DBMaker

SQL Stored Procedure User's Guide

CASEMaker®

CASEMaker Inc./Corporate Headquarters

1680 Civic Center Drive

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

www.casemaker.com

www.casemaker.com/support

Document No. 645049-235039/DBM53-M12302012-SLSP

Publication Date: 2012-12-30

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.

Contens

1 Introduction

Welcome to the *DBMaker SQL Stored Procedure User's Guide*. DBMaker is a powerful and flexible *SQL Database Management System (DBMS)* that supports an interactive *Structured Query Language (SQL)*, a *Microsoft Open Database Connectivity (ODBC)* compatible interface, and *Embedded SQL* for *C (ESQL/C).* The unique open architecture and native *ODBC* interface give you the freedom to build custom applications using a wide variety of programming tools or to query databases using existing *ODBC*-compliant applications.

DBMaker is easily scalable from personal single-user databases to distributed enterprise-wide databases. The advanced security, integrity, and reliability features of *DBMaker* ensure the safety of critical data. Extensive cross-platform support permits you to leverage existing hardware, allows for expansion and upgrades to more powerful hardware as your needs grow.

DBMaker provides excellent multimedia handling capabilities to store, search, retrieve, and manipulate all types of multimedia data. *Binary Large Objects (BLOBs)* ensures 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 the source application.

This manual includes the basic operations of *SQL stored procedure* and provides systematic instructions that guide you through the management of a database. The *User's Guide* content is intended for designers and administers of *DBMaker* databases. It will assist those unfamiliar with using *DBMaker*, but have some understanding of

how a relational database works. The user should have some operating systems knowledge working with *Windows* and/or *UNIX* environments. Information in this manual may also be helpful for experienced users for reference purposes.

The manual shows various commands and procedures used in maintaining a database with *SQL stored procedure*. Although the manual is for *DBMaker* on *Windows NT* and *Windows 98* environments, it can perform all functions on a *UNIX* platform. For clarity purposes, portions of sample databases appear through out this manual.

SQL is a dual-mode language. It is both an interactive tool to communicate and access a database, commonly referred as an Interactive *SQL*, and a database programming language used by application programs for database access.

Generally, all major RDBMS provide their own user interface for using *SQL*. For example, *DBMaker* provides dmSQL/C. Users can input *SQL* syntax directly through the tool to access and maintain their database.

DBMaker also comes with a variety of Java Tools. For more information on a particular subject, refer to the "*Additional Resources*" section that follows and select the appropriate manual.

1.1 Other Sources of Information

DBMaker provides a complete set of RDBMS manuals in addition to this one. For more information on a particular subject, consult one of the books listed below:

- For an introduction to DBMaker's capabilities and functions, refer to the *DBMaker Tutorial*.
- For more information on designing, administering, and maintaining a DBMaker database, refer to the *Database Administrator's Guide*.
- 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 DCI COBOL interface tool, refer to the *DCI User's Guide*.
- For more information on the SQL language used in dmSQL, refer to the *SQL Command and Function Reference*.
- For more information on the dmSQL, refer to the *dmSQL 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 ESQL/C programming, refer to the *ESQL/C User's Guide*.
- For more information on the Java stored procedure, refer to the *JAVA stored procedure Guide*.

1.2 Technical Support

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

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

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

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

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

1.3 Document Conventions

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

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

2 DBMaker Stored Procedure Overview

We provide a general description of ESQL, Java stored procedure and SQL stored procedure in this chapter.

2.1 What is a Stored Procedure

A stored procedure is a program that is kept and executed within a database server. Once the stored procedure has been created, it is stored in executable format in the database as an object. This allows the database engine to bypass repeated SQL compilation and optimization, increasing the performance of frequently repeated tasks. Stored procedure is executed as a command in interactive SQL, or invoked in application programs, trigger actions, or other stored procedures.

Accomplish a wide range of objectives with stored procedures including improving database performance, simplifying the writing of applications, and limiting or monitoring access to a database.

Because stored procedures run in the DBMS itself, they can help to reduce latency in applications. Rather than executing four or five SQL statements in your applications, you just execute one stored procedure that does the operations for you on the server side. Reducing the number of network trips alone can have a dramatic effect on performance

2.2 DBMaker Supported Stored Procedure Languages

DBMaker supports three types of stored procedure languages: ESQL/C, Java and SQL.

DBMaker version before 4.3 only support ESQL/C stored procedure, but DBMaker 4.3 and later support ESQL/C stored procedures and Java stored procedure. SQL Stored Procedures (SQL SP) are now supported in DBMaker 5.2 and later.

This book concentrates on the development of stored procedures written in the SQL language.

ESQL/C Stored Procedure

DBMaker ESQL/C provides the convenience of entering SQL statements directly into the C language source code. An ESQL stored procedure is an ESQL/C program. Stored procedures can perform any function a C application can, including calling other C functions and system calls. Therefore, a C compiler is required for writing stored procedures. You can write C application programs that use ESQL commands to access a DBMS. The DBMaker ESQL/C preprocessor prepares the application program containing the SQL commands for the C compiler. The preprocessor then converts the SQL commands to C statements, with C comments, to perform the database operations.

An ESQL/C program for a stored procedure consists of a **CREATE PROCEDURE** statement, a declare section if needed, and the code section. If your program does not use any host variables, the declare section can be omitted. For more information on ESQL, refer to the *"ESQL User's Guide"* and select the appropriate sections.

JAVA Stored Procedure

Given Java's popularity today, it is certainly possible that members of a development team are more proficient in Java than ESQL. DBMaker supports Java stored procedures to give Java programmers the ability to code in their preferred language.

For experienced ESQL developers, Java allows you to extend the functionality of database applications. Java also allows you to reuse code and dramatically increase productivity.

Each Java stored procedure is implemented as a Java method. When you call it, the name of the procedure and the parameters you specify are sent over the JDBC connection to the DBMS, which executes the procedure and returns the results back over the connection.

Java stored procedures represent an open, database-independent alternative to ESQL/C. Furthermore, Java stored procedures bring the power, richness, and object-orientation of the Java language.

For more information on Java stored procedure, refer to the *Java Stored Procedure Guide* and select the appropriate sections.

SQL Stored Procedure

Using ESQL and Java to create stored procedures is inefficient. Directly using SQL statements to create an SQL stored procedure if much more efficient.

SQL stored procedures are stored procedures that are implemented with only SQL statements. An SQL stored procedure is a set of SQL statements that is stored on the server. Clients can refer to SQL stored procedures rather than repetitively executing statements.

