub xml element, or xml comment.

➲ **Example 1**

```
dmSQL> SELECT XMLELEMENT(name foo, XMLATTRIBUTES(current_date as bar), 'cont',
'ent');


XMLELEMENT(NAME FOO, XMLATTRIBUTES(CURRENT_DATE AS BAR), 'CONT', 'ENT')
=======================================================================
<foo bar="2011-08-18">content</foo>

1 rows selected
```

The syntax for xmlforest is as follows:

```
xmlforest(content [AS name] [, ...])
```

The xmlforest expression produces an XML forest (sequence) of elements using the given name and content. If name is not specified and the content value is a column reference, then the default would be the column name.

➲ **Example 2**

```
dmSQL> SELECT XMLFOREST(empname, phone) FROM employee;


XMLFOREST(EMPNAME, PHONE)
=======================================================================
<empname>Abby</empname><phone>123-1234</phone>
<empname>Alice</empname><phone>234-1234</phone>
<empname>Amber</empname><phone>567-1234</phone>
3 rows selected
```

The syntax for xmlagg(xml) is as follows:

```
xmlagg(xml) .......
```

Unlike the other functions, xmlagg is an aggregate function. It concatenates the input values across rows. The input of xmlagg should be an xml fragment. The output is CLOB type.If there is no content, then the xml element would be displayed as an empty element like <ABC/>. No extra new line added after the start or end tag.

➲ **Example 3**

```
dmSQL> SELECT XMLAGG(XMLELEMENT(name person, XMLELEMENT(name name, empname)))
FROM employee;
```

```
XMLAGG(XMLELEMENT(NAME PERSON, XMLELEMENT(NAME NAME, EMPNAME)))
============================================================================
<person><name>Abby</name></person><person><name>Alice</name></person><person><nam
e>Amber</name></person>

1 rows selected
```

The syntax for xmlcomment(text) is as follows:

xmlcomment(text) . . . . .

Xmlcomment is an udf. The input is sql expression which can produce nchar or char data. The output is a string in XML comment format which starts with <!-- and ends with -->.

text . . . . If the text contains any escape characters (eg, < > &), those characters will be replaced with entity display.

➲ **Example 4**

```
dmSQL> SELECT XMLCOMMENT(empname) FROM employee;

XMLCOMMENT(EMPNAME)
============================================================================
<!--Abby-->
<!--Alice-->
<!--Amber-->

3 rows selected
```

There are restrictions on using the four XML functions:

- Always automatically cast input as char. Because udf's input type is predefined.

- Output is char type, except xmlagg. The size is limited (related to page size). For example, if **DB_PgSiz** = 8, then the output string size is limited as 8056. Oversize data will be truncated without warning.

- If input is nchar with characters which cannot be converted to lcode, the output might be something invalid. For example, the original data is stored in nchar column and the data contain both traditional Chinese character and Japanese

character. If lcode = 2, then those traditional Chinese character cannot be converted properly.

- No new line will be added after end tag. No special xml format is provided.

3.89 SET CONNECTION OPTIONS

The SET CONNECTION OPTIONS command provides syntax so users can set connection options through SQL statements. Useful for users that use front-end tools like Delphi to connect to the database and cannot get ODBC connection handles, they can set connection options needed directly instead.

The following is the detailed description of all of the options used with this command. The options fall into six categories: *no value options*, *on/off options*, *number options*, *string options*, *symbol options,* and *transaction options*.

no_value_options............ Option which has no option value

*on_off_options…………*Option with a value of *on* or *off*

*string_options…………*Option whose value a single quoted string, such as *'FOB'*

*number_options………*Option whose value is an integer

*symbol_options…………*Option whose value is one of a set of symbols, such as *{delete | close | preserve}*

*transaction_options……*Option specifying transaction's actions
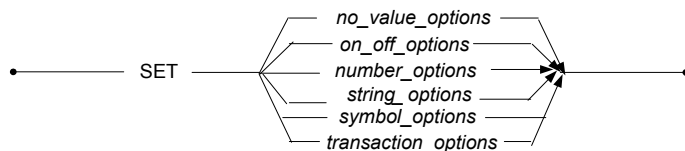
Figure 3-134 SET CONNECTION OPTIONS syntax

No Value Options

Options in this category have no option values and are simple commands.

SET FLUSH

The SET FLUSH is a replication server option that flushes replication to the slave site(s).

SET SYSINFO CLEAR

Clear system information resets system table, SYSINFO.
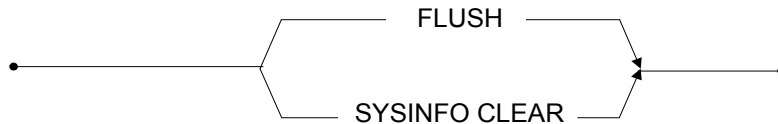
Figure 3-135 No Value Options syntax

ON/OFF Options

In this category, all valid option values are *ON* or *OFF*. Some only allow the value of ON or OFF; others accept both.

SET AUTOCOMMIT *ON/OFF*

Turn autocommit ON or OFF.

SET BACKUP *OFF*

Set backup mode to non-backup. The setting is the same as setting the **DB_BMode** to 0.

SET BKSVR CMP *ON/OFF*

Set backup server's compact backup option ON or OFF.

SET BLOB BACKUP *ON*

Set backup mode to backup-data-and-blob. This setting is the same as setting **DB_BMode** to 2.

SET BROWSE ON/OFF

Set connection option SQL_ATTR_TXN_ISOLATION to SQL_TXN_READ_UNCOMMITTED (*ON*) or SQL_TXN_SERIALIZABLE (*OFF*). For more information, please refer to the *ODBC Programming Guide* in the function *SQLGetInfo* with the option *SQL_DEFAULT_TXN_ISOLATION*.

SET DATA BACKUP *ON*

Set backup mode to backup-data. This setting is the same as setting the **DB_BMode** to 1.

SET FASTCOPY *ON/OFF*

This option is used to set the connection attribute on client side. The default setting is *OFF*. Each user connecting to the database has a key attribute, and a user's setting will not affect other users.

SET FREE CATALOG CACHE *ON/OFF*

Set the system catalog cache ON to free it or OFF to save.

SET ITCMD *ON/OFF*

Turn implicit data conversion ON or OFF.

SET JOURNAL *ON/OFF*

Only a DBA may turn Journal writing ON or OFF.

SET LOADAUTOINDEX *ON/OFF*

This option is used to specify wheter load all indexes while user implemented LOAD DB command. Users implement LOAD DB command load all index if setting LOADAUTOINDEX ON, but if set LOADAUTOINDEX OFF, user implement LOAD DB command load all index except auto index. The default setting is *OFF*.

SET LOAD SYSTEM DEFAULT *ON/OFF*

If users assign value to the column by using the INSERT/UPDATE statement, this option is used 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 users. The default setting for this option is *OFF*.

SET REMOVE SPACE PADDING *ON/OFF*

Turn ON/OFF the facility that removes the space padding after a string data automatically.

SET STRING CONCAT *ON/OFF*

This option is used for the string concatenate operator (||). If you set this option to ON, all space padding in CHAR type data will be removed before the operator is applied. If this option is OFF, all space padding will be kept.

SET SYSTEM DEFAULT *ON/OFF*

Data to be updated, this option is used 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*.

SET SYSTEM INIT *ON/OFF*

Only a DBA may turn system mode ON or OFF. In the system mode, create system tables.
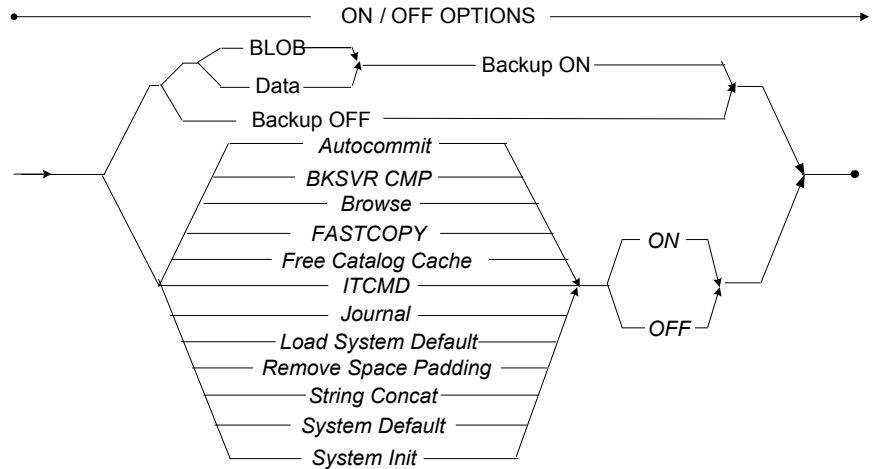
Figure 3-136 ON/OFF Options syntax

Number Options

This group contains options with values as integers. Each option may have its own range of valid integers.

SET BKSVR JOURNAL FULL *NUMBER*

Set the backup server's Journal full percent rate, from 0 to 100.

SET BKSVR PID *NUMBER*

Set the backup server process ID to a number. Currently the number must be 0.

SET DDB LOGIN TIMEOUT *NUMBER*

Set the login timeout for a DDB connection.

SET DDB LOCK TIMEOUT *NUMBER*

This option sets the lock timeout for a DDB connection.

SET INPUT PARAM *N* AS CFILE | ASCII

This set option is used before an INSERT or UPDATE statement that uses parameters. It is used if the user wants to bind one or more of the parameters in the statement to a client file. The input data for the corresponding parameter or parameters in the succeeding statement will be bound to a client file. The data to insert must be character type data, and the parameter must correspond to either a LONG VARCHAR or LONG VARBINARY type column.

Use the ALL option to bind all parameters to a client file. The CFILE option must be used to set the parameters to bind to the client file. To reset DBMaker so that it does not bind parameters to a client file, use the SET INPUT PARAM statement with the ASCII option.

number ....Specifies, in sequence, which parameter is bound to the client file
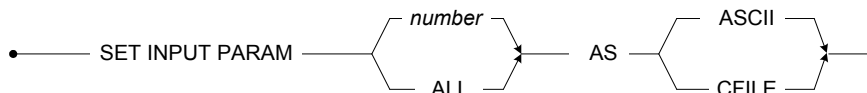
Figure 137 Syntax of the SET INPUT PARAM option

➲ **Example**

In this example, the file '**dmconfig.ini**' can be inserted into column **c3** using a host variable.

```
dmSQL> CREATE TABLE tb_attri (c1 INT, c2 INT, c3 LONG VARBINARY);
dmSQL> SET INPUT PARAM 3 AS CFILE;
dmSQL> INSERT INTO tb_attri VALUES (?,?,?);
dmSQL/Val> 2,2,'dmconfig.ini';
dmSQL/Val> end;
```

SET LOCK TIMEOUT *NUMBER*

Set the number of seconds to wait for the lock before returning to the application. If the number is positive, the timeout is in seconds. If the number is zero, it does not wait. If the number is negative, it will always wait.

SET MAXTBROW *NUMBER*

Set the maximum number of rows to be returned when retrieving table data. All rows are returned when the number is zero or negative.

SET RPSVR RETRY *NUMBER*

The number of retries after a network failure occurs when replicating.

NUMBER OPTIONS

BKSVR Journal Full
BKSVR PID
DDB Login Timeout
DDB Lock Timeout
Lock Timeout
MAXTBROW
RPSVR Retry
String Length

integer
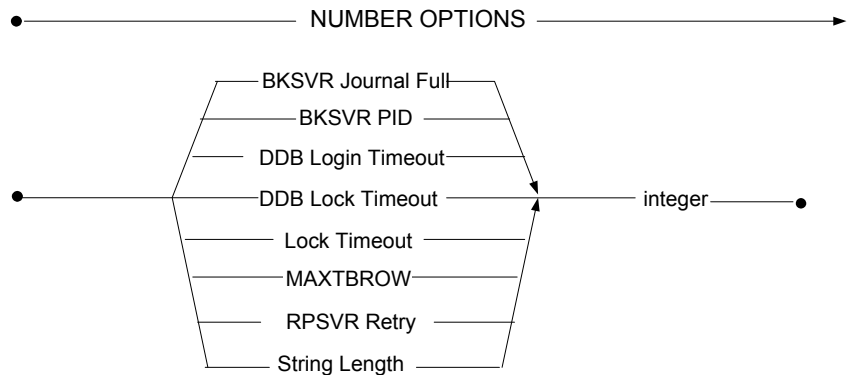
Figure 3-138 Number Options syntax

String Options

Options in this group use single-quoted strings as the value. For some options, the values must fit in the special formats.

SET BKSVR PATH *STRING*

Set the backup Journal file path.

SET DATE INPUT FORMAT {ALL | *STRING}*

Set input format for DATE columns.

The valid formats are:

FORMAT	EXAMPLE
'mm/dd/yy'	'02/18/99'
'mm-dd-yy'	'02-18-99'
'dd-mon-yy'	'18-Feb-99'
'mm/dd/yyyy'	'02/18/1999'
'dd/mon/yyyy'	'18/Feb/1999'
'dd-mon-yyyy'	'18-Feb-1999'
'dd.mm.yyyy'	'18.2.1999'

Table 3-2 (yy/yyyy: year, mm*: month,* dd*: day)*

When the ALL command is specified, all of the above date formats are allowed.

SET DATE OUTPUT FORMAT *STRING*

Set the output format for DATE columns. The formats are listed in the *SET DATE INPUT FORMAT* command.

SET EXTNAME TO *STRING*

Set extension name of the server file objects to *string*.

SET TIME INPUT FORMAT { ALL | *STRING* }

Set the input formats for the TIME columns. Setting the input format to ALL allows all formats.

Alternately, use one of the following formats for input and output formats:

FORMAT	EXAMPLE
'hh:mm:ss.fff '	22:10:20.30
'hh:mm:ss'	22:10:20
'hh:mm'	22:10
'hh'	22

'hh:mm:ss.fff tt'	10:10:20.30 PM
'hh:mm:ss tt'	10:10:20 PM
'hh:mm tt'	10:10 PM
'hh tt'	10 PM
'tt hh:mm:ss.fff '	PM 10:10:20.30
'tt hh:mm:ss'	PM 10:10:20
'tt hh:mm'	PM 10:10
'tt hh'	PM 10

*Table 3-3 (*hh*: hour,* mm*: minute,* ss*: second,* fff*: fraction,* tt*: AM/PM)*

When the *ALL* command is applied, all of the above formats can be used to input TIME columns.

SET TIME OUTPUT FORMAT *STRING*

Set output format for the TIME columns. The possible formats in the string are the same options as "SET TIME INPUT FORMAT" (See Table 3-3).
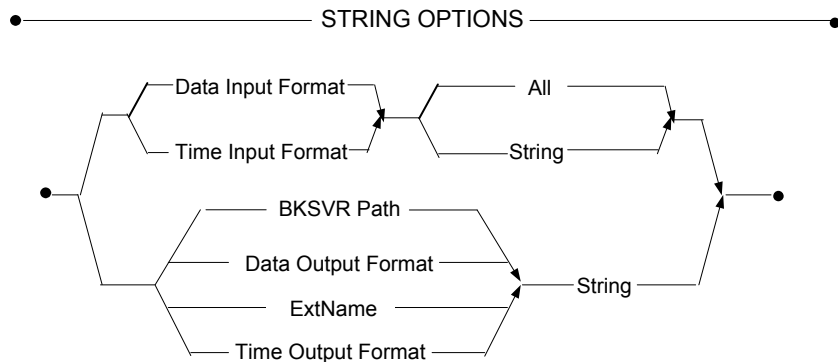
Figure 3-139 String Options syntax

Symbol Options

In this group, all option values are a set of symbols that mainly match ODBC symbols. Please refer to the corresponding ODBC connection options for more information.

SET CB MODE { *CLOSE* | *DELETE* | *PRESERVE* }

Set cursor behavior, as transactions are committed. For more information about these three modes, please refer to the *ODBC Programmer's Guide* in the *SQLGetInfo* function section with the *SQL_CURSOR_COMMIT_BEHAVIOR* option.

SET CONCAT NULL RETURN { *NULL* | *STRING* }

This option is used for string concatenation with null for the CONCAT built-in function or concatenate operator (||). The default setting for this option is *NULL*. If this option is set to *NULL*, then any string concatenated with a null value will return null. If the option is set to *STRING*, then any string concatenated with a null value will return the string, because the null value will be treated as an empty string.

SET DISCONNECT { *DISCONNECT* | *TERMINAT* | *WAIT* }

Sets the action of SQLDisconnect(). If *disconnect* is set, it just disconnects from the server. The *terminate* call will shutdown the database. The *wait* call option will cause the call to wait for the server to completely shutdown before it returns. This is an internal option of DBMaker for developing tools to shutdown the database by calling the SQLDisconnect().

SET DFO DUPMODE { *COPY* | *NULL* }

This option determines file objects duplication when executing the "*select into*" on the file object columns from the remote tables. If set to *null*, the FILE columns will be set to NULL. Otherwise, the remote file objects will be copied into local tables.

SET FO TYPE { *BLOB* | *FILE* }

Selects the SQL types to map to a FILE column. If a file is selected, SQL_FILE will be returned for FILE columns. Otherwise, the SQL_LONGVARBINARY will be used.
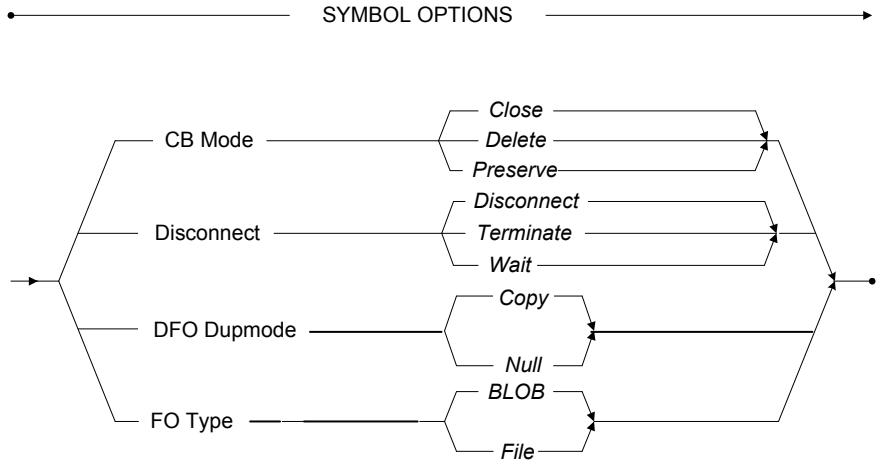
SYMBOL OPTIONS

Figure 3-140 Symbol Options

⮞ **Example 1**

SET BKSVR PID

```
dmSQL> SET BKSVR PID 0;
```

⮞ **Example 2**

SET BKSVR PATH

```
dmSQL> SET BKSVR PATH 'd:\data\backup';
```

⮞ **Example 3**

SET DATE INPUT FORMAT

```
dmSQL> SET DATE INPUT FORMAT ALL;
dmSQL> SET DATE INPUT FORMAT 'yyyy/mm/dd';
```

⊃ **Example 4**

SET DATE OUTPUT FORMAT
```
dmSQL> SET DATE OUTPUT FORMAT 'mm-dd-yy';    // result of DATE column will be like
12-31-99
```

⊃ **Example 5**

SET DDB LOCK TIMEOUT:
```
dmSQL> SET DDB LOCK TIMEOUT 20;                    // timeout is 20
```

⊃ **Example 6**

SET DDB LOGIN TIMEOUT
```
dmSQL> SET DDB LOGIN TIMEOUT 15;
```

The remaining examples use two tables named *t1* on database *db1* and *db2*. The definitions of both tables named *t1* are included.

⊃ **Example 7**

SET DFO DUPMODE
```
dmSQL> CREATE TABLE t1 (c1 INT, c2 FILE);
```

Now, we use *db2* as a remote database of *db1*.

⊃ **Example 8**

SET DFO DUPMODE
```
dmSQL> SET DFO DUPMODE null;
```

Insert data into *t1*.

⊃ **Example 9**

SET DFO DUPMODE
```
dmSQL> SELECT c1, c2 from DB2:SYSADM.t1 INTO t1;
```

Then column *c2* of *t1* will be NULL. On the other hand, if we use.

⊃ **Example 10**

SET DFO DUPMODE
```
dmSQL> SET DFO DUPMODE copy;
```

Insert data into *t1* by selecting tuples from *db2:t1*, column *c2* of newly inserted rows are copied from column *c2* of *db2:t1*.

➲ **Example 11**

SET EXTNAME TO

```
dmSQL> SET EXTNAME TO 'FOB';
```

➲ **Example 12**

SET LOCK TIMEOUT

```
dmSQL> SET LOCK TIMEOUT 30;                  // timeout is 30 seconds
dmSQL> SET LOCK TIMEOUT 0;          // always wait
dmSQL> SET LOCK TIMEOUT -5;                  // always wait
```

➲ **Example 13**

SET MAXTBROW

```
dmSQL> SET MAXTBROW 10;               // return only first 10 tuples of data
dmSQL> SET MAXTBROW -3;               // return all tuples
```

➲ **Example 14**

SET SYSTEM INIT

```
dmSQL> SET SYSTEM INIT ON;
dmSQL> CREATE TABLE SYSTEM.t1 (c1 int);
```

➲ **Example 15**

SET TIME INPUT FORMAT

```
dmSQL> SET TIME INPUT FORMAT ALL;           // all formats accepted
dmSQL> SET TIME INPUT FORMAT 'hh:mm';       // 10:20
```

➲ **Example 16**

SET TIME OUTPUT FORMAT

```
dmSQL> SET TIME OUTPUT FORMAT 'hh:mm:ss';   // 10:20:55
```

Transaction Options

Set connection option SQL_ATTR_TXN_ISOLATION to
SQL_TXN_READ_UNCOMMITTED, SQL_TXN_READ_COMMITTED,
SQL_TXN_REPEATABLE_READ or SQL_TXN_SERIALIZABLE. For more
information, please refer to the *ODBC Programming Guide* in the function
SQLGetInfo with the option *SQL_DEFAULT_TXN_ISOLATION*.

TRANSACTION OPTION

SET TRANSACTION ISOLATION LEVEL

READ UNCOMMITED
READ COMMITED
REPEATABLE READ
SERIALIZABLE
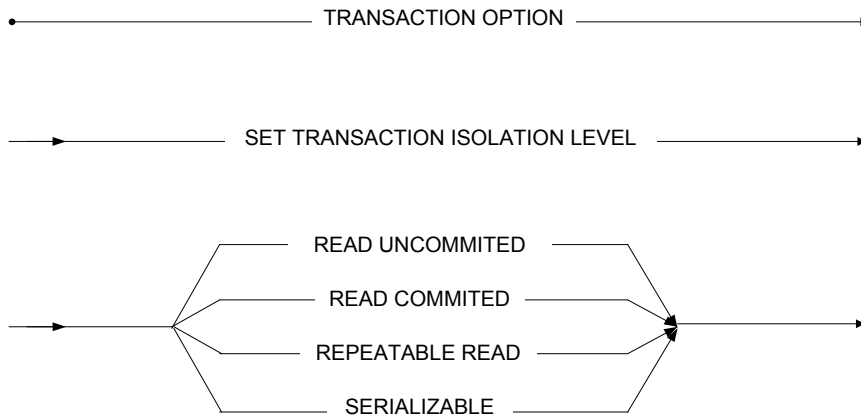
Figure 3-141 TRANSACTION OPTIONS syntax

3.90 SET CLIENT_CHAR_SET

The **SET CLIENT_CHAR_SET** command specifies the character set on the database client side.

In mutilingual database, client side can use several local codes to connect to UTF-8 database. So client side can set its own character set to distinguish with the server side. Key word **DB_LCode** is used to set the server's language code, while uses **DB_CliLCODE** to set the client's character set in the dmconfig.ini file. In addition, user can aslo set client's character set throung the command **SET_CLIENT_CHAR_SET** But, this command is only valid for the current session, once disconnect this session, the setting set by this command can be useless.

Aslo, user may need to know what the character set in the database server or in the client side. And the **UDF GETSYSINFO**() can help to return the settings.

To get server character set, the syntax is SELECT GETSYSINFO('LCODE');

To get the client character set, the command is SELECT GETSYSINFO('CLILCODE');

client-character-set-string   Character sets can be set in client side

      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)

●————————SET CLIENT_CHAR_SET————————————*client-character-set-string*————●

Figure 3-142 SET CLIENT CHARACTER SET syntax

➲ **Example**

Set the client character set to **BIG5**.

```
dmSQL> SET CLIENT_CHAR_SET 'BIG5';
```

3.91   SET ERRMSG_CHAR_SET

The SET ERRMSG_CHAR_SET command specifies the error message output character set of the database client.

Clients can set their own error message output character set for mutilingual databases.

The command must be specified like 'language*[_locale][.code]*'. **'language'** string follows ISO-639 standards, it is must be lowercase; and **'locale'**stirng follows ISO-3166 standards,it is must be capital letter; **'code'** string is the character set name that DBMaker supported. For a language which has more than one locale, it should be specified to which locale. For example, zh_CN or zh_TW, zh alone is invalid.

But, this command is only valid for the current session, once disconnect this session, the setting set by this command can be useless.

DBMaker currently supports four languages for client error messages: English, simplified Chinese, traditional Chinese and Japanese.

The Error table is stored in the *dbmaker/5.4/shared/locale/locale_LANG/ directory*.

To get client error message set, user can execute command SELECT GETSYSINFO('ERRLCODE');

The valid values: en，jp，zh_CN and zh_TW or the combination of them and character set.

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.UTF-8

ja.EUC-JP

zh_CN

zh_CN.GBK

zh_CN.UTF-8

zh_CN.GB18030

zh_TW

zh_TW.BIG5

zh_TW.UTF-8

●————— SET ERRMSG_CHAR_SET —————— *language [_locale][.encode]* —————●

Figure 3-143 SET ERRMSG_CHAR_SET syntax

⮞ **Example1**

The following sets the client error message output character set with the locale '**ja**'.

```
dmSQL> SET ERRMSG_CHAR_SET 'ja';
```

⮞ **Example2**

The following sets the client error message output character set with the locale '**ja**' and the character set '**EUC_JP**'.

```
dmSQL> SET ERRMSG_CHAR_SET 'ja.EUC-JP';
```

⮞ **Example3**

The following sets the client error message output character set with the locale '**ja**' and the character set '**UTF-8**'.

```
dmSQL> SET ERRMSG_CHAR_SET 'ja.UTF-8';
```

3.92   SUSPEND SCHEDULE

The SUSPEND SCHEDULE command suspends the replication schedule for an asynchronous table replication. The local database will not try to connect to the remote database until the replication schedule resumes. Only the local table owner, a DBA, a SYSDBA, or a SYSADM can execute the SUSPEND SCHEDULE command.

Use the SUSPEND SCHEDULE command to suspend a replication schedule for an asynchronous table replication. To resume the replication schedule use the RESUME SCHEDULE command.

remote_database_name….Name of the remote database to remove the replication schedule from

●──── SUSPEND SCHEDULE FOR REPLICATION TO ──── *remote_database_name* ────●

Figure 3-144 SUSPEND SCHEDULE syntax

➲   **Example**

The following suspends the replication schedule for the remote database named **DivOneDb**.

```
dmSQL> SUSPEND SCHEDULE FOR REPLICATION TO DivOneDb;
```

3.93   SYNC AUTO INDEX

The SYNC AUTO INDEX command wakes up the auto index daemon handing mechanisms immediately. Only users with DBA, SYSDBA or SYSADM security privileges can execute the SYNC AUTO INDEX command when the auto index daemon starts.

Only when auto index daemon startups and the keyword AUTOCOMMIT is set to ON, users can execute the SYNC AUTO INDEX command.

```
•────────────    SYNC AUTO INDEX    ────────────•
```

Figure 3-145 SYNC AUTO INDEX syntax

➲   **Example**

The following example wakes up the auto index daemon when **AUTOCOMMIT** is set to **ON**.

```
dmSQL> SYNC AUTO INDEX;
```

©Copyright 1995-2017 CASEMaker Inc.

3.94   SYNCHRONIZE SCHEDULE

The SYNCHRONIZE SCHEDULE command synchronizes all data in the remote database with data in the local database without waiting for the next scheduled time. Only the local table owner, a DBA, a SYSDBA, or a SYSADM can execute the SYNCHRONIZE SCHEDULE command.

Use the SYNCHRONIZE SCHEDULE command to synchronize data in the local and remote tables for an asynchronous table replication.

remote_database_name….Name of the remote database to synchronize the replication schedule for
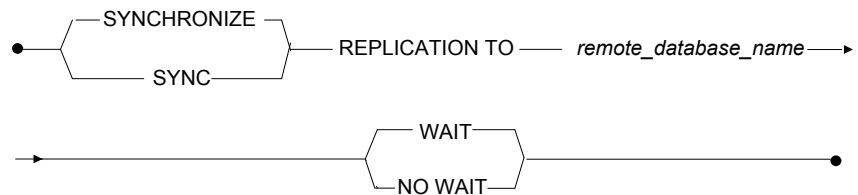
Figure 3-146 SYNCHRONIZE SCHEDULE syntax

➲ **Example**

The following example synchronizes the replication schedule for the remote database named **DivOneDb**.

```
dmSQL> SYNCHRONIZE REPLICATION TO DivOneDb;
```

3.95 UNLOAD STATISTICS

The UNLOAD STATISTICS command unloads database statistics into an ASCII text file. Edit the file and load the desired statistics data back into the database. Only users with DBA, SYSDBA or SYSADM security privileges can execute the UNLOAD STATISTICS command.

Load statistical information for an entire database, or for one or more tables. For each table specify whether to load the table statistics information, the column statistics information, the index statistics information, or a combination of the three.

DBMaker records table data statistics on the number of pages, the number of rows, and the average row length of sampled rows in a table. DBMaker records column data statistics on the number of distinct column values, the average column length, the low value, and the high value for all sampled values in a column. DBMaker records index data statistics on the number of index pages, the number of index tree levels, the number of leaf pages, the number of distinct key values, the number of pages per key, and the cluster count for the index.

object_list ....List of database objects to unload statistics data for

file_name ....Name of the ASCII text file that statistics data will be saved in
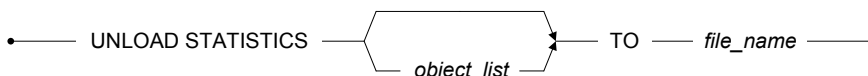
Figure 3-147 UNLOAD STATISTICS syntax

UNLOAD STATISTICS Object List
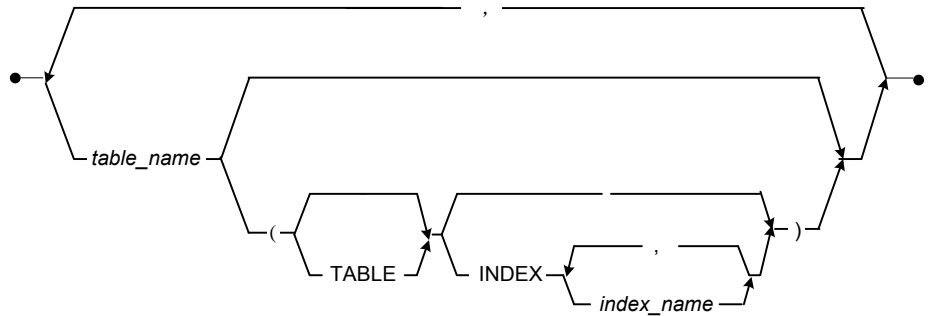
Figure 3-148 UNLOAD STATISTICS Object List syntax

➲ **Example**

The following unloads all **STATISTICS** to the file **stat.dat**.

```
dmSQL> UNLOAD STATISTICS TO stat.dat;
```

3.96   UPDATE

The UPDATE command updates rows in a table. Rows in the system catalog tables can not updated with this command. Only the table owner, a DBA, a SYSDBA, a SYSADM, or a user with the UPDATE privilege for the entire table or for the specific column can execute the UPDATE command.

When updating a column the new column values must satisfy the column constraints and referential integrity. Use the DEFAULT keyword to set the value of the column to the default.

table_name...............Name of the table containing the rows to update

column_name...........Name of the column to update values in

literal.......................Literal value to update the column with

expression ....Expression that returns a value to update the column with

constant....................Constant value to update the column with

search_condition ....Conditions a row must meet to be updated

cursor_name ....Name of the cursor to use for a positioned update (cursors are only available within ODBC programs)
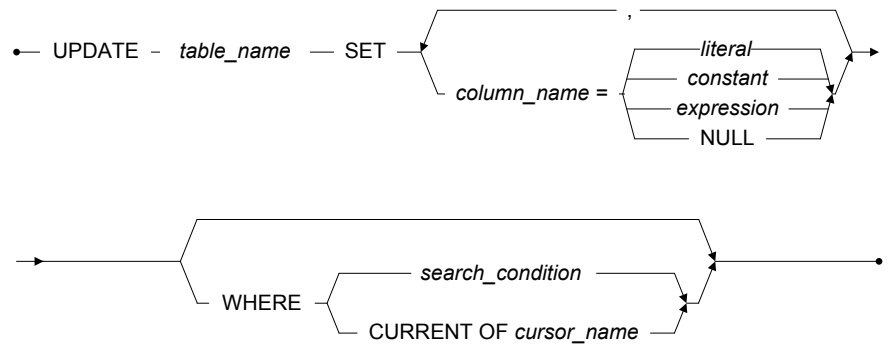
Figure 3-149 UPDATE syntax

➲ **Example 1**

The following shows how to update the **Employeesinfo** table and change the salary of all employees named **Chris**.

```
dmSQL> UPDATE Employeesinfo SET Salary = 5000 WHERE FName = 'Chris';
```

➲ **Example 2**

The following shows how to give a salary raise of **10%** to all **employees** named **Chris**.

```
dmSQL> UPDATE Employeesinfo SET Salary = Salary*1.10 WHERE FName = 'Chris';
```

3.97 UPDATE STATISTICS

The UPDATE STATISTICS command updates database statistics information. Keeping statistics information current helps the database to perform queries more efficiently. Only the owner of the object, a DBA, a SYSDBA, or a SYSADM can execute the UPDATE STATISTICS command.

Update statistical information for the entire database or take update statistical information for one or more tables. For each table specify whether to update statistical information for the table, the column, the index, or a combination of the three. In addition, specifying a number between 1 and 100 for the SAMPLE keyword can set the percentage of data to sample.

DBMaker records index data statistics on the number of index pages, the number of index tree levels, the number of leaf pages, the number of distinct key values, the number of pages per key, and the cluster count for the index.

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.

object_list ....List of database objects to update statistics data for

number ....Percentage of data to use when updating statistics data
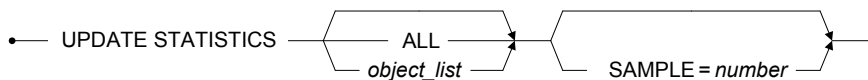
Figure 3-150 UPDATE STATISTICS syntax

UPDATE STATISTICS Object List

DBMaker records table data statistics on the number of pages, the number of rows, and the average row length of sampled rows in a table.

DBMaker also records column data statistics on the number of distinct column values, the average column length, the low value, and the high value for all sampled values in a column.

Figure 3-151 UPDATE STATISTICS Object List syntax

➲ **Example 1**

The following updates all **STATISTICS** in the database with a sampling of **30%**.
```
dmSQL> UPDATE STATISTICS SAMPLE = 30;
```

➲ **Example 2**

The following updates all **STATISTICS** on **table1**.
```
dmSQL> UPDATE STATISTICS table1 SAMPLE = 50;
```

➲ **Example 3**

The following updates **STATISTICS** for index **ix1** on **table1**.
```
dmSQL> UPDATE STATISTICS table1 (INDEX ix1);
```

➲ **Example 4**

The following updates **STATISTICS** for all indexes on **table1**.
```
dmSQL> UPDATE STATISTICS table1 (INDEX);
```

⮊ **Example 5**

The following forcibly updates **STATISTICS** for all objects in a database.
```
dmSQL> UPDATE STATISTICS ALL;
```

3.98   UPDATE STATISTICS SET

The UPDATE STATISTICS SET command specifies every table's update statistics method and sample ratio for update statistics daemon when it starts in **every table setting** mode, that is to say, the value of **DB_StMod** is 1.

Every table's update statistics and sample ratio are stored in system table SYSTABLE. The column UPD_STS_MODE stored the table statistics method, and the column UPD_STS_SAMPLE stored the table statistics sample ratio.

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.

table_name ....Name of the table

mode_value..............The table update statistics method

> 0: Sample ratio of table uses value of **DB_StsSp** in **dmconfig.ini** but need to consider the above four filter conditions. The default value is *0*.
> 1: Sample ratio of table uses table update statistics sample ratio which be set in *sample_value* but need to consider the above four filter conditions.

2: Sample ratio of table uses table update statistics sample ratio which be set in *sample_value* regardless of the above four filter conditions.

sample_value ....The table update statistics sample ratio

-1 : Intelligently obtain sample ratio

0: The database does not need to update statistics value

0 ~ 100: Table update statistics sample ratio, the default value is **100**
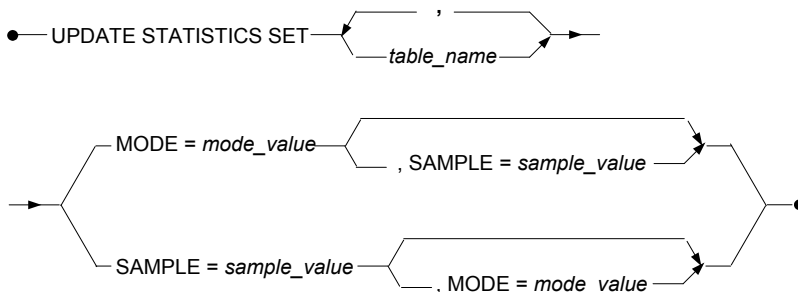
Figure 3-152 UPDATE STATISTICS SET syntax

➲ Example 1

Setting the update statistics method and sample ratio for the 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 the update statistics method and sample ratio for the table **jeff.tb_staff** and **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
```

3.99 UPDATE TABLESPACE STATISTICS

The UPDATE TABLESPACE STATISTICS command updates tablespace statistical information. Keeping statistical information current helps the tablespace to perform queries more efficiently. Only users with DBA, SYSDBA or SYSADM security privileges can execute the UPDATE TABLESPACE STATISTICS command.

DBMaker will update the tablespaces and associated file statistical value to update tablespace statistics.

DBMaker records tablespace data statistics on the number of pages, the number of free pages, the number of frames, and the number of free frames.

DBMaker records file data statistics on the number of pages/frames, and the number of free pages/frames.

object_list ....List of database objects to update statistical data for

●————— UPDATE TABLESPACE STATISTICS ————— *object_list* —————●

Figure 3-153 UPDATE TABLESPACE STATISTICS syntax
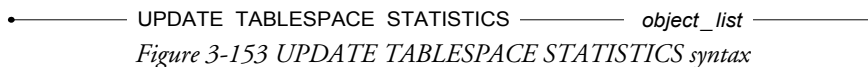
➲ **Example**

The following updates the **DEFTABLESPACE STATISTICS**.

```
dmSQL> UPDATE TABLESPACE STATISTICS DEFTABLESPACE;
```

4 Functions

DBMaker provides a number of built-in functions, and also allows programmers to build their own user-defined functions (UDF). For details, please refer to the following sections.

4.1   Built-in Functions

DBMaker provides a number of built-in functions. These functions can be used on columns in a result set or columns that restrict rows in a result set. This chapter lists each function by type. The arguments and returned values for each function are listed below the syntax diagram providing the name, data type, and value.

The Built-in Functions types are:

◆   String functions

◆   Numeric functions

◆   Date and time functions

◆   System functions

4.1.1. ABS

The ABS function returns the absolute value of *number,* as a double precision floating-point number.

number ....Double: Number to find the absolute value for

Return value ....Double: Absolute value of *number*

ABS (*number*)
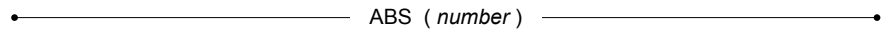
Figure 4-1 ABS syntax

➲ **Example**

The following syntax returns **3.14000000000000e+012**.

```
ABS(-3.14E12)
```

4.1.2. ACOS

The ACOS function returns the arc cosine for a number in the double precision floating-point number format. The number argument must be in the range 0 to π radians.

number ....Double: Number to find the arc cosine for

Return value ....Double: The arc cosine for a *number*

ACOS (*number*)

Figure 4-2 ACOS syntax

➲ **Example**

The following syntax returns **1.04719755119660e+000**.
```
ACOS(0.5)
```

4.1.3. ADD_DAYS

The ADD_DAYS function returns a result from adding the *number* of days to the *date*. The *number* argument may be a negative number.

date .... Date: Date to add days to

number .... Integer: Number of days to add

Return value .... Date: Result of adding *number* days to *date*

ADD_DAYS (*date, number*)

Figure 4-3 ADD_DAYS syntax

➲ **Example 1**

The following syntax returns the date**1999-03-01**.
```
ADD_DAYS('1999-02-24', 5)
```

➲ **Example 2**

The following syntax returns the date **2000-02-29**.
```
ADD_DAYS('2000-02-24', 5)
```

4.1.4. ADD_HOURS

The ADD_HOURS function returns a result after adding the *number* in hours to *time*. The *number* argument may be a negative number.

time ....Time: Time to add hours to

number ....Integer: Number of hours to add

Return value.............Time: Result of adding *number* hours to *time*

—————————— ADD_HOURS (*time, number*) ——————————

Figure 4-4 ADD_HOURS syntax

➲ **Example 1**

The following syntax returns the time **20:11:12**.
```
ADD_HOURS('10:11:12', 10)
```

➲ **Example 2**

The following syntax returns the time **22:11:12**.
```
ADD_HOURS('10:11:12', -12)
```

4.1.5. ADD_MINS

The ADD_MINS function returns a result after adding the *number* in minutes to *time*. The *number* argument may be a negative number.

time .... Time: Time to add minutes to

number .... Integer: Number of minutes to add

Return value .... Time: Result of adding *number* minutes to *time*

ADD_MINS (*time, number*)
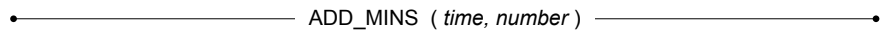
Figure 4-5 ADD_MINS syntax

➲ **Example 1**

The following syntax returns the time **10:21:12**.
```
ADD_MINS('10:11:12', 10)
```

➲ **Example 2**

The following syntax returns the time **09:59:12**.
```
ADD_MINS('10:11:12', -12)
```

4.1.6. ADD_MONTHS

The ADD_MONTHS function returns a result after adding a *number* in months to *date*. The *number* argument may be a negative number.

date.........................Date: Date to add months to

number ....Integer: Number of months to add

Return value.............Date: Result of adding *number* months to *date*

ADD_MONTHS (*date, number*)

Figure 4-6 ADD_MONTHS syntax

➲ **Example 1**

The following syntax returns the date **1999-07-24**.
```
ADD_MONTHS('1999-02-24',5)
```

➲ **Example 2**

The following syntax returns the date **2001-01-01**.
```
ADD_MONTHS('2000-01-01',12)
```

4.1.7. ADD_SECS

The ADD_SECS function returns a result after adding a *number* in seconds to *time*. The *number* argument may be a negative number.

time .... Time: Time to add seconds to

number .... Integer: Number of seconds to add

Return value .... Time: Result of adding *number* seconds to *time*

ADD_SECS (*time, number*)
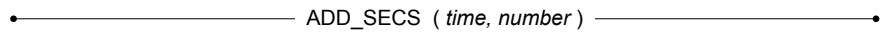
Figure 4-7 ADD_SECS syntax

➲ **Example 1**

The following syntax returns the time **10:11:22**.
```
ADD_SECS('10:11:12',10)
```

➲ **Example 2**

The following syntax returns the time **10:10:52**
```
ADD_SECS('10:11:12', -20)
```

4.1.8. ADD_YEARS

The ADD_YEARS function returns a result after adding a *number* in years to *date*. The *number* argument may be a negative number.

date..........................Date: Date to add years to

number ....Integer: Number of years to add

Return value.............Date: Result of adding *number* years to *date*

<div align="center">

──────────── ADD_YEARS (*date, number*) ────────────

</div>

<div align="center">

Figure 4-8 ADD_YEARS syntax

</div>

➲ **Example 1**

The following syntax returns the date **2001-03-04**.
```
ADD_YEARS('1999-03-04', 2)
```

➲ **Example 2**

The following syntax returns the date **1995-02-28**.
```
ADD_YEARS('2000-02-29', -5)
```

4.1.9. ASCII

The ASCII function returns the ASCII code value of the first character in *string*. If string contains no characters, a value of 0 (NULL) is returned. An error will be returned when a value for the *string* argument is not specified.

string ....String: Character, in the first position to obtain an ASCII code

Return value ....Integer: ASCII code of the character specified in *string*

ASCII (*string*)
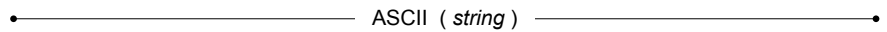
Figure 4-9 ASCII syntax

➲ **Example 1a**

The following syntax returns **65**, which is the **ASCII** code for "**A**".
```
ASCII('A')
```

➲ **Example 1b**

The following syntax also returns **65**, which is the **ASCII** code for "**A**".
```
ASCII('ABC')
```

➲ **Example 2a**

The following syntax returns **97**, which is the **ASCII** code for "**a**".
```
ASCII('a')
```

➲ **Example 2b**

The following syntax also returns **97**, which is the **ASCII** code for "**a**".
```
ASCII('abc')
```

⮞ **Example 3a**

The following syntax returns **49**, which is the **ASCII** code for "**1**".

```
ASCII('1')
```

⮞ **Example 3b**

The following syntax returns **33**, which is the **ASCII** code for "**!**".

```
ASCII('!')
```

4.1.10.   ASIN

The ASIN function returns a double precision floating-point number from the arc sine of *number* (in the range from -π/2 to π/2).

number....................Double: Number to find the arc sine for

Return value ....Double: Arc sine of *number*

ASIN (*number*)
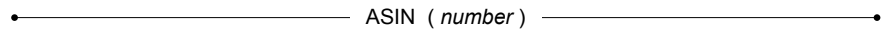
Figure 4-10 ASIN syntax

➲ **Example**

The following syntax returns the arc sine of number: **5.23598775598299e-001**.

```
ASIN(0.5)
```

4.1.11.   ATAN

The ATAN function returns a double precision floating-point number from the tangent of *number* (in the range from -π/2 to π/2).

number ....Double: Number to find the arc tangent for

Return value.............Double: Arc tangent of *number*

<center>———————— ATAN (*number*) ————————</center>

Figure 4-11 ATAN syntax

➲ **Example**

The following syntax returns the arc tangent of *number*: **4.63647609000806e-001**.

```
ATAN(0.5)
```

4.1.12.   ATAN2

The ATAN2 function returns the arc tangent of x/y in the range -π to π as a double precision floating-point number.

x.............................Double: Numerator in the ratio x/y to find the arc tangent for

y.............................Double: Denominator in the ratio x/y to find the arc tangent for

Return value ....Double: Arc tangent of x/y

ATAN2 (x, y)
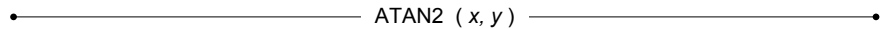
Figure 4-12 ATAN2 syntax

➲ **Example**

The following syntax returns the arc tangent of *x/y*, **4.63647609000806e-001**.

```
ATAN2(0.1, 0.2)
```

4.1.13.   ATOF

The ATOF function returns the value represented by the character string in the *string* argument as a double precision floating-point number.

string........................String: String to convert to a double-precision floating-point

number

Return value.............Double: Value of the character string in *string*

—————— ATOF (*string*) ——————

Figure 4-13 ATOF syntax

➲   **Example 1**

The following returns **-1.23400000000000e+001**, which is the double precision floating-point value of the character string "**-12.34**".

```
ATOF('-12.34')
```

➲   **Example 2**

The following returns **-1.23400000000000e+035**, which is the double-precision floating-point value of the character string "**-12.34E34**".

```
ATOF('-12.34E34')
```

4.1.14.　BLOBLEN

The BLOBLEN function returns the data length of an input BLOB. Please note, BLOBLEN reports at most $(2^{31} - 1)$B even when if the size is greater than or equal to 2^{31}B. BLOBLEN can get the data length for CLOB, BLOB, NCLOB and FILE type objects.

object........................BLOB: Source BLOB

Return value ....Integer: Get BLOB type data length of source BLOB

BLOBLEN (blob)

Figure 4-14 BLOBLEN syntax

● **Example**

The following returns the BLOB length of "**content**".
```
BLOBLEN(content)
```

4.1.15.   BLOBLENEX

The BLOBLENEX function returns the data length of an input BLOB as a decimal value. BLOBLENEX can get the data length for CLOB, BLOB, NCLOB and FILE type objects. BLOBLENEX reports the correct BLOB size, unlike BLOBLEN, even for BLOB size > 2^{31}B.

object ....BLOB: Source BLOB

Return value.............Decimal: Get BLOB type data length of source BLOB

BLOBLEN (blob)

Figure 4-15 BLOBLENEX syntax

➲ **Example**

The following returns the BLOB length of "**content**".
```
BLOBLENEX(content)
```

4.1.16.   CEILING

The CEILING function returns the integral value, greater than or equal to *number,* as a double precision floating-point number.

number....................Double: Number to find the nearest larger integer value for

*Return value ....*Double: The next integer value greater than *number*

CEILING (*number*)
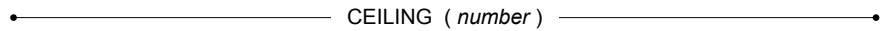
Figure 4-16 CEILING syntax

➲ **Example 1**

The following syntax returns **1.30000000000000e+001**, which is the next integer value with a value greater than **12.3**.
```
CEILING(12.3)
```

➲ **Example 2**

The following syntax returns **-1.20000000000000e+001**, which is the next integer value with a value greater than **-12.3**.
```
CEILING(-12.3)
```

4.1.17.   CHAR

The CHAR function returns the character that has the ASCII code value specified by *number*. The value specified for *number* should be a valid ASCII code value between 0 and 255; other values are not valid ASCII codes and are not supported by the CHAR function. Specifying a value that is not a valid ASCII code value may return incorrect or invalid results. An error will be returned when a value for the *number* argument is not provided.

number ....Integer: ASCII code of the character to obtain

Return value String: Character represented by the ASCII code specified by *number*

•————————— CHAR (*number*) —————————•

Figure 4-17 CHAR syntax

⮑ **Example 1**

The following syntax returns the string "**A**", which has an ASCII code value of **65**.
```
CHAR(65)
```

⮑ **Example 2**

The following syntax returns the string "**a**", which has an ASCII code value of **97**.
```
CHAR(97)
```

⮑ **Example 3**

The following syntax returns the string "**1**", which has an ASCII code value of **49**.
```
CHAR(49)
```

➲ **Example 4**

The following syntax returns the string "!", which has an ASCII code value of **33**.

```
CHAR(33)
```

4.1.18.   CHAR_LENGTH

The CHAR_LENGTH function returns the number of characters in *string*, excluding trailing blanks and the string termination character, when present. An error will be returned if a value for the *string* argument is not provided.

string........................String: String to find the length of

Return value............Integer: Leftmost *count* characters in *string*

CHAR_LENGTH (*string* _expression)
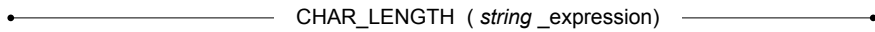
Figure 4-18 CHAR_LENGTH function syntax

⮊ **Example**

The following function command returns "4".

```
dmSQL> SELECT CHAR_LENGTH(' abc  ');
CHAR_LENGTH(' ABC   ')
=========================
                        4
```

4.1.19.   CHARACTER_LENGTH

The CHARACTER_LENGTH function returns the number of characters in *string*, excluding trailing blanks and the string termination character, when present. An error will be returned if a value for the *string* argument is not provided.

string ....String: String to find the length of

Return value ....Integer: Leftmost *count* characters in *string*

CHARACTER_LENGTH (*string_expression*)

Figure 4-19 CHARACTER_LENGTH function syntax

➲   **Example**

The following function command returns "4".

```
dmSQL> SELECT CHARACTER_LENGTH(' abc   ');
CHARACTER_LENGTH(' ABC    ')
==========================
                        4
```

4.1.20. CHECKMEDIAFORMAT

The CHECKMEDIAFORMAT function is used to check whether the BLOB content matches the specified media format.

blob ....Column name on which to perform the check

Media format: ....String: specify media format. The supported format is: DOC, XLS, PPT, HTM, XML and PDF.

Return value: ....True if the record in the column matches media format

CHECKMEDIAFORMAT(blob , media_format)

Figure 4-20 Syntax for CHECKMEDIAFORMAT

➲ **Example:**

The following check whether the blob column match the DOC format.

```
dmSQL> CHECKMEDIAFORMAT(wordcol, 'DOC');
```

4.1.21.   CONCAT

The CONCAT function returns a string expression formed by joining *string1* and *string2*. A return value will occur only if the string expression in *string1* is placed at the beginning of the result string, and the string expression in *string2* is placed at the end of the result string; an error will be returned if both values for the arguments have not been provided.

DBMaker uses the following rule to determine the value returned if one of the string expressions contains a NULL value.

Any string that is concatenated with a null value using the CONCAT built-in function or concatenate operator (||) will return *NULL*. If you want to return the string value when concatenating a string value with a null value, you must set the SET CONCAT NULL RETURN option to STRING. A null value concatenated with a null value will always return a null value, regardless of the value of the SET CONCAT NULL RETURN built-in-function.

string1 .... String: String to place at the beginning of the result string

string2 .... String: String to place at the end of the result string

Return value .... String: Formed by joining *string1* and *string2*

CONCAT (*string1, string2*)

Figure 4-21 CONCAT syntax

➲   **Example 1**

The following returns "**master plan**". Take notice the space at the end of the first string.

```
CONCAT('master ', 'plan')
```

➲ **Example 2**

The following returns "**mastermind**".
```
CONCAT('master', 'mind')
```

4.1.22.   COS

The COS function returns the cosine of *number,* expressed in radians, as a double precision floating-point number.

number....................Double: Number to find the cosine for

Return value ....Double: The cosine of *number*

COS (*number*)
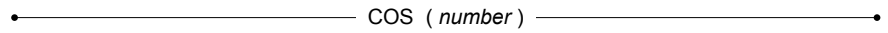
Figure 4-22 COS syntax

⮥ **Example**

The following syntax returns a value of **8.77582561890373e-001**.
```
COS(0.5)
```

4.1.23.   COSH

The COSH function returns the hyperbolic cosine of *number,* expressed in radians, as a double precision floating-point number.

number ....Double: Number to find the hyperbolic cosine for

Return value.............Double: The hyperbolic cosine of *number*

COSH (*number*)

Figure 4-23 COSH syntax

➲ **Example**

The following returns the hyperbolic cosine of *number;* **1.12762596520638e+000**.
```
COSH(0.5)
```

4.1.24.   **COT**

The COT function returns the cotangent of *number,* expressed in radians, as a double precision floating point number.

number....................Double: Find the cotangent for *number*

Return value ....Double: The cotangent of *number*

COT (*number*)
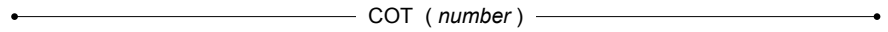
Figure 4-24 COT syntax

➲ **Example**

The following returns the cotangent of *number,* **1.83048772171245e+000**.

```
COT(0.5)
```

4.1.25.   CURDATE

The CURDATE function returns the current date.

*Return value.............*Date: The current date

CURDATE ( )

Figure 4-25 CURDATE syntax

➲ **Example**

The following returns the current date.
```
CURDATE()
```

4.1.26. CURRENT_DATE

The CURRENT_DATE function returns the current date from the default date/time/timestamp DBMaker output format.

Return value .... DATE: The current date

CURRENT_DATE ( )

Figure 4-26 CURRENT_DATE syntax

➲ **Example 1**

The following returns the current date.
```
dmSQL> INSERT INTO t1 VALUES (CURRENT_DATE);
dmSQL> SELECT CURRENT_DATE;
dmSQL> SELECY c1 FROM t1 WHERE c2 = CURRENT_DATE;
```

➲ **Example 2**

The following will insert the CURRENT_TIME, CURRENT_DATE, CURRENT_TIMESTAMP, and CURRENT_USER into one row, display the values, and then update the values.
```
dmSQL> INSERT INTO sql99t5 VALUES(CURRENT_TIME, CURRENT_DATE, CURRENT_TIMESTAMP,
CURRENT_USER);
1 row inserted

dmSQL> SELECT CURRENT_TIME, CURRENT_DATE, CURRENT_TIMESTAMP, CURRENT_USER;

CURRENT_TIME   CURRENT_D*    CURRENT_TIMESTAMP            CURRENT_USER
============   ==========  ======================= ==========================
16:53:09      2001-09-26 2001-09-26 16:53:09       SYSADM
```

```
dmSQL> UPDATE sql99t5 SET c1 = CAST(CURRENT_TIMESTAMP AS CHAR(20)),
                   c2 = CURRENT_DATE,
                   c3 = CURRENT_TIME,
                   c4 = CURRENT_TIMESTAMP WHERE c1 = CURRENT_USER;
1 row updated
```

4.1.27.   CURRENT_TIME

The CURRENT_TIME function returns the current time from the default time DBMaker output format.

Return value .... TIME: The current time
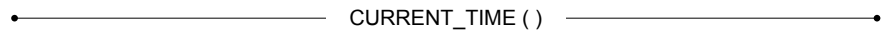
CURRENT_TIME ( )

Figure 4-27 CURRENT_ TIME syntax

➲   **Example 1**

The following returns the current time.
```
dmSQL> INSERT INTO t1 VALUES (CURRENT_TIME);
dmSQL> SELECT CURRENT_TIME;
dmSQL> SELECT c1 FROM t1 WHERE c2 = CURRENT_TIME;
```

➲   **Example 2**

The following will insert the CURRENT_TIME, CURRENT_DATE, CURRENT_TIMESTAMP, and CURRENT_USER into one row, display the values, and then update the values.
```
dmSQL> INSERT INTO sql99t5 VALUES(CURRENT_TIME, CURRENT_DATE, CURRENT_TIMESTAMP,
CURRENT_USER);
1 row inserted

dmSQL> SELECT CURRENT_TIME, CURRENT_DATE, CURRENT_TIMESTAMP, CURRENT_USER;

CURRENT_TIME   CURRENT_D*    CURRENT_TIMESTAMP              CURRENT_USER
============ ========= ======================= ============================
16:53:09     2001-09-26 2001-09-26 16:53:09      SYSADM

dmSQL> UPDATE sql99t5 SET c1 = CAST(CURRENT_TIMESTAMP AS CHAR(20)),
                c2 = CURRENT_DATE,
                c3 = CURRENT_TIME,
```

```
                    c4 = CURRENT_TIMESTAMP WHERE c1 = CURRENT_USER;
1 row updated
```

4.1.28.   CURRENT_TIMESTAMP

The CURRENT_TIMESTAMP function returns the current timestamp from the default timestamp DBMaker output format.

Return value .... TIMESTAMP: The current timestamp
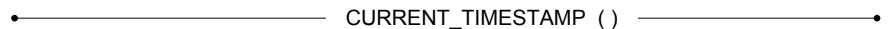
———————————•———— CURRENT_TIMESTAMP ( ) ————•———————————

Figure 4-28 CURRENT_ TIMESTAMP syntax

➲ **Example 1**

The following returns the current timestamp.
```
dmSQL> INSERT INTO t1 VALUES(CURRENT_TIMESTAMP);
dmSQL> SELECT CURRENT_TIMESTAMP;
dmSQL> SELECT c1 FROM t1 WHERE c2 = CURRENT_TIMESTAMP;
```

➲ **Example 2**

The following will insert the CURRENT_TIME, CURRENT_DATE, CURRENT_TIMESTAMP, and CURRENT_USER into one row, display the values, and then update the values.
```
dmSQL> INSERT INTO sql99t5 VALUES(CURRENT TIME, CURRENT DATE, CURRENT TIMESTAMP,
CURRENT_USER);
1 row inserted

dmSQL> SELECT CURRENT_TIME, CURRENT_DATE, CURRENT_TIMESTAMP, CURRENT_USER;

CURRENT_TIME   CURRENT_D*    CURRENT_TIMESTAMP            CURRENT_USER
============= ========== ====================== ==========================
16:53:09      2001-09-26 2001-09-26 16:53:09      SYSADM

dmSQL> UPDATE sql99t5 SET c1 = CAST(CURRENT_TIMESTAMP AS CHAR(20)),
```

```
                    c2 = CURRENT_DATE,
                    c3 = CURRENT_TIME,
                    c4 = CURRENT_TIMESTAMP WHERE c1 = CURRENT_USER;
1 row updated
```

4.1.29.   CURRENT_USER

The CURRENT_USER function returns the current user connected to DBMaker.

Return value .... USER: The current user
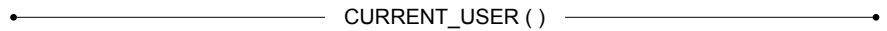
CURRENT_USER ( )

Figure 4-29 CURRENT_ USER syntax

 Example 1

The following returns the current user.
```
dmSQL> INSERT INTO t1 VALUES (CURRENT_USER);
dmSQL> SELECT CURRENT_USER;
dmSQL> SELECT c1 FROM t1 WHERE c2 = CURRENT_USER;
```

 Example 2

The following will insert the CURRENT_TIME, CURRENT_DATE,
CURRENT_TIMESTAMP and CURRENT_USER into one row, display the
values, and then update the values.
```
dmSQL> INSERT INTO sql99t5 VALUES(CURRENT_TIME, CURRENT_DATE, CURRENT_TIMESTAMP,
CURRENT_USER);
1 row inserted

dmSQL> SELECT CURRENT_TIME, CURRENT_DATE, CURRENT_TIMESTAMP, CURRENT_USER;

CURRENT_TIME   CURRENT_D*   CURRENT_TIMESTAMP           CURRENT_USER
============   ==========   ======================   ==========================
16:53:09      2001-09-26 2001-09-26 16:53:09       SYSADM

dmSQL> UPDATE sql99t5 SET c1 = CAST(CURRENT_TIMESTAMP AS CHAR(20)),
                c2 = CURRENT_DATE,
                c3 = CURRENT_TIME,
```

```
                   c4 = CURRENT_TIMESTAMP WHERE c1 = CURRENT_USER;
1 row updated
```

4.1.30.   CURTIME

The CURTIME function returns the current time.

Return value ....Time. The current time

CURTIME ( )

Figure 4-30 CURRENTTIME syntax

➲ **Example**

The following syntax returns the current time.
```
CURTIME ()
```

4.1.31.   DATABASE

The DATABASE function returns the name of the database corresponding to the current connection. Alternately, determine the name of the database in an ODBC program by calling the SQLGetConnectOption with the SQL_CURRENT_QUALIFIER connection option.

Return value.............String: The name of the database on the current connection

DATABASE ( )

Figure 4-31 DATABASE syntax

➲   **Example**

The following returns the name of the database corresponding to the current connection.

```
DATABASE()
```

4.1.32.   DATEPART

The DATEPART function returns the date part of *timestamp*.

timestamp .... Timestamp: Timestamp to extract the date part from

Return value .... Date: Date part of *timestamp*

DATEPART (*timestamp*)
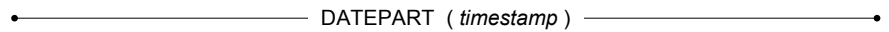
Figure 4-32 DATEPART syntax

➲ **Example**

The following syntax returns the date **1999-08-07**.
```
DATEPART('1999-08-07 10:11:12.123')
```

4.1.33. DAYNAME

The DAYNAME function returns a character string containing the data-source specific name of the day (for example, Sunday, Monday, …, Saturday) that *date* falls on.

date..........................Date: Date to find the name of the day for

Return value.............String: Weekday that *date* falls on

<div style="text-align:center;">

DAYNAME (*date*)

</div>

Figure 4-33 DAYNAME syntax

➲ **Example**

The following returns "**Saturday**".
```
DAYNAME('1999-12-25')
```

4.1.34. DAYOFMONTH

The DAYOFMONTH function returns the day of the month found in *date* as an integer value in the range 1-31.

date .... Date: Find the day of the month for *date*

Return value .... Integer: Day of the month that *date* falls on

DAYOFMONTH (*date*)

Figure 4-34 DAYOFMONTH syntax

➲ **Example**

The following returns **23**.
```
DAYOFMONTH('1999-01-23')
```

4.1.35.   DAYOFWEEK

The DAYOFWEEK function returns the day of the week found in *date* as an integer value in the range 1-7, where 1 is Sunday, 2 is Monday, …, and 7 is Saturday.

date..........................Date: Find the day of the week for *date*

Return value.............Integer: Day of the week that *date* falls on

—————————— DAYOFWEEK (*date*) ——————————

Figure 4-35 DAYOFWEEK syntax

➲ **Example 1**

The following returns **3**.
```
DAYOFWEEK('2000-02-29')
```

➲ **Example 2**

The following returns **6**.
```
DAYOFWEEK('2000-03-03')
```

4.1.36.   DAYOFYEAR

The DAYOFYEAR function returns the day of the year found in *date* as an integer value in the range 1-366, 366 is only returned for the last day of a leap year.

date .... Date: Find the day of the year for *date*

Return value .... Integer: Day of the year that *date* falls on

DAYOFYEAR (*date*)

Figure 4-36 DAYOFYEAR syntax

➲ **Example 1**

The following returns **31**.
```
DAYOFYEAR('1999-01-31')
```

➲ **Example 2**

The following returns **365**.
```
DAYOFYEAR('1999-12-31')
```

4.1.37.   DAYS_BETWEEN

The DAYS_BETWEEN function returns the number of days between two dates. The *date1* argument can be earlier or later than the *date2* argument.

date1........................Date: First of two dates

date2........................Date: Second of two dates

Return value.............Integer: Number of days between *date1* and *date2*

—————————————— DAYS_BETWEEN (*date1, date2*) ——————————————

Figure 4-37 DAYS_BETWEEN syntax

âž¡ **Example 1**

The following returns **31**.
```
DAYS_BETWEEN('1999-01-15', '1999-02-15')
```

âž¡ **Example 2**

The following returns **31**.
```
DAYS_BETWEEN('1999-02-15', '1999-01-15')
```

4.1.38.   DEGREES

The DEGREES function returns the number of degrees in *radians* as a double precision floating-point number.

radians .... Date: Radians value to convert to degrees

Return value .... Double: Number of degrees in *radians*

DEGREES (*radians*)
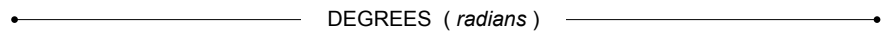
Figure 4-38 DEGREES syntax

⮞ **Example**

The following returns **1.79908747671078e+002**.
```
DEGREES(3.14)
```

4.1.39.   DOCTOTXT

The DOCTOTXT function is used for converting Microsoft Word documents into a temporary BLOB containing the pure text of blob as unicode. It returns temp blob or NULL. In DBMaker current version, UDF will support office 2007- 2010 version.

Blob: ....Column name to be converted to pure text

Return value:...........temp BLOB as NCLOB type if blob is convertable to pure text.

$$\longmapsto \text{COS} \ (\ number\ ) \longmapsto$$

Figure4-39 Syntax for DOCTOTXT

➲ **Example**

The following example illustrates converting the column memo to puretext.