3 SQL Stored Procedures

A SQL stored procedure is a special kind of user-defined function. Once the stored procedure has been created, it is stored in executable format in the database as an object, it can be used to create simple scripts for quickly querying transforming, and updating data or for generating basic reports, for improving application performance, for modularizing applications, and for improving overall database design, and database security. This allows the database engine to bypass repeated SQL compilation and optimization, increasing the performance of frequently repeated tasks. Stored procedure is executed as a command in interactive SQL, or invoked in application programs, trigger actions, or other store procedures.

Because a stored procedure is stored as an object in the database, it is available to every application running on the database. Several applications can use the same stored procedure to reduce development time for an application.

Before deciding to implement a SQL stored procedure, it is important that you first understand what SQL stored procedures are, how they are implemented. With that knowledge you can learn more about SQL stored procedure from the following concept topics so that you can make informed decisions about when and how to use them in your database environment:

- SQL stored procedures advantages
- Tools for working with SQL stored procedures

- SQL Stored Procedure Function
- SQL Stored Procedure Syntax
- Working with SQL Stored Procedure
- SQL Stored Procedures Migration
- Restriction on SQL Stored Procedure

3.1 Advantages of SQL Stored Procedures

There are many useful applications of SQL stored procedures within a database or database application architecture. Improving database performance, simplifying application writing, and limiting or monitoring access to a database are just some of the benefits from utilizing stored procedures.

Systems Run Faster

In the absence of a compiler, so SQL stored procedure is not same as program that written by external language (such as ESQL/C) running so fast. However, the primary method for enhancing speed is to reduce network traffic, if you need to deal with the task is checking, recycling and multi-statement, but there is no user interactive duplication tasks, you can use stored procedures to reduce server load. So that each step to execute task, between the server and the client side is not have so much information exchange.

Reusable

SQL stored procedure can cross platforms and support Win32/64 and Linux32/64 systems.

Whether you change the host language has no impact on the SQL stored procedures, because it is the database logic rather than the application. Stored procedures can be migrated, when you use the SQL language creating stored procedures, you know that

it can run on any platform supported by DBMaker, does not require adding operating specific details, nor the need for implementation setting license in operating system for procedure, or configuring different software packages for your computer, which is the advantage of using SQL statement compared with Java and ESQL/C external languages.

Persistent

If you write a complete procedure, such as indicates the check cancel operation in the banking transaction processing. Then the person who wants to understand the checks could find your program. It will be in the form of source code stored in the database, which will make the process of data and processing data associated.

Easy to Migrate

DBMaker support the highly portable SQL 99 standard. Other DBMS also support this standard, so the code written using other DBMS can easily migrate to DBMaker.

Easy to Upgrade

SQL Stored Procedure is same as the DBMaker ESQL data types, creation, execution and delete, so that convenience of existing customers to upgrade, and the client program does not require any changes.

Other Advantages

Because they do not use the extra C compiler, so it will not appear the problem that is affected by the compiler, such as the installation path or the compiler unloading.

Easy to use, syntax simple, clear, structural strong. This feature will be introduced in later in *SQL Stored Procedure Syntax*.

Do not need to compile, the create speed quickly.

3.2 Tools for working with SQL stored procedures

SQL stored procedure development tools make it faster and easier to create and execute SQL stored procedures. The following GUI and command line tool can be used to create and execute SQL stored procedures:

- JDBA Tool
- dmSQL

JDBA Tool

JDBA Tool is a cross-platform user-friendly graphical user interface (GUI) that helps users to easily manage database objects in DBMaker, a powerful and flexible SQL Database Management System. JDBA tool hides the complexity of the DBMS and query language and provides an easy to understand and convenient to use interface. You can perform database functions without having to learn SQL. JDBA Tool's monitoring functions also provides statistical data and information on who is using your database. Refer to the *JDBA Tool Reference* for more information about how to use JDBA tool.

dmSQL

dmSQL is a command line tool, you can do many more things within the DBMaker dmSQL tool that can assist you in developing SQL stored procedures, including: querying, modifying, loading, and extracting table data, working with XML functions, executing Java routines, and more. Refer to the *dmSQL User's Guide* for more information about how to use dmSQL command line tool.

4 SQL Stored Procedures Function

SQL stored procedures have a rich set of features, as follows:

- Can contain Stored Procedural Language statements and features which support the implementation of control-flow logic around traditional static and dynamic SQL statements.
- Are easy to implement, because they use a simple high-level, strongly typed language.
- SQL Store procedures are more reliable than equivalent external procedures.
- Adhere to the SQL99 ANSI/ISO/IEC SQL standard.
- Support input, output parameter passing modes and the following data types: SMALLINT, INTEGER, BIGINT, FLOAT, DOUBLE, DECIMAL, REAL, DATE, TIME, TIMESTAMP, BINARY, CHAR, VARCHAR, NCHAR, NVARCHAR.
- Support a simple, but powerful condition and error-handling model.
- Allow you to return single result sets to the caller or to a client application.
- Allow you to easily access the SQLSTATE and SQLCODE values as special variables but not support assignment values to SQLSTATE and SQLCODE.
- Can be invoked wherever the CALL statement is supported.

- Support nested SQL stored procedure calls to other stored procedures implemented in other languages.
- Support recursion.
- Can be called from triggers.
- Support define variables in stored procedures: cursor, condition, handle
- Support cursor operation: OPEN, FETCH, CLOSE
- Support SET syntax for variables assignment
- Support control flow statement: IF, CASE, WHILE, LOOP, FOR, REPEAT, GOTO, RETURN
- Support TRACE, using the same method as the ESQL

SQL Stored procedures provide extensive support not limited to what is listed above. When implemented according to best practices, they can play an essential role in database architecture, database application design, and in database system performance.

5 SQL Stored Procedures Syntax

The SQL Stored Procedure Language (SQL SP) is accord with ANSI SQL99 language standard, it is a set of SQL statements that was introduced in *SQL Command and Function Reference* to provide the procedural constructs necessary for implementing control flow logic around traditional SQL queries and operations.

SQL Stored Procedure has a simple syntax that includes support for variables, conditional statements, looping statements, transfer of control statements, error management statements, and result set manipulation statements.

5.1 Architecture of the SQL Stored Procedure

SQL Stored procedures consist of several logic parts and the format is quite straight-forward and easy to follow and is intended to simplify the design and semantics of routines.