```
DOCTOTXT(memo)
```

4.1.40.   EXISTSNODE

Function existsNode is used to check if specified node is found or not.

*Xmldata………………*XML content to be queried

*Xpath-expression……*user will use to query xmldata

Namespaces……….. optionally specifies the namespace(s) used in xpath-expression

*Returnvalue…………*the result will be serialized into NCLOB.

$$\bullet\text{———} \text{EXISTSNODE}\,(\text{XMLdata}, \text{xpath - expression}, \text{namespaces}) \text{———}\bullet$$
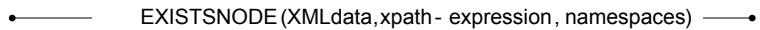
Figure 4-40 EXISTSNODE syntax

➲ **Example**

This example illustrates creatring an index using the **existsnode** XML UDF:

```
dmSQL> CREATE INDEX idx1 ON t1 (EXISTSNODE(c1, '/order/items/item/@product',
NULL));
```

4.1.41.   EXP

The EXP function returns the exponential function e^x as a double precision floating-point number.

x ....Double: Power to raise the natural logarithm to

Return value ....Double: Natural logarithm (e) to the power of *x*

<p style="text-align:center">━━━━━━━━━━━━ EXP (x) ━━━━━━━━━━━━</p>

Figure 4-41 EXP syntax

➲ **Example**

The following returns **2.71828182845905e+000**.
```
EXP(1)
```

4.1.42.   EXTRACT

The EXTRACT function returns the multi-value, one value or zero value. It not allow asc/desc and unique index

Return value: .... *UDF: allows multi values, one value and zero value of the UDF results*

EXTRACT ()

Figure 4-42 EXTRACT syntax

➲   **Example**

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

```
dmSQL> CREATE INDEX idx1 ON t1 (EXTRACT(c1, '/order/items/item/@product', NULL));
```

4.1.43.   EXTRACTVALUE

The EXTRACTVALUE function only returns the one value or zero value. It allow asc/desc and unique index

Return value:............UDF: *allow one and zero value UDF results, but not* multi-value

•———————————— EXTRACTVALUE () ————————————•

Figure 4-43 EXTRACTVALUE syntax

➲ **Example**

To create an index use the **extractValue** XML UDF:
```
dmSQL> CREATE INDEX idx2 ON t1 (EXTRACTVALUE(c1, '/order/items/item/@product',
NULL));
```

4.1.44.   FILEEXIST

The FILEEXIST function determines if the file object specified by *fileobject* exists as a physical file. Possible return values are 1 for a file that exists, and 0 file a file that does not exist.

fileobject ....File: File object to check the existence of

Return value ....Integer: Boolean value indicating whether the file exists

FILEEXIST (*fileobject*)

Figure 4-44 FILEEXIST syntax

⮕ **Example 1**

The following returns **1**, indicating the file exists.
```
FILEEXIST(file_column)
```

⮕ **Example 2**

The following returns **0**, indicating the file does not exist.
```
FILEEXIST(nofile_column)
```

4.1.45.   FILELEN

The FILELEN function returns the file size of *fileobject* as an integer value. And it will report at most $(2^{31} - 1)$B even if the size is greater than or equal to 2^{31}B. The *fileobject* argument must be a column in the database of the FILE data type.

fileobject...................File: File to find the length of

Return value.............Integer: Length of the file in bytes

FILELEN (*fileobject*)

Figure 4-45 FILELEN syntax

➲ **Example**

The following returns **211** for a file that is **211** bytes in size.
```
FILELEN(file_column)
```

4.1.46.   FILELENEX

The FILELENEX function returns the file size of *fileobject* as a decimal value. The *fileobject* argument must be a column in the database of the FILE data type. And unlike FILELEN function, it can report the correct size for FOs > 2^{31}B.

fileobject ....File: File to find the length of

Return value ....Decimal: Length of the file in bytes

FILELEN (*fileobject*)
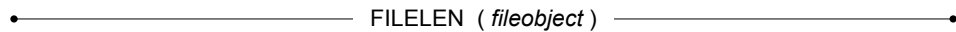
Figure 4-46 FILELENEX syntax

➲ **Example**

The following returns **211** for a file that is **211** bytes in size.

```
FILELENEX(file_column)
```

4.1.47.   FILENAME

The FILENAME function returns the file name of *fileobject* as a string. The *fileobject* argument must be a column in the database of the FILE data type.

fileobject...................File: File to find the name of

Return value.............String: Name of the file

FILENAME (*fileobject*)

Figure 4-47 FILENAME syntax

➲ **Example**

The following returns **C:\PATH\MYFILE.FIL**.

```
FILENAME(file_column)
```

4.1.48.   FIX

The FIX function returns an integer value for the integral part of *number*.

number....................Double: Number to find the integral part of

Return value ....Bigint: Integral part of *number*

FIX (*number*)

Figure 4-48 FIX syntax

➲   Example 1

The following returns **11**.
```
FIX(11.99)
```

➲   Example 2

The following returns **12**.
```
FIX(12.01)
```

➲   Example 3

The following returns a value of **–11**.
```
FLOOR(-11.99)
```

➲   Example 4

The following returns a value of **–12**.
```
FLOOR(-12.01)
```

4.1.49.   FLOOR

The FLOOR function returns a double-precision floating-point value for the greatest integral value less than or equal to *number*.

number ....Double: Number to find the next integral value less than

Return value.............Double: Integral part of *number*

FLOOR (*number*)

Figure 4-49 FLOOR syntax

➲ **Example 1**

The following returns **1.20000000000000e+001**.
```
FLOOR(12.01)
```

➲ **Example 2**

The following returns **1.10000000000000e+001**.
```
FLOOR(11.99)
```

➲ **Example 3**

The following returns -**1.20000000000000e+001**.
```
FLOOR(-11.99)
```

➲ **Example 4**

The following returns -**1.30000000000000e+001**.
```
FLOOR(-12.01)
```

4.1.50.   FREXPE

The FREXPE function returns the exponent n from the equation $number = X \times 2^{n}$ as an integer value, where the value of X is in the range $0.5 < \left| X \right| < 1$.

number....................Double: Number to find the next exponent n for from the

equation $number = X \times 2^{n}$

Return value ....Integer: Exponent n from the equation $number = X \times 2^{n}$
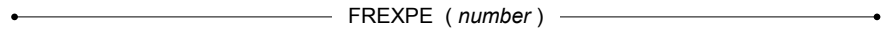
FREXPE (*number*)

Figure 4-50 FREXPE syntax

➲   **Example**

The following returns **3**, where ***n*** must equal **3** when *number* equals **4.0** and **X** is restricted to values between **0.5** and **1**.

```
FREXPE(4.0)
```

4.1.51.   FREXPM

The FREXPM function returns the mantissa X from the equation $number = X \times 2^n$ as a double-precision floating-point number, where the value of X is in the range $0.5 < \left| X \right| < 1$.

number ....Double. Number to find the next mantissa X for from the equation $number = X \times 2^n$.

Return value.............Integer. Mantissa X from the equation $number = X \times 2^n$.
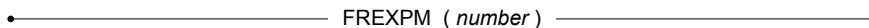
———————————————— FREXPM (*number*) ————————————————

Figure 4-51 FREXPM syntax

➲   **Example**

The following returns the value of **5.00000000000000e-001**, which means **X** must equal **0.5** or **5.00000000000000e-001** when *number* equals **4.0** and ***n*** equals an exact integer value.

```
FREXPM(4.0)
```

4.1.52.   FTOA

The FTOA function returns a string containing *number* with a fixed amount of digits after the decimal point. The *digits* argument specifies the number of digits after the decimal point, and the *format* argument specifies whether the return value should be in regular decimal format or exponential format.

The format argument has four possible values, "f", "F", "e", and "E". Using "f" or "F" returns a string in regular decimal format, for example, 123.45, when *digits* is 2. Using "e" or "E" returns a string in exponential format, for example, 1.23e+02. After conversion, the exponential digits will be converted to the regular decimal equivalent.

number .... Double: Number to convert to a string

digits .... Integer: Number of digits after the decimal

format .... String: Format to return the number in

Return value .... String: String containing *number* with a fixed number of digits in the specified format

FTOA (number, digits, format)
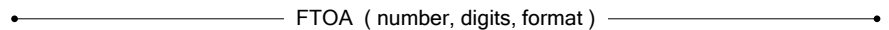
Figure 4-52 FTOA syntax

⮡ **Example 1**

The following syntax returns the value "**123.46**".
```
FTOA(123.456789, 2, 'f')
```

⮡ **Example 2**

The following syntax returns the value "**1.23e+02**".
```
FTOA(123.456789, 2, 'e')
```

4.1.53.   HIGHLIGHT

The HIGHLIGHT function returns the modified source text in which all of the matching text patterns will be highlighted with **preTag** and **endTag** before and after.

At most 10000 (MaxTagSpace) byte tags can be added. If the pattern contains Boolean operators [&, |, !, (,)], all the simple searching pattern will be tagged except the ! (NOT) patterns. The inputted text's type can be CLOB, file, char or media type.

If the inputted text's type is XMLTYPE, the HIGHLIGHT function returns Error 6536, in this case, users can call the function PURETEXT to convert data with XMLTYPE into that with NCLOB and then highlight patterns that matches conditions.

text ....CLOB: Source Text

BoolPatn ....Char: Patterns to be hilighted, can be Boolean expression pattern

sensitive ....Integer: Whether the match is case sensitive, 1 means yes and 0 means no

PreTag ....Char: Tag before pattern, NULL denotes none

EndTag ....Char: Tag after pattern, NULL denotes none

Return value ....NCLOB: Modified source text after highlighting patterns

HIGHLIGHT(*text, BoolPatn, sensitive, PreTag, EndTag*)

Figure 4-53 HIGHLIGHT syntax

➲ **Example 1**

The following will return the modified content in which all "**Intel**" or "**AMD**" are highlighted with **preTag** "**<<**" and **endTag** "**>>**".

```
dmSQL> SELECT HIGHLIGHT(content,'Intel | AMD',0,'<<','>>') FROM news WHERE
content MATCH 'Intel| AMD';
```

 Example 2

The following will return the modified content in which all "**dbmaker**" is highlighted with **preTag "<"** and **endTag ">"**.

```
dmSQL> CREATE TABLE tpdf(c1 SERIAL, c2 pdffiletype);
dmSQL> SELECT HIGHLIGHT(c2,'dbmaker',0,'<','>') FROM tpdf;
```

 Example 3

The following will return the modified content in which all "**dbmaker**" is highlighted with **preTag "<"** and **endTag ">"**.

```
dmSQL> CREATE TABLE txml(c1 SERIAL,c2 XMLTYPE);
dmSQL> SELECT HIGHLIGHT(PURETEXT(c2), 'dbmaker',0,'<','>') FROM txml;
```

4.1.54. HITCOUNT

The HITCOUNT function returns the frequency of patterns found in source text.

Rule of count values for Boolean patterns are:

- a AND b : min(count(a), count(b))

- a OR b : count(a) + count(b)

- NOT a : count = 0

text ....CLOB: Source text

BoolPatn ....Char: Patterns to be highlighted can be Boolean expression
patterns

sensitive ....Integer: Whether the match is case sensitive, 1/0 means yes/no,
respectively

Return value ....Integer: The frequency of searched text patterns in the source text

<div align="center">

HITCOUNT (text*, BOOlPatn, sensitive*)

</div>

Figure 4-54 HITCOUNT syntax

➲ **Example**

The following returns the frequency of "**target**" found in source data "**content**", and
the finding is case insensitive.

```
HITCOUNT(content, "target", 0)
```

4.1.55.   HITPOS

The HITPOS function shows the position information of the nth pattern found in source text, the offset can be: start offset, end offset, pattern length, begin offset (higher than 24 bits), BINARY, OR end offset (lower 8 bits). The offset starts at 1.

text .... CLOB: Source Text

BoolPatn .... Char: Patterns to be hilighted can be Boolean expression pattern

sensitive .... Integer: Whether the match is case sensitive, 1/0 means yes/no, respectively

n .... Integer: The nth pattern in source text

RetType .... Char: Return position type:

> 0: begin offset (default setting)
>
> 1: end offset
>
> 2: pattern length (endoff - begoff + 1)
>
> 3: begin offset (higher 24 bits) BINARY OR end offset
>
>   (lower 8 bits)

Return value .... Integer: Get position information of the nth pattern found in source text. If nth pattern is not found, the value is 0

<div align="center">

HITPOS(*text* , *BoolPatn, sensitive,n* , *RetType*)

</div>

<div align="center">

Figure 4-55 HITPOS syntax

</div>

➲ **Example**

The following examples return **5**, **3**, **5** and **7** using the source text "**a b A c**".

```
HITPOS(src,'A', 1, 1, 0) = 5 ('A')
```

```
HITPOS(src,'A&B' 0, 2, 0) = 3 ('b')
HITPOS(src,'a|b|c', 0, 3, 0) = 5 ('A')
HITPOS(src,'!a&c' 0, 1, 0) = 7 ('c')
```

4.1.56.   HMS

The HMS function returns the time *hours*: *minutes: seconds* in time format. The *hours'* argument represents the hours' component of the time, and has valid values from 0 to 23. Hours must be entered using the 24-hour format; there is no method provided for entering values for AM and PM to indicate the time in 12-hour format. The *minutes'* argument represents the minutes' component of the time, and has valid values from 0 to 59. The *seconds'* argument represents the seconds' component of the time, and has valid values from 0 to 59.

hours .... Integer: Hours component of the time

minutes .... Integer: Minutes component of the time

seconds .... Integer: Seconds component of the time

Return value .... Time: Time format composite of *hours*, *minutes,* and *seconds*

HMS (*hours, minutes, seconds*)
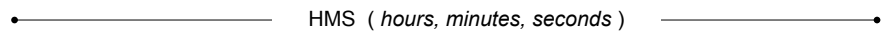
Figure 4-56 HMS syntax

➲ **Example 1**

The following returns **10:11:12**, which is equivalent to **10:11:12 AM**.
```
HMS(10, 11, 12)
```

➲ **Example 2**

The following returns **22:11:12**, which is equivalent to **10:11:12 PM**.
```
HMS(22, 11, 12)
```

4.1.57. HOUR

The HOUR function returns the hour in *time* as an integer value in the range from 0 to 23.

time ....Time: Time to find the hour component of

Return value.............Integer: Hour component of *time*

HOUR (*time*)

Figure 4-57 HOUR syntax

➲ **Example 1**

The following returns **10**.
```
HOUR('10:11:12')
```

➲ **Example 2**

The following returns **22**.
```
HOUR('PM 10:11:12')
```

4.1.58.   HTMLHIGHLIGHT

The HTMLHIGHLIGHT function returns modified source data in which all text matching patterns will be highlighted with **preTag** and **endTag** before and after. HTMLHIGHLIGHT also provides a highlight function to quote the patterns in an HTML file without destroying the HTML document structure.

At most 10000 (MaxTagSpace) byte tags can be added. If the pattern contains Boolean operators [&, |, !, (,)], all the simple searching pattern will be tagged expect the ! (NOT) patterns. The input text can be CLOB, file or char type. No content inside tags, including comments, will be highlighted. All tags (include comments) are treated as SPACE character. For example, if pattern is "DBMaker License", then the HTML data "DBMaker
License" will be highlighted. However, if the HTML data is "<B>DBM</B>aker", it will not match "DBMaker" pattern! Only the data after <BODY> can be highlighted.

text .... CLOB: Source text.

BoolPatn .... Char: Patterns to be highlighted can be Boolean expression pattern

sensitive .... Integer: Whether the match is case sensitive, 1/0 means yes/no, respectively

PreTag .... Char: The tag after pattern, NULL denotes none

EndTag .... Char: The tag after pattern, NULL denotes none

Return value .... BLOB: The modified text after highlighting patterns

HTMLHIGHLIGHT(*text, BoolPatn, sensitive, PreTag, EndTag*)

Figure 4-58 HTMLHIGHLIGHT syntax

➲ Example

The following returns modified content in which all text matching "**Intel**" or "**AMD**" will be highlighted with "**<<**" and "**>>**" before and after.

```
HTMLHIGHLIGHT(content,'Intel | AMD',0,'<<','>>')
```

4.1.59.   HTMLTITLE

The HTMLTITLE function finds the title (text between *html tags* "*<title>*" and "*</title>*" in source HTML data) of HTML data.

object .... BLOB: Source HTML data

Return value .... Varchar: Return the title of the source HTML data

HTMLTITLE(*object*)
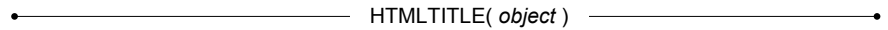
Figure 4-59 HTMLTITLE syntax

➲ **Example**

The following returns title in source HTML data "**htmlFile**".

```
HTMLTITLE(htmlFile)
```

4.1.60.   HTMTOTXT

The HTMTOTXT function can be used to convert html document to a temporary BLOB containing the pure text of blob as local code.

Blob: ....Column name on which to be converted to pure text

Return value:............temp blob as CLOB type if blob could be converted to pure text

HTMTOTXT (*blob*)

Figure4-60 Syntax for HTMTOTXT

➔ **Example**

The following will convert the column memo to puretext.

```
HTMTOTXT (memo)
```

4.1.61.   HYPOT

The HYPOT function returns the length of the hypotenuse of a right angle triangle as a double precision floating-point number. The hypotenuse is calculated according to the equation $z^2 = x^2 + y^2$ (Pythagorean Theorem), where z is the length of the hypotenuse.

x..............................Double: Length of one leg of the right triangle you are finding the hypotenuse for

y..............................Double: Length of the other leg of the right triangle you are finding the hypotenuse for

Return value ....Double: Length of the hypotenuse of the right triangle

$$\text{---------- HYPOT } (\, x, y \,) \text{ ----------}$$
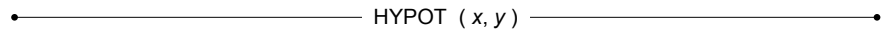
Figure 4-61 HYPOT syntax

➲   **Example**

The following returns **5**.
```
HYPOT(3,4)
```

4.1.62.   INSERT

The INSERT function returns a character string where *length* characters from *string1* have been replaced by *string2* beginning at *start*. The value of *start* indicates the position in *string1* where the first character of *string2* is placed. If the value of *length* is zero, *string2* is inserted into *string1* without replacing any characters. An error is returned if a value for all arguments is not provided.

DBMaker uses the following rules to determine the value returned if one of the string expressions contains a NULL value or if one of the integer arguments contains an *atypical* value:

- ◆   If *string1* contains a *NULL* value, the function returns a *NULL* value

- ◆   If *start*, *length*, or *string2* contains a *NULL* value, the function returns the string expression in *string1*

- ◆   If the value of *start* is less than or equal to zero, or the value of length is less than zero, the function returns the string expression in *string1*

- ◆   If the value of *start* is greater than the length of *string1* plus one, the function returns the string expression in *string1*

string1.....................String: String to insert characters into

start ....Integer: Position where the first character from *string2* is inserted in *string1*

length......................Integer: Number of characters to replace in *string1*

string2.....................String: String to insert into the original source string

Return value.............String: String formed by inserting *string1* in *string2*

INSERT (*string1, start, length, string2*)

Figure 4-62 INSERT syntax

⮕ **Example 1**

The following returns the string "**Good ng!**"
```
INSERT('morning!', 1, 5, 'Good ')
```

⮕ **Example 2**

The following returns the string "**Good morning!**"
```
INSERT('Good ', 6, 8, 'morning!')
```

⮕ **Example 3**

The following returns the string "**Good night!**"
```
INSERT('Good morning!', 6, 7, 'night')
```

⮕ **Example 4**

The following returns the string "**Good morning, sir. Here is your coffee.**"
```
INSERT('Good morning! Here is your coffee.', 13, 1, ', sir.')
```

4.1.63.   INVDATE

The INVDATE function determines if the date specified by the *date* argument is valid. Possible return values are:

- 1 for invalid dates (e.g., out of date range)

- 0 for valid dates (e.g., '0001-01-01' to '9999-12-31')

- -1 for dates with unknown values (e.g., *NULL* values)

date..........................Date: Date to check the validity of

Return value.............Integer: Boolean value indicating whether the date is valid

INVDATE (date)

Figure 4-63 INVDATE syntax

➲ **Example**

The following returns a **0**, indicating the date is valid.

```
INVDATE('2000-01-01')
```

4.1.64.   INVTIME

The INVTIME function determines if the time specified by the *time* argument is valid. Possible return values are:

◆   1 for invalid times (e.g., out of time range)

◆   0 for valid times (e.g., '00:00:00' to '24:00:00')

◆   -1 for times with unknown values (e.g., *NULL* values)

time......................... Time: Time to check the validity of

Return value .... Integer: Boolean value indicating whether the time is valid
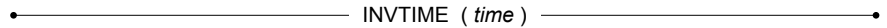
INVTIME (*time*)

Figure 4-64 INVTIME syntax

➲   **Example**

The following returns a **0**, indicating the time is valid.
```
INVTIME('01:01:01')
```

4.1.65. INVTIMESTAMP

The INVTIMESTAMP function determines if the timestamp specified with a *timestamp* argument is valid. Possible return values are:

- ◆ 1 for invalid timestamps (e.g., out of timestamp range)

- ◆ 0 for valid timestamps (e.g., '00:00:00' to '24:00:00')

- ◆ -1 for timestamps with unknown values (e.g., *NULL* values)

timestamp.................Timestamp: Timestamp to check the validity of

Return value.............Integer: Boolean value indicating whether the timestamp is valid

INVTIMESTAMP (*timestamp*)

Figure 4-65 INVTIMESTAMP syntax

➲ **Example**

The following returns a **0**, indicating the timestamp is valid.
```
INVTIMESTAMP('1999-08-07 10:11:12.123')
```

4.1.66. LAST_DAY

The LAST_DAY function returns the last date in the same month as the date specified in the *date* argument.

date .... Date: Date to find the last date in the same month of

Return value .... Date: Last date in the same month as *date*

LAST_DAY (*date*)

Figure 4-66 LAST_DAY syntax

➲ **Example 1**

The following returns '**1996-02-29**'.
```
LAST_DAY('1996-02-08')
```

➲ **Example 2**

The following returns '**2002-12-31**'.
```
LAST_DAY('2002-12-25')
```

4.1.67.   LCASE

The LCASE function converts all upper case letters in *string* to lower case; numbers and symbols are not affected. If the string argument is NULL, a NULL value is returned. If you do not provide a value for the *string* argument, an error will be returned.

string........................String: Text to convert to lower case

Return value.............String: Text from the *string* argument in lower case

●────────────────────── LCASE (*string*) ──────────────────────●

Figure 4-67 LCASE syntax

➲ **Example 1**

The following returns the string "**abcdef**".
```
LCASE('ABCdef')
```

➲ **Example 2**

The following returns the string "**abc123**".
```
LCASE('ABC123')
```

➲ **Example 3**

The following returns the string "**abc@#$**".
```
LCASE('ABC@#$')
```

4.1.68. LDEXP

The LDEXP function returns the result of the equation $number = X \times 2^n$ as a double precision floating-point number.

x .... Double: Mantissa *x* from the equation $number = X \times 2^n$

n .... Integer: Exponent *n* from the equation $number = X \times 2^n$

Return value .... Double: Result of the equation $number = X \times 2^n$

LDEXP (*x*, *n*)

Figure 4-68 LDEXP syntax

➲ **Example**

The following returns **8.00000000000000e+000**.
```
LDEXP(0.5, 4)
```

4.1.69.   LEFT

The LEFT function returns the leftmost *count* characters in *string*. If the value of *count* is less than zero, a NULL value is returned. All arguments must be provided otherwise an error is returned.

string........................String: String to extract characters from

count........................Integer: Number of characters to extract

*Return value.............*String: Leftmost *count* characters in *string*

LEFT (*string,count*)

Figure 4-69 LEFT syntax

➲ **Example**

The following returns the string "**Good**".
```
LEFT('Good morning!', 4)
```

4.1.70.   LENGTH

The LENGTH function returns the number of characters in *string*, excluding trailing blanks and the string termination character, when present. An error is returned if a value for the *string* argument is not provided.

string .... String: String to find the length of

Return value .... Integer: Leftmost *count* characters in *string*
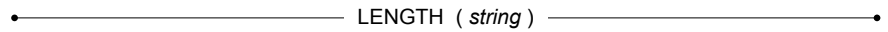
LENGTH (*string*)

Figure 4-70 LENGTH syntax

➲   **Example**

The following returns **13**.
```
LENGTH('Good morning!   ')
```

4.1.71. LOCATE

The LOCATE function returns the starting position of the first occurrence of *string1* in *string2*. The search for the first occurrence of *string1* begins with the character position specified by *start*. Assigning a value of 1 to *start* indicates the search should begin with the first character in *string2*. If *string1* is not found in *string2*, a value of 0 is returned. DBMaker uses the following rules to determine the value returned if one of the string expressions contains a NULL value or when *start* contains an atypical value:

◆   If *string1* contains a *NULL* value, the function will return a *NULL* value

◆   If *string2* or *start* contain a *NULL* value, the function will return 0

◆   If *start* is less than or equal to zero, the function will return the correct value

◆   If *start* is greater than the length of *string2* plus one, the function will return 0

string1......................String: String to locate

string2......................String: String to search

start ....Integer: Position in *string2* to start searching

Return value.............Integer: Starting position of *string1* in *string2*

LOCATE (*string_exp1, string_exp2, 1*)

Figure *4-71 LOCATE syntax*

➲ **Example 1**

The following syntax returns a value of **4**.
```
LOCATE('def', 'abcdefghi', 1)
```

➲ **Example 2**

The following syntax returns the value of **0**.

```
LOCATE('def', 'abcdefghi', 5)
```

‌ **Example 3**

The following syntax returns a value of **4**.

```
LOCATE('def', 'abcdefghi', 4)
```

‌ **Example 4**

The following syntax returns a value of **4**.

```
LOCATE('def', 'abcdefghi', -1)
```

‌ **Example 5**

The following syntax returns a value of **0**.

```
LOCATE('def', 'abcdefghi', 10)
```

4.1.72.   LOG

The LOG function returns the natural logarithm of *x* as a double-precision floating-point number.

x..............................Double: Value to find the natural logarithm of

Return value.............Double: Natural logarithm of *x*

LOG (*x*)

Figure 4-72 LOG syntax

➲ **Example**

The following returns **1.00000000000000e+000**.

```
LOG(2.71828182845905e+000)
```

4.1.73.   LOG10

The LOG10 function returns the logarithm with base 10 of x as a double precision floating-point number.

x...............................Double: Value to find the natural logarithm with base 10 of x

Return value ....Double: Natural logarithm with base 10 of x
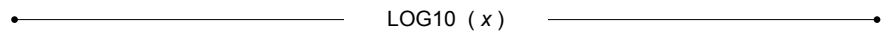
LOG10 (x)

Figure 4-73 LOG10 syntax

➲ **Example**

The following returns **2**.
```
LOG10(100)
```

4.1.74.   LOWER

The LOWER function performs the same calculation as LCASE. It makes all characters in the string lower case characters.

String_expression ......   String: string to convert all characters in lower case

Return value.................String: the returned characters in lower case converted from characters in upper case

LOWER (*string_expression*)

Figure 4-74 Lower function syntax

➲ **Example**

```
dmSQL> SELECT LOWER('ABCDEF');
LOWER('ABCDEF')
================
abcdef
```

4.1.75.   LTRIM

The LTRIM function returns the characters of *string* with leading blanks removed. All arguments must be provided otherwise an error is returned.

string .... String: String to trim characters from the left of

Return value .... String: String with leading blanks removed
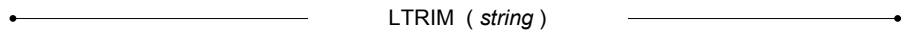
LTRIM (*string*)

Figure 4-75 LTRIM syntax

➲ **Example**

The following returns the string "**Good morning!**"
```
LTRIM('   Good morning!')
```

4.1.76.   MDY

The MDY function returns the date *month*/*day*/*year* in the current date format. The *month* argument represents the month component of the date, and has valid values from 1 to 12. The *day* argument represents the day component of the time, and has valid values from 1 to 31. The *year* argument represents the year component of the time, and has valid values from 0001 to 9999.

month ....Integer: Month component of the date

day ....Integer: Day component of the date

year ....Integer: Year component of the date

Return value ....Date: Date format composite of *hours*, *minutes* and *seconds*

MDY (*month, day, year*)

Figure 4-76 MDY syntax

➲ **Example 1**

The following returns the date **1996-02-08** when the current date format is set to **yyyy-mm-dd**.
```
MDY(2,8,1996)
```

➲ **Example 2**

The following returns the date **02/08/2001** when the current date format is set to **mm/dd/yyyy**.
```
MDY(2,8,2001)
```

4.1.77.   MINUTE

The MINUTE function returns the minutes in *time* as an integer value in the range from 0 to 59.

time .... Time: Time to find the minute component of

Return value .... Integer: The minute component of *time*

Figure 4-77 MINUTE syntax

➲ **Example**

The following returns **11**.
```
MINUTE('10:11:12')
```

4.1.78.   MOD

The MOD function returns the remainder, modulus, of *x* divided by *y* as a double precision floating-point number.

x..............................Double: Dividend

y..............................Double: Divisor

Return value.............Double: Remainder

MOD (*x, y*)
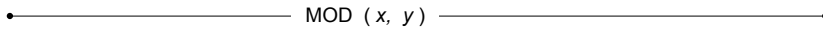
Figure 4-78 MOD syntax

➲ **Example**

The following returns **2.00000000000000e+000**.

```
MOD(17, 3)
```

4.1.79.   MODFI

The MODFI function returns a double precision floating-point number for the integer part of *number*.

number....................Double: Number to determine the integer part of

Return value ....Double: Integer part of *number*

MODFI (*number*)
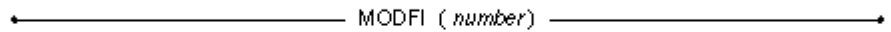
Figure 4-79 MODFI syntax

➲ **Example 1**

The following returns **3.00000000000000e+000**.
```
MODFI(3.1415926535897936)
```

➲ **Example 2**

The following returns -**3.00000000000000e+000**.
```
MODFI(-3.1415926535897936)
```

4.1.80. MODFM

The MODFM function returns a double-precision floating-point number for the mantissa part of *number*.

number ....Double: Number to determine the mantissa part of

Return value ....Double: Mantissa part of *number*

MODFM (*number*)
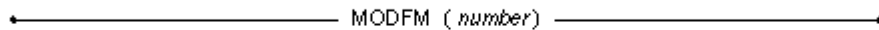
Figure 4-80 MODFM syntax

⮕ **Example 1**

The following returns the value of **1.41592653589790e-001**.
```
MODFM(3.1415926535897936)
```

⮕ **Example 2**

The following returns the value of **-1.41592653589790e-001**.
```
MODFM(-3.1415926535897936)
```

4.1.81. MONTH

The MONTH function returns the month in *date* as an integer value in the range from 1 to 12.

date .... Date: Date to find the month component of

Return value .... Integer: The month component of *date*

MONTH (*date*)
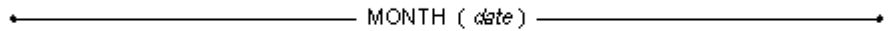
Figure 4-81 MONTH syntax

➜ **Example**

The following returns **2**.
```
MONTH('1996-02-29')
```

4.1.82. MONTHNAME

The MONTHNAME function returns a character string containing the data-source specific name of the month (e.g., JAN, FEB, …, DEC) that *date* falls on. The *date* argument must be a valid date or DBMaker will return an error.

date..........................Date: Date to find the name of the month for

Return value.............String: The name of the month that *date* falls in
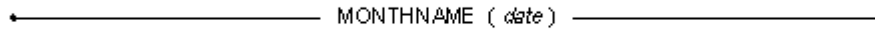
<div align="center">MONTHNAME (date)</div>

Figure 4-82 MONTHNAME syntax

➲ **Example**

The following returns "**FEB**".
```
MONTHNAME('1996-02-29')
```

4.1.83.   NEXT_DAY

The NEXT_DAY function returns the date proceeding the *date* that *weekday* falls on. Valid values for the *weekday* argument are the names of the days of the week (Monday, Tuesday, …, Sunday) or their abbreviations (Mon, Tue, …, Sun). Values for *weekday* are not case-sensitive.

date .... Date: Date after which to find the next date that a weekday falls on

weekday .... String: Weekday the date will fall on

Return value .... Date: Next date after *date* that *weekday* falls on

NEXT_DAY (*date, weekday*)
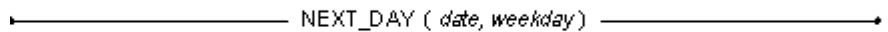
Figure 4-83 NEXT_DAY syntax

➲ **Example 1**

The following syntax returns the date **1996-03-04**.
```
NEXT_DAY('1996-02-29', 'Monday')
```

➲ **Example 2**

The following syntax returns the date **1996-03-05**.
```
NEXT_DAY('1996-02-29', 'Tuesday')
```

4.1.84.   NOW

The NOW function returns the current date and time as a timestamp value.

Return value.............Timestamp: The current date and time

NOW ()

Figure 4-84 NOW syntax

4.1.85.   **PDFTOTXT**

The PDFTOTXT function can be used to converts pdf document to a temporary BLOB containing the pure text of blob as unicode. it will return temp blob or NULL. Please note that PDF's formats supported by DBMaker are 1.2, 1.3, 1.4, 1.5, 1.6 and 1.7.

Blob: ....Column name on which to be converted to pure text

Return value: ....temp blob as NCLOB type if blob could be converted to pure text

PDFTOTXT (*blob*)

Figure 4-85 Syntax for PDFTOTXT

➲ **Example**

The following will convert the column memo to puretext.
```
PDFTOTXT(memo)
```

4.1.86.   PI

The PI function returns the constant value of π, 3.1415926535897936, as a decimal number with a precision of 38 and a scale of 16.

Return value.............Decimal: The constant value π

$$\longleftarrow \text{PI ()} \longrightarrow$$
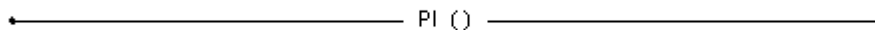
Figure 4-86 PI syntax

4.1.87.   POSITION

The POSITION function returns the starting position of the first occurrence of *string1* in *string2*. If *string1* is not found in *string2*, a value of 0 is returned. DBMaker uses the following rules to determine the value returned if one of the string expressions contains a NULL value or when *start* contains an atypical value:

- If *string1* contains a *NULL* value, the function will return a *NULL* value

- If *string2* or *start* contain a *NULL* value, the function will return 0

string1 .... String: String to locate

string2 .... String: String to search

Return value .... Integer: Starting position of *string1* in *string2*

POSITION (*string_exp1 IN string_exp2*)
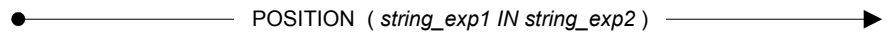
Figure *4-87 POSITION function syntax*

⮑ **Example 1**

The following function command returns the value of "4".
```
dmSQL> SELECT POSITION('abc' in 'defabcjlkjl');
POSITION('ABC' IN 'DEFABCJLKJL')
==============================
                              4
```

⮑ **Example 2**

The following function command returns the value of "1".
```
dmSQL> SELECT POSITION('abc' in 'abcdefghihj');
POSITION('ABC' IN 'ABCDEFGHIHJ')
==============================
                              1
```

➲ **Example 3**

The following function command returns the value of "**0**".

```
dmSQL> SELECT POSITION('abc' in 'jlkjlkklj');
POSITION('ABC' IN 'JLKJLKKLJ')
==============================
                             0
```

4.1.88.   POW

The POW function returns x^y as a double-precision floating-point number.

x...............................Double: Number to raise to a power *y*

y..............................Double: Power to raise number *x* to

Return value ....Double: Value of *x* to the power *y*

POW (x, y)
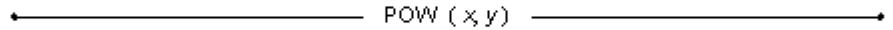
Figure 4-88 POW syntax

➲ **Example**

The following returns **8.00000000000000e+000**.
```
POW(2, 3)
```

4.1.89.   PPTTOTXT

The PPTTOTXT function can be used to convert Microsoft PowerPoint document to a temporary BLOB containing the pure text of blob as unicode.it will return temp blob or NULL. In DBMaker current version, UDF will support office 2007- 2010 version.

Blob: ....Column name on which to be converted to pure text

Return value:............temp blob as NCLOB type if blob could be converted to pure text.

PPTTOTXT (*blob*)

Figure 4-89 Syntax for PPTTOTXT

➔ **Example**

The following will convert the column memo to puretext.

```
PPTTOTXT (memo)
```

4.1.90.   PURETEXT

The PURETEXT function can be use to convert blob to a temporary BLOB containing the pure text of blob as unicode.

When use PURETEXT on the column with media type or a domain with text converter will implicitly call the text converter function.

Blob: .... *Column name on which to be converted to pure text*

Return value: .... *temp blob as NCLOB type if blob could be converted to pure text*

PURETEXT (*blob*)
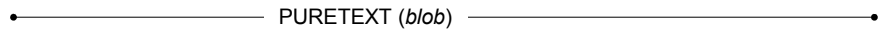
Figure 4-90 Syntax for PURETEXT

➲   **Example**

The following will convert the column memo to puretext.
```
PURETOTXT (memo)
```

4.1.91.   QUARTER

The QUARTER function returns the quarter that *date* falls in as an integer value in the range 1 to 4, where 1 represents January 1 through March 31.

date.........................Date: Date to find the quarter for

Return value.............Integer: The quarter that *date* falls in

```
                    ———————————— QUARTER ( date ) ————————————
```

Figure 4-91 QUARTER syntax

➲ **Example**

The following returns the value of **1**.
```
QUARTER('2002-01-20')
```

4.1.92.   RADIANS

The RADIANS function returns the number of radians in *degrees* as a double precision floating-point number.

degrees .... Double: Number of degrees to convert to radians

Return value .... Double: Number of radians in *degrees*

RADIANS (*degrees*)
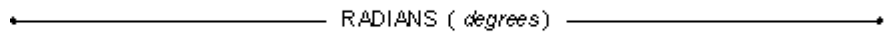
Figure 4-92 RADIANS

➲ **Example**

The following returns **3.14159265358979e+000**.
```
RADIANS(180)
```

4.1.93.   RAND

The RAND function returns a random Integer value.

Return value.............Integer: Random number

<pre>
─────────────────── RAND ( ) ───────────────────
</pre>

Figure 4-93 RAND syntax

4.1.94.   REPEAT

The REPEAT function returns a character string composed of *string* repeated *count* times. DBMaker uses the following rules to determine the value returned if the string expression contains a NULL value or is an empty string. If *string* or *count* contained in a *NULL* value, the function returns a *NULL* value. If *count* is less than 0 or *string* is an empty string, the function returns an empty string. If you do not provide a value for all arguments, an error will be returned.

string .... String: String to repeat

count .... Integer: Number times to repeat *string*

Return value .... String: String composed of *string* repeated *count* times

REPEAT (*string, count*)
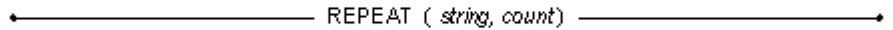
Figure 4-94 REPEAT syntax

⮞ **Example 1**

The following returns the string "**Good morning! Good morning!**"
```
REPEAT('Good morning! ', 2)
```

⮞ **Example 2**

The following returns the string "***Zzzz Zzzz Zzzz Zzzz***".
```
REPEAT('Zzzz ', 4)
```

4.1.95. REPLACE

The REPLACE function replaces all occurrences of *string2* in *string1* with *string3*. DBMaker uses the following rules to determine the value returned if one of the string expressions contains a NULL value or is an empty, zero length, and string:

- ◆ If *string1* is *NULL* return *NULL*

- ◆ If *string2* or *string3* is *NULL* return *string1*

- ◆ If *string2* is empty return *string1*

string1......................String: String to replace characters in

string2......................String: String to replace

string3......................String: String to replace with

Return value.............String: String composed of *string1* with all occurrences of *string2* replaced with *string3*

REPLACE (string1, string2, string3)

Figure 4-95 REPLACE syntax

⮑ **Example 1**

The following returns the string "**Good evening! Good evening!**"
```
REPLACE('Good morning! Good morning!', 'morning', 'evening')
```

⮑ **Example 2**

The following example returns the string "**Goodbye Dave.**"
```
REPLACE('Hello, Dave.', 'Hello,', 'Goodbye')
```

4.1.96.   RIGHT

The RIGHT function returns the rightmost *count* characters in *string*. If the value of *count* is less than zero, a NULL value is returned. All arguments must be provided otherwise an error is returned.

string ....String: String to extract characters from

count ....Integer: Number of characters to extract

Return value ....String: Rightmost *count* characters in *string*

RIGHT (*string, count*)
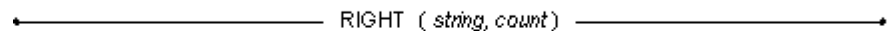
Figure 4-96 RIGHT syntax

● **Example**

The following returns the string "**morning!**"
```
RIGHT('Good morning!  ', 10)
```

NOTE   *There are two spaces after the exclamation point in both the function argument and the return value.*

4.1.97. RND

The RND function rounds number to the nearest integer.

number ....Double: Number to round

Return value.............Bigint: Nearest integer value to *number*

Figure 4-97 RND syntax

➲ **Example 1**

The following returns **12**.
```
RND(12.01)
```

➲ **Example 2**

The following returns **12**.
```
RND(12.49)
```

➲ **Example 3**

The following returns **13**.
```
RND(12.50)
```

➲ **Example 4**

The following returns **13**.
```
RND(12.99)
```

4.1.98.   ROUND

The ROUNDS function returns *number* rounded according to the *decimal_places* argument. The *decimal_places* argument may be a negative number, and the *decimal_places* argument must be an integer.

Rule of rounding for ROUND function are:

◆ If the *decimal_places* argument is omitted, the ROUND function will round the number to 0 decimal places.

◆ If the *decimal_places* argument is bigger than 0, the ROUND function will round off digits right of the decimal point.

◆ If the *decimal_places* argument is equal 0, the ROUND function will round the number to the nearest integer.

◆ If the *decimal_places* argument is smaller than 0, the ROUND function will round off digits left of the decimal point.

number....................Double: Number to round

decimal_places..........Integer: Number of decimal places rounded to

Return value ....Bigint: Nearest integer or decimal value to *number*

ROUND   (*number*, *decimal_places*)
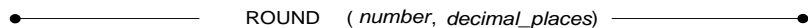
Figure 4-98 ROUND syntax

➲ **Example 1**

The following returns **124**.
```
ROUND(123.56)
```

➲ **Example 2**

The following returns **37.2690000000000000000**.
```
ROUND(37.269412, 3)
```

➲ **Example 3**

The following returns **125.3611000000000000000**.

```
ROUND(125.361080, 4)
```

➲ **Example 4**

The following returns **8912341.0000000000000000000**.

```
ROUND(8912341.123456, 0)
```

➲ **Example 5**

The following returns **1234600.0000000000000000000**.

```
ROUND(1234591.123450, -2)
```

NOTE *Round function return type is decimal(38,19). Therefore, dmsql display would have 19 digit on the right of the decimal point.*

4.1.99.   RTRIM

The RTRIM function returns the characters of *string* with trailing blanks removed. All arguments must be provided otherwise an error is returned.

string .... String: String to trim characters from the right of

Return value .... String: String with trailing blanks removed
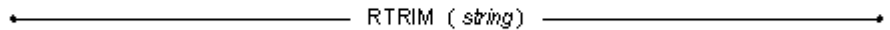
RTRIM (*string*)

Figure 4-99 RTRIM syntax

➲ **Example**

The following returns the string "**Good morning!**"
```
RTRIM('Good morning!  ')
```

NOTE   *There are two spaces after the exclamation point in the function argument.*

4.1.100.   SECOND

The SECOND function returns the seconds in *time* as an integer value in the range from 0 to 59.

time ....Time: Time to find the second component of

Return value.............Integer: The second component of *time*

Figure 4-100 SECOND syntax

➲   **Example**

The following returns **12**.
```
SECOND('10:11:12')
```

4.1.101.   SECS_BETWEEN

The SECS_BETWEEN function returns the number of seconds between two times. The *time1* argument can be earlier or later than the *time2* argument.

time1 ....Time: First time of two to calculate the number of seconds

between

time2 ....Time: Second time of two to calculate the number of

seconds between

Return value ....Integer: Number of seconds between *time1* and *time2*

SECS_BETWEEN (*time1, time2*)
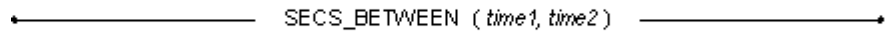
Figure 4-101 SECS_BETWEEN syntax

➲ **Example**

The following returns **36000**.
```
SECS_BETWEEN('10:10:10', '20:10:10')
```

4.1.102.   SESSION_USER

The SESSION_USER function returns the current user connected to DBMaker.

Return value.............The current session user

```
•────────────────────  SESSION_USER  ─────────────────•
```

Figure 4-102 SESSION_ USER syntax

➲ **Example**

The following returns the current SESSION_USER.
```
dmSQL> INSERT INTO t1 VALUES (SESSION_USER);
dmSQL> SELECT SESSION_USER;
dmSQL> SELECT c1 FROM t1 WHERE c2 = SESSION_USER;
```

4.1.103.   SIGN

The SIGN function returns an integer indicating the sign of *number*. The values returned are +1 for positive numbers, 0 for zero, and -1 for negative numbers.

number....................Double: Number to find the sign of

Return value ....Integer: Value corresponding to the sign of *number*

SIGN (*number*)
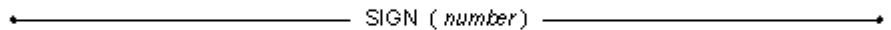
Figure 4-103 SIGN syntax

➲ **Example 1**

The following returns the value of **1**.
```
SIGN(12.3)
```

➲ **Example 2**

The following returns the value of **0**.
```
SIGN(0)
```

➲ **Example 3**

The following returns the value of **–1**.
```
SIGN(-12.3)
```

4.1.104.   SIN

The SIN function returns the sine of *number,* expressed in radians, as a double precision floating-point number.

number ....Double: Number to find the sine for

*Return valu*e.............Double: The sine of *number*

SIN (*number*)

Figure 4-104 SIN syntax

⮥ Example

The following returns the value of **4.79425538604203e-001**.
```
SIN(0.5)
```

4.1.105.   SINH

The SINH function returns the hyperbolic sine of *number,* expressed in radians, as a double precision floating-point number.

number...................Double: Number to find the hyperbolic sine for

Return value ....Double: The hyperbolic cosine of *number*

SINH (*number*)
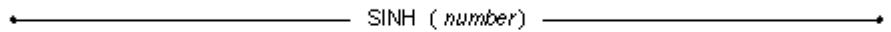
Figure 4-105 SINH syntax

⮕ **Example**

The following returns the value of **5.21095305493747e-001**.
```
SINH(0.5)
```
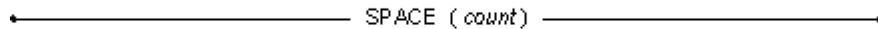
4.1.106. SPACE

The SPACE function returns a character string consisting of *count* spaces. If the value of *count* is less than zero, a NULL value is returned.

count........................Integer: Number of spaces

Return value.............String: String containing *count* spaces

—————————————— SPACE (*count*) ——————————————

Figure 4-106 SPACE syntax

➲ **Example 1**

The following returns a string consisting of three blank spaces "   ".
```
SPACE(3)
```

➲ **Example 2**

The following returns the string "   **Good morning!**" with three blank spaces in front.
```
CONCAT(SPACE(3), 'Good morning!')
```

NOTE *There are three spaces before the first letter in the return value.*

4.1.107.   SQRT

The SQRT function returns the square root of *x* as a double-precision floating-point number.

x..............................Double: Number to find the square root of

Return value ....Double: Square root of *x*

SQRT (*x*)
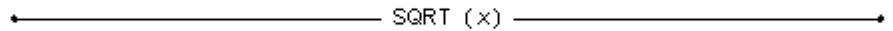
Figure 4-107 SQRT syntax

➜   **Example**

The following returns **1.30000000000000e+001**.
```
SQRT(169)
```

4.1.108.   STRTOINT

The STRTOINT function converts the string to an integer, when the string argument is NULL a NULL value is returned. An error is returned if the string cannot be converted to an integer.

string........................String: String to convert to number

Return value.............Bigint: integer converted by string

STRTOINT (string)

Figure 4-108 STRTOINT syntax

➲ **Example**

The following returns **1234**.
```
STRTOINT('1234')
```

4.1.109. SUBBLOB

The SUBBLOB function returns a temporary BLOB from an input *blob* beginning at the byte position specified by *start* for *length* bytes. The first BLOB byte is counted from 1. This function is an add-on; run the script **libblob.sql** provided by DBMaker to install it. DBMaker uses the following rules to determine the value returned if one of the expressions contains a NULL value or is zero.

◆ If blob is *NULL* the function returns a *NULL* value

◆ If *start* or *length* is *NULL* the function returns a temporary *BLOB*

◆ If *start* < 0 or *length* < 0 the function returns a *NULL* value

◆ If *start* > length of *blob* the function returns a *NULL* value

◆ If *length* is 0, the function returns an empty temporary *BLOB*

blob .... BLOB: CLOB, FILE to extract partial data from

start .... Integer: Position to begin extracting the data of *blob*

length .... Integer: Number of bytes to extract

Return value .... BLOB: Temporary BLOB extracted from *blob*

SUBBLOB (*blob, start, length*)
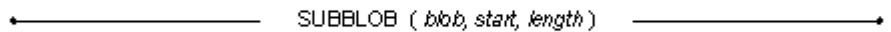
Figure 4-109 SUBBLOB syntax

➲ **Example**

The following returns temporary BLOB data extracted from Data BLOB from byte position **1001** to byte position **1100**.

```
SUBBLOB(Data, 1001, 100)
```

4.1.110.   SUBBLOBTOBIN

The SUBBLOBTOBIN function returns a binary string derived from input *blob,* beginning at the byte position specified by *start* for *length* bytes. The first byte of BLOB is counted from 1. This function is an add-on; run the **libblob.sql** script provided by DBMaker to install it. DBMaker uses the following rules to determine the value returned if one of the expressions contains a NULL value or is zero.

- ◆   If *blob* is *NULL* the function returns a *NULL* value

- ◆   If *start* or *length* is *NULL* the function returns a string with the same data as *blob*

- ◆   If *start* < 0 or *length* < 0 the function returns a *NULL* value

- ◆   If *start* > *length* of *blob* the function returns a *NULL* value

- ◆   If *length* is 0 the function returns an empty string

blob ....BLOB (BLOB, CLOB, FILE) to extract partial data from

start ....Integer. Position to begin extracting the data of *blob*

length.......................Integer. Number of characters to extract

Return value.............Binary string. Data extracted from *blob*

SUBBLOBTOBIN (*string, start, length*)

Figure 4-110 SUBBLOBTOBIN syntax

➥ **Example**

A binary string with data extracted from the Data BLOB byte position **1001** to **1100**.
```
SUBBLOBTOBIN(Data, 1001, 100)
```

4.1.111. **SUBBLOBTOCHAR**

The SUBBLOBTOCHAR function returns a character string that is derived from the input *blob* beginning at the byte position specified by *start* for *length* bytes. The first byte of BLOB is counted from 1. This function is an add-on, run the **libblob.sql** script provided by DBMaker to install it. DBMaker uses the following rules to determine the value returned if one of the expressions contains a NULL value or is zero.

- ◆ If *blob* is *NULL* the function returns a *NULL* value

- ◆ If *start* or *length* is *NULL* return the string, which is the same data as *blob*

- ◆ If *start* < 0 or *length* < 0 the function returns a *NULL* value

- ◆ If *start* > *length* of *blob* the function returns a *NULL* value

- ◆ If *length* is 0 the function returns an empty string

blob .... BLOB: BLOB, CLOB, FILE to extract partial data from

start .... Integer: Position to begin extracting the data of *blob*

length .... Integer: Number of characters to extract

Return value .... Character String: Data extracted from *blob*

SUBBLOBTOCHAR (*string, start, length*)

Figure 4-111 SUBBLOBTOCHAR syntax

➲ **Example**

A character string with data extracted from Data BLOB byte position **1001** to **1100**.
```
SUBBLOBTOCHAR(Data, 1001, 100)
```

4.1.112.   SUBSTRING

The SUBSTRING function returns *length* characters beginning at *start* from *string*. DBMaker uses the following rules to determine the value returned if one of the expressions contains a NULL value or is zero.

- ◆   If *string* is *NULL* the function returns a *NULL* value

- ◆   If *start* or *length* is *NULL* the function returns *string*

- ◆   If *start* < 0 or *length* < 0 the function returns a *NULL* value

- ◆   If *start* > *length* of *string* the function returns a *NULL* value

- ◆   If *length* is 0 the function returns an empty string

string.......................String: String to extract a substring from

start ....Integer: Position to begin extracting the substring

length......................Integer: Number of characters to extract

Return value.............String: Substring extracted from *string*
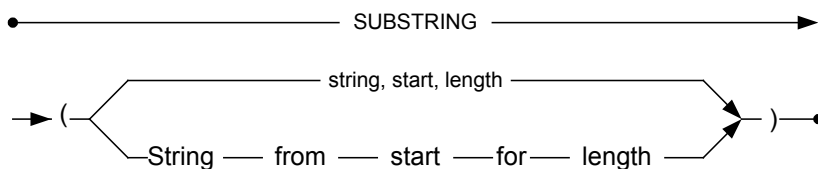
Figure 4-112 SUBSTRING syntax

⊃   **Example 1**

The following returns the string "**morning**".
```
SUBSTRING('Good morning!', 6, 7)
```

➲ **Example 2**

```
dmSQL> SELECT SUBSTRING(CAST(123456 AS CHAR(10)) FROM LENGTH('abc') for
LENGTH('abc'));
SUBSTRING(CAST(123456 AS CHAR(10)
================================
345
```

➲ **Example 3**

```
dmSQL> SELECT SUBSTRING('abcdef', 2, 2);
```

4.1.113.   TAN

The TAN function returns the tangent of *number,* expressed in radians, as a double-precision floating-point number.

 number ....Double: Number to find the tangent for

Return value.............Double: The tangent of *number*

Figure 4-113 TAN syntax

➲ **Example**

The following returns the value of **5.46302489843790e-001**.

```
TAN(0.5)
```

4.1.114.   TANH

The TANH function returns the hyperbolic tangent of a *number* as a double precision floating-point number expressed in radians.

Number ....Double: Number to find the hyperbolic tangent for

Return value ....Double: The hyperbolic tangent of Number

$$\longleftarrow\text{TANH (}\textit{number}\text{)}\longrightarrow$$

Figure 4-114 TANH syntax

➲ **Example**

The following returns the value of **4.62117157260010e-001**.
```
TANH(0.5)
```

4.1.115.   TIMEPART

The TIMEPART function returns the time part of Timestamp.

timestamp.................Timestamp: Timestamp to extract the time part from

Return value.............Time: Time part of *Timestamp*
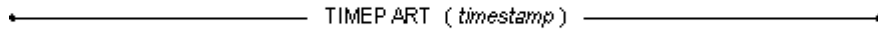
Figure 4-115 TIMEPART syntax

➲   **Example**

The following returns **10:11:12**.
```
TIMEPART('1996-02-29 10:11:12.123')
```

4.1.116.   TIMESTAMPADD

The TIMESTAMPADD function returns the timestamp calculated by adding Numbered Intervals to Timestamp.

IF INTERVAL	UNIT INTERVAL
"f " (or SQL_TSI_FRAC_SECOND for ODBC programs)	Fractions of a second
"s" (or SQL_TSI_SECOND for ODBC programs)	Seconds
"m" (or SQL_TSI_MINUTE for ODBC programs)	Minutes
"h" (or SQL_TSI_HOUR for ODBC programs)	Hours
"D" (or SQL_TSI_DAY for ODBC programs)	Days
"W" (or SQL_TSI_WEEK for ODBC programs)	Weeks
"M" (or SQL_TSI_MONTH for ODBC programs)	Months
"Q" (or SQL_TSI_QUARTER for ODBC programs)	Quarters
"Y" (or SQL_TSI_YEAR for ODBC programs)	Years

Table 4-1 TIMESTAMPADD NUMBERED INTERVAL table

interval ....String: Unit interval to add

number ....Integer: Number of unit intervals to add

timestamp ....Timestamp: Timestamp to add interval to

Return value ....Timestamp: Result of Timestamp + Interval × Number

TIMESTAMPADD (*interval, number, timestamp*)

Figure 4-116 TIMESTAMPADD syntax

⊃ **Example**

The following returns **1996-01-17 06:10:10**.

```
TIMESTAMPADD('h',20,'1996-01-16 10:10:10')
```

4.1.117.   TIMESTAMPDIFF

The TIMESTAMPDIFF function returns the number of unit intervals between *timestamp2* and *timestamp1*.

IF INTERVAL	UNIT INTERVAL
"f" (or SQL_TSI_FRAC_SECOND for ODBC programs)	Fractions of a second
"s" (or SQL_TSI_SECOND for ODBC programs)	Seconds
"m" (or SQL_TSI_MINUTE for ODBC programs)	Minutes
"h" (or SQL_TSI_HOUR for ODBC programs)	Hours
"D" (or SQL_TSI_DAY for ODBC programs)	Days
"W" (or SQL_TSI_WEEK for ODBC programs)	Weeks
"M" (or SQL_TSI_MONTH for ODBC programs)	Months
"Q" (or SQL_TSI_QUARTER for ODBC programs)	Quarters
"Y" (or SQL_TSI_YEAR for ODBC programs)	Years

Table 4-2 TIMESTAMPDIFF NUMBERED INTERVAL table

interval .... String: Unit Interval to return the difference in

timestamp1 .... Timestamp: First Timestamp to find the interval between

timestamp2 .... Timestamp: Second Timestamp to find the Interval between

Return value .... Double: Result of Timestamp2 - Timestamp1

TIMESTAMPDIFF (*interval, timestamp1, timestamp2*)
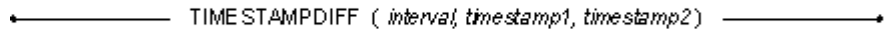
Figure 4-117 TIMESTAMPDIFF syntax

➔ **Example**

The following returns **2.40000000000000e+001**.
```
TIMESTAMPDIFF('h','1996-01-16 10:10:10', '1996-01-17 10:10:10')
```

4.1.118.   TRIM

The TRIM function combines the LTRIM and RTRIM functions. More than one character can be specified in the trim_char_value_expr and each character is viewed as a valid trim character.

The default trim option is BOTH when at least one LEADING, TRAILING, or BOTH options are not specified. The default trim_char_value_expr character is the space character (' '). In addition, if the trim_char_value_expr were an empty string (''), the resulting string would be trim_source string. If the trim_source is NULL, than the result would also be NULL, no matter which trim option and trim character were used. The LENGTH function can also be used with the TRIM function as shown in some of the examples that follow.

leading.....................Remove trim_string from the front of *trim_source*

trailing.....................Remove trim_string from the end of *trim_source*

both ....remove trim_string from the front and end of *trim_source*

If none of these are chosen (i.e.: leading, trailing, both), the trim function will remove *trim_expr* from both the front and end of *trim_source*.

trim_expr................. The character that will be removed from *trim_source*.

If this parameter is omitted, the trim function will remove all leading and trailing spaces from *trim_source*.

trim_source...............The string to trim.
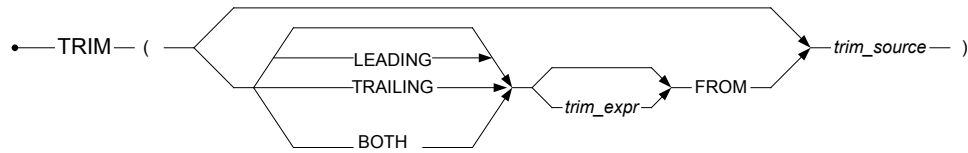
Figure 4-118 TRIM function syntax

➲ **Example 1**
```
dmSQL> SELECT TRIM(both 'a' FROM 'aabcaa');
 TRIM(BOTH 'A' FROM 'AABCAA')
============================
bc
```

➲ **Example 2**
```
dmSQL> SELECT TRIM(FROM 'aabcaa');
 TRIM(FROM 'AABCAA')
====================
aabcaa
```

➲ **Example 3**
```
dmSQL> SELECT TRIM('a' FROM 'aabcaa');
 TRIM('A' FROM 'AABCAA')
========================
bc
```

➲ **Example 4**
```
dmSQL> SELECT TRIM('abc' FROM 'abckjkjjdcba');
  TRIM('ABC' FROM 'ABCKJKJJDCBA')
=================================
kjkjjd
```

➲ **Example 5**
```
dmSQL> SELECT TRIM('a c' FROM 'ac ddbc');
  TRIM ('A C' FROM 'AC DDBC')
============================
ddb
```

⮞ **Example 6**

```
dmSQL> SELECT LENGTH(TRIM(leading FROM ' abc '));
    LENGTH(TRIM(LEADING FROM ' ABC '))
=========================================
                                       3
```

⮞ **Example 7**

```
dmSQL> SELECT LENGTH(TRIM(leading 'a' FROM 'aabc '));
    LENGTH(TRIM(LEADING 'A' FROM 'AA'))
=========================================
                                       2
```

⮞ **Example 8**

```
dmSQL> SELECT LENGTH(TRIM(trailing FROM 'aabc '));
    LENGTH(TRIM(TRAILING FROM 'AABC'))
=========================================
                                       4
```

⮞ **Example 9**

```
dmSQL> SELECT LENGTH(TRIM(trailing 'a' FROM 'aabcaa'));
 LENGTH(TRIM(TRAILING 'A' FROM 'AABCAA'))
=========================================
                                       4
```

4.1.119.   UCASE

The UCASE function converts all lower case characters in *string* to uppercase. If the *string* argument is NULL, a NULL value is returned. All arguments must be provided otherwise an error is returned.

string .... String: Text to convert to upper case

Return value .... String: Text from the *string* argument in upper case

UCASE (*string*)

Figure 4-119 UCASE syntax

➲ **Example 1**

The following returns the string "**ABCDEF**".
```
UCASE('ABCdef')
```

➲ **Example 2**

The following returns the string "**ABC123**".
```
UCASE('abc123')
```

➲ **Example 3**

The following returns the string "**ABC@#$**".
```
UCASE('abc@#$')
```

4.1.120.   UPPER

This function performs the same calculation as UCASE. It capitalizes all characters in the string. NULL string argument will return NULL.

String-expression ....String: function change LOWER case into UPPER case

Return value.............String: returns all characters in UPPER case

UPPER (string_expression)

Figure 4-120 UPPER function syntax

➲ **Example**
```
dmSQL> SELECT UPPER('abcdef');
 UPPER('ABCDEF')
================
ABCDEF
```

4.1.121.   USER

The USER function returns the authorization name of the current user. The authorization name of the user is also available by calling the SQLGetInfo with the SQL_USER_NAME option.

Return value ....String: The name of the current user

```
——————————————— USER () ———————————————
```

Figure 4-121 USER syntax

4.1.122.   UTFConvert

The UTFConvert function is used to convert the character set between UTF-8 and UTF-16. It contains two functions U8TOU16 and U16TOU8.

The UTFConvert function is in DBMaker's installation directory*shared**udf*. It's not created by default in the database. If users want to use it, they have to create it manually.

To execute the following command to create these functions:
```
dmSQL> CREATE FUNCTION UTFConvert.U8TOU16(long varbinary) RETURNS nclob;
dmSQL> CREATE FUNCTION UTFConvert.U16TOU8(nclob) RETURNS long varbinary;
```

long varbinary ....the UTF-8 content which will be converted to UTF-16

nclob ....the UTF-16 content which will be converted to UTF-8

```
●─────────────U8TOU16(long varbinary)─────────────●


●───────────────U16TOU8(nclob)───────────────●
```
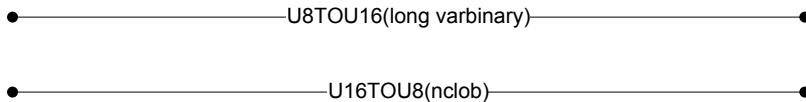
Figure 4-122 UTFConvert syntax

©Copyright 1995-2017 CASEMaker Inc.

4.1.123.   WEEK

The WEEK function returns the week *date* that falls in the integer value range from **1** to **53**.

date ....Date: Date to find the week for

Return value ....Integer: The week that *date* falls in
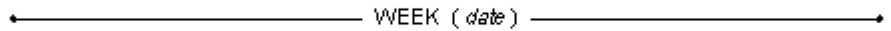
Figure 4-123 WEEK syntax

➲   **Example**

2002-02-11 is in the **5**th week of **2002**, the following returns **5**.
```
WEEK('2002-02-01')
```

4.1.124.   XLSTOTXT

The XLSTOTXT function can be used to convert excel document to a temporary BLOB containing the pure text of blob as unicode.it will return temp blob or NULL. In DBMaker current version, UDF will support office 2007- 2010 version.

blob ....Column name on which to be converted to pure text

Return value.............temp blob as NCLOB type if blob can be converted to pure text.

COS (*number*)
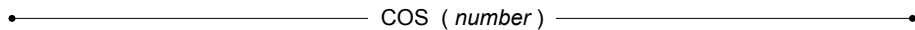
Figure4-124 Syntax for XLSTOTXT

➲ **Example**

The following will convert the column memo to puretext.
```
XLSTOTXT (memo)
```

4.1.125.   XMLUPDATE

The xmlupdate function is using XPath to locate the part of xml data to be updated.

xmldata………………the XML content to be updated

xpath-expression……specifies the location of the xmldata to be updated

namespaces…………optionally specifies the namespace used in Xpath-expression

replace-content…… the value to replace the content located by Xpath

returnvalue…………the entire XML document after updating

XMLUPDATE(XMLdata, modification-type, xpath-expression, namespaces, replace content)

Figure 4-125 XMLUPDATE syntax

4.1.126.   YEAR

The YEAR function returns the year in *date* as an integer value in the range from **1** to **9999**.

date..........................Date: Date to find the year component of

Return value.............Integer: The year component of *date*

<p style="text-align:center;">YEAR (date)</p>

Figure 4-126 YEAR syntax

➲   **Example**

The following example illustratrates returning **2002**.
```
YEAR('2002-02-01');
```

4.2 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.

4.2.1. AES_DECRYPT

The AES_ENCRYPT function is used to encrypt the data to safeguard the significant data. Correspondingly, The AES_DECRYPT function is used to decrypt the encrypted data to get the raw data.

The AES_DECRYPT function is in DBMaker's installation *directory\shared\udf*. It's not created by default in the database. If users want to use it, they have to create it manually, or run the SQL script located under the same directory.

To execute the following command to create these functions:

```
dmSQL> CREATE FUNCTION LIBCRYPT.AES_DECRYPT(BINARY(4096),STRING) RETURNS
BINARY(4096);
```

ciphertext ....the ciphertext

cipher key ....the inputed passphrase key

plaintext ....the raw data

AES_DECRYPT (*ciphertext* , *cipher key* , *plaintext*)

Figure 4-127 AES_DECRYPT syntax

The AES_DECRYPT function supports the following five data types: BINARY(N), CHAR(N), VARCHAR(N), NCHAR(N), NVARCHAR(N). If the original data's type is other types, ERROR (6536): [DBMaker] function arguments do not match definition will be returned.

Encryption uses 16 byte alignment, which makes the string bigger. Note that the original data's length must matches value of the defined UDF argument. Under different environment, users should modify the parameter BINARY(n) to make sure the length is big enough for the encrypted data. In addition, value of AES_DECRYPT's parameter BINARY(n) must matches the value of the AES_ENCRYPT's parameter BINARY(n).

➲ **Example**

Users can use the following syntax to run the AES_DECRYPT function.

```
dmSQL> SELECT AES_DECRYPT (Column, 'key') FROM table;
```

The following example decribes usage of the AES_DECRYPT function.

```
dmSQL> CREATE TABLE DAES(C1 BINARY(1024));
dmSQL> SELECT AES_DECRYPT (C1, 'key') FROM DAES; //the result' data type is
BINARY, and please cast the value;
dmSQL> SELECT CAST(AES_DECRYPT (C1, 'key') AS CHAR(200)) FROM DAES;
```

4.2.2. AES_ENCRYPT

The AES_ENCRYPT function is used to encrypt the data to safeguard the significant data.

The AES_ENCRYPT function is in DBMaker's installation *directory\shared\udf*. It's not created by default in the database. If users want to use it, they have to create it manually, or run the SQL script located under the same directory.

To execute the following command to create these functions:

```
dmSQL> CREATE FUNCTION LIBCRYPT.AES_ENCRYPT(BINARY(4096),STRING) RETURNS
BINARY(4096);
```

plaintext ....the raw data to be encrypted

cipher key ....the inputted passphrase key

ciphertext ....the ciphertext

AES_ENCRYPT (*plaintext* , *cipher key* , *ciphertext*)

Figure 4-128 AES_ENCRYPT syntax

The AES_ENCRYPT function supports the following five data types: BINARY(N), CHAR(N), VARCHAR(N), NCHAR(N), NVARCHAR(N). If the original data's type is other types, ERROR (6536): [DBMaker] function arguments do not match definition will be returned.

Encryption will make the string bigger and 16 byte alignment. Note that the original data's length must matches value of the defined UDF argument. Under different environment, users should modify the parameter BINARY(n) to make sure the length is big enough for the encrypted data. In addition, value of AES_DECRYPT's parameter BINARY(n) must matches the value of the AES_ENCRYPT's parameter BINARY(n).

⮑ **Example**

Users can use the following syntax to run the AES_ENCRYPT function.