The core of an SQL stored procedure is a compound statement. Compound statements are bounded by the keywords BEGIN and END. The following is an architecture of SQL stored procedure statement:

```
BEGIN                          #block header
Variable declarations
```

```
Condition declarations
Cursor declarations
Condition handler declarations
Assignment ,flow of control. SQL statements and other compound
statements
END;                    #block end
```

Above shows that SQL stored procedures can consist of one or more optionally atomic compound statements (or blocks) and that these blocks can be nested or serially introduced within a single SQL stored procedure. Within each of these atomic blocks there is a prescribed order for the optional variable, condition, and handler declarations. These must precede the introduction of procedural logic implemented with SQL-control statements and other SQL statements and cursor declarations. Cursors can be declared anywhere with the set of SQL statements contained in the SQL stored procedure body.

5.2 Syntax of the SQL Stored Procedure

The following naming rules apply to the parameter names and variable names in SQL stored procedures:

- SQL stored procedure names can contain at most 128 characters
- SQL stored procedure names can contain any alphanumeric characters, including the underscore
- Character may be in any position
- SQL stored procedure names are not case-sensitive
- SQL stored procedure names must be unique

➲ **Syntax: SQL stored procedure syntax**

```
<SQL stored procedure syntax> ::=
CREATE [OR REPLACE] PROCEDURE <procedure_name>
[< procedure_parameters > [ { <comma> < procedure_parameters
> }... ]]
```

```
[RETURNS STATUS]
LANGUAGE SQL
BEGIN
  [stored_procedure_statement]
END;
<procedure_parameters> ::=
[IN | OUT | INPUT | OUTPUT] <parameter_name> <data_type>
```

Above syntax demonstrate the SQL stored procedure overall frame, a complete SQL stored procedure must contain a block header and block end, block header is used to describe name and its parameters of the SQL stored procedure, block end is only a end sign, such as:

```
CREATE [OR REPLACE] PROCEDURE project_name.module_name.sp_name
[ arg_list ]
LANGUAGE SQL
BEGIN
----------------->Block header
[ declare_list ]
[ statement_list ]
----------------> Block end
END;
```

Arg_list is a parameter list, input parameters and output parameters in the declare process, and will not return parameter if it is NULL.

```
<Arg_list> ::=
        [ Variable declarations ]
        [ Condition declarations ]
        [ Cursor declarations ]
        [ Condition handler declarations ]
```

Declare_list is a variable declare list used in declare process.

```
<declare_list> ::=
        [ Variable declarations ]
        [ Condition declarations ]
        [ Cursor declarations ]
        [ Condition handler declarations ]
```

Statement_list is a process control statement, consisted by various control statements and SQL statements to implement database operations generated by process.

```
<statement_list> ::=
        [ sp_block ]
        [ procedure_control_statement ]
```

5.3 Parameters in SQL Stored Procedure

SQL stored procedures support parameters for the passing of SQL values into and out of procedures.

Parameters can be useful in SQL stored procedures when implementing logic that is conditional on a particular input or set of input scalar values or when you need to return one or more output scalar values but do not want to return a result set.

It is good to understand the features of and limitations of parameters in SQL Stored procedures when designing or creating SQL stored procedures.

- DBMaker supports the optional use of a large number of input, output parameters in SQL Stored Procedures. The keywords IN/INPUT, OUT/OUTPUT in the routine signature portion of CREATE PROCEDURE statements indicate the mode or intended use of the parameter. IN/INPUT and OUT/OUTPUT parameters are passed by value.
- When multiple parameters are specified for a procedure they must each have a unique name.
- If a variable is to be declared within the procedure with the same name as a parameter, the variable must be declared within a labeled atomic block nested within the procedure. Otherwise DBMaker will detect what would otherwise be an ambiguous name reference.

- SQL stored procedure supports data types is: SMALLINT, INTEGER, BIGINT, FLOAT, DOUBLE, DECIMAL, REAL,DATE, TIME, TIMESTAMP, BINARY, CHAR, VARCHAR, NCHAR, NVARCHAR..

➲ **Syntax: SQL stored procedure parameter syntax**

```
<procedure_parameters> ::=
[IN | OUT| INPUT | OUTPUT] <parameter_name> <data_type>
```

This clause is optional. SQL stored procedure is same as ordinary function that have it's parameters to convey data, Main function can convey data to stored procedure by adding 'in/input' after parameters. SQL stored procedure also can convey the disposed date to main function by adding 'out/output' after parameters.

➲ **Example 1: The following SQL stored procedure named myparams illustrates the use of IN/INPUT, and OUT/OUTPUT parameter modes.**

```
CREATE PROCEDURE myparams (IN p1 INT, OUT p2 INT, OUT p3 INT)
LANGUAGE SQL
BEGIN
SET p2 = p1 + 1;
SET p3 = 2 * p1;
END;
```

➲ **Example 2: The parameter of SQL stored procedure is ignored.**

```
CREATE PROCEDURE CRETB
LANGUAGE SQL
BEGIN
        CREATE TABLE TB_1(V1 INT,V2 BIGINT,V3 SMALLINT,V4 CHAR(10),
                          V5 VARCHAR(20),V6 FLOAT,V7 DOUBLE,
                          V8 BINARY(10),V9 DECIMAL(20,4),V10 TIME,V11
DATE,
                          V12 TIMESTAMP);
END;
```

The parameters of SQL stored procedure CRETB is ignored, its function is to create a table named TB_1.

➲ **Example 3: SQL stored procedure with the input parameters:**

```
###############################################################################
#
#       Module Name = INS.SP
#       Purpose = Store Procedure testing program
#               1. Test IN parameter store procedure
#               2. Test all type which can use in Store Procedure
#       Function = 1. Create table INS
#       Use Database: "DBNAME" "SYSADM"  ""
#            table : INS(V1 int,V2 BIGINT,V3 smallint,V4 INT,
#                        V5 float,V6 DOUBLE, V7 DECIMAL(20,4),
#                        V8 binary(20), V9 CHAR (20), V10 VARCHAR (20),
#                        V11 NCHAR (40),V12 NVARCHAR (40),
#                         V13 DATE, V14 TIME, V15 TIMESTAMP,V16 REAL)
###############################################################################
#
CREATE PROCEDURE INS(IN V1 int, IN V2 BIGINT, IN V3 smallint, IN V4 INT,
                     IN V5 FLOAT, IN V6 DOUBLE,IN V7 DECIMAL(20,4),
                     IN V8 BINARY(20), IN V9 CHAR(20),
                     IN V10 VARCHAR(20),IN V11 NCHAR(40),
                     IN V12 NVARCHAR(40), IN V13 DATE,
                     IN V14 TIME, IN V15 TIMESTAMP, IN V16 REAL)
LANGUAGE SQL
BEGIN
        INSERT INTO ins VALUES(V1,
V2,V3,V4,V5,V6,V7,V8,V9,V10,V11,V12,V13,V14,V15,V16);
END;
```

All the parameter types of SQL stored procedure INS is INPUT. Input parameter data type in INS contains all the data type supported by SQL stored procedure. The function as shown in above case is to insert input data into the existing INS table.