```
dmSQL> SELECT AES_ENCRYPT (Column, 'key') FROM table;
```

The following example decribes usage of the AES_ENCRYPT function.

```
dmSQL> CREATE TABLE AES(C1 CHAR(200));
dmSQL> INSERT INTO AES VALUES('abc');
dmSQL> SELECT AES_ENCRYPT (C1, 'key') FROM AES INTO DAES;
```

4.2.3. DATETOSTR

The DATETOSTR function is used to convert a value in DATE type into the character string in specified format. The value in DATE type must be a valid date.

The DATETOSTR function is in DBMaker's installation directory\shared\udf. It's not created by default in the database. If users want to use it, they have to create it manually, or run the SQL script located under the same directory.

To execute the following command to create these functions:

```
dmSQL> CREATE FUNCTION datetostr.DATETOSTR(DATE, varchar(20)) RETURNS
varchar(20);
```

date....................................the date to be converted into a character string

date_format_string.............the format of the returned character string into which the date is converted. Currently, the following 13 formats are supported: 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 and yyyymmdd. Additional, the format must be small letters.

Return value ....the character string into which the date is converted

DATETOSTR(*date* , *date_format_ string*)
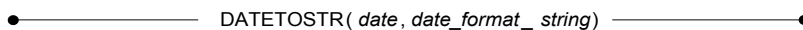
Figure 4-129 DATATOSTR syntax

➲ **Example**

The following will convert the date "**2012-2-12**" into the character string in "**mm/dd/yy**" format.

```
DATETOSTR('2012-12-12','mm/dd/yy')
```

4.2.4. TIMETOSTR

The TIMETOSTR function is used to convert a value in TIME type into the character string with specified format. The value in TIME type must be a valid time.

The TIMETOSTR function is in DBMaker's installation directory*shared**udf*. It's not created by default in the database. If users want to use it, they have to create it manually, or run the SQL script located under the same directory.

To execute the following command to create these functions:
```
dmSQL> CREATE FUNCTION datetostr.TIMETOSTR(TIME, varchar(20)) RETURNS
varchar(20);
```

time...............................the time to be converted into a character string

time_format_string.........the format of the returned character string into which the time is converted. Currently, the following 13 formats are supported: hh:mm:ss.fff, hh:mm:ss, hh:mm, hh, hh:mm:ss.fff tt, hh:mm:ss tt, hh:mm tt, hh tt, tt hh:mm:ss.fff, tt hh:mm:ss, tt hh:mm, tt hh and hhmmss. Additional, the format must be small letters.

Return value ....the character string into which the time is converted
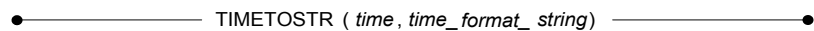
Figure 4-130 TIMETOSTR syntax

➲ **Example**

The following will convert the time "**12:10:10**" into characters in "**hh:mm:ss tt**" format.
```
TIMETOSTR('12:10:10','hh:mm:ss tt')
```

4.2.5. TIMESTAMPTOSTR

The TIMESTAMPTOSTR function is used to convert a value in TIMESTAMP type into the character string in specified format. The value in TIMESTAMP type must be a valid date and time.

The TIMESTAMPTOSTR function is in DBMaker's installation *directory\shared\udf*. It's not created by default in the database. If users want to use it, they have to create it manually, or run the SQL script located under the same directory.

To execute the following command to create these functions:

```
dmSQL> CREATE FUNCTION datetostr.TIMESTAMPTOSTR(TIMESTAMP, varchar(20),
varchar(20)) RETURNS varchar(30);
```

timestamp..........................the date and time to be converted into a character string

date_format_string.............the format of the returned character string into which the date is converted. Currently, the following 13 formats are supported: 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 and yyyymmdd. Additional, the format must be small letters.

time_format_string ....the format of the returned character string into which the time is converted. Currently, the following 13 formats are supported: hh:mm:ss.fff, hh:mm:ss, hh:mm, hh, hh:mm:ss.fff tt, hh:mm:ss tt, hh:mm tt, hh tt, tt hh:mm:ss.fff, tt hh:mm:ss, tt hh:mm, tt hh and hhmmss. Additional, the format must be small letters.

Return value ....the character string into which the date and time is converted

TIMESTAMPTOSTR (*timestamp* , *date_format_ string* , *time_format_ string*)

Figure 4-131 TIMESTAMPTOSTR syntax

➲ **Example**

The following will convert the date and time "**2012-12-12 12:12:12**" into the character string in "**mm/dd/yy**" and "**tt hh:mm:ss**" format.

```
TIMESTAMPTOSTR('2012-12-12 12:12:12','mm/dd/yy' 'tt hh:mm:ss')
```

4.2.6. TO_DATE

The TO_DATE function converts a selected character string to a DATE format. The string may be of any data type, but must conform to a valid date when converted to a date. The TO_DATE function consists of two parameters, char_string and date_format_string. The char_string parameter represents the string that is to be matched, while the date_format_string represents the format that the DATE type data result set will take.

The TO_DATE UDF function is in DBMaker's installation directory*shared**udf*, it's not created by default in the database. If users want to use it, they have to create it manually, or run the SQL script located under the same directory.

To execute the following command:

```
dmSQL> CREATE FUNCTION to_date.TO_DATE(varchar(20), varchar(20)) RETURNS DATE;
```

string_expr.....................String expression from which the expression is matched

date_format_string...........The format that the date format should take. Use **Y** or **y** to denote years, **M** or **m** to denote months, and **D** or **d** to denote days. Use **/** or **–** to denote a separator.

Return value.....................The string expression returned as a DATE type data string.

TO_DATE (*string_expr, date_format_string*)

Figure 4-132 TO_DATE syntax

⮑ **Example 1**
```
TO_DATE('991031', 'YYMMDD')
```

⮑ **Example 2**
```
dmSQL> SELECT TRIM(FROM 'aabcaa');
dmSQL> SELECT TO_DATE('2009-Jan-01', 'YYYY-mon-DD');
```

5 System-Stored Procedures

System-Stored Procedures are dynamic library modules that are not be loaded until called. System-stored procedures include shared objects and XML import and XML export procedures.

A shared object is a signed integer variable existing in the database shared memory (DCCA). The access of a shared object is more efficient and independent of the transaction. Unlike data records, shared objects are not stored in a database file. As a result, the lifecycle of the shared object ends when it is dropped or the database is shut down.

Every user connected to the database can see the shared objects added by the SYSADM. Users can set or get the shared object's values unless a lock has been placed on them by another user. A shared object is a 4 byte signed integer. All users also have equal rights and permissions to the shared objects, thus any user can override or reset an objects' settings except for the lock permission.

The other two system-stored procedures (XMLEXPORT and XMLIMPORT) can only be used by a SYSADM, a SYSDBA or a DBA to import and export xml files.

5.1   APPENDBLOB

The APPENDBLOB system-stored procedure is used to insert a huge file into a
BLOB/CLOB/FILE type column piece by piece. It is built in the add-on executable
file. DBMaker doesn't initialize it when creating a database, so, before using it, users
need to declare it by running *<DBMaker home installation directory> /shared/sp/*
AppendBlob.sql.

For simplicity, without special note, we use BLOB type to represent the
BLOB/CLOB/FILE type in the following sections.

If no record or more than one record matches the condition specified by
WHERE_STR, an error will occur and a relevant error message will be returned.

If the cell specified by TABLE_NAME, COLUMN_NAME and WHERE_STR is
Null or its type is not BLOB, an error will occur and a relevant error message will be
returned.

The maximum size of DATA_BUFF is 10M bytes, so if the value of DATA_BUFF or
DATA_LEN is bigger than 10485760, an error will occur and a relevant error
message will be returned.

➲ **The prototype for APPENDBLOB is:**

```
APPENDBLOB(VARCHAR(128) TABLE_NAME INPUT,
          VARCHAR(128) COLUMN_NAME INPUT,
          VARCHAR(2048) WHERE_STR INPUT,
          BINARY(10485760) DATA_BUFF INPUT,
          INTEGER DATA_LEN INPUT)
```

table_name................the name of the table which contains BLOB type column

column_name.............the name of the BLOB type column to which the new data will
be appended

where_str...................the condition string used to specify the single row

data_buff..................the data in the buffer will be appended to on the BLOB type column specified by TABLE_NAME, COLUMN_NAME and WHERE_STR.

data_len.....................the length of the valid data in DATA_BUFF. Its unit is **Byte**. If it is less than the DATA_BUFF's length, DBMaker appends only the data with the length DATA_LEN, or else, appends the whole buffer.

➲ **Example**

Firstly, create a database named **bbsp**, and then declare the stored procedure APPENDBLOB.

```
dmSQL> CREATE DB bbsp;
USE db #1 connected to db:<bbsp> by user:<SYSADM>
dmSQL> RUN 'C:\DBMaker\5.4\shared\sp\AppendBlob.sql';
```

Create a table named **test_blob1**, and then insert a record.

```
dmSQL> CREATE TABLE test_blob1(c1 INT,c2 BLOB);
dmSQL> INSERT INTO test_blob1 VALUES(1,?);
dmSQL/Val> &file1;
1 rows inserted
dmSQL/Val> END;
dmSQL> SELECT c1,BLOBLEN(c2) FROM test_blob1;
    C1         BLOBLEN(C2)
==========  ============
        1         81920
1 rows selected
```

Call the stored procedure APPENDBLOB to append more data to the BLOB.

```
dmSQL> CALL APPENDBLOB('test_blob1','c2','c1=1',?,10);
dmSQL/Val> 'xxxxxyyyyy';
dmSQL/Val> END;
dmSQL> SELECT c1,BLOBLEN(c2) FROM test_blob1;
    C1         BLOBLEN(C2)
==========  ============
        1         81930
1 rows selected
```

5.2   APPENDBLOBBYOID

The APPENDBLOBBYOID system-stored procedure is used to insert a huge file piece by piece.It has the same usage as the APPENDBLOB system-stored procedure. It is also built in the add-on executable file and users also need to declare it before using.

If DBMaker cannot find a row through ROW_ID, an error will occur and a relevant error message will be returned.

If the cell specified by ROW_ID and COLUMN_ORDER is Null or its type is neither BLOB nor FILE, an error will occur and a relevant error message will be returned.

The maximum size of DATA_BUFF is 10M bytes, so if the value of DATA_BUFF or DATA_LEN is bigger than 10485760, an error will occur and a relevant error message will be returned.

⮕   **The prototype for APPENDBLOBBYOID is:**

```
APPENDBLOBBYOID(BINARY(16) ROW_ID INPUT,
                INT COLUMN_ORDER INPUT,
                BINARY(10485760) DATA_BUFF INPUT,
                INT DATA_LEN INPUT)
```

row_id_input ....id of the row which contains BLOB type columns

column_order..............the order number of the BLOB type column to which the new data will be appended

data_buff...................the data in the buffer will be appended to the BLOB type column specified by ROW_ID and COLUMN_ORDER.

data_len.....................the length of the valid data in DATA_BUFF. Its unit is **Byte**. Ifit is less than the DATA_BUFF's length, DBMaker appends only the data with the length DATA_LEN, or else, appends the whole buffer.

➲ **Example**

Create a table named **test_blob2**, and then insert a record.

```
dmSQL> CREATE TABLE test_blob2(c1 INT,c2 BLOB);
dmSQL> INSERT INTO test_blob2 VALUES(1,?);
dmSQL/Val> &file1;
1 rows inserted
dmSQL/Val> END;
dmSQL> SET AUTOMIT OFF;
dmSQL> COMMIT;
1 rows selected
dmSQL> SELECT oid,c1,BLOBLEN(c2) FROM test_blob2;
             OID                   C1      BLOBLEN(C2)
============================== ========== ===========
0300000002000000000000000000000            1        81920
1 rows selected
dmSQL> SELECT column_order FROM system.syscolumn WHERE table_name = 'TEST_BLOB2'
 AND column_name = 'C2';
COLUMN_ORDER
============
          2
1 rows selected
```

Call the stored procedure APPENDBLOBBYOID to append more data to the file1.

```
dmSQL> CALL APPENDBLOBBYOID('0300000002000000000000000000000'x,2,?,15);
dmSQL/Val> 'xxxxxyyyyyzzzzz';
dmSQL/Val> END;
dmSQL> COMMIT;
dmSQL> SELECT c1,BLOBLEN(c2) FROM test_blob2;
    C1      BLOBLEN(C2)
========== ===========
         1        81935
1 rows selected
```

5.3 COPYTABLE

The COPYTABLE system-stored procedure is used to copy one table's definition and data into another table. The source table's index, table and column constraints, triggers, and data are copied to the destination table.

The COPYTABLE stored procedure must be run when autocommit mode is on. An error is returned when the destination table already exists. If the rename index flag is set to 1, then the index name is renamed to the new table name if it's prefixed by the table name. If a user sets the commit count, a command is issued to commit when copying every n^{th} data into the destination table.

When an error occurs, operations executed prior to the error are committed. The commands that did not execute for COPYTABLE are recorded in the _spusr.log. Users can set the **DB_SPLog** directory in **dmconfig.ini** or find it in the directory where the user executes the application.

➲ **The prototype for COPYTABLE is:**

```
COPYTABLE(VARCHAR(32) source_schema_name INPUT,
         VARCHAR(32) source_table_name INPUT,
         VARCHAR(32) destination_schema_name INPUT,
         VARCHAR(32) destination_table_name INPUT,
         VARCHAR(128) tablespace_lock_mode_option_string INPUT,
         VARCHAR(2048) where_condition_string INPUT,
         INT fg_rename_index INPUT,
         INT commit_count INPUT)
```

schema_name.............The schema name of the table represents the default current user when specified as NULL or an empty string.

table_name...............The name of the source or destination table.

tablespace_lock_mode_option_string... Specified in the IN tablespace or lock mode syntax as in the created table in the string. The identifier specified in the string must follow the SQL syntax rule.

where_condition_string…Specified in the where condition as the SELECT statement in the string. The identifier specified in the string must follow the SQL syntax rule.

fg_rename_index............This flag indicates whether to rename the index to be prefixed by new_table_name if the source index name is prefixed by the table name. The valid values are 0 or 1.

commit_count ....Commit after every n^{th} record is inserted. The valid value range is 0 to n.

➲ **Example**

The following syntax copies the **Scores** table to table **Scores70** in a different tablespace, where the **Math** score > 70. It does not rename the index. It commits after every **10** records.

```
dmSQL> CALL COPYTABLE('SYSADM', 'Scores', 'SYSADM', 'Scores70', 'in tablespace1',
'Math > 70', 0, 10);
```

5.4   GETCPUNUMBER

The GETCPUNUMBER system-stored procedure is used to get the number of logical processors in the machine.

Using GETCPUNUMBER and SETAFFINITY system stored procedures, user 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

➲ **Example**

The following syntax gets the number of CPU by calling GETCPUNUMBER:
```
dmSQL> CALL GETCPUNUMBER(?);
```

5.5 GETSYSTEMOPTION

The GetSystemOption system-stored procedure is used to get the system option value during run time. That is to say, user can use GetSystemOption stored procedure to get all validsystem option values during the database running.

The following table lists all of the **option_name** system option values that obtained by calling the system-stored procedure GetSystemOption, and a brief description of what keyword is contained in each **option_name**. For more details of related keywords, please refer to *Database Administrator's Guide*.

OPTION_NAME	DESCRIPTION
FODIR	The system file object directory (DB_FoDir)
LGSVR	Server log level (DB_LgSvr)
LGERR	Server log error level (DB_LgErr)
LGSTM	Server log statment execution time over n secs (DB_LgSTm)
LGSYS	Server log the system info (DB_LgSys)
LGFSZ	Server log file size (DB_LgFSz)
LGFNO	Server log file number (DB_LgFNo)
LGSQL	Server log the sql command (DB_LgSQL)
LGPLN	Server log the execution plan (DB_LgPLn)
LGPAR	Server log the input parameter value (DB_LgPar)
LGLCK	Server log extra lock time out informatin when it exceed error argument's length (DB_LgLck)
LGDIR	Server log directory (DB_LgDir)
LGDAY	The number of days to keep server log files (DB_LgDay)
LGZIP	Zip closed log files (DB_LgZip)
BKCHK	Whether check database before full backup and differential backup (DB_ BkChk)
BKCMP	The compact backup mode (DB_BkCmp)
BKDIR	Directory to store backup journal files (DB_BkDir)

BKFOM	The file object (FO) backup mode (DB_BkFoM)
BKFRM	The format Backup Server used to name incremental backup journal files (DB_BkFrm)
BKFUL	The percentage full of the journal files that triggers the backup server to perform an incremental backup (DB_BkFul)
BKITV	The backup time interval (DB_BkItv)
BKODR	The directories where the backup server puts the previous version of full backup files (DB_ BkOdr)
BKRTS	Whether the backup server includes the read-only tablespace files when performing a full-backup (DB_BkRTs)
BKSPM	The store procedure(SP) backup mode(DB_BkSPm)
BKSVR	Whether a backup server is activated (DB_BkSvr)
BKTIM	The first time a backup server performs an incremental backup (DB_BkTim)
BKZIP	Whether the backup files are compressed by a backup server when performing full backups (DB_BkZip)
CTBLM	The default lock mode used when creating a table (DB_CTbLM)
DBKMX	The maximum number of differential backup after a full backup (DB_DbKmx)
DBKTV	The differential backup time interval (DB_DbKtv)
DBNAME	The database name of current connection
DDBMD	Whether the DDB (Distributed DataBase) function is enabled on the database server (DD_DDBMd)
EATRPT	The database server's Subscriber Daemon TCP/IP port number (DB_EtrPt)
EXTNP	A size for DBMaker to extend autoextend tablespace (DB_ExtNp)
FBKTM	The first time the Backup Server will perform a full backup (DB_FBkTm)
FBKTV	The full backup time interval (DB_FBkTv)
FOSUB	The maximum number of file objects that may be stored

	in each system file object subdirectory (DB_FoSub)
FULLBKID	The full backup id
IDXDP	Auto drop index threshold for auto index daemon (DB_IdxDp)
IDXLN	Auto create index threshold for auto index daemon (DB_IdxLn)
IDXTM	The start time for auto index daemon (DB_IdxTm)
IDXTV	The auto index daemon interval (DB_IdxTv)
IDXSV	Activate auto index daemon (DB_IdxSv)
ISOLV	The default transaction isolation level when a user connects to the database (DB_ISOLV)
LETPT	The Lock Escalation Threshold for escalating a page lock to a table lock (DB_LetPT)
LETRP	The Lock Escalation Threshold for an escalating rowlock to a page lock (DB_LetRP)
LIC_ACL	Access Control List
LIC_BKSERVER	Backup Server
LIC_DBREP	Database Replication
LIC_DCI	Database Cobol Interface
LIC_DDB	Distributed Database
LIC_EDITION	Edition
LIC_EXPIREDATE	License Expiration Date
LIC_FREETRIAL	Free Trial Period
LIC_FULLTEXT	FullText Indexing
LIC_HOSTCONN	Host Connection
LIC_IOSERVER	IO Server
LIC_LOCALE	Locale Language
LIC_MAXCONN	Max Connection
LIC_MAXDBSIZE	Max Database Size
LIC_MAXJNFSZ	Max Journal File Size
LIC_MAXPGSIZE	Max Page Size

LIC_NETZC	Network Compression
LIC_PLATFORM	Platform
LIC_PRODUCT	Product Name
LIC_SERIALID	Serial ID
LIC_STARTDATE	License Start Date
LIC_UPGRADE	Upgradable
LIC_USERINFO	User Information
LIC_VERSION	Version
SQLST	The display mode of the SQL command monitor (DB_SQLSt)
STARTBACKUP	Option to act backup server to process full or incremental backup
STSVR	Start update statistics daemon (DB_StSvr)
STMOD	The incremental update statistics mode for a database (DB_StMod)
STSTM	The start time for update statistics (DB_StsTm)
STSTV	The update statistics daemon interval (DB_StsTv)
STSSP	The update statistics sample (DB_StsSP)
USRFO	User file objects can be inserted in a database (DB_UsrFo)

➲ **The prototype for GETSYSTEMOPTION is:**
```
GETSYSTEMOPTION('optionName', ?)
```

optionName ....……system option name.

➲ **Example 1**

The following syntax gets option value of backup server:
```
dmSQL> CALL GETSYSTEMOPTION('BKSVR',?);
```

➲ **Example 2**

The following syntax gets option value of database expired date:
```
dmSQL> CALL GETSYSTEMOPTION('LIC_EXPIREDATE',?);
```

5.6   SCHEDULE_ALTER

The SCHEDULE_ALTER system-stored procedure is used to alter an existing schedule.

Except for SCHEDULE_NAME, all schedule parameters can be altered. If a parameter of a schedule is altered while the task is running, dmschsvr will load and use the new parameters of this schedule when the task runs next time according to users' schedule.

➲ **The prototype for SCHEDULE_ALTER is:**

```
SCHEDULE_ALTER(VARCHAR(128) SCHEDULE_NAME INPUT,
               VARCHAR(128) TASK_NAME INPUT,
               VARCHAR(512) TIMETABLE INPUT,
               VARCHAR(32) STARTTIME INPUT,
               VARCHAR(32) ENDTIME INPUT)
```

schedule_name ....the name of the existing schedule to alter

task_name..................the name of the task involved into the schedule

starttime....................the date and time when the schedule starts; its format is yyyy-mm-dd hh:mm:ss.

Endtime....................the date and time when the schedule expires; its format is yyyy-mm-dd hh:mm:ss. Because the minimum time unit of scheduel daemon is minutes, so end time must be later than start time by at least one minute. Please note that, usually users must set endtime, but if value of timetable is set to @once or @once m n, it is allowed for users not to set endtime, and under this situation, system will automatically regard the time later than start time by one minute and (m*n+1) minutes as the end time respectively.

Timetable..................the timetable of the task's execution; it is composed of fivefields in sequence: minute, hour, day-of-month, month, day-of-week, and the five fields should be separated by a space. Their value's

range are 0–59, 0–23, 1-31, 1–12 and 0–7 respectively and all values can be replaced with the following wildcard: asterisk (*), comma (,), hyphen (-) and slash (/). Details are as follows.

-Asterisk (*)

It's valid to specify a * to represent all possible values for a position, e.g. a * on 2nd position is same as specifying all the possible values for hour.

-Comma (,)

It's valid to specify several values separated by commas, e.g. if a user want a command to be executed every 10th minute, he can specify 0,10,20,30,40,50 for minute.

-Hyphen (-)

It's valid to specify the range of a value with a -, e.g. a user can specify 0-12 for hour to represent every hour a.m.

-slash (/)

It's valid to specify a regular interval with a /, e.g. a user can specify */3 for minute to represent every 3 minutes.

For convenience, some simple and specific characters are set as valid value for TIMETABLE. Details are as follows.

CHARACTER	DESCRIPTION
@MINUTE	the first second of every minute
@HOURLY	the first minute of every hour
@MIDNIGHT	the first minute of every day
@DAILY	the first minute of every day
@WEEKLY	the first minute of every Monday
@MONTHLY	the first minute of the 1st of every month
@ONCE	This task will be executed only once, whether it is executed successfully or not. The starting time is specified by STARTTIME; if STARTTIME is 'now()', this task will be exected at next minute.
@ONCE M N	This task will be executed sccussfully only once. If fail, this task

	will be executed once every n minutes until a success, and this kind of attempts can be done at most m times. m' range is 1 ~ 525600; n's range is 1 ~ 1440.

Table 5-1 Valid Special Characters table

➲ **Example**

The following syntax is used to 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 5.7, *SCHEDULE_CREATE*.

```
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.
```

5.7   SCHEDULE_CREATE

The SCHEDULE_CREATE system-stored procedure is used to create a schedule.

➲ **The prototype for SCHEDULE_CREATE is:**

```
SCHEDULE CREATE(VARCHAR(128) SCHEDULE NAME INPUT,
                VARCHAR(128) TASK NAME INPUT,
                VARCHAR(512) TIMETABLE INPUT,
                VARCHAR(32) STARTTIME INPUT,
                VARCHAR(32) ENDTIME INPUT)
```

schedule_name......the name of the schedule to create. It can contains 1 to 128 letters, numbers, and underscores, but the first character cannot be numbers.

task_name.................. the name of the task involved into the schedule

starttime......................the date and time when the schedule starts; its format is yyyy-mm-dd hh:mm:ss.

Endtime......................the date and time when the schedule expires; its format is yyyy-mm-dd hh:mm:ss. Because the minimum time unit of scheduel daemon is minutes, so end time must be later than start time by at least one minute. Please note that, usually users must set endtime, but if value of timetable is set to @once or @once m n, it is allowed for users not to set endtime, and under this situation, system will automatically regard the time later than start time by one minute and (m*n+1) minutes as the end time respectively.

Timetable....................the timetale of the task's execution; it is composed of five fields in sequence: minute, hour, day-of-month, month, day-of-week, and the five fields should be separated by a space. Their value's range are 0–59, 0–23, 1-31, 1–12 and 0–7 respectively and all values can be replaced with the following wildcard: asterisk (*), comma (,), hyphen (-) and slash (/). Details are as follows.

-Asterisk (*)

It's valid to specify a * to represent all possible values for a position, e.g. A * on 2nd position is same as specifying all the possible values for hour.

-Comma (,)

It's valid to specify several values separated by commas, e.g. If a user want a command to be executed every 10th minute, he can specify 0,10,20,30,40,50 for minute.

-Hyphen (-)

It's valid to specify the range of a value with a -, e.g. A user can specify 0-12 for hour to represent every hour a.m.

-slash (/)

It's valid to specify a regular interval with a /, e.g. A user can specify */3 for minute to represent every 3 minutes.

For convenience, some simple and specific characters are set as valid value for TIMETABLE. Details are as follows.

CHARACTER	DESCRIPTION
@MINUTE	the first second of every minute
@HOURLY	the first minute of every hour
@MIDNIGHT	the first minute of every day
@DAILY	the first minute of every day
@WEEKLY	the first minute of every Monday
@MONTHLY	the first minute of the 1st of every month
@ONCE	This task will be executed only once, whether it is executed successfully or not. The starting time is specified by STARTTIME; if STARTTIME is 'now()', this task will be exected immediately.
@ONCE M N	This task will be executed sccussfully only once. If fail, this task will be executed once every n minutes until a success, and this kind of attemps can be done at most m times. m' range is 1 ~ 525600; n's range is 1 ~ 1440.

Table 5-2 Valid Special Characters table

➲ **Example**

The following syntax is used to create a schedule named **insert_into_t1** for task **insert_t1**. For more information of task **insert t1**, please refer to the example in Chapter 5.27, *TASK_CREATE*.

```
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.
```

5.8 SCHEDULE_DISABLE

The SCHEDULE_DISABLE system-stored procedure is used to disable a schedule.

Disabling a schedule means that, although the metadata of the schedule is there, it should not run and dmschsvr will not load the schedule for processing. When a schedule is disabled, its state in the system table is changed to disabled. A newly created schedule, except a one-off one, is disabled by default.

➲ **The prototype for SCHEDULE_DISABLE is:**

```
SCHEDULE_DISABLE(VARCHAR(128) SCHEDULE_NAME INPUT)
```

schedule_name............……the name of the schedule to disable

➲ **Example**

The following syntax is used to disable schedule **insert_into_t1**. For more information of schedule **insert_into_t1**, please refer to the example in Chapter 5.7, *SCHEDULE_CREATE.*

```
dmSQL> CALL SCHEDULE_DISABLE('insert_into_t1');
```

5.9 SCHEDULE_DROP

The SCHEDULE_DROP system-stored procedure is used to delete an existing schedule. If a schedule is dropped, the record about it stored in SYSSCHEDULE also is dropped.

➲ **The prototype for SCHEDULE_DROP is:**

```
SCHEDULE_DROP(VARCHAR(128) SCHEDULE_NAME INPUT)
```

schedule_name ....the name of the schedule to delete

➲ **Example**

The following syntax is used to delete schedule **insert_into_t1**. For more information of schedule **insert_into_t1**, please refer to the example in Chapter 5.7, *SCHEDULE_CREATE*.

```
dmSQL> CALL SCHEDULE_DROP('insert_into_t1');
```

5.10 SCHEDULE_ENABLE

The SCHEDULE_ENABLE system-stored procedure is used to enable a schedule.

The effect of using this procedure is that the schedule will now be loaded by dmschsvr for processing. Usually a newly created schedule is disabled by default, so users need to enable it before running it.

➲ **The prototype for SCHEDULE_ENABLE is:**

```
SCHEDULE_ENABLE(VARCHAR(128)  SCHEDULE_NAME INPUT)
```

schedule_name............……the name of the schedule to enable

➲ **Example**

The following syntax is used to ensable schedule **insert_into_t1**. For more information of schedule **insert_into_t1**, please refer to the example in Chapter 5.7, *SCHEDULE_CREATE.*

```
dmSQL> CALL SCHEDULE_ENABLE('insert_into_t1');
```

5.11   SCHEDULE_RELOAD

The SCHEDULE_RELOAD system-stored procedure is used to reload all enabled schedules into system. **Dmschsvr** automatically check whether there are altered or newly created schedule every minute, and if find some, reload all enabled schedules.

➲   **The prototype for SCHEDULE_RELOAD is:**
```
SCHEDULE_RELOAD
```

➲   **Example**

The following syntax is used to reload all enabled schedules into system.
```
dmSQL> CALL SCHEDULE_RELOAD;
```

5.12 SCHELOG_CLEAN

The SCHELOG_CLEAN system-stored procedure is used to clean excessive logs and only keep logs of recently days. Only a user with DBA authority or higher can call it.

⇨ **The prototype for SCHELOG_CLEAN is:**

```
SCHELOG_CLEAN(INT RESERVE_DAY INPUT)
```

reserve_day...................the number of days between creation time of schedule logs to delete and that of the most recent schedule logs. Schedule logs are stored in SYSSCHELOG, and the range of this value is 0 ~ 7300 (20years).

⇨ **Example**

A user's the most recent logs are created 10 days ago, and the following syntax is used to clean logs of which creation is earlier than the most recent logs by 20 days, namely logs created 30 days ago.

```
dmSQL> CALL SCHELOG_CLEAN(20);
```

5.13 SETAFFINITY

The SETAFFINITY system-stored procedure is used to set CPU affinity of processes and threads. Please note that only SYSADM can call the SETAFFINITY system-stored procedure.

Using GETCPUNUMBER and SETAFFINITY system-stored procedures, user can get the current system state and set a connection's CPU affinity without restarting DBMaker during runtime.

CPU affinity is difined by affinity mask in which each bit represents one processor. DBMaker define affinity mask as char(64), so it most set 64 CPU.

➲ **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. Users 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 1**

There are affinity mask values for an 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

➥ **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.

Get the number of CPU by calling GETCPUNUMBER:

```
dmSQL> CALL GETCPUNUMBER(?);
```

Get the CPU usage of every connection, correct affinity mask:

```
dmSQL> SELECT connection_id, affinity_mask, priority_level, cpu_usage FROM
sysuser;
```

Set CPU affinity 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');
```

Get CPU affinity mask by querying sysuser for a precise connection:

```
dmSQL> SELECT affinity_mask FROM sysuser WHERE connection_id = ?;
```

5.14 SETPRIORITY

The SETPRIORITY system-stored procedure is used to set the priority of processes and threads. Please note that only sysadm can call the PRIORITY system-stored procedure.

Using the SETPRIORITY system-stored procedure, users can set a connection's priority without restarting DBMaker during runtime. Please note that user can't set a higher priority on Linux because it need root privilege. So you can only set lower level on Linux, but there are no limits on Windows.

➲ **The prototype for SETPRIORITY is:**

```
SETPRIORITY(INT CONNECTION_ID INPUT,
            INT PRIORITY_LEVEL INPUT)
```

connection_id…….……input parameter, the ID of connections or servers. Users 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.

➲ **Example**

Users must get some system information before setting priority, 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：

```
dmSQL> SELECT connection_id , user_name FROM sysuser;


    CONNECT*                USER_NAME
===============    =======================
         30420    BACKUP_SERVER
         30418    SYSADM


2 rows selected
dmSQL> CALL SETPRIORITY(30418,3);
```

To get the priority level by querying sysuser for a precise connection:

```
dmSQL> SELECT priority_level FROM sysuser WHERE connection_id = ?;
```

5.15   SETSYSTEMOPTION

The SetSystemOption system-stored procedure is used to set system option during run time, that is to say, these validsystem option values can be changed during the run time with the SetSystemOption system stored procedure and call GetSystemOption to get the value of system option.

The following table lists all of the **option_name** system option values that obtained by calling the system-stored procedure SetSystemOption, and a brief description of what keyword is contained in each **option_name**. For more details of related keywords, please refer to *Database Administrator's Guide.*

OPTION_NAME	DESCRIPTION
FODIR	On-line change the system file object directory (DB_FoDir). option_name is the new full path. Empty string, i.e. ' ', disables the feature of the system file object.
LGSVR	Server log level (DB_LgSvr)
LGERR	Server log error level (DB_LgErr)
LGSTM	Server log statment execution time over n secs (DB_LgSTm)
LGSYS	Server log the system info (DB_LgSys)
LGFSZ	Server log file size (DB_LgFSz)
LGFNO	Server log file number (DB_LgFNo)
LGSQL	Server log the sql command (DB_LgSQL)
LGPLN	Server log the execution plan (DB_LgPLn)
LGPAR	Server log the input parameter value (DB_LgPar)
LGLCK	Server log extra lock time out informatin when it exceed error argument's length (DB_LgLck)
LGDIR	Server log directory (DB_LgDir)
LGDAY	The number of days to keep server log files (DB_LgDay)
LGZIP	Zip closed log files (DB_LgZip)
BKCHK	Whether check database before full backup and differential backup (DB_ BkChk)

BKCMP	The compact backup mode (DB_BkCmp)
BKDIR	Directory to store backup journal files (DB_BkDir)
BKFOM	The file object (FO) backup mode (DB_BkFoM)
BKFRM	The format Backup Server used to name incremental backup journal files (DB_BkFrm)
BKFUL	The percentage full of the journal files that triggers the backup server to perform an incremental backup (DB_BkFul)
BKITV	The backup time interval (DB_BkItv)
BKODR	The directories where the backup server puts the previous version of full backup files (DB_ BkOdr)
BKRTS	Whether the backup server includes the read-only tablespace files when performing a full-backup (DB_BkRTs)
BKSPM	The store procedure(SP) backup mode(DB_BkSPm)
BKSVR	whether a backup server is activated (DB_BkSvr)
BKTIM	The first time a backup server performs an incremental backup (DB_BkTim)
BKZIP	Whether the backup files are compressed by a backup server when performing full backups (DB_BkZip)
CTBLM	The default lock mode used when creating a table (DB_CTbLM)
DBKMX	The maximum number of differential backup after a full backup (DB_DbKmx)
DBKTV	The differential backup time interval (DB_DbKtv)
DDBMD	whether the DDB (Distributed DataBase) function is enabled on the database server (DD_DDBMd)
EXTNP	A size for DBMaker to extend autoextend tablespace (DB_ExtNp)
FBKTM	The first time the Backup Server will perform a full backup (DB_FBkTm)
FBKTV	The full backup time interval (DB_FBkTv)
FOSUB	The maximum number of file objects that may be stored in each system file object subdirectory (DB_FoSub)

IDXDP	Auto drop index threshold for auto index daemon (DB_IdxDp)
IDXLN	Auto create index threshold for auto index daemon (DB_IdxLn)
IDXTM	The start time for auto index daemon (DB_IdxTm)
IDXTV	The auto index daemon interval (DB_IdxTv)
IDXSV	Activate auto index daemon (DB_IdxSv)
LETPT	The Lock Escalation Threshold for escalating a page lock to a table lock (DB_LetPT)
LETRP	The Lock Escalation Threshold for an escalating rowlock to a page lock (DB_LetRP)
LIC_RELOAD	Reload License
SQLST	The display mode of the SQL command monitor (DB_SQLSt)
STSVR	Start update statistics daemon (DB_StSvr)
STMOD	The incremental update statistics mode for a database (DB_StMod)
STSTM	The start time for update statistics (DB_StsTm)
STSTV	The update statistics daemon interval (DB_StsTv)
STSSP	The update statistics sample (DB_StsSP)
STS_ABORT	Abort ongoing update statistics
USRFO	User file objects can be inserted in a database (DB_UsrFo)

➲ **The prototype for SETSYSTEMOPTION is:**

```
SETSYSTEMOPTION(VARCHAR(32) OPTION NAME INPUT,
                VARCHAR(8576) OPTION_VALUE INPUT)
```

option_name .... the name of the system option

option_value .... the value of the system option

➲ **Example 1**

The following syntax is used to activate the Backup Server.

```
dmSQL> CALL SETSYSTEMOPTION('BKSVR','1');
```

● **Example 2**

When Backup Server is activated, the following syntax is used to set the appropriate backup parameters in the **dmconfig.ini** configuration file.

```
dmSQL> CALL SETSYSTEMOPTION('STARTBACKUP','1'); //do full backup
dmSQL> CALL SETSYSTEMOPTION('STARTBACKUP','2'); //do incremential backup
dmSQL> CALL SETSYSTEMOPTION('STARTBACKUP','3'); //do differential backup
```

● **Example 3**

The following syntax is used to set update statistics sample to **60** during run time.

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

● **Example 4**

The following syntax is used to abort an ongoing update statistics.

```
dmSQL> CALL SETSYSTEMOPTION('STS_ABORT', '14076'); // abort an ongoing update
statistics which connection ID is 14076.
dmSQL> CALL SETSYSTEMOPTION('STS_ABORT', '0'); // the value 0 is a special
connection ID.It means abort all command related to update statistics.
```

● **Example 5**

The following syntax is used to activate an auto index daemon.

```
dmSQL> CALL SETSYSTEMOPTION ('IDXSV', '1');
```

● **Example 6**

After Auto Index Daemon is activated, the following syntax is used to set the appropriate auto index daemon parameters in the **dmconfig.ini** configuration file.

```
dmSQL> CALL SETSYSTEMOPTION('IDXTM', '2012-12-12 00:00:00'); // The first time
the auto index daemon starts for the first time at 2012-12-12 00:00:00.
dmSQL> CALL SETSYSTEMOPTION('IDXTV', '2-00:00:00'); // The interval of performing
the auto index daemon is 2 days.
dmSQL> CALL SETSYSTEMOPTION('IDXDP', '60'); // An index which is not used reaches
or exceeds 60 days will be dropped by auto index daemon.
dmSQL> CALL SETSYSTEMOPTION('IDXLN', '10'); // If the same scan log number
reaches or exceeds 10, an auto index will be created according to these log.
```

● **Example 7**

If user update the license, call this function may reload the license.

```
dmSQL> CALL SETSYSTEMOPTION('LIC_RELOAD', '1');
```

5.16 SETSYSTEMOPTIONW

The SetSystemOptionW system-stored procedure is used to set system option during run time and writing run time setting to **dmconfig.ini** file.

This stored procedure is an extension of SetSystemOption, and support all system options that SetSystemOption can change, The following table lists all of the **option_name** system option values that obtained by calling the system-stored procedure SetSystemOptionW, and a brief description of what keyword is contained in each **option_name**. For more details of related keywords, please refer to *Database Administrator's Guide.*

DBMaker can set system option at run time by calling the system stored procedure setSystemOption(), and now add new system stored procedure setSystemOptionW() to support setting system option at run time and writing run time setting to **dmconfig.ini** file. User also can get new option value through calling getSystemOtion.

OPTION_NAME	DESCRIPTION
FODIR	On-line change the system file object directory (DB_FoDir). option_name is the new full path. Empty string, i.e. ' ', disables the feature of the system file object.
LGSVR	Server log level (DB_LgSvr)
LGERR	Server log error level (DB_LgErr)
LGSTM	Server log statment execution time over n secs (DB_LgSTm)
LGSYS	Server log the system info ((DB_LgSys)
LGFSZ	Server log file size (DB_LgFSz)
LGFNO	Server log file number (DB_LgFNo)
LGSQL	Server log the sql command (DB_LgSQL)
LGPLN	Server log the execution plan (DB_LgPLn)
LGPAR	Server log the input parameter value (DB_LgPar)
LGLCK	Server log extra lock time out informatin when it exceed error argument's length (DB_LgLck)
LGDIR	Server log directory (DB_LgDir)

LGDAY	The number of days to keep server log files (DB_LgDay)
LGZIP	Zip closed log files (DB_LgZip)
BKCHK	Whether check database before full backup and differential backup (DB_ BkChk)
BKCMP	The compact backup mode (DB_BkCmp)
BKDIR	Directory to store backup journal files (DB_BkDir)
BKFOM	The file object (FO) backup mode (DB_BkFoM)
BKFRM	The format Backup Server used to name incremental backup journal files (DB_BkFrm)
BKFUL	The percentage full of the journal files that triggers the backup server to perform an incremental backup (DB_BkFul)
BKITV	The backup time interval (DB_BkItv)
BKODR	The directories where the backup server puts the previous version of full backup files (DB_ BkOdr)
BKRTS	Whether the backup server includes the read-only tablespace files when performing a full-backup (DB_BkRTs)
BKSPM	The store procedure(SP) backup mode(DB_BkSPm)
BKSVR	whether a backup server is activated (DB_BkSvr)
BKTIM	The first time a backup server performs an incremental backup (DB_BkTim)
BKZIP	Whether the backup files are compressed by a backup server when performing full backups (DB_BkZip)
CTBLM	The default lock mode used when creating a table (DB_CTbLM)
DBKMX	The maximum number of differential backup after a full backup (DB_DbKmx)
DBKTV	The differential backup time interval (DB_DbKtv)
DDBMD	whether the DDB (Distributed DataBase) function is enabled on the database server (DD_DDBMd)
EXTNP	A size for DBMaker to extend autoextend tablespace (DB_ExtNp)
FBKTM	The first time the Backup Server will perform a full

	backup (DB_FBkTm)
FBKTV	The full backup time interval (DB_FBkTv)
FOSUB	The maximum number of file objects that may be stored in each system file object subdirectory (DB_FoSub)
IDXDP	Auto drop index threshold for auto index daemon (DB_IdxDp)
IDXLN	Auto create index threshold for auto index daemon (DB_IdxLn)
IDXTM	The start time for auto index daemon (DB_IdxTm)
IDXTV	The auto index daemon interval (DB_IdxTv)
IDXSV	Activate auto index daemon (DB_IdxSv)
LETPT	The Lock Escalation Threshold for escalating a page lock to a table lock (DB_LetPT)
LETRP	The Lock Escalation Threshold for an escalating rowlock to a page lock (DB_LetRP)
SQLST	The display mode of the SQL command monitor (DB_SQLST)
STSVR	Start update statistics daemon (DB_StSvr)
STMOD	The incremental update statistics mode for a database (DB_StMod)
STSTM	The start time for update statistics (DB_StsTm)
STSTV	The update statistics daemon interval (DB_StsTv)
STSSP	The update statistics sample (DB_StsSP)
USRFO	User file objects can be inserted in a database (DB_UsrFo)

➲ **The prototype for SETSYSTEMOPTIONW is:**

```
SETSYSTEMOPTIONW(VARCHAR(32) OPTION_NAME INPUT,
                 VARCHAR(8576) OPTION_VALUE INPUT)
```

option_name ....the name of the system option

option_value.............the value of the system option

➲ **Example**

To start update statistics daemon, set value of STSVR to **1** by calling **setSystemOptionW** during runtime, and then the run time setting will be written into **dmconfig.ini** file, as following:

The **dmconfig.ini** file before calling stored procedure **setSystemOptionW()** is:

```
[DBSAMPLE5]
; Here omit other keywords
DB_StSvr = 0
```

Execute call **setSystemOptionW** ('**optionName**', '**optionValue**'):

```
dmSQL> CALL SETSYSTEMOPTIONW('STSVR', '1');
dmSQL> CALL SETSYSTEMOPTION('STSVR',?);
OPTION_VALUE : 1
```

The **dmconfig.ini** file after calling stored procedure **setSystemOptionW** is:

```
[DBSAMPLE5]
; Here omit other keywords
DB_StSvr = 1
```

5.17   SOADD

The SOADD system-stored procedure is used to increase the shared object's value.

➲ **The prototype for SOADD is:**

```
SOADD(INTEGER SHID,
      INTEGER ADDEND,
      INTEGER NEW_VAL OUTPUT)
```

shid..........................the id of the shared object

addend.....................the positive or negative value to add

new_val...................the value after adding

➲ **Example**

The following syntax is used to add 3 to shared object 2 and get the new value = 3.

```
dmSQL> CALL SYSADM.SOADD(2,3,?);
new_val: 3
```

5.18   SOCREATE

The SOCREATE system-stored procedure is used to create shared objects. To use a shared object, use SOCreate to create the shared object with a specified identifier and initial value. Then, to read, modify, or to increase the shared object value use SORead, SOSet or SOAdd respectively by indicating its identifier. Since the shared object can be accessed by any connection, it supports SOLock and SOUnlock for concurrency control. When the shared object is no longer in use it can be dropped with SODrop.

➲ **The prototype for SOCREATE is:**
```
SOCREATE(INTEGER SETID,
         INTEGER INIT_VAL,
         INTEGER SHID OUTPUT)
```

Setid……………… the assigned id of the shared object

   0: system assigned, otherwise:user assigned

init_val....................initial value

shid ....id of the created shared object

➲ **Example 1**

The following syntax is used to create a shared object with an initial value = 0 with a system assigned id = 0.
```
dmSQL> CALL SYSADM.SOCREATE(0,0,?);
Shid: 1
```

➲ **Example 2**

The following syntax is used to create shared object 2 with an initial value = 0.
```
dmSQL> CALL SYSADM.SOCREATE(2,0,?);
Shid: 2
```

5.19 SODROP

The SODROP system-stored procedure is used to drop a shared object. This can be used when the object is no longer in use.

➲ **The prototype for SODROP is:**

```
SODROP(INTEGER SHID)
```

shid..........................id of the shared object to drop

➲ **Example**

The following syntax is used to drop shared object **1**.

```
dmSQL> CALL SYSADM.SODROP(1);
```

5.20　SOLOCK

The SOLOCK system-stored procedure is used to lock a shared object. After a shared object has been locked, other users cannot read, set, add, drop, lock, or unlock it. Only the user that set the lock can use the other six system-stored procedures on it.

➲　**The prototype for SOLOCK is:**

```
SOLOCK(INTEGER SHID)
```

shid ....id of shared object which are desired to lock

➲　**Example**

The following syntax is used to lock shared object **1**.

```
dmSQL> CALL SYSADM.SOLOCK(1);
```

5.21   SOREAD

The SOREAD system-stored procedure is used to read (get) the value of a shared object.

⮑ **The prototype for SOREAD is:**

```
SOREAD(INTEGER SHID,
       INTEGER VAL OUTPUT)
```

shid..........................the id of shared object

val ....value of the shared object

⮑ **Example**

The following syntax is used to get the value of shared object **2**.

```
dmSQL> CALL SYSADM.SOREAD(2,?);
val: 3
```

5.22 SOSET

The SOSET system-stored procedure is used to set or modify a shared object's values.

➲ **The prototype for SOSET is:**

```
SOSET(INTEGER SHID,
      INTEGER NEW VAL,
      INTEGER OLD_VAL OUTPUT)
```

shid .... the id of shared object

new_val .... value to assign

old_val .... value before the assignment

➲ **Example**

The following syntax is used to set the value of shared object **2** to **–2**.

```
dmSQL> CALL SYSADM.SOSET(2,-2,?);
old_val: 3
```

5.23   SOUNLOCK

The SOUNLOCK system-stored procedure is used to unlock a shared object. After a shared object has been locked, other users cannot read, set, add, drop, lock, or unlock it. Only the user that placed a lock on the shared object may unlock it.

➲ **The prototype for SOUNLOCK is:**

```
SOUNLOCK(INTEGER SHID)
```

shid..........................id of shared object to be unlocked

➲ **Example**

The following syntax is used to unlock shared object 1.

```
dmSQL> CALL SYSADM.SOUNLOCK(1);
```

5.24 START_DMSCHSVR

The START_DMSCHSVR system-stored procedure is used to start **dmschsvr**.

➲ **The prototype for START_DMSCHSVR is:**

```
START DMSCHSVR(VARCHAR(8)   TASKRUNNUM INPUT,
              VARCHAR(128)  SCHELOGDIR INPUT)
```

Taskrunnum.............the task numbers that can be aroused by dmschsvr at the same time. The range of this value is 1 ~ 50, and the default value is *30*.

schelogdir..................the path indicating the directory of dmschsvr's log files. The default path is same with the path specified by **DB_DbDir**. The log filename format is <DB_NAME><_><Date>, e.g. DBSAMPLE5_20150135.log.

➲ **Example**

The following syntax is used to start **dmschsvr**.

```
dmSQL> CALL START_DMSCHSVR ('30', 'C:\DBMaker\5.4\SAMPLES\DATABASE');
```

5.25   STOP_DMSCHSVR

The STOP_DMSCHSVR system-stored procedure is used to stop **dmschsvr**.

➲   **The prototype for STOP_DMSCHSVR is:**

```
STOP_DMSCHSVR
```

➲   **Example**

The following syntax is used to stop **dmschsvr.**

```
dmSQL> CALL STOP_DMSCHSVR;
```

5.26   TASK_ALTER

The TASK_ALTER system-stored procedure is used to alter an existing task.

Except for TASK_NAME, all task parameters can be altered. If a parameter of a task is altered while the task is running, the task will use the new parameter when it runs next time according to users' schedule.

➲ **The prototype for TASK_ALTER is:**

```
TASK_ALTER(VARCHAR(128) TASK_NAME INPUT,
           VARCHAR(16) TASK_TYPE INPUT,
           VARCHAR(2048) ACTIONS INPUT)
```

task_name................the name of the existing task to alter.

task_type..................the type of the task. There are three options: SQL_STATEMENT (abbr. SQL), STORE_PROCEDURE (abbr. SP), and EXECUTABLE (abbr. EXEC). SQL_STATEMENT means the task is a sql statement; STORE_PROCEDURE means the task is a procedure; EXECUTABLE means the task is an executable program.

actions....................the actions that the existing task performs regularly. It must match the type of the existing task.

➲ **Example**

The following syntax is used to alter task "**insert_t1**". In this example, alter the action "**INSERT INTO t1 VALUES(1, 2)**" to "**INSERT INTO t1 VALUES(1, 3)**". For more information of task **insert_t1**, please refer to the example in Chapter 5.27, *TASK_CREATE*.

```
dmSQL> CALL TASK_ALTER('insert_t1','SQL_STATEMENT','INSERT INTO t1 VALUES(1,3)');
```

5.27   TASK_CREATE

The TASK_CREATE system-stored procedure is used to create a task. A task is user-defined and scheduled to run one or more times. It is a combination of actions (what needs executions) and executed by a schedule.

➲ **The prototype for TASK_CREATE is:**
```
TASK_CREATE(VARCHAR(128) TASK_NAME INPUT,
            VARCHAR(16) TASK_TYPE INPUT,
            VARCHAR(2048) ACTIONS INPUT)
```

task_name..............the name of the task to create. It can contain 1 to 128 letters, numbers, and underscores, but the first character cannot be numbers.

task_type...............the type of the task to create. There are three options: SQL_STATEMENT (abbr. SQL), STORE_PROCEDURE (abbr. SP), and EXECUTABLE (abbr. EXEC). SQL_STATEMENT means the task is a sql statement; STORE_PROCEDURE means the task is a procedure; EXECUTABLE means the task is an executable program.

actions...................the actions that the task will perform regularly. It must match the type of the task to create, and its maximum length is 2K bytes.

➲ **Example**

The following syntax is used to 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)');
```

5.28 TASK_DROP

The TASK_DROP system-stored procedure is used to delete an existing task.

If a task has been added into a schedule, an error will occur when a user uses TASK_DROP to delete this task, that is to say, users should make sure that no user use this task before dropping it.

➲ **The prototype for TASK_DROP is:**

```
TASK_DROP(VARCHAR(128) TASK_NAME INPUT)
```

task_name................the name of the task to delete

➲ **Example**

The following syntax is used to delete task **insert_t1**. For more information of task **insert_t1**, please refer to the example in Chapter 5.27, *TASK_CREATE*.

```
dmSQL> CALL TASK_DROP('insert_t1');
```

5.29 XMLEXPORT

The XMLEXPORT system-stored procedure provides a programmable interface for users to export XML data from DBMaker. Only a user with SYSADM, SYSDBA or DBA security privilege can call these stored procedures. In addition, the execute privilege cannot be granted to other users because XMLEXPORT is a system-stored procedures.

XMLEXPORT exports tables from a DBMaker database to an XML file and can process multiple tables within one call of the corresponding stored procedures. Descriptions on the mapping between the content of XML files and DBMaker tables are outlined in a description string. This description string is used as one of the arguments passed into the stored procedure.

⇨ **The prototype for XMLEXPORT is:**

```
XMLEXPORT(VARCHAR(256) FILE_PATH,
          VARCHAR(256) DB_TAG,
          VARCHAR(256) XML_HEADER,
          VARCHAR(16000) OBJECT_STR,
          VARCHAR(256) OPTION_STR,
          VARCHAR(256) LOG_PATH)
```

NAME	TYPE	LENGTH (BYTES)	DESCRIPTION	CASE SENSITIVITY
file_path	varchar	256	Full path of exported xml file	Depends on operating system
db_tag	varchar	256	Customized database tag	Yes (output has the same capitalization)
xml_header	varchar	256	Customized xml header	Yes (output has the same capitalization)
object_str	varchar	16000	Description string for exported objects	Depends on DBMaker setting
option_flag	varchar	256	Description string for option flags	No
log_path	varchar	256	Full path of error log file on the client	Varies by operating system

Table 5-3 XMLEXPORT Arguments table

Constructing XMLEXPORT Arguments

The XML file designated for exporting from a database must first be generated on the server. The file_path is specified by a full path string passed in as one of the arguments of the corresponding stored procedure.

Next, the db_tag is used to customize a tag. The default value (i.e., database name) is used if a NULL or empty string is present.

Next, the argument object_str is used as shown here:

```
Object_str=:
    { <element> [; <element>…]


<element>=:
    {TABLE_NAME | <select_query>} [#TABLE_TAG]
```

An <element> represents a table and is delimited by semi-colons. If the first token from <element> is "select" (case insensitive comparison), this <element> is seen as <select_query> [#TABLE_TAG]. Otherwise, this <element> is seen as

TABLE_NAME [#TABLE_TAG]. "If <element> = TALBE_NAME [#TABLE_TAG]", all columns in this table are selected and no customized column tag can be specified. That is to say, in the exported XML file, the names of column tags are the same as their corresponding table column names. Customized table tags are specified with TABLE_TAG. The table name in the database is used as table tag name when a TABLE_TAG is not specified.

If users want to specify a customized column tag name, they can only use <select_query>[#TABLE_TAG] in the <element> string. The customized column tag names are specified by using column alias names in the <select_query> statement. The user must use "AS" in their <select_query>, for example, "select c1 as name, c2 as type from t2" as the <select_query> statement, then column c1 becomes the "name" tag and column c2 becomes the "type" tag in the exported XML file.

Next, users can specify an option string using option_flag. Each option is separated by a semicolon. For example, to treat column names as attributes, use "column_as_attribute" in the option string. If users do not specify a certain option, that option is not set. The option flag string is case-insensitive.

OPTION FLAG	SET	NOT SET
blob_in_separate_file	BLOB/CLOB column data is exported as a temp file separate from the XML file. The name of that temp file is recorded in the exported DTD.	Blob/Clob column data is exported as part of the XML file.
column_as_attribute	Columns are exported as attributes instead of an element in the XML file.	Columns are exported as an element in the XML file.
capitalize_tag_name	All tag names are capitalized in the XML file.	The capitalization of all tag names stays the same as that of the corresponding names in database.
file_type_as_link	File type data content is not exported. Only the name of the file is exported to the XML file.	File type data content will be exported as part of the XML file.

no_schema_dtd	Will not generate a schema DTD when the XML file is generated.	Will generate a corresponding DTD along with the XML file exported.

Table 5-4 XMLEXPORT Options

Lastly, log files generated during XML file exporting are saved on the client machine in the log_path.

Exporting XML Files

Suppose that we want to export two tables named *tb_card,* and *tb_contact* as one file /usr/john/xmlexport.xml from a DBMaker database called *Customer*. In the xmlexport.xml file, we want to use "EMPLOYEE" as our customized database tag, "TITLE" as our customized table tag for the table "tb_card" and "NUMBER" as our customized table tag for the table "tb_contact".

In addition, the customized column tags for ID, FNAME, LNAME and WORK of the table **tb_card** are NO, FIRST_NAME, LAST_NAME and JOB respectively. We will not use customized column tags for the table "**tb_contact**". We also want to capitalize all tag names in the XML file and all BLOB column data (if any) will be saved in another temporary file. Finally, our log file name is going to be saved as /client/john/xmlexport.log. The contents of these two tables are as follows:

```
dmSQL> SELECT * FROM tb_card;
    ID          FNAME               LNAME               WORK
========== ==================== ==================== ====================
         1 Eddie                Chang                Manager
         2 Hook                 Hu                   SoftwareEngineer
         3 Jackie               Yu                   SoftwareEngineer
         8 Jerry                Liu                  Manager

dmSQL> SELECT * FROM tb_contact;
    NO          FIRST_NAME          LAST_NAME           PHONE
========== ==================== ==================== ====================
         1 Eddie                Chang                2145678
         2 Hook                 Hu                   2335678
```

```
      3  Jackie          Yu             2346678
      4  Jerry           Liu            2345671
```

➲ **To export an XML file**

1. File_path is the full path of the XML file to be exported. The generated file will be on the server, thus the specified file path must also be on the server. The string '/usr/john/xmlexport.xml' will be used for this argument.

2. db_tag is a customized database tag. A NULL or empty string means that a default value is used. The string EMPLOYEE will be used for this argument.

3. In this example, we will use the object_str string;

```
'SELECT ID AS NO, FNAME AS FIRST_NAME, LNAME AS LAST_NAME, WORK
AS JOB FROM tb_card#TITLE;tb_contact#NUMBER'
```

4. We will use the "capitalize_tag_name;blob_in_separate_file" tag as our option string for this argument.

5. For this argument, we will use "/client/john/xmlexport.log" for log path.

6. The resulting CALL XMLExport statement will have the following form:

```
CALL XMLExport(

'/usr/john/xmlexport.xml',

'EMPLOYEE',

'SELECT ID AS NO, FNAME AS FIRST_NAME, LNAME AS LAST_NAME, WORK
AS JOB FROM tb_card#TITLE;tb_contact#NUMBER',

'capitalize_tag_name;blob_in_separate_file',

'/client/john/xmlexport.log');
```

7. Part of the export file xmlexport.xml would be:

```
<EMPLOYEE>

  <TITLE>

    <NO>1</NO>

    <FIRST_NAME>Eddie</FIRST_NAME>
```

```
    <LAST_NAME>Chang</LAST_NAME>

    <JOB>Manager</JOB>

</TITLE>

<TITLE>

    <NO>2</NO>

    <FIRST_NAME>Hook</FIRST_NAME>

    <LAST_NAME>Hu</LAST_NAME>

    <JOB>SoftwareEngineer</JOB>

</TITLE>

<TITLE>

    <NO>3</NO>

    <FIRST_NAME>Jackie</FIRST_NAME>

    <LAST_NAME>Yu</LAST_NAME>

    <JOB>SoftwareEngineer</JOB>

</TITLE>

<TITLE>

    <NO>4</NO>

    <FIRST_NAME>Jerry</FIRST_NAME>

    <LAST_NAME>Liu</LAST_NAME>

    <JOB>Manager</JOB>

</TITLE>

<NUMBER>

    <NO>1</NO>

    <FIRST_NAME>Eddie</FIRST_NAME>
```

```
        <LAST_NAME>Chang</LAST_NAME>

        <PHONE>2145678</PHONE>

     </NUMBER>

     <NUMBER>

        <NO>2</NO>

        <FIRST_NAME>Hook</FIRST_NAME>

        <LAST_NAME>Hu</LAST_NAME>

        <PHONE>2335678</PHONE>

     </NUMBER>

     <NUMBER>

        <NO>3</NO>

        <FIRST_NAME>Jackie</FIRST_NAME>

        <LAST_NAME>Yu</LAST_NAME>

        <PHONE>2346678</PHONE>

     </NUMBER>

     <NUMBER>

        <NO>4</NO>

        <FIRST_NAME>Jerry</FIRST_NAME>

        <LAST_NAME>Liu</LAST_NAME>

        <PHONE>2345671</PHONE>

     </NUMBER>

  </EMPLOYEE>
```

➲ **Alternatively**

1. Using the option "**column_as_attribute**" and calling XMLExport:

```
CALL XMLExport(

'/usr/john/xmlexport.xml',

'EMPLOYEE',

'SELECT ID AS NO, FNAME AS FIRST_NAME, LNAME AS LAST_NAME, WORK
AS JOB FROM tb_card#TITLE ',

'capitalize_tag_name;blob_in_separate_file;column_as_attribute
','/client/john/xmlexport.log');
```

2. The partial result will become:

```
<EMPLOYEE>

  <TITLE NO="1" FIRST_NAME="Eddie" LAST_NAME="Chang"
JOB="Manager" />

  <TITLE NO="2" FIRST_NAME="Hook"  LAST_NAME="Hu"
JOB="SoftwareEngineer" />

  <TITLE NO="3" FIRST_NAME="Jackie" LAST_NAME="Yu"
JOB="SoftwareEngineer" />

  <TITLE> NO="4" FIRST_NAME="Jerry" LAST_NAME="Liu"
JOB="Manager" />

</EMPLOYEE>
```

5.30   XMLIMPORT

The XMLIMPORT system-stored procedure provides a programmable interface for users to import XML data to DBMaker. Only a user with SYSADM, SYSDBA or DBA security privilege can call these stored procedures. In addition, the execute privilege cannot be granted to other users because XMLIMPORT is a system-stored procedures.

XMLIMPORT will import tables from XML files to tables in DBMaker. When importing from an XML file, users can simply store the whole XML file in the database instead of parsing, (analyzing the file content and importing data into tables). The XML file being imported must be on the server and the log file generated during the importing of an XML file is saved on the client machine.

If users just want to store the whole XML file instead of parsing it, they must specify the "key" used for storing the XML file. The key value can then be used when querying a database for the stored XML file.

➲   **The prototype for XMLIMPORT is:**

```
XMLIMPORT(VARCHAR(256) FILE_PATH,
          VARCHAR(16000) OBJECT_STR,
          VARCHAR(256) OPTION_STR,
          VARCHAR(256) LOG_PATH)
```

NAME	TYPE	LENGTH (BYTES)	DESCRIPTION	CASE SENSITIVITY
file_path	varchar	256	Full path of exported xml file	Depends on operating system
object_str	varchar	16000	Description string for exported objects	XML tags are case sensitive; table names and table column names depends on DBMaker setting
option_flag	varchar	256	Description string for option flags	No
log_path	varchar	256	Full path of error log file on the client	Varies by operating system

Table 5-5 XMLIMPORT Arguments table

Constructing XMLIMPORT Arguments

First, the XML file being imported from a database must be generated on the server. The file_path is specified by a full path string passed in as one of the arguments of the corresponding stored procedure.

Second, the object_str argument is used to describe imported objects. This information includes document levels, the mapping between customized column tag names, and inserted table column names, as well as the mapping between customized table tag name and table name in the database. The format is as follows:

```
object_str =:
      { <table_element> [; <table_element>]...}


<table_element> =
      { <document mapping information>#<table mapping information> }


<document mapping information> =:
      {<document level string>[(<column tag names>)]}


<document level string> =: {/<level1> [/<level2>/.....]}
```

```
<column tag names> =: {<tag1> [, <tag2>]...}

<table mapping information> =: <table import definition>

<table import definition> =: { <insert sql statement> | <target table
name>[(<table column names>)] }

<insert sql statement> =: INSERT INTO <target table name> [(<table column
names>)] VALUES (<value list>)

<table column names> =: {<col1> [, <col2>] ...}

<value list> =: {<insert value>, <insert value>,...}

<insert value> =: {<constant> | <expression>}
```

Figure 5-1 object_str Argument Syntax

If users want to store the entire XML file instead of parsing it and storing the content in tables, they should use special handling in <column tag names>. Please see example 5.

<table_element>represents a table. The delimiter used between <element> is a semi-colon. In the <document level string>, the document levels from the root level to the table level are specified.

```
<root>
   <database>
      <table1>
          <column1>
          </column1>
          <column2>
          </column2>
      </table1>
      <table2>
      </table2>
   </database>
</root>
```

Figure 5-2 Sample XML File

Based on the sample XML file shown in Figure 5.2, to import data stored in the <table1> tag of the<database>, specify a <document level string> of the "*/root/database/table1*".

In <column tag names>, specify which column tags to insert into the table. If a <column tag names> is not specified, all column tags under a certain table tag are inserted.

In the <table import definition>, use either the format of <INSERT SQL statement> or TABLE_NAME [<table column names>]. When using the <INSERT SQL statement>, the INSERT SQL statement will be like this:

```
INSERT INTO <target table name> [(<table column names>)] VALUES (<value list>)
```

The <table column names> columns to be inserted are specified. If a <table column names> is not specified, it is implied that the user is trying to insert all columns in the target table (this is the same as the syntax for the ordinary INSERT SQL statement.) Also, if there is a <column tag names> located in the <document mapping information>, than the number of column tags specified in <column tag names> must be equal to the number of host variables in the <value list>. If there are no <column tag names> located in the <document mapping information>, it is implied that all column tags under the base element are to be inserted into the target table. The schema information in the dtd file is also used to check whether the number of tags is equal to the number of host variables located in the <value list>.

The mapping between <table column names>, <value list> and <column tag names> in the <document mapping information> file must be appropriate. The <column tag names> are mapped to host variables in the <value list> file. The sequence of columns in <table column names> combined with the sequence values in <value list> and the sequence of tags <column tag names> decides what values are inserted into <value list>.

When using <target table name>[(<table column names>)], specify the table to be inserted into <target table name>. This <target table name> is mapped to the last level in <document level string>.

When this format is used, a constant value insert or expression insert cannot be used. If there is no <column tag names> specified in <document mapping information>,

there should be no <table column names> present either. If there is <column tag names> in <document mapping information>, the number of tags in <column tag names> must be equal to the number of columns in <table column names>.

In <table column names>, specify mapped table columns to be inserted. If no <table column names> are specified, all table columns will be inserted. If that is the case, there should be no <column tag names> in <document mapping information>. The schema information in the dtd file will be used to check whether the number of all tags under the base element is equal to the number of all columns in the target table.

Users are responsible for the mapping between <table column names> and <column tag names>. The location of tags in <column tag names> should be mapped to that of columns in <table column names>.

⮕ **Example 1**

If the <table column names> is (c1, c2, c3), <value list> is (?,?,?) and <column tag names> is (tg1, tg2, tg3), the value in tg1 is inserted into c1, the value in tg2 is inserted into c2 and the value in tg3 is inserted into c3.

⮕ **Example 2**

Assume that table t1 has four columns, c1, c2, c3, and c4, and that we have four tags, tg1, tg2, tg3, tg4, in the xml element we are trying to import. Also, assume that the obj_str is, "/root/book/order(tg1, tg2)#insert into t1 (c1, c2, c3) values (?,?+3, 5)". From the string, we decide that table t1 is our target table, that the column c1 in table t1 has the inserted value of tag tg1, that column c2 has the inserted value of tag tg2 plus 3, and that column c3 has the inserted constant value of 5.

⮕ **Example 3**

If the user does not specify the usage of the <column tag names> file in the <document mapping information>, it is implied that the sequence of xml column tags matches the sequence of what is located in the <table column names>, and that all column tags under the base element are to be inserted into the target table.

Assume that our target table t2 has five columns, c1, c2, c3, c4, and c5. Also, assume that in our xml file, the sequence of tags is tg1, tg2, tg3, and tg4. If the obj_str is,

"/root/book/order#insert into t2 (c1, c2, c3, c4, c5) values (?, ?, ?, ?, 6)", the value of tg1 is inserted into c1 of t2, the value of tg2 is inserted into c2 of t2, the value of tg3 is inserted into c3 of t2, the value of tg4 is inserted into c4 of t2 and the constant value of 6 is inserted into c5 of t2.

If the obj_str is "/root/book/order(tg1, tg2, tg3, tag4)#insert into t1 values (?, ?, ?, ?)". This tells us that users are trying to insert 4 tags into all columns of our target table. The value of tg1 is inserted into c1 of t1, the value of tg2 is inserted into c2 of t1, the value of tg3 is inserted into c3 of t1, and the value of tg4 is inserted into c4 of t1.

⮞ **Example 4**

If obj_str is "/root/book/order(tg1, tg2)#insert into t1 values (?, ?, acos(1))", the result of acos(1) is inserted into c3 of t1.

⮞ **Example 5**

For users who want to store the whole XML file in the record instead of parsing the whole XML file and storing the content (i.e., parsing the whole XML file and then storing the data in XML file in table), they have to specify a "virtual tag" in <column tag names>. This special "virtual tag" is named "_XML_FILE_".

If this "_XML_FILE_" is used as the column tag name, the columns represented by the column tags preceding this special "virtual tag" are used as the key value. In addition, the mapped value in the <value list> file must be a single host variable without any further calculation.

If the following object string, "/root/book/order(tag1, tag2, XML_FILE_)#insert into t2 (c1, c2, c3, c4, c5) values (?+2, ?*5, ?, 7, 8)", is used then the whole file will be inserted into c3 of table t2.

If <table_element> in the object string, "/root/book/order(tag1, tag2, _XML_FILE_)#customer(firstname, lastname, xml_file)", is used for the table "customer", then firstname is inserted from the tag1 tag into the XML file. In addition, the lastname is inserted from the tag2 tag into the XML file and the xml_file will be inserted from the whole XML file. The firstname and lastname are used as keys for finding a specific XML file.

➲ **Example 6**

In <column tag names> = <tag1, tag2, tag3> and <table column names> = <c1, c2, c3>, there are three pairs of mapping: tag1 <-> c1, tag2 <-> c2, tag3 <-> c3. Tag names and column names are all-or-nothing. That means that empty tag names such as (tag1, ,tag3) are not permissible, neither are empty column names. All customized tag names must specify or none of them at all.

So, the object string "/root/book/order(tag1, , tag2)#insert into t2 (c1, c2) values (?, ?, ?)" is not permissible. An object string of "/root/book/order(tag1, tag2, tag3)#insert into t2 (c1, c2, c3, c4) values (?, ?, ?,) is permissible. What is inserted into c4 of t2 depends on the table schema information.

Thirdly, the option_flag string is case-insensitive. When the option_flag string is set, the column_as_attribute columns in the imported XML file are treated as attributes. When the option_flag string is not set, the columns are treated as elements in the XML file.