NOTE *All the current row with "#" began in this manual example as a comment*

➲ **Example 4:** SQL stored procedures with input\output parameters

```
###############################################################################
####
```

```
#       Module Name = DRPTB.SP
#       Purpose  = Store Procedure testing program
#                  1. Test DROP TABLE in Stored Procrdure
#       Function = 1. create table TB_1
#       Use Database : DBNAME
#       table : TB_1(V1 INT, V2 BIGINT, V3 SMALLINT, V4 CHAR(10),
#                    V5 VARCHAR(20),V6 FLOAT,V7 DOUBLE,V8 BINARY,
#                    V9 DECIMAL,V10 TIME,V11 DATE,V12 TIMESTAMP,)
################################################################################
####
CREATE PROCEDURE DRPTB(IN V1 CHAR(20),OUT WORNING CHAR(40))
LANGUAGE SQL
BEGIN
        IF V1 = 'DROP TABLE' THEN
                DROP TABLE TB_1;
                SET WORNING ='NORMAL! TABLE TB_1 DROPED';
        ELSEIF V1 = 'CREATE TABLE' THEN
                CALL CRETB;
                SET WORNING ='NORMAL! CREATE A NEW TABLE NAMED TB_1';
        ELSE
                SET WORNING = 'YOU INPUT WRONG PARAMETER!';
        END IF;
END;
```

The SQL stored procedure DRPTB in the above case is not only have input parameters, but also have output parameters. It determines the input parameters in the process of body, if the input parameter is 'DROP TABLE', then the SQL stored procedure executes the DROP TABLE command and sets the output parameters to 'NORMAL! TABLE TB_1 DROPED'. So it can control the output parameters based on the input parameters.

5.4 Variables in SQL Stored Procedures

Local variable support in SQL stored procedures allows you to assign and retrieve SQL values in support of SQL stored procedure logic.

Variables in SQL Stored procedures are defined by using the DECLARE statement. The DECLARE statement is used to define various items local to a routine: Local variables. Conditions , handles and Cursors. DECLARE is allowed only inside a BEGIN ... END compound statement and must be at its start, before any other statements. Declarations must follow a certain order. Cursors must be declared before declaring handlers, and variables and conditions must be declared before declaring either cursors or handlers.

➲ **Syntax: SQL stored procedure DECLARE variable syntax.**

```
<sp_declare_main> ::=
DECLARE <variable_name> [ { <comma> <variable_name> }... ] <data_type>
[DEFAULT <default_value>]
```

This statement is used to declare local variables. To provide a default value for the variable, include a default clause. The value can be specified as an expression, it need not be a constant if the default clause is missing. The initial value is null.

The scope of a local variable is within the begin…end block where it is declared. The variable can be referred to in blocks nested within the declaring block except those blocks that declare a variable with the same name.

➲ **Example 1: declare data without default**

```
CREATE PROCEDURE DECALRES
LANGUAGE SQL
BEGIN
        declare v1 int;
        declare v2 bigint;
        declare v3 char(10);
        declare v4 varchar(10);
        declare v5 integer;
```

```
        declare v6 time;
        declare v7 date;
        declare v8 timestamp;
        declare v9 nchar(20);
        declare v10 binary(20);
        declare v11 decimal(4,8);
        declare v12 double;
        declare v13 float;
        declare v14 real;
END;
```

➲ **Example 2: declare data with default**

```
CREATE PROCEDURE DECLARES
LANGUAGE SQL
BEGIN
        declare v1 int default 50;
        declare v2 bigint default 7396;
        declare v3 char(10) default 'char';
        declare v4 varchar(10) default 'varchar';
        declare v5 integer default 20;
        declare v6 time default '11:11:11';
        declare v7 date default '2008-08-08';
        declare v8 timestamp default '2008-08-08 11:11:11';
        declare v9 nchar(20) default 'nchar';
        declare v10 binary(20) default 'binary';
        declare v11 decimal(4,8) default 1.123;
        declare v12 double default 123;
        declare v13 float default 6546;
        declare v14 real default 123.3;
END;
```

5.5 Cursors in SQL Stored Procedures

In SQL stored procedures, a cursor make it possible to define a result set (a set of data rows) and perform complex logic on a row by row basis. By using the same mechanics, an SQL Stored procedure can also define a result set and return it directly to the caller of the SQL Stored procedure or to a client application.

A cursor can be viewed as a pointer to one row in a set of rows and can only reference one row at a time, but can move to other rows of the result set as needed.

The DECLARE CURSOR statement defines a cursor. Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is not an executable statement and cannot be dynamically prepared.

To use cursors in SQL stored procedures, you need to do the following:

1. Declare a cursor that defines a result set
2. Open the cursor to establish the result set
3. Fetch the data into local variables as needed from the cursor, one row at a time
4. Close the cursor when done

To work with cursors you must use the following SQL statements:

- DECLARE CURSOR
- OPEN
- FETCH
- CLOSE

➲ **Syntax: DECLARE CURSOR statement syntax**

```
<sp_declare_main> ::=

DECLARE <cursor_name> [[NO] SCROLL] CURSOR [WITH RETURN] FOR { CALL
<procedure_name> | <select_statement> }
```

cursor_name

Specifies the name of the cursor created when the source program is run. The name must not be the same as the name of another cursor declared in the source program. The cursor must be opened before use.

[NO] SCROLL

If the SCROLL clause isn't used or only the NO SCROLL clause is used in the definition of the DECLARE CURSOR statement, the FECTCH statement will not execute any operation expect the NEXT behavior. Contrarily, if the SCROLL clause is used, then all behavior of the FETCH statement can be used. Default value is non-scrollable cursor.

WITH RETURN

Within an SQL stored procedure, cursors declared using the WITH RETURN clause that are still open when the SQL stored procedure ends, define the result sets from the SQL stored procedure. All other open cursors in an SQL stored procedure are closed when the SQL stored procedure ends. Within an external stored procedure (one not defined using LANGUAGE SQL), the default for all cursors is WITH RETURN TO CALL. Therefore, all cursors that are open when the procedure ends will be considered result sets.

Select_statement

Identifies the SELECT statement of the cursor. The *select-statement* must not include parameter markers, but can include references to host variables. The declarations of the host variables must precede the DECLARE CURSOR statement in the source program.

call

Specifies that the cursor can return a result set to the caller. For example, if the caller is another stored procedure, the result set is returned to that stored procedure. If the caller is a client application, the result set is returned to the client application.

➲ **Example 1 : cursor with select_statements**

```
CREATE PROCEDURE getbd_sql(IN name char(12), OUT bd DATE)
```

```
LANGUAGE SQL
BEGIN
        DECLARE cur CURSOR FOR SELECT birthday FROM birthd WHERE NAME =
name;

        OPEN cur;
        FETCH cur INTO bd;
        CLOSE cur;
END;
```

➲ **Example 2: cursor with result set**

```
CREATE PROCEDURE t42_sql(argn TIMESTAMP)
LANGUAGE SQL
BEGIN
                DECLARE cur CURSOR WITH RETURN FOR select n,ts from t2
where ts < argn;
                OPEN cur;
END;
```

➲ **Example 3: cursor with call**

```
CREATE PROCEDURE call_test
LANGUAGE SQL
BEGIN
                DECLARE cur CURSOR WITH RETURN FOR select * from call_tb;
                OPEN cur;
END;
```

➲ **Example 4: call4 will use call_test to declare a result set.**

```
CREATE PROCEDURE call4
LANGUAGE SQL
BEGIN
                DECLARE cur CURSOR WITH RETURN FOR call call_test;
                OPEN cur;
END;
```

➲ **Example 5: The following example demonstrates the basic usage of a read-only cursor within an SQL stored procedure:**

```
CREATE PROCEDURE sum_salaries(OUT sum INTEGER)
LANGUAGE SQL
BEGIN
DECLARE p_sum INTEGER;
DECLARE p_sal INTEGER;
DECLARE c CURSOR FOR SELECT SALARY FROM EMPLOYEE;
SET p_sum = 0;
OPEN c;
FETCH FROM c INTO p_sal;
WHILE(SQLCODE = 0)
                DO
                SET p_sum = p_sum + p_sal;
                FETCH FROM c INTO p_sal;
        END WHILE;
CLOSE c;
SET sum = p_sum;
END;
```

FETCH in SQL Stored Procedures

The FETCH statement positions a cursor on the next row of its result table and assigns the values of that row to host variables. Although an interactive SQL facility might provide an interface that gives the appearance of interactive execution, this statement can only be embedded within an application program. It is an executable statement that cannot be dynamically prepared.

➲ **Syntax: FETCH statement syntax**

```
<FETCH statement main> ::=
FETCH [NEXT|PRIOR|FIRST|LAST|ABSOLUTE n|RELATIVE n] FROM <cursor_name>
INTO <fetch_target_arg>
fetch_target_arg ::=
<variable_name> [ { <comma> <variable_name> }... ]
```

cursor-name

Identifies the cursor to be used in the fetch operation. The cursor-name must identify a declared cursor, as explained in "DECLARE CURSOR". The DECLARE CURSOR statement must precede the FETCH statement in the source program. When the FETCH statement is executed, the cursor must be in the open state.

NEXT

Returns the next row within the results set. NEXT is the default cursor fetch.

LAST

The LAST command moves the cursor to the last row within the result set and returns the last row.

FIRST

The FIRST command moves the cursor to the first row within the result set and

returns the first row.

PRIOR

The PRIOR command returns the previous row within the results set.

ABSOLUTE n

Returns the n row from the first row within the result set

RELATIVE n

Returns the n row from the current row within the result set

INTO fetch_target_arg

Identifies one or more variables that must be described in accordance with the rules for declaring variables. The first value in the result row is assigned to the first variable in the list, the second value to the second variable, and so on.

➲ **Example 1: fetch**

```
CREATE PROCEDURE t43_sql(i SMALLINT, OUT ts1 TIMESTAMP)
LANGUAGE SQL
```

```
BEGIN
        DECLARE cur CURSOR FOR select TS from t2 where n = i;
        OPEN cur;
        FETCH cur INTO ts1;
        CLOSE cur;
END;
```

➲ **Example 2: fetch with first, last, next, prior**

```
Table used in example procedure
dmSQL> SELECT * FROM TB_1;
         C1
====================
FIRST
NEXT
PRIOR
LAST


CREATE PROCEDURE FETCH_TEST(OUT FHTY_1 CHAR(20),OUT FHTY_2 CHAR(20),OUT
FHTY_3 CHAR(20),OUT FHTY_4 CHAR(20),OUT FHTY_5 CHAR(20),OUT FHTY_6
CHAR(20))
LANGUAGE SQL
BEGIN
        DECLARE V1 CHAR(20);
        DECLARE V2 CHAR(20);
        DECLARE V3 CHAR(20);
        DECLARE V4 CHAR(20);
        DECLARE V5 CHAR(20);
        DECLARE V6 CHAR(20);
        DECLARE CUR SCROLL CURSOR FOR SELECT * FROM TB_1;
        OPEN CUR;
        FETCH FIRST FROM CUR INTO V1;
                IF V1 = 'FIRST' THEN
                        SET FHTY_1 = V1;
                ELSE
                        SET FHTY_1 = 'NULL';
```

```
        END IF;
FETCH NEXT FROM CUR INTO V2;
        IF V2 = 'NEXT' THEN
                SET FHTY_2 = V2;
        ELSE
                SET FHTY_2 = 'NULL';
        END IF;
FETCH NEXT FROM CUR INTO V3;
FETCH PRIOR FROM CUR INTO V3;
        IF V3 = 'NEXT' THEN
                SET FHTY_3 = V3;
        ELSE
                SET FHTY_3 = 'NULL';
        END IF;
FETCH RELATIVE 1 FROM CUR INTO V4;
        IF V4 = 'PRIOR' THEN
                SET FHTY_4 = V4;
        ELSE
                SET FHTY_4 = 'NULL';
        END IF;
FETCH ABSOLUTE 1 FROM CUR INTO V5;
    IF V5 = 'FIRST' THEN
                SET FHTY_5 = V5;
        ELSE
                SET FHTY_5 = 'NULL';
        END IF;

FETCH LAST FROM CUR INTO V6;
        IF V6 = 'LAST' THEN
                SET FHTY_6 = V6;
        ELSE
                SET FHTY_6 = 'NULL';
        END IF;
CLOSE CUR;
```

```
END;
```

➲ **Call fetch_test result:**

```
dmSQL> CALL FETCH_TEST(?,?,?,?,?,?);
FHTY_1                          : FIRST
FHTY_2                          : NEXT
FHTY_3                          : NEXT
FHTY_4                          : PRIOR
FHTY_5                          : FIRST
FHTY_6                          : LAST
```

DECLARE CONDITION in SQL Stored Procedures

Certain conditions may require specific handling. These conditions can relate to errors, as well as to general flow control inside a routine.