```
Option_flag=:{[<attribute>[;<attribute>]…]}
<attribute>=:
{
column_as_attribute
}
```

Lastly, the log file of errors generated during the importing of XML files are saved on the client machine in the log_path.

Importing XML Files

Assume that we have an XML file, xmlimport.xml under the */usr/john* directory. The file is listed as follows.

```
<ROOT>
   <EMPLOYEE>
     <TITLE>
       <TAG1>1</TAG1>
       <TAG2>Eddie</TAG2>
       <TAG3>Chang</TAG3>
       <TAG4>Manager</TAG4>
     </TITLE>
```

```
<TITLE>
  <TAG1>2</TAG1>
  <TAG2>Hook</TAG2>
  <TAG3>Hu</TAG3>
  <TAG4>SoftwareEngineer</TAG4>
</TITLE>
<TITLE>
  <TAG1>3</TAG1>
  <TAG2>Jackie</TAG2>
  <TAG3>Yu</TAG3>
  <TAG4>SoftwareEngineer</TAG4>
</TITLE>
<TITLE>
  <TAG1>4</TAG1>
  <TAG2>Jerry</TAG2>
  <TAG3>Liu</TAG3>
  <TAG4>Manager</TAG4>
</TITLE>
<NUMBER>
  <NO>1</NO>
  <FIRST_NAME>Eddie</FIRST_NAME>
  <LAST_NAME>Chang</LAST_NAME>
  <PHONE>2145678</PHONE>
</NUMBER>
<NUMBER>
  <NO>2</NO>
  <FIRST_NAME>Hook</FIRST_NAME>
  <LAST_NAME>Hu</LAST_NAME>
  <PHONE>2335678</PHONE>
</NUMBER>
<NUMBER>
  <NO>3</NO>
  <FIRST_NAME>Jackie</FIRST_NAME>
  <LAST_NAME>Yu</LAST_NAME>
  <PHONE>2346678</PHONE>
</NUMBER>
<NUMBER>
  <NO>4</NO>
  <FIRST_NAME>Jerry</FIRST_NAME>
```

```
      <LAST_NAME>Liu</LAST_NAME>
      <PHONE>2345671</PHONE>
    </NUMBER>
  </EMPLOYEE>
<ROOT>
```

We are trying to import the data recorded in the **xmlimport.xml** file into the following database schema:

```
Database Name: DB_TEST
Table Name: TB_CARD(ID CHAR(30), FNAME CHAR(30), LNAME CHAR(30), WORK CHAR(30))
Table Name: TB_CONTACT(NO CHAR(30), FIRST_NAME CHAR(30), LAST_NAME CHAR(30),
PHONE CHAR(30))
```

From the content of the above .xml file, we can see that under the <EMPLOYEE> element, there are two sub-elements. We can map <EMPLOYEE> element as the database level, the <TITLE> as the table level and the <NUMBER> as another table level in the import database.

Assume that we want to import <TITLE> into **TB_CARD** table and <NUMBER> into **TB_CONTACT** table. The mapping of xml document tags to database tables is as follows:

```
/ROOT/EMPLOYEE/TITLE -> /DB_TEST/TB_CARD
/ROOT/EMPLOYEE/NUMBER -> /DB_TEST/TB_CONTACT
```

The mapping between the XML document tags and table columns is as follows:

The elements under */ROOT/EMPLOYEE/TITLE*(the mapping between **<TITLE>** and **TB_CARD** table):

```
TAG1 -> NO
TAG2 -> FIRST_NAME
TAG3 -> LAST_NAME
TAG4 -> JOB
```

The elements under */ROOT/EMPLOYEE/NUMBER* (the mapping between **<NUMBER>** and the **TB_CONTACT** table):

```
NO -> NO
FIRST_NAME -> FIRST_NAME
LAST_NAME -> LAST_NAME
PHONE -> PHONE
```

In addition, we can see in **xmlimport.xml** that columns are treated as elements in the target XML file. Finally let us assume that our log file is */client/john/xmlimport.log*.

For importing into table **TB_CARD**, the elements under */ROOT/EMPLOYEE/TITLE* are imported. **TAG1** is mapped to column **ID**, **TAG2** is mapped to column **FNAME**, **TAG3** is mapped to column **LNAME** and **TAG4** is mapped to column **WORK**.

For Importing into table **TB_CONTACT**, the elements under */ROOT/EMPLOYEE/NUMBER* are imported. All elements under the <NUMBER> tag are imported and they are assumed a direct mapping to columns in table **TB_CONTACT**.

Note that xml tags are case-sensitive subsequently, **ROOT**, **EMPLOYEE**, **TITLE**, **TAG1**, **TAG2**, and **TAG3** in this example must be capitalized. The case-sensitivity of table names and table column names depends on DBMaker settings.

⊃ **To use XMLIMPORT with the above files:**

1. The file must be on the server, thus the specified full path must also be on the server. The file_path used in the argument is "/usr/john/xmlimport.xml".

2. The **object_str** can be used like this

```
'/ROOT/EMPLOYEE/TITLE(TAG1, TAG2, TAG3, TAG4)#INSERT INTO
TB_CARD (ID,FNAME,LNAME,WORK) VALUES
(?, ?, ?, ?);/ROOT/EMPLOYEE/NUMBER#tb_contact'
```

or

```
'/ROOT/EMPLOYEE/TITLE#INSERT INTO TB_CARD (ID,FNAME,LNAME,WORK) VALUES
(?, ?, ?, ?);/ROOT/EMPLOYEE/NUMBER#tb_contact'
```

or

```
'/ROOT/EMPLOYEE/TITLE(TAG1, TAG2, TAG3, TAG4)#CARD
(C1,C2,C3,C4);/ROOT/EMPLOYEE/NUMBER#contact'
```

3. The object string used can have several formats:

```
'/ROOT/EMPLOYEE/TITLE(TAG1, TAG2, TAG3, TAG4)#INSERT INTO
TB_CARD (ID,FNAME,LNAME,WORK) VALUES
(?, ?, ?, ?);/ROOT/EMPLOYEE/NUMBER#TB_CONTACT'
```

or, since there are four tags mapping four columns and the sequence of tags are the same as the columns:

```
'/ROOT/EMPLOYEE/TITLE#INSERT INTO TB_CARD (ID,FNAME,LNAME,WORK)
VALUES  (?, ?, ?, ?);/ROOT/EMPLOYEE/NUMBER#TB_CONTACT'
```

or,

```
'/ROOT/EMPLOYEE/TITLE#INSERT INTO TB_CARD VALUES
(?, ?, ?, ?);/ROOT/EMPLOYEE/NUMBER#TB_CONTACT'
```

or, since no further calculation of host variables is required:

```
'/ROOT/EMPLOYEE/TITLE(TAG1, TAG2, TAG3, TAG4)#TB_CARD(ID, FNAME,
LNAME, WORK);/ROOT/EMPLOYEE/NUMBER#TB_CONTACT'
```

4.   Since columns are treated as elements in the XML file, we will not set the option_flag here. If these columns were not treated as elements, the option_flag could be set.

```
option flag =: {[<attribute> [;<attribute>]...]}

<attribute> =:

{

column_as_attribute

}
```

5.   The log_path will be: "/client/john/xmlimport.log. This is where errors are recorded during the process of XMLIMPORT. "

6.   Call XMLIMPORT using one of possible forms of **obj_str**:

```
CALL XMLImport (

'/usr/john/xmlimport.xml',
```

```
'/ROOT/EMPLOYEE/TITLE(TAG1, TAG2, TAG3,
TAG4)#TB_CARD(ID,FNAME,LNAME,WORK);/ROOT/EMPLOYEE/NUMBER#tb_cont
act',

'',

'/client/john/xmlimport.log');
```

6   dmSQL Commands

The commands presented in this chapter require CASEMaker's dmSQLTool included with DBMaker.

6.1   CONNECT

The CONNECT command establishes a connection to a database. The user name and password are case-sensitive, while the database name is not. Any user with CONNECT or higher security privileges can execute the CONNECT command.

Before connecting to a database, the **dmconfig.ini** file on the computer must contain a database configuration section for the target database. The database configuration section should already exist if the database was created on the local computer. If the database was created on a remote computer, add the database configuration section.

Use the CONNECT command to connect to a single-user database. This starts the database and establishes a connection. Only one user may be connected to a single-user database.

Before connecting to a single-user database, specify the database directory. Use the **DB_DbDir** *keyword* to set the directory containing the database in the **dmconfig.ini** file.

Use the CONNECT command to connect to a client/server database while the database server is running. If the database server is not running, start it before trying to connect.

Before connecting to a client/server database, specify the IP address of the host computer running the DBMaker server and the port number of the database. Use the **DB_SvAdr** and **DB_PtNum** keywords to set the IP address and the port number in the **dmconfig.ini** file. Alternatively, substitute a host name in place of an IP address when using the **DB_SvAdr** keyword.

DBMaker will try to connect to a client/server database until the connection timeout period expires. The connection timeout period is specified by the **DB_CTimO** keyword in the **dmconfig.ini** file. The **DB_CTimO** keyword does not apply to single-user databases.

The user name and password are not optional with one exception; if the password is NULL omit it. You may also omit the user name and password from the CONNECT command using the **DB_UsrId** and **DB_PasWd** keywords in the **dmconfig.ini** file.

The **DB_UsrId** keyword specifies a default user name and the **DB_PasWd** keyword specifies a default password. You cannot specify one parameter on the command line and the other in the configuration file; DBMaker always takes the user name and password from the same location. DBMaker ignores the values specified by the **DB_UsrId** and **DB_PasWd** keywords if you provide a username and password with the CONNECT command.

database_name.........Name of the database being connected to

user_name ....Name of the user connecting to the database

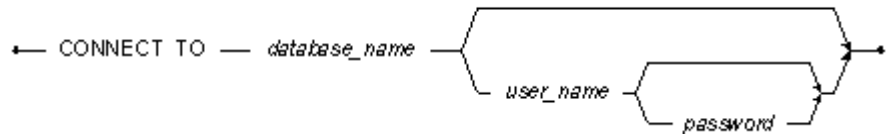*password* ....Current password of user *user_name*

Figure 6-1 CONNECT syntax

➲ **Value 1**

The **dmconfig.ini** file will provide a value for the **DB_DbDir** keyword in the **Tutor1** configuration section.

```
[TUTOR1]
DB_DbDir = C:\DBMAKER\DATABASE\TUTOR1
```

➲ **Example 1**

The following connects the user **jenny** with password **grala833** to the single-user **Tutor1** database.

```
dmSQL> CONNECT TO Tutor1 jenny grala833;
```

➲ **Value 2a**

The **dmconfig.ini** file will provide a value for the **DB_SvAdr** and **DB_PtNum** keywords in the **Tutor2** configuration section.

```
[TUTOR2]
DB_SvAdr = 192.72.116.137
```

```
DB_PtNum = 35400
```

➲ **Value 2b**

Alternatively use a host name for the **DB_SvAdr** keyword instead of an IP address.

```
[TUTOR2]
DB_SvAdr = mars.syscom.com.tw
DB_PtNum = 35400
```

➲ **Example 2**

The following connects the user **amanda** with password **grixa944** to the multi-user **Tutor2** database.

```
dmSQL> CONNECT TO Tutor2 amanda grixa944;
```

➲ **Value 3**

The **dmconfig.ini** file provides values for the **DB_SvAdr**, **DB_PtNum**, **DB_UsrId**, and **DB_PasWd** keywords in the **Tutor2** configuration section.

```
[TUTOR2]
DB_SvAdr = 192.72.116.137
DB_PtNum = 35400
DB_UsrId = vivian
DB_PasWd = shuka828
```

Alternatively, substitute a host name for the IP address for **DB_SvAdr**, the same as in Value **2b**.

➲ **Example 3**

The following connects the user **vivian** with password **shuka828** to the multi-user **Tutor2** database. The user name and password are not provided in the command since they are specified by the **DB_UsrId** and **DB_PasWd** keywords in the **dmconfig.ini** configuration section. If you provide a user name and password in the command, DBMaker ignores the values specified by the **DB_UsrId** and **DB_PasWd** keywords.

```
dmSQL> CONNECT TO Tutor2;
```

6.2 CREATE DATABASE

The CREATE DATABASE command creates a new database. To execute the CREATE DATABASE command, DBMaker must have write permission (e.g., from the operating system) on the directory where the database will be created. Any user can execute the CREATE DATABASE command.

DBMaker stores all configuration information for each database in **dmconfig.ini**. This file contains a database configuration section for each database you can connect to from the computer. The **dmconfig.ini** file is an ASCII text file, and can be edited with a text editor.

Each database configuration section is comprised of a section header followed by one or more keyword lines. The section header is the name of the database enclosed in square brackets. The keyword lines consist of a keyword and a corresponding value(s). If a keyword requires or supports multiple values, delimit individual values with either spaces or commas. Depending on their purpose, keywords may be used, at start time or connect time.

Key words in the **dmconfig.ini** file are not case-sensitive. Keyword values may be case-sensitive, depending on the keyword and the operating system the database is running on. When creating a database, DBMaker will examine the **dmconfig.ini** file for a database configuration section. If a database configuration section with the same name as the database exists, DBMaker uses the values specified in this section when it creates the database. If a database configuration section with the same name as the database does not exist, DBMaker uses default values when it creates the database and adds a new configuration section.

Choose a database name that is unique from all computers that will be connecting. Since, DBMaker stores configuration information for all local and remote databases in the **dmconfig.ini** file, using the same name for two databases will cause a conflict. You cannot change the database name once it has been created, unless you unload all data and recreate the database with a new name. Database names have a maximum length of 128 characters, and may contain letters, numbers, and the underscore character. Database names are not case-sensitive.

In the DBMaker physical storage model, files are physical units of storage that contain the data. Files are managed by the operating system, while data in the files is managed by the DBMS. DBMaker uses three types of files Data, BLOB, and Journal.

Data and BLOB files store user and system data. Although they have similar characteristics, DBMaker manages these two file types in different ways to improve performance. Data files store table and index data, while BLOB files store only binary large objects.

Journal files are special files that provide a real-time, historical record of all changes made to a database and the status of each change. This allows the database to undo changes made by a transaction that fails, or redo changes made successfully but not written to disk after a database crash. Journal files are used only by the database management system, and are not used to store user data.

In the DBMaker logical storage model, tablespaces are the logical storage structures used to partition information in a database into manageable areas. Each tablespace may contain several tables and indexes. Data in the tablespace is managed by the DBMS, but is physically stored in files. There are three types of tablespaces regular, autoextend, and system.

Regular tablespaces have a fixed size and contain one or more Data or BLOB files. They may be extended manually by enlarging existing files in the tablespace or adding new files to the tablespace. A regular tablespace may contain a maximum of 32,767 files, with a maximum cumulative file size of 8 TB. On UNIX platforms, regular tablespaces may be placed on raw devices.

NOTE *For more information on raw devices, see your UNIX system documentation.*

Autoextend tablespaces automatically increase in size to a maximum of 8 TB to hold additional data as required. They must contain one data file, and may contain one BLOB file. To add new files to an autoextend tablespace, first convert it to a regular tablespace. If an autoextend tablespace is created with only one Data file and no BLOB file, a BLOB file may be added later. Autoextend tablespaces do not support raw devices.

DBMaker generates system tablespaces, while a database is created. Each database has one system tablespace, which contains the system catalog tables used to store schema,

security, and status information. The system tablespace is created as an autoextend tablespace, unless created on a *UNIX raw device*. System tablespaces may be converted to regular tablespaces. System tablespaces are created with an initial data file size of 600 KB, and an initial BLOB file size of 20 KB.

DBMaker will create one system data file and one system BLOB file in the system tablespace, and create one user data file and one user BLOB file in the default user tablespace. DBMaker also creates at least one system Journal file to log database transactions.

The default names for the system files are DATABASE.SDB, DATABASE.SBB, and DATABASE.JNL, where DATABASE is the name of the database. To change the default names, use the **DB_DbFil**, **DB_BbFil**, and **DB_JnFil** keywords in the **dmconfig.ini** file. Use **DB_DbFil** to specify the name of the system data file, **DB_BbFil** to specify the name of the system BLOB file, and **DB_JnFil** to specify the name of the system Journal file. Specify a new name before creating a database or the default name will be used. The name of a system file may not be changed after creating the database.

The default user files names are DATABASE.DB and DATABASE.BB.DATABASE is the name of the database. To change the default names, use the **DB_UsrDb** and **DB_UsrBb** keywords in the **dmconfig.ini** file. Use **DB_UsrDb** to specify the name and size of the default user data file, and **DB_UsrBb** to specify the name and size of the default user BLOB file. When using these two keywords to specify new names for the default user files, also include the size of the file in Data pages or BLOB frames, separated from the filename by a space or comma. If the default name is not used for either of the default user files, specify a new name before creating the database.

DBMaker can use up to eight Journal files to log database transactions. To create multiple Journal files, add additional filenames after the **DB_JnFil** keyword, separated by spaces or commas. DBMaker automatically creates these Journal files when it creating the database. It is possible to add additional Journal files to a database after creating it by adding additional Journal filenames and restarting the database in new Journal mode.

To include a path with a filename, include the drive and full path on Windows systems. On UNIX systems, include either a full or a relative path. By default, the file

will be created in the directory specified by the **DB_DbDir** keyword in the **dmconfig.ini** file, or the application directory if the **DB_DbDir** keyword is not present. DBMaker system files may have filenames with a maximum length of 256 characters, and may contain any characters and symbols permitted by the operating system, except spaces.

The default sizes for the system files are 600 KB for the data file, 20 KB for the BLOB file, and 4,000 KB for the Journal file. To change the default file sizes, use the **DB_BfrSz** and **DB_JnlSz** keywords in the **dmconfig.ini** file.

The **DB_BfrSz** keyword specifies the size of frames in the system BLOB file, which also changes the size of the system BLOB file. Provide a value for **DB_BfrSz** when you create your database if you do not want to use the default, and it cannot be changed after creating the database.

The **DB_JnlSz** keyword specifies the size of the system Journal file in Journal blocks, which are the primary unit of storage in a Journal file. Journal blocks store a record of every transaction performed on the database. The size of each Journal block is determined by the **DB_PgSiz** in dmconfig.ini file. Each Journal block can store information on as many transactions as will fit into a block. To specify a size for a system Journal file, set the **DB_JnlSz** keyword to a value between 23 and 524,287 blocks. To calculate the actual size of the file in kilobytes, multiply this value by the value of the **DB_PgSiz** as specified in the dmconfig.ini file. If your database has multiple Journal files, DBMaker creates each Journal file with the size specified by **DB_JnlSz**. The default value for **DB_JnlSz** is *1,000* pages. The **DB_JnlSz** keyword may be changed at any time, but it will not take effect until the next time the database is started in New Journal Mode.

The default sizes for the default user files are *600* KB for the default user data file, and *20* KB for the default user BLOB file. To change the default file sizes, use the **DB_UsrDb** and **DB_UsrBb** keywords in the **dmconfig.ini** file.

The **DB_UsrDb** keyword specifies the size of the default user data file in data pages, which are the primary unit of storage. Data pages store table records, index keys, and any BLOB data small enough to fit onto the data page. Each data page can store as many table rows or index keys as will fit onto a page. The size of each data page is determined by the **DB_PgSiz** as specified in the dmconfig.ini file. To specify a size for

the default user data file, set the size parameter of the **DB_UsrDb** keyword to a value between 2 and 524,287 pages. To calculate the actual size of the file in kilobytes, multiply this value by the value of the keyword **DB_PgSiz** as specified in the **dmconfig.ini** file. The default value of **DB_UsrDb** is *150*.

The **DB_UsrBb** keyword specifies the size of the default user BLOB file in BLOB frames, which are the primary unit of storage in a BLOB file. BLOB frames store large binary data objects, graphics, audio and video, or large text, which does not fit onto a data page. Each BLOB frame can only store a single BLOB. The size of each BLOB frame is specified by the **DB_BfrSz** keyword, which can range from 8 KB to 256 KB. To specify a size for the default user BLOB file, set the size parameter of the **DB_UsrBb** keyword to a value between 2 and 524,287 frames. To calculate the actual size of the file in kilobytes, multiply this value by the value of **DB_BfrSz**. The default value for **DB_UsrBb** is *2*.

Security mode determines whether DBMaker uses security privileges to control access to the database. There are five levels of security privileges: CONNECT, RESOURCE, DBA, SYSDBA and SYSADM.

CONNECT security privilege permits a user to connect to the database, view the system tables, and access any database objects granted privileges on by the owner, a DBA, a SYSDBA or a SYSADM. New database objects cannot be created with the CONNECT security privilege. The CONNECT security privilege must be granted before being granted any other privilege.

RESOURCE security privilege permits users to create and drop tables, indexes, views, synonyms, and domains. A user can only drop tables, views, synonyms, and domains they created. In addition, a user can grant and revoke object privileges to other users on any database objects created by them. Users with RESOURCE security privilege also have all privileges of the CONNECT security privilege.

DBA security privilege permits a user to start, terminate, and back up databases, manage database resources, tablespaces and files, and access all tables, indexes, views, synonyms, and domains without having been granted privileges. Also grant, change, and revoke object privileges on any database object owned by any user. A DBA may not grant security privileges to new users or create new groups, but may add and

remove users from existing groups. Users with DBA security privilege also have all privileges of RESOURCE and CONNECT.

SYSDBA security privilege permits a user to grant and revoke security privileges to all users except users with the SYSADM and SYSDBA authority, create and drop groups, and add or remove users from groups. Also, change the password of all users except users with the SYSADM and SYSDBA authority. Users with SYSDBA security privilege also have all privileges of DBA, RESOURCE and CONNECT.

SYSADM security privilege permits a user to grant and revoke security privileges to all users, create and drop groups, and add or remove users from groups. Also, change the password of any user. There is only one user in each database with SYSADM security privileges. DBMaker automatically creates this user when creating the database, and assigns the user name SYSADM. A SYSADM may not grant SYSADM security privileges to any other users. The SYSADM also has all privileges of SYSDBA, DBA, RESOURCE, and CONNECT.

Set the security mode before creating a database. After creating a database, the security mode cannot change unless the database is unloaded and recreated. Use the **DB_Secur** keyword in the **dmconfig.ini** file to set the security mode. If the **DB_Secur** keyword is not used when creating a database, the security mode is *ON* by default.

When security mode is ON, only users with appropriate security privileges can connect to the database. A user name and password are required to connect to a database. DBMaker maintains a list of authorized users and their security privileges for the database, and checks this list to determine the specific commands each user can execute.

When security mode is OFF, any user can connect to a database with any user name. Passwords are not required to connect to a database, and DBMaker ignores passwords. DBMaker does not maintain a list of users or security privileges for the database, and any user can execute any command.

When executing the CREATE DATABASE command, DBMaker creates a new database, starts the database, and connects you as the SYSADM. DBMaker does not assign a password to the SYSADM user when it is created. Change the SYSADM password immediately after creating the database to prevent unauthorized access to

the database. DBMaker starts a newly created database in single-user mode to prohibit other users from logging on to the database before you can change the SYSADM password. To put the new password into effect and allow other users to connect, shut down the database and restart it in single or multi-user mode.

DBMaker starts all databases in single-user mode by default. To start a database in multi-user mode, use the **DB_SvAdr** and **DB_PtNum** keywords in the client-side **dmconfig.ini** file and the **DB_PtNum** keyword in the server-side **dmconfig.ini** file.

The **DB_SvAdr** keyword specifies the IP address or host name of the computer the DBMaker server is running on. This keyword is required only on the client side; it is optional on the server side. To specify an IP address or host name, set the **DB_SvAdr** keyword to any valid IP address or host name. Use a hostname; also ensure that the Domain Name Service (DNS) is properly set up on your computer.

The **DB_PtNum** keyword specifies the port number the DBMaker server is bound to. This keyword is required on both the client and server sides. To specify a port number, set the **DB_PtNum** keyword to a value between 1,025 and 65,535. If not specifying a port number, DBMaker uses port number *23,000* by default.

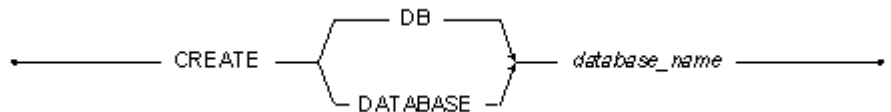*database_name*.........Name of the new database to create

Figure 6-2 CREATE DATABASE syntax

⮌ **Example 1**

The following creates a new database named **Accounts** with the default settings for all parameters. A database configuration section for this database does not exist in the **dmconfig.ini** file when this command is executed. This creates a single-user database in the application directory using the default file names **ACCOUNTS.SDB**, **ACCOUNTS.SBB**, **ACCOUNTS.DB**, **ACCOUNTS.BB** and **ACCOUNTS.JNL** and the default file sizes of 600 KB for the .**SDB** and .**DB** files, 20 KB for the .**SBB** and .**BB** files, and 4,000 KB for the .**JNL** file. To start this database in multi-user

mode, add the **DB_SvAdr** and **DB_PtNum** keywords to the Accounts database configuration section in the **dmconfig.ini** file after creating the database.

```
dmSQL> CREATE DATABASE Accounts;
```

➲ **Example 2**

The following creates a new database named **Accounts** using the settings shown in the **dmconfig.ini** section below.

```
dmSQL> CREATE DATABASE Accounts;
```

➲ **Excerpt**

This database configuration section exists in the **dmconfig.ini** file when the command is executed. This creates a single-user database with security in the *C:\DATABASE\ACCOUNTS* directory, using file names **ACCOUNTS.SDB** for the system data file, **ACCOUNTS.SBB** for the system BLOB file, **ACNTDATA.DB** for the default-user data file, **ACNTBLOB.BB** for the default user BLOB *file*, and **ACNTHIST.JN1**, **ACNTHIST.JN2**, and **ACNTHIST.JN3** for the three Journal files. The file sizes are 600 KB for the system data file, 20 KB for the system BLOB file, 1,000 KB for the default user data file, 8,000 KB for the default user BLOB file, and 2,000 KB for each of the three Journal files. To start this database in multi-user mode, add the **DB_SvAdr** and **DB_PtNum** keywords to the Accounts database configuration section in the **dmconfig.ini** file after creating the database.

```
[ACCOUNTS]
DB_DbDir = C:\DATABASE\ACCOUNTS
DB_DbFil = ACCOUNTS.SDB
DB_BbFil = ACCOUNTS.SBB
DB_UsrDb = ACNTDATA.DB 250
DB_UsrBb = ACNTBLOB.BB 250
DB_BfrSz = 32
DB_JnFil = ACNTHIST.JN1, ACNTHIST.JN2, ACNTHIST.JN3
DB_JnlSz = 500
```

6.3   DEF TABLE

The dmSQL command DEF TABLE is used to display schema information for a
specified table. This command should not be used on system tables.

Figure 6-3 DEF TABLE Command

 ➲ **Example 1a**

Create a table:

```
dmSQL> CREATE TABLE tb_tmp(c00_serial SERIAL, c01_int INTEGER, c02_char
CHAR(20));
```

 ➲ **Example 1b**

Execute the command:

```
dmSQL> DEF TABLE tb_tmp;
```

 ➲ **Result**

```
dmSQL> DEF TABLE tb_tmp;
dmSQL> create table SYSADM.TB_TMP (
 C00_SERIAL  SERIAL(1),
 C01_INT  INTEGER default null ,
 C02_CHAR  CHAR(20) default null )
 in DEFTABLESPACE  lock mode row  fillfactor 100;
```

6.4   DEF VIEW

The dmSQL command DEF VIEW is used to display the construction of definitions. This command should not be used on system views.

```
──────────────── DEF VIEW ──────────────── view_name ────────
```

Figure 6-4 DEF VIEW Command

⮞ **Example 1a**

Create a view:
```
dmSQL> CREATE VIEW view_tmp AS SELECT c00_serial, c01_int FROM tb_tmp;
```

⮞ **Example 1b**

Execute the command:
```
dmSQL> DEF VIEW view_tmp;
```

⮞ **Result**
```
dmSQL> DEF VIEW view_tmp;
dmSQL> CREATE VIEW SYSADM.VIEW_TMP AS SELECT c00_serial, c01_int FROM
SYSADM.TB_TMP;
```
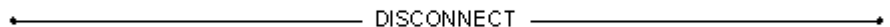
6.5 DISCONNECT

The DISCONNECT command closes an active database connection. Any user with CONNECT or higher security privileges can execute the command.

AUTOCOMMIT mode controls when DBMaker will commit a transaction. When AUTOCOMMIT mode is on, each command is treated as a separate transaction. DBMaker automatically commits each command executed if it completes successfully, or rolls it back if an error occurs during execution. When AUTOCOMMIT mode is off, all commands between successive COMMIT WORK commands form a single transaction.

Executing the COMMIT WORK command commits any changes made in the transaction, and executing the ROLLBACK WORK command rolls back all changes. When disconnecting from a database and AUTOCOMMIT mode is off, the active transaction is aborted. Any changes made by the transaction are not recorded in the database.

When disconnecting from a multi-user database, the database remains active and accessible to other users. When disconnecting from a single-user database running on *UNIX* the database shuts down. When disconnecting from a multiple-connection database running on Windows, the database shuts down only if you are the last connected user.

◆━━━━━━━━━━━ DISCONNECT ━━━━━━━━━━━◆

Figure 6-5 DISCONNECT syntax

➲ **Example**

The following disconnects an active database.
```
dmSQL> DISCONNECT;
```

6.6   EXPORT

The Export command facitates the extraction of data from database tables and inserts the data into text files. There are two configurations used. The export command interface is used for specifying command options. The description file is used for specifying the export file format.

EXPORT COMMAND INTERFACE

The Export command syntax is as follows:

<data_file> ....This is the target file into which you will insert the data. It should be in full path. If you do not specify data_file, the export file name will be *<table_name>_out.txt*.

TABLE ....Please specify the table you want to export.

[DESCRIPTION <description_file>] … This is the description file for the data format in the resulting data file. In the description file, users will specify some rules for the resulting data file. Refer to the *DESCRIPTION FILE FORMAT* section for more information. If the description file is not specified, the description file name is *<table_name>_out.dsc*. If this file does not exist, DBMaker uses the default output format.

The default file format is the variable format, meaning:

* TAB is the column delimiter

* New line characters are row terminators

* No quotation marks

* All columns in the source table are exported in the order as they are in the table

[LOG <log_file>] .....This file logs the errors that occur during the course of unloading data. If this option is not specified, the default log file name, *export.log*, will be used.

[STOP_ON_ERROR]… Specifies that you want want to stop unloading data if an error occurs. If this option is not specified, the unloading of data will continue even if an error has occured.

```
EXPORT
[INTO <data_file>]
TABLE [<owner_name>.]<table_name>
[DESCRIPTION <description_file>]
[LOG <log_file>]
[STOP_ON_ERROR]
```

DESCRIPTION FILE

You can specify the format of the description file for formatting the unloading result. Two types of format can be used, fixed format and variable format.

FIXED FORMAT DESCRIPTION FILE

When the fixed format description file is used, users want each column of the export result to be aligned vertically. The separators used for alignment will be space characters.

FORMAT = FIXED …This specifies the description file format for fixed length data files.

[LOB_FORMAT= INTERNAL | EXTERNAL]…This specifies that when exporting columns of large object types (such as blob, clob, nclob, nblob and other files) external files will be generated. For each column of large object type in each row, an external file will be generated. If this option is not specified, the content of data will be embedded in a datafile.

When naming external files it's important to keep the following in mind: ***blobtempdir<m>\blbtmpf<n>.<tmp | txt>***.

m specifies the minimum un-used number counted from 1 in the directory.

For example, if there are already directories named blobtempdir1, blobtempdir2 and blobtempdir3, the newly created directory for containing external files will be blobtempdir4.

n specifies the minimum un-used number counted from 1 in the directory.

Whether the file extension name is **tmp** or **txt** depends on whether the exported column is BLOB type, FILE type or CLOB type. If the column type is BLOB or FILE, the file extension name will be **tmp**. Otherwise, the column type is **txt**.

server_column_name…This lists the names of the source table columns that are going to be exported from the database. If there are spaces in table name, use double quotes to enclose the column names.

column_position........ Specifies the column byte position in data file.
server_columnname and *column_position* are separated by space character(s).
column_position is specified by two numbers that are separated by (:).For example a 1:40 means the data loader should look for data from 1st byte to 40th byte in data file. We will use space characters to align the data field vertically. If the data in the source table exceeds the field length, the data output will be truncated.