➲ **Syntax: DECLARE CONDITION statement syntax**

```
<condition declaration> ::=
DECLARE <condition name> CONDITION  FOR <sqlstate value>
<sqlstate value> ::=
SQLSTATE [ VALUE ] <character string literal>
```

➲ **Example :**

```
DECLARE con1 CONDITION FOR SQLSTATE '23000';
DECLARE con2 CONDITION FOR SQLSTATE VALUE '23001';
```

DECLARE HANDLE in SQL Stored Procedures

Associate a handler with exception or completion conditions to be handled in a module or compound statement.

➲ **The user can redefine the DECLARE HANDLER statement, then previous actions defined by the DECLARE HANDLER statement will be re-set.Syntax: DECLARE HANDLE statement syntax**

```
<handler declaration> ::=
DECLARE <handler type> HANDLER FOR <condition value list> <handler
action>
```

```
<handler type> ::=
CONTINUE
| EXIT
<handler action> ::= <SQL procedure statement>
<condition value list> ::= <condition value> [ { <comma> <condition
value> }... ]
<condition value> ::=
<sqlstate value>
| <condition name>
| SQLEXCEPTION
| SQLWARNING
| NOT FOUND
```

➲ **Example 1:**

```
create procedure sphdler
language sql
begin
        declare continue handler for sqlexception;
        drop table tb_1;
        declare exit handler for sqlexception;
        drop table tb_1;
end
```

➲ **Example 2:**

```
DECLARE val INT;
DECLARE con1 CONDITION FOR SQLSTATE '23000';
DECLARE CONTINUE HANDLER FOR SQLSTATE '23000';
DECLARE CONTINUE HANDLER FOR con1 SET val = 100;
```

5.6 Assignment statements in SQL Stored Procedures

Assignment statement used to assign a value to an SQL variable or SQL parameter. Values can be assigned to variables using the SET statement or the CURSOR FOR

SELECT FROM statement or as a default value when the variable is declared. Literals, expressions, the result of a query, and special register values can be assigned to variables. Variable values can be assigned to SQL stored procedure parameters, other variables in the SQL stored procedure, and can be referenced as parameters within SQL stored procedure statements that executed within the routine.

The SET Variable statement assigns values to local variables, output parameters, or new transition variables. It is under transaction control. SET assignment statements accept simple expressions and complex expressions.

NOTE *For the string data type variable assignment, assignment length is less than 1024 bytes.*

➲ **Example 1: The following example demonstrates various methods for assigning and retrieving variable values**

```
CREATE PROCEDURE proc_vars
LANGUAGE SQL
BEGIN
DECLARE v_rcount INTEGER;
DECLARE v_max DECIMAL (9,2);
DECLARE v_adate, v_another DATE;
DECLARE v_total INTEGER DEFAULT 0;           # (1)
SET v_total = v_total + 1;                   # (2)
DECLARE CUR CURSOR FOR SELECT * FROM TB_1;   # (3)
END;
```

When declaring a variable, you can specify a default value using the DEFAULT clause as in line (1). Line (2) shows that a SET statement can be used to assign a single variable value. Line (3) shows you can specify a value using the CURSOR FOR SELECT FROM.

NOTE *DECLARE must be in front of the SET definition, or the preprocessor compile error.*

➲ **Example 2: The following example demonstrates for assigning and truncating variable values in SET assignment statement**

In the SET assignment statement, for values exceeding the range of double and integer data type, which can be accepted if the assigned variables are corresponding data types, or will be truncated according to double and integer values range.

```
DECLARE d1, d2 DECIMAL(20, 10);
DECLARE b1, b2 BIGINT;
SET d1 = 1234.43534534531;     # exceeding the double data type range
SET d2 = 1234.43534534532;

SET b1 = 1234454654645645651;  # exceeding the integer data type range
SET b2 = 1234454654645645652;
```

The comparison result of the above **SET** equation is d1 less than d2, b1 less than b2.

However such as the following IF statement, d1 and d2, b1 and b2 are equal, because the exceeding rang parts were truncated.

```
IF 1234.43534534531 = 1234.43534534532 THEN ……..    #as double data type
intercepted
```

IF 1234454654645645651 = 1234454654645645652 THEN …….#as integer data type intercepted.

Simple expression

Simple expression is divided into numeric data types: INTEGER, BIGINT, SMALLINT, DOUBLE, FLOAT, DECIMAL,REAL; Character data types: CHAR, NCHAR, VARCHAR, NVARCHAR; BINARY data types and timestamp data types: DATE, TIME, TIMESTAMP.

A simple expression just includes '+', '-', '*', '/', variables, constants, values, string. The implementation efficiency of a simple expression is much higher than a complex expression, it applies in multi-loop statement to use, can greatly improve the speed of execution. For more information on SQL functions, refer to the *SQL Command and Function Reference*.