```
FORMAT=FIXED
[LOB_FORMAT=INTERNAL | EXTERNAL]
<server_column_name> <column_position>
```
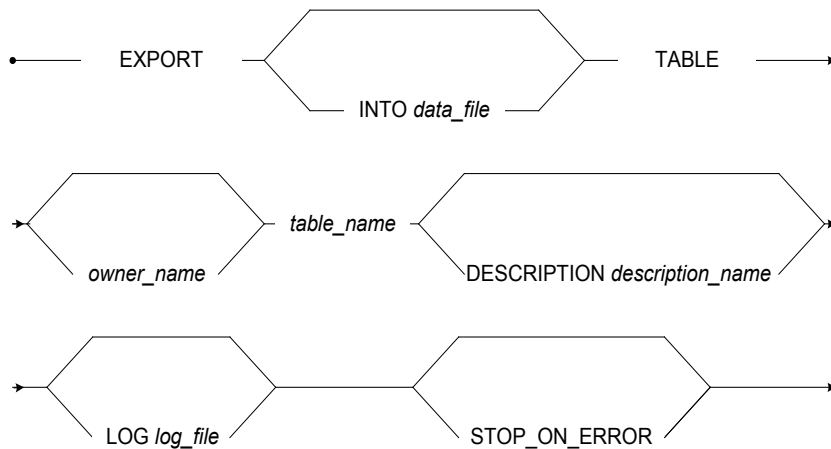

Figure 6-6 EXPORT syntax

VARIABLE FORMAT DESCRIPTION FILE

When variable format description file is chosen, the fields of resulting data output will be separated by a user specified delimiter.

FORMAT=VARIABLE… specifies the resulting output file is in variable format.

[COLUMN_DELIMITER=<delimiter>]… This specifies a character that separates each column in datafile. The character should be single quoted. For example, to indicate that a **SPACE** is used as column delimiter, use ' '. Aside from normal characters, take the following escape sequences that represent special characters.

CHARACTER	ESCAPE SEQUENCE REPRESENTATION
TAB	\t
NEW LINE	\n

Table 6-1 Character and Escape Sequence

For example, if the delimiter is a TAB, users will use '\t' in *<delimiter>*. If the column delimiter is not specified, we will use TAB (\t) as the column delimiter. Use discretion when choosing a delimiter.

If the number of column delimiters is fewer than the number of target table columns specified by users, NULL will be used for the insert value.

[ROW_TERMINATOR=<row_terminator>]… This string denotes the end of a row.

[QUOTATION=SINGLE_QUOTE | DOUBLE_QUOTE] …This indicates that the output data will be quoted by either single quotes or double quotes. If there is quotation mark in the data, the output will show two consecutive quotation marks.

[LOB_FORMAT=INTERNAL | EXTERNAL]… This specifies that when exporting columns of large object types, such as blob, clob, nclob, nblob and other large files, external files will be generated. For each column of large object type in each row, an external file will be generated. If this option is not specified, the content of the data will be embedded in a datafile.

When naming external files it's important to keep the following in mind: **blobtempdir<m>\blbtmpf<n>.<tmp | txt>**.

m specifies the minimum un-used number counted from 1 in the directory.

For example, if there are already directories named blobtempdir1, blobtempdir2 and blobtempdir3, the newly created directory for containing external files will be blobtempdir4.

n specifies the minimum un-used number counted from 1 in the directory.

Whether the file extension name is **tmp** or **txt** depends on whether the exported column is BLOB type, FILE type or CLOB type. If the column type is BLOB or FILE, the file extension name will be **tmp**. Otherwise, the column type is **txt**.

server_column_name… This variable lists the names of columns of a server table which are to be exported. The order of these names represents the order of column export. If there is no such list, all the columns in source table will be export in the same order as that of table columns.

```
FORMAT=VARIABLE
[COLUMN_DELIMITER=<delimiter>]
[ROW_TERMINATOR=<row_terminator>]
[QUOTATION=SINGLE_QUOTE | DOUBLE_QUOTE]
[LOB_FORMAT=INTERNAL | EXTERNAL]
[<server_column_name>]
```

IMPORT/EXPORT DATA RULES

The following table outlines the rules that must be applied when attempting to import or export data to or from a file.

DATA TYPE	IMPORT/EXPORT FORMAT	EXAMPLE
BINARY	Use HEX format	To import the binary number "0x004D2", use 004D2 in datafile
CHAR	Characters are used exclusively	To import the word "inception", use inception in the datafile
NCHAR	Three formats can be used: auto, hex format or character. Use description flag IMPORT_NCHAR_FORMAT to indicate user's option in	To import the word "word", use 77006f0072006400 or word in datafile

DATA TYPE	IMPORT/EXPORT FORMAT	EXAMPLE
	description file. NCHAR_AUTO option is trying to import data as HEX format first. If fails, then try import data as character. NCHAR_HEX format option is importing data as HEX format. NCHAR_CHAR format option is importing data as characters.	
VARCHAR	See CHAR data type	
NVARCHAR	See NCHAR data type	
DATE	The format YYYY/MM/DD will be used for exporting	To import the date "2003/07/25", use 2003/07/25 in the datafile
TIME	Export and import will use the format HH:MM:SS	To import the time "14:30:25", use 14:30:25 in the datafile
TIMESTAMP	The combination of DATE format and TIME format forms the format of TIMESTAMP	to import the timestamp "2003/07/25 14:30:25", use 2003/07/25 14:30:25 in data file
DECIMAL	Use numeric data representation	To import the number "36.82", use 36.82 in data file
DOUBLE	Use numeric data as described in DECIMAL or scientific notation of numbers	To import the number "13e+12", use 13e+12 in datafile
FLOAT	See DOUBLE	
INTEGER	Use integer data	To import the integer "576", use 576 in datafile
LONG VARBINARY	Two formats can be used: embedded or external file format. For embedded format, HEX	(1) embedded format: The format used will be the same as BINARY.

DATA TYPE	IMPORT/EXPORT FORMAT	EXAMPLE
	characters are used. For external file format, the URL is provided. Use description flag LOB_FORMAT to indicate your option. For details see description file specifications.	(2) external file format: For example, if users want to import a binary file whose full path is "c:\My Document\GRAPH.GIF". The URL provided will be c:\My Document\GRAPH.GIF
LONG VARCHAR	Similar to the case for LONG VARBINARY, two formats can be used. The input data will be in ASCII string instead of HEX string.	(1) embedded format: Same as CHAR format. (2) external file format: Same as LONG VARBINARY.
FILE	For FILE type, import/export will adopt the same rule for LONG VARBINARY.	
OID	Same rule as INTEGER	
SERIAL	Same rule as INTEGER	
SMALLINT	Same rule as INTEGER	
NULL DATA	For variable format, NULL data is recognized by the fact that there's nothing between two consecutive delimiters. For fixed format, NULL data are recognized by the fact that there are only space characters between columns.	

Table 6-2 Import/Export Data Rules

6.7   IMPORT

The Import command is used for extracting data from a text file and then inserting the data into database tables. The import command interface is used for specifying command options. The description file is used for specifying the import file format.

IMPORT COMMAND INTERFACE

The Import Command Interface provides you with several options for importing data. Options include controling the stoppage criteria for data loading, the logging of errors and the data encoding of source data files. The format, of source data files, is described in the description file.

[<owner_name>.]<table_name> … This identifies the table to be loaded from the datafile. If you do not specify the *<owner_name>*, the current connection user will be assigned as the owner.

[FROM <data_file>]… This is the actual file that contains data to be loaded. If you do not specify *data_file*, the datafile name will be *<table_name>_in.txt*. For example, if the import table name is **t1** and datafile name is not specified in command, the datafile name will be **t1_in.txt**.

[DESCRIPTION <description_file>]… This is the description file for describing the data format in the datafile. If this option is not specified, the description file name will be assigned as *<table_name>_in.dsc*. For example, if the import table name is **t1** and description file is not specified, the description file name will be assigned as **t1_in.dsc**. If this file is not found, a default description file format will be used, variable description file format.

[LOG <log_file>]….. This identifies the log file, which logs any errors during the course of data loading. It will show the content of the record, which triggers the error as well as the corresponding error message. If you do not specify this option, the default log name will be *import.log*.

[STOP_ON_ERROR]… The loading of data will stop if an error occurs during the import process if this variable is set. If it is not specified, the loading will continue even when an error occurs.

```
IMPORT [<owner_name>.]<table_name>
[FROM <data_file>]
[DESCRIPTION <description_file>]
[LOG <log_file>]
[STOP_ON_ERROR]
```
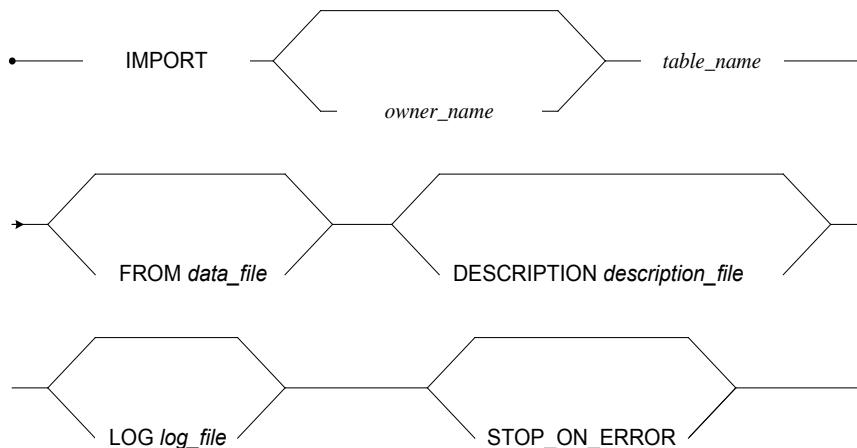

Figure 6-7 IMPORT syntax

DESCRIPTION FILE

Two types of description file are used. One is fixed format and the other is variable format. Parse errors in the description file will be shown as clearly as possible. You will know why the error has happened by checking the error message. The error message will display the problem that occurred when parsing a specific word.

FIXED FORMAT DESCRIPTION FILE

FORMAT=FIXED…When the format is set to fixed this means the description file describes the format for fixed length datafiles.

[START_WITH_ROW=<row_number>]… You can specify from which record you want to start loading data. The default number is 1, if you do not specify this option. If *START_WITH_ROW* is greater than total rows of data in datafile, no data will be loaded. The *row_number* is must be a positive number.

[NUMBER_OF_ROWS_FOR_EACH_TRANSACTION=<number>] … This lets you specify the interval of the rows of records loaded between each commit-transaction. If this option is not specified, DBMaker will commit transaction for every 5 rows. If the variable is set at **-1**, there will be no commit. In this case you must commit transaction manually if you want the load to be effective. If the variable is set at **0**, the entire import is seen as a single transaction. The system will then issue a commit after the loading is finished.

The number of rows committed will still count a record even if an error occurs when loading the record.

For example, you set NUMBER_OF_ROWS_FOR_EACH_TRANSACTION=10, and an error occurs when the 4th record is loaded. The 1st through 3rd records and 5th through 10th records will still be committed and the 1st through 10th records are still seen as one transaction unit. Of course, when STOP_ON_ERROR is specified, the 5th record through the 10th record are not committed; only the 1st through 3rd records are committed.

This option is valid only when auto-commit is off.

[LOB_FORMAT=INTERNAL | EXTERNAL] … If clob/blob format is internal, the text in data file is seen as the data that is going to be imported. Otherwise, the text is seen as a URL to external files that are going to be imported.

server_column_name … This lists the names of the target table columns that are going to be imported from a datafile. If there are spaces or equal signs in the table column name, use double quotes to enclose it.

column_position …… This is the column byte position in datafiles. *server_column_name* and *column_position* are separated by space characters. *column_position* is specified by two numbers that are separated by (:). For example, a *1:40* means the data loader should look for data from the 1st byte through the 40th byte in a datafile. Use space characters to align the data field vertically. If the data in the source table exceeds the

field length, the rest of row data will be truncated. Each line is terminated by either new line or a carriage return and a new line, depending on whether the loader is a Windows platform. If a line is smaller than the maximum position, spaces will be padded to fill the hole. If a line is longer than the maximum position, the rest of the line is ignored.

```
FORMAT=FIXED
[START_WITH_ROW=<row_number>]
[NUMBER_OF_ROWS_FOR_EACH_TRANSACTION=<number>]
[LOB_FORMAT=INTERNAL | EXTERNAL]
<server_column_name> <column_position>
```

NOTE *The fields, server_column_name, and column_position are separated by space characters.*

⮕ **An example for importing a file with fix format description file is as follows:**

The datafile exists as follows:

```
Davolio Nancy ......      Sales Representative      Ms.
Fuller Andrew ......      Vice President, Sales     Dr.
Leverling Janet ....      Sales Representative      Ms.
Peacock Margaret ...      Sales Representative      Mrs.
Buchanan Steven ....      Sales Manager             Mr.
Suyama Michael .....      Sales Representative      Mr.
King Robert ............  Sales Representative      Mr.
```

The description file for this datafile may look like this:

```
START_WITH_ROW=1
NUMBER_OF_ROWS_FOR_EACH_TRANSACTION=5
Name 1:20
Position 20:45
Gender 50:54
```

VARIABLE FORMAT DESCRIPTION FILE

```
FORMAT=VARIABLE
[START_WITH_ROW=<row_number>]
[NUMBER_OF_ROWS_FOR_EACH_TRANSACTION=<number>]
[{COLUMN_DELIMITER=<delimiter>}]
[ROW_TERMINATOR=<row_terminator>]
[QUOTATION=SINGLE_QUOTE | DOUBLE_QUOTE]
[ESCAPE_CHAR=YES|NO]
```

```
[LOB_FORMAT=INTERNAL | EXTERNAL]
[<server_column_name> <column_number>]
```

FORMAT=VARIABLE… This means this file contains the format for variable length description files.

[START_WITH_ROW=<row_number>]… You can specify from which record you want to start loading data. The default number is *1*, if you do not specify this option. If *START_WITH_ROW* is greater than total rows of data in datafile, no data will be loaded. The *row_number* is must be a positive number.

[NUMBER_OF_ROWS_FOR_EACH_TRANSACTION=<number>] … This lets you specify the interval of the rows of records loaded between each commit-transaction. If this option is not specified, DBMaker will commit transaction for every 5 rows. If the variable is set at **-1**, there will be no commit. In this case you must commit transaction manually if you want the load to be effective. If the variable is set at **0**, the entire import is seen as a single transaction. The system will then issue a commit after the loading is finished.

The number of rows committed will still count a record even if an error occurs when loading the record.

For example, you set NUMBER_OF_ROWS_FOR_EACH_TRANSACTION=10, and an error occurs when the 4th record is loaded. The 1st to 3rd records and 5th to 10th records will still be committed and the 1st to 10th records still seen as one transaction unit. Of course, when STOP_ON_ERROR is specified, the 5th record to 10th record won't be committed at all only the 1st to 3rd records will be committed.

This option is valid only when auto-commit is off.

[COLUMN_DELIMITER=<delimiter>]… This specifies a character that separates each column in datafile. The character should be single quoted. For example, to indicate that a **SPACE** is used as column delimiter, use ' '. Aside from normal characters, take the following escape sequences that represent special characters.

CHARACTER	ESCAPE SEQUENCE REPRESENTATION
TAB	\t

NEW LINE	\n

Table 6-3 Character and Escape Sequence

For example, if the delimiter is a TAB, users will use '\t' in *<delimiter>*. If the column delimiter is not specified, we will use TAB (\t) as the column delimiter. Use discretion when choosing a delimiter.

If the number of column delimiters is fewer than the number of target table columns specified by users, NULL will be used for the insert value.

[ROW_TERMINATOR=<row_terminator>] … This is a string that denotes the end of a row. The *row_terminator* should be double-quoted. The escape sequence rule for column delimiter applies to row terminator. In addition to that, the carriage-return also can be the escape sequence:

CHARACTER	ESCAPE SEQUENCE REPRESENTATION
CARRIAGE RETURN	\r

Table 6-4 Character and Escape Sequence

For example, if a carriage return and a new line character form a row terminator, the *<row_terminator>* should be "\r\n". If no row terminator is specified, a new line character ('\n') will be used as row terminator. The number of characters in row terminator should not be greater than 2.

Note that, no column delimiter should be in *row_terminator*.

[QUOTATION=SINGLE_QUOTE | DOUBLE_QUOTE] … This indicates whether the alphabetic data in one field of a data source file is quoted. If SINGLE_QUOTE is specified, the data enclosed by single quotes is seen as one column of data. If DOUBLE_QUOTE is specified, the data enclosed by double quotes is seen as one column of data.

[ESCAPE_CHAR=YES | NO] … This indicates whether an escape character (\) is used or not. The default is YES. If the escape character is used, the column delimiter character after escape character is seen as real data. For example, if we specify that a TAB be used as the column delimiter, and *ESCAPE_CHAR* is YES, a \TAB data is

seen as TAB in data instead of column delimiter. For row terminator, this escape character means the line continues, and the \n is seen as real data. This rule also applies to the quotation mark.

[LOB_FORMAT=INTERNAL | EXTERNAL] … If clob/blob format is internal, the text in the datafile is seen as the data that is going to be imported. Otherwise, the text is seen as a URL to external files that are going to be imported.

server_column_name … This lists the names of the target table columns that are going to be imported from a datafile. If there are spaces or equal signs in the table column name, use double quotes to enclose it.

column_number ......This is the cardinal number of each field in data file. *server_column_name* and *column_number* are separated by space characters.

NOTE *Note that if* server_column_name *and* column_number *are not specified, all columns in datafile will be imported into target table columns in the same order as datafile columns. That is to say, the 1st column in datafile will be imported as 1st column in the table, and the 2nd column in datafile will be imported as the 2nd column in table, etc. If the number of columns in datafile is greater than that of the target table, the remaining columns in datafile will be ignored. If, on the other hand, the number of columns in datafile is smaller than that of the target table, the remaining columns in target table will be inserted with NULL.*

DEFAULT VARAIBLE FORMAT DESCRIPTION FILE

It's optional that users specify the description file for their datafile format. If users do not specify the description file, a default description format is assumed. The default format means the following description file is used (On Win32 platform, the ROW_DELIMITER="\r\n"):

```
START_WITH_ROW=1
NUMBER_OF_ROWS_FOR_EACH_TRANSACTION=5
COLUMN_DELIMITER="\t"
ROW_TERMINATOR="\n"
```

➲ **An example for importing a file with variable format description file is as follows:**

A datafile exists:
```
Davolio Nancy,Sales Representative,Ms.
```

```
Fuller Andrew,"Vice President, Sales",Dr.
Leverling Janet,Sales Representative,Ms.
Peacock Margaret,Sales Representative,Mrs.
Buchanan Steven,Sales Manager,Mr.
Suyama Michael,Sales Representative,Mr.
King ..............Robert,Sales Representative,Mr.
```

The description file for this data file may look like this:

```
START_WITH_ROW=1
NUMBER_OF_ROWS_FOR_EACH_TRANSACTION=5
COLUMN_DELIMITER=","
ROW_TERMINATOR="\n"
DOUBLE_QUOTE
Name 1
Position 2
Gender 3
```

IMPORT/EXPORT DATA RULES

The following table outlines the rules that must be applied when attempting to import or export data from or to a file.

DATA TYPE	IMPORT/EXPORT FORMAT	EXAMPLE
BINARY	Use HEX format	To import the binary number "0x004D2", use 004D2 in data file
CHAR	Characters are used exclusively	To import the word "inception", use inception in the data file
NCHAR	Three formats can be used: auto, hex format or character. Use description flag IMPORT_NCHAR_FORMAT to indicate user's option in description file. NCHAR_AUTO option is trying to import data as HEX format first. If fails, then try import data as	To import the word "word", use 77006f0072006400 or word in data file

DATA TYPE	IMPORT/EXPORT FORMAT	EXAMPLE
	character. NCHAR_HEX format option is importing data as HEX format. NCHAR_CHAR format option is importing data as characters.	
VARCHAR	See CHAR data type	
NVARCHAR	See NCHAR data type	
DATE	The format YYYY/MM/DD will be used for exporting	To import the date "2003/07/25", use 2003/07/25 in the data file
TIME	Export and import will use the format HH:MM:SS	To import the time "14:30:25", use 14:30:25 in the data file
TIMESTAMP	The combination of DATE format and TIME format forms the format of TIMESTAMP	to import the timestamp "2003/07/25 14:30:25", use 2003/07/25 14:30:25 in data file
DECIMAL	Use numeric data representation	To import the number "36.82", use 36.82 in data file
DOUBLE	Use numeric data as described in DECIMAL or scientific notation of numbers	To import the number "13e+12", use 13e+12 in data file
FLOAT	See DOUBLE	
INTEGER	Use integer data	To import the integer "576", use 576 in data file
LONG VARBINARY	Two formats can be used: embedded or external file format. For embedded format, HEX characters are used. For external file format, the URL is provided.	(1) embedded format: The format used will be the same as BINARY. (2) external file format: For example, if users

DATA TYPE	IMPORT/EXPORT FORMAT	EXAMPLE
	Use description flag LOB_FORMAT to indicate your option. For details see description file specifications.	want to import a binary file whose full path is "c:\My Document\GRAPH.GIF". The URL provided will be c:\My Document\GRAPH.GIF
LONG VARCHAR	Similar to the case for LONG VARBINARY, two formats can be used. The input data will be in ASCII string instead of HEX string.	(1) embedded format: Same as CHAR format. (2) external file format: Same as LONG VARBINARY.
FILE	For FILE type, import/export will adopt the same rule for LONG VARBINARY.	
OID	Same rule as INTEGER	
SERIAL	Same rule as INTEGER	
SMALLINT	Same rule as INTEGER	
NULL DATA	For variable format, NULL data is recognized by the fact that there's nothing between two consecutive delimiters.

For fixed format, NULL data is recognized by the fact that there are all space characters between columns. | |

Table 6-5 Import/Export Data Rules

6.8   LOAD

The Load command is a tool provided by dmSQL, it is used to transfer a database object, already 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. Only load the file that is unloaded in the same option. For example, load a database from the text file that is unloaded with database option.

When loading a text file, set the number of commands to automatically commit the transaction. The default number is 1000. The size of *n* will affect whether the transaction succeeds or not and the speed of loading. The Journal will fill easily with a large *n* value and could cause the transaction to fail. A small *n* value will increase the commit times and slow down the speed of loading.

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.
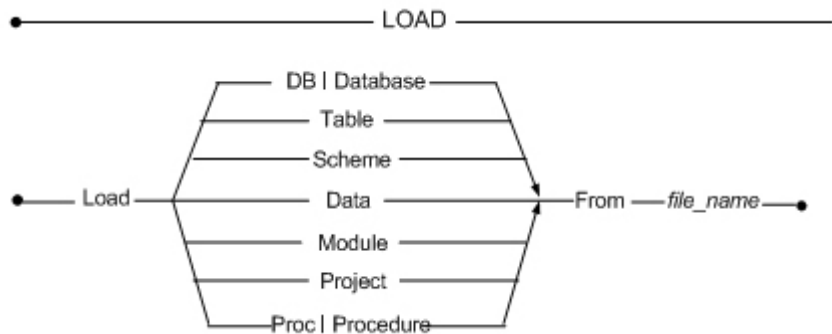

Figure 6-8 LOAD syntax

LOAD DB [DATABASE]

Use the command to transfer the contents of a database to a new database. First, unload the database to transfer to an external text file, and then 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 user with DBA, SYSDBA or SYSADM security priviledge 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, and, 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 utility will work in Journal mode if the loaddb is set in safe mode. The load utility will rollback to the last committed command if an error occurs during loading, the error messages will return to screen, and write to the log file of the load utility.

When using the set loaddb 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 loaddb 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 loaddb 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 loaddb is set to use the fast option, then the following error message occurs, "ERROR(30017), [DBMaker] errors occurred on no-Journal mode, shut down database". The default option is "set loaddb safe".

⊃ **Example**

The following set option for **loaddb** has been added to versions above DBMaker 3.6.

```
SET LOADDB [SAFE | FAST]
```

LOAD TABLE

The 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

The 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.

If *loaddata 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 near or at

ERROR(6015)........incomplete SQL statement input

The error 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 the load data stop or stop on error is set, the whole load command will rollback if errors occur. The default value for this option is set *loaddata skip* [*error*]. All the error messages occurred during the loading of data will be written into the log file.

➲ **Example**

DBMaker 3.6 and later versions support the following options.

```
SET LOADDATA SKIP [ERROR] | STOP [ON ERROR]
```

LOAD MODULE

The option allows a user to load a module from an external text file.

LOAD PROJECT

The option allows a user to load a project from an external text file.

LOAD PROC [PROCEDURE]

The 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 **1000** commands using the default setting.

```
dmSQL> LOAD DATA FROM datafile;
```

6.9　SET DUMP PLAN

A dump plan consists of several ON blocks. The query optimizer divides and optimizes a query into several logical ON blocks. Simple and joined queries usually only generate one ON block, where as a complex query like a sub-query may generate more than one ON block which includes a main-block and sub-blocks.

The optimizer will find the best execution method based on the cost for each ON block. It will divide each ON block into several PL blocks, and each PL block will represent an operation like a scan, join, etc.

Set dump plan on ..... turns the dump plan on, accepts queries and executes commands

Set dump plan off ..... turns the dump plan off, this is the default

Set dump plan only ... turns the dump plan on, accepts queries, but doesn't execute

　　　　　　　　　　　commands

Figure 6-9 SET DUMP PLAN Syntax

● **Example**

```
dmSQL> SET DUMP PLAN ON;
dmSQL> SELECT * FROM tb_tmp ORDER BY c01_int;
dmSQL> SET DUMP PLAN OFF;
```

6.10 START DATABASE

The START DATABASE command starts a database to allow users to connect. This command is normally only used with client/server databases. Only a user with DBA, SYSDBA or SYSADM security privilege may execute the command.

To start a database without specifying a user-name and password in the START DATABASE command, use the **DB_UsrId** and **DB_PasWd** keywords in the **dmconfig.ini** file.

The password is in plain text and can be seen by anyone with the read permission for the **dmconfig.ini** file. This keyword is included for convenience only, and may pose a security risk to the database Use it on an unsecured computer.

database_name ....Name of the database to start

user_name ....Name of the user starting the database

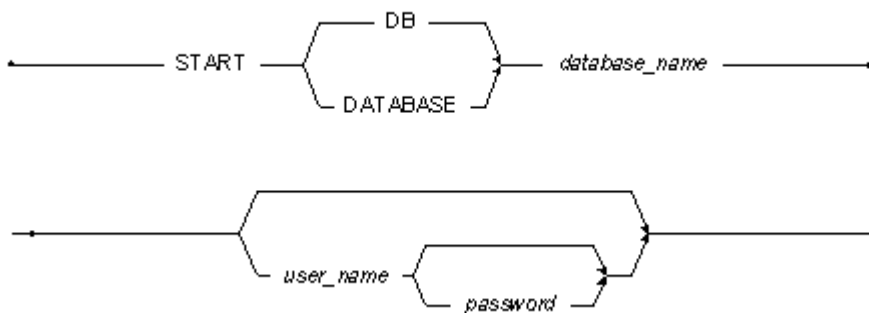*password* ....Current password of *user_name*
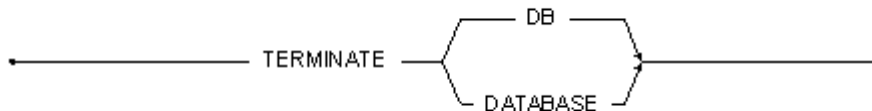

Figure 6-10 START DATABASE syntax

⮚ **Example**

The following starts the **Employees** database; the user **vivian** has DBA, SYSDBA or SYSADM privileges.

```
dmSQL> START DATABASE Employees vivian shuka828;
```

6.11 TERMINATE DATABASE

The TERMINATE DATABASE command shuts down a database so other users cannot connect. This command is normally used with client/server databases. Only a user with DBA, SYSDBA or SYSADM security privilege may execute the command.

Figure 6-11 TERMINATE DATABASE syntax

➲ **Example**

The following terminates the database on an active connection.

```
dmSQL> TERMINATE DATABASE;
```

6.12 UNLOAD

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. Only unload the object that you have the select privilege on. For instance, if you have the select privilege on a table, then you can only unload the content of this table. Only a user with DBA, SYSDBA or SYSADM security privilege may unload the database.

To Unload tables with names containing wild cards like the escape character "\", or double quotes on the name.
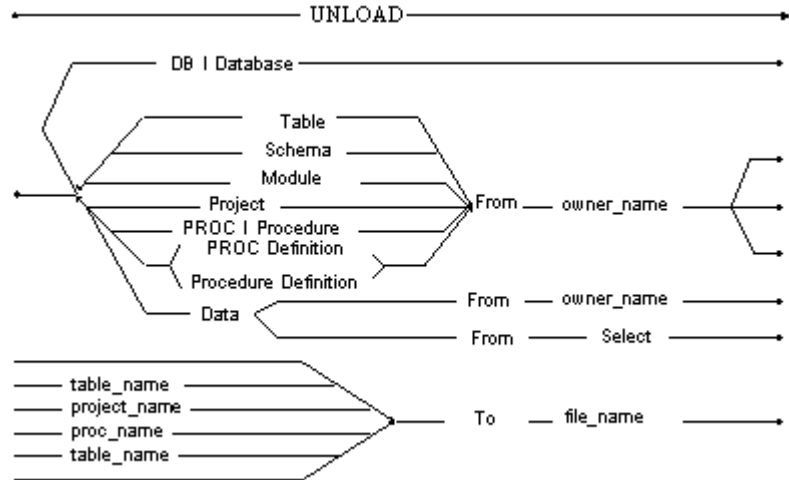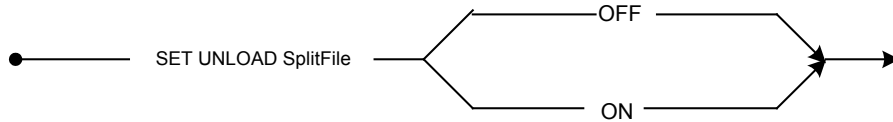

Figure 6-12 UNLOAD syntax

User can use the **set unload splitfile on/off** command to split the unloaded script file according to each table's definition, data, index and other related information. The default is off.

Set unload splitfile on …… split the unloaded script file.

Set unload splitfle off …… by default, it only unload the database content to <external text file name>.bn and <external text file name>.so files.
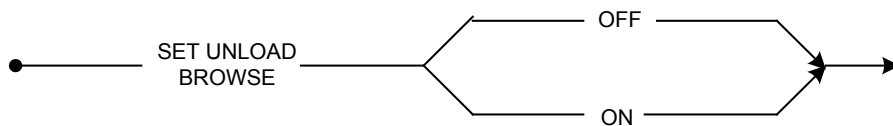

⊃ Example

```
dmSQL> SET UNLOAD SPLITFILE ON;
dmSQL> UNLOAD DB TO empdb;
dmSQL> SET UNLOAD SPILTFILE OFF;
```

User can use the **set unload browse on/off** command to ensure the unload command and other DML can be used simultaneously and the consistency of the unloaded data before doing unload operation. The default is off.

*Set unload browse on ……*the unload command and other DML can be used simultaneously, meanwhile, dirty data will be unload.

*Set unload browse off ……*the unload command and other DML can not be used simultaneously.

➲ **Example**
```
dmSQL> SET UNLOAD BROWSE ON;
dmSQL> UNLOAD DB TO empdb;
dmSQL> SET UNLOAD BROWSE OFF;
```

UNLOAD DB [DATABASE]

Only a user with DBA, SYSDBA or SYSADM security privilege may unload the content of a database to an external text file. This file includes information about security, tablespaces, definitions, indices, synonyms, data, etc. For each database, dmSQL will generate at least two external files, one script, and one BLOB data.

empdb is the name of the external text file. By default, dmSQL will create these files in the current working directory. In the statement below, 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 generate empdb.b1, empdb.b2 through to empdb.bn sequentially up to a maximum number of 99. dmSQL will always generate one script file emodb.s0, and its maximum size is set to the operating system limitation.

➲ **Example 1**
```
dmSQL> UNLOAD DB TO empdb;
```

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 l*ocal_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

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 is similar with "?" 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 with unload table. It can only unload the definition of a table, and does not unload the data in a table. Uses the same wild cards as illustrated in the above unload table option.

UNLOAD DATA

This option will unload all data from a table and does 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: **dmSQL>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 tb_doc.c01_int, tb_doc.c02_char FROM tb_doc,
tb_txt WHERE tb_doc.c01_int= tb_txt.c01_int) TO f1;
```

➲ **Example 2**

Illegal syntax
```
dmSQL> UNLOAD DATA FROM (SELECT tb_doc.c01_int, tb_txt.c01_int FROM tb_doc,
tb_txt WHERE tb_doc.c01_int = tb_txt.c01_int) TO f1;
```

➲ **Example 3**

Illegal syntax, no aggregate or built-in functions are permitted in the projection columns.
```
dmSQL> UNLOAD DATA FROM (SELECT AVG(c01 int) FROM tb doc) TO f1;
dmSQL> UNLOAD DATA FROM (SELECT NOW() FROM tb_doc) TO f1;
```

➲ **Example 4**

Valid syntax, views and synonyms are permitted.
```
dmSQL> UNLOAD DATA FROM (SELECT * FROM syn_tmp WHERE c01_int > 10) TO f1;
dmSQL> UNLOAD DATA FROM (SELECT * FROM view_tmp WHERE c01_int < 10) TO f1;
```

UNLOAD PROJECT

This option allows a user to unload a 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**, … etc.
```
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;
```

➲ **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;
```