➲ **Syntax: set statement syntax**

```
SET <variable_name> ::= <variable_name> [{ <+ | - | * | / | || >
<variable_name> }...]
```

➲ **Example 1:**

```
CREATE PROCEDURE SETTS(out rc1 int,out rc2 int, out rc3 char(10) )
LANGUAGE SQL
BEGIN
        DECLARE d1 INT DEFAULT 2;
        DECLARE d2,d3 INT DEFAULT 2;
        DECLARE c1,c2 char(10) DEFAULT '12345';
        SET d1 = 1 + d2 * d3*100/10;
        SET d2 = 6;
        SET c2 = c1;
        SET d3 = d1+d2;
        SET rc1 = d1;
        SET rc2 = d2-3;
        SET rc3 = c2;
END;
```

➲ **Example 2:**

```
CREATE PROCEDURE OUTPUTS(OUTPUT V1 INT,OUTPUT V2 BIGINT,
                         OUTPUT V3 FLOAT,OUTPUT V4 DOUBLE,
                         OUTPUT V5 DECIMAL(8,4),
                         OUTPUT V6 BINARY(20),OUTPUT V7 CHAR(20),
                         OUTPUT V8 VARCHAR(20),OUTPUT V9 NCHAR(40),
                         OUTPUT V10 NVARCHAR(40),OUTPUT V11 DATE,
                         OUTPUT V12 TIME,OUTPUT V13 TIMESTAMP)
LANGUAGE SQL
BEGIN
        SET V1 = 1;
        SET V2 = 7396;
        SET V3 = 2.2;
        SET V4 = 3.3;
        SET V5 = 4.4;
```

```
        SET V6 = 'ASSIGNMENT';
        SET V7 = 'CHAR';
        SET V8 = 'VARCHAR';
        SET V9 = 'NCHAR';
        SET V10 = 'NVARCHAR';
        SET V11 = '2008-08-08';
        SET V12 = '11:11:11';
        SET V13 = '2008-08-08 11:11:11';
END;
```

Complex expression

A complex expression not only includes the assignment value in simple expression, but also includes SQL functions. such as build-in function and user defined function.

Build-in function can be used on columns in a result set or columns that restrict rows in a result set. Please refer to the *SQL Command and Function Reference* chapter 4 for more detail information about DBMaker build-in function, which listed the arguments and returned values for each function.

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 function with the same usages. Please refer to the *Database Administrator's Guide* chapter 13 for more detail information about user-defined function.

➲ **Example: set statement usage**

```
################################################################################
####
#       Module Name = SETTS.SP
#       Purpose  = Store Procedure testing program
#                  1. Test keyword "SET" in Store Procedure
#       Function = 1. decalare d1 d2 d3 c1 c2 for pass parameter
#       Use Database : DBNAME
################################################################################
####
CREATE PROCEDURE SETTS(out rc1 int,out rc2 int, out rc3 char(10) )
```

```
LANGUAGE SQL
BEGIN
        DECLARE d1 INT DEFAULT 2;
        DECLARE d2,d3 INT DEFAULT 2;
        DECLARE c1,c2 char(10) DEFAULT '12345';

        SET d1 = 1 + d2 * d3*100/10;
        SET d2 = 6;
        SET c2 = c1;
        SET d3 = d1+d2;
        SET rc1 = d1;
        SET rc2 = d2-3;
        SET rc3 = c2;
END;
```

5.7 Control flow statements in SQL Stored Procedures

Sequential execution is the most basic path that program execution can take. With this method, the program starts execution at the first line of the code, followed by the next, and continues until the final statement in the code has been executed. This approach works fine for very simple tasks, but tends to lack usefulness because it can only handle one situation. Programs often need to be able to decide what to do in response to changing circumstances. By controlling a code's execution path, a specific piece of code can then be used to intelligently handle more than one situation.

SQL control statements provide support for variables and flow of control statements that can be used to control the sequence of statement execution. Statements such as IF and CASE are used to conditionally execute blocks of SQL control statements, while other statements, such as WHILE and REPEAT, are typically used to execute a set of statements repetitively until a task is complete.

Although there are many types of SQL control statements, there are a few categories into which these can be sorted:

- Variable related statements
- Conditional statements
- Loop statements
- Goto statements
- Return statements
- Transfer of control statements
- Labels and SQL stored procedure compound statements

Variable related statements

Variable related SQL statements are used to declare variables and to assign values to variables. There are a few types of variable related statements:

- DECLARE <variable> statement in SQL stored procedures
- DECLARE <condition> statement in SQL stored procedures
- DECLARE <condition handler> statement in SQL stored procedures
- DECLARE CURSOR in SQL stored procedures

These statements provide the necessary support required to make use of the other types of SQL control statements and SQL statements that will make use of variable values.

Conditional statements

Conditional statements are used to define what logic is to be executed based on the status of some condition being satisfied. There are two types of conditional statements supported in SQL stored procedures:

- CASE
- IF

These statements are similar; however the CASE statements extends the IF statement.

CASE STATEMENT IN SQL STORED PROCEDURES

CASE statements can be used to conditionally enter into some logic based on the status of a condition being satisfied. There are two types of CASE statements:

- Simple case statement: used to enter into some logic based on a literal value
- Searched case statement: used to enter into some logic based on the value of an expression

The WHEN clause of the CASE statement defines the value that when satisfied determines the flow of control.

The CASE statement for stored routines implements a complex conditional construct. If a search_condition evaluates to true, the corresponding SQL statement list is executed. If no search condition matches, the statement list in the ELSE clause is executed. Each statement_list consists of one or more statements.

➲ **Syntax: CASE statement syntax**

```
<case_statement_main> ::=
CASE
<variable_case> | <condition_case>
ELSE
<sp_statement_main>
END CASE
<variable_case> ::= <variable_name> <variable_case_list>
<variable_case_list> ::= WHEN <var_value> THEN <sp_statement_main>
                          [ { < ; > WHEN <var_value> THEN
                         <sp_statement_main> }... ]
<condition_case> ::= WHEN <condition> THEN <sp_statement_main>
                     [ { < ; > WHEN <condition> THEN
                     <sp_statement_main> }... ]
```

➲ **Example 1: The following is an example of an SQL stored procedure with a CASE statement with a simple-case-statement-when-clause**

```
CREATE PROCEDURE UPDATE_DEPT (IN p_workdept char(3))
LANGUAGE SQL
BEGIN
```

```
DECLARE v_workdept CHAR(3);
SET v_workdept = p_workdept;
CASE v_workdept
WHEN 'A00' THEN
UPDATE department SET deptname = 'D1';
WHEN 'B01' THEN
UPDATE department SET deptname = 'D2';
ELSE
UPDATE department SET deptname = 'D3';
END CASE;
END;
```

➲ **Example 2: The following is an example of CASE statement with a searched-case-statement-when-clause**

```
CREATE PROCEDURE UPDATE_DEPT (IN p_workdept char(3))
LANGUAGE SQL
BEGIN
DECLARE v_workdept CHAR(3);
SET v_workdept = p_workdept;
CASE
WHEN v_workdept = 'A00' THEN
UPDATE department SET deptname = 'D1';
WHEN v_workdept = 'B01' THEN
UPDATE department SET deptname = 'D2';
ELSE
UPDATE department SET deptname = 'D3';
END CASE;
END;
```

The examples provided above are logically equivalent, however it is important to note that CASE statements with a searched-case-statement-when-clause can be very powerful. Any supported SQL expression can be used here. These expressions can contain references to variables, parameters and more.

➲ **Example 3: case usage**

```
###############################################################################
#
#       Module Name = CASE_TEST.SP
#       Purpose  = Store Procedure testing program
#                       1. Test keyword "CASE" in Store Procedure
#       Use Database : "DBNAME" "SYSADM"  ""
###############################################################################
#
CREATE PROCEDURE CASE_TEST (IN INVAL INT, OUT outval1 INT, OUT outval2
INT)
LANGUAGE SQL
BEGIN
        DECLARE VAL INT;
        SET VAL = INVAL;
        CASE VAL
                WHEN 1 THEN
                        SET OUTVAL1 = 1;
                WHEN 2 THEN
                        SET OUTVAL1 = 2;
                WHEN 3 THEN
                        SET OUTVAL1 = 3;
                ELSE
                        SET OUTVAL1 = 10;
        END CASE;
        CASE
                WHEN VAL = 1 THEN
                        SET OUTVAL2 = 11;
                WHEN VAL = 2 THEN
                        SET OUTVAL2 = 22;
                WHEN VAL = 3 THEN
                        SET OUTVAL2 = 33;
                ELSE
                        SET OUTVAL2 = 100;
        END CASE;
```

```
END;
```

IF STATEMENT IN SQL STORED PROCEDURES

IF statements can be used to conditionally enter into some logic based on the status of a condition being satisfied. The IF statement is logically equivalent to a CASE statements with a searched-case-statement-when clause.
The IF statement supports the use of optional ELSE IF clauses and a default ELSE clause. An END IF clause is required to indicate the end of the statement.

IF implements a basic conditional construct, and if the search_condition evaluates to true, the corresponding SQL statement list is executed. If no seatch_condition matches, the statement listed in the ELSE clause executed. Each statement_list consists of one or more statements.

➲ **Syntax: IF statement syntax**

```
<IF statement main> ::=
IF <condition_value> THEN <sp_statement_main>
[ELSEIF <condition_value> THEN <sp_statement_main>]
[ELSE <sp_statement_main>]
END IF
```

➲ **Example 1: The following is an example of SQL stored procedure that contains an IF..CALL statement**

```
#############################################################################
#
#       Module Name = IF_CALL.SP
#       Purpose = Store Procedure testing program
#                   1. Test keyword "IF" and "CALL" in Store Procedure
#       Function = 1. Store Procedure: CRETB CASE_TEST_2 INS
#       Use Database: "DBNAME" "SYSADM"  ""
#############################################################################
#
CREATE PROCEDURE IF_CALL (INPUT C1 CHAR (20))
LANGUAGE SQL
BEGIN
        DECLARE OBJ CHAR (10);
```

```
        IF C1 = 'CRETB' THEN
                CALL CRETB;
        ELSEIF C1 = 'CASE' THEN
                CALL CASE_TEST_2 (OBJ);
        ELSE
                CALL
INS(1,2,3,4,5,6,7,'binary','char','varchar',N'NCHAR',N'NVARCHAR',
'2008-01-01','11:11:11','2008-01-01 11:11:11',1);
        END IF;
END;
```

➲ **Example 2: The following is an example of SQL stored procedure that contains an IF…SET statement:**

```
#############################################################################
#
#       Module Name = SETTS.SP
#       Purpose  = Store Procedure testing program
#                  1. Test keyword "SET" and "IF" in Store Procedure
#       Function = 1. create table TB_1 and insert data into TB_1
#       Use Database: "DBNAME" "SYSADM"  ""
#                       table : TB_1(V1 INTEGER)
#############################################################################
CREATE PROCEDURE IFTEST_1(OUT SUM INTEGER)
LANGUAGE SQL
BEGIN
        DECLARE P_SUM INTEGER;
        DECLARE P_SAL INTEGER;
        DECLARE CUR CURSOR FOR SELECT  V1 FROM TB_1;
        SET P_SUM = 0;
        OPEN CUR;
        FETCH FROM CUR INTO P_SAL;
                IF P_SAL = 2 THEN
                SET P_SUM = 10;
                ELSE
                SET P_SUM = 20;
```

```
                END IF;
                FETCH NEXT FROM CUR INTO P_SAL;
                SET P_SUM = P_SUM+P_SAL;
                FETCH NEXT FROM CUR INTO P_SAL;
                SET P_SUM = P_SUM+P_SAL;
                FETCH NEXT FROM CUR INTO P_SAL;
                SET P_SUM = P_SUM+P_SAL;
                FETCH NEXT FROM CUR INTO P_SAL;
                SET P_SUM = P_SUM+P_SAL;
        CLOSE CUR;
        SET SUM = P_SUM;
END;
```

➲ **Example 3: The following is an example of SQL stored procedure that contains an IF…ELSEIF…ELSE statement:**

```
################################################################################
####
#       Module Name = IF_UPDATE.SP
#       Purpose  = Store Procedure testing program
#                  1. Test keyword "IF" and "UPDATE" in Store Procedure
#       Function = 1. create table TB_1 and insert data into TB_1
#       Use Database : "DBNAME" "SYSADM"  ""
#       table : TB_1(V1 INT,V2 DOUBLE,V3 CHAR(6))
################################################################################
####
CREATE PROCEDURE IF_UPDATE (IN C1 CHAR(6), IN C2 CHAR(20))
LANGUAGE SQL
BEGIN
        IF C2 = 'FIRST' THEN
                UPDATE TB_1 SET V2 = V2 * 1.10, V1 = 1000
                WHERE V3 = C1;
        ELSEIF C2 = 'SECOND' THEN
                UPDATE TB_1 SET V2 = V2 * 1.05, V1 = 500
                WHERE V3 = C1;
```

```
        ELSE
                UPDATE TB_1 SET V2 = V2 * 1.03, V1 = 0
                WHERE V3 = C1;
        END IF;
END;
```

➲ **Example 4: The following is an example of SQL stored procedure that contains an IF…ELSEIF statement:**

```
#############################################################################
#
#       Module Name = ELSEIFS.SP
#       Purpose  = Store Procedure testing program
#                  1. Test keyword "ELSEIF" in Store Procedure
#       Use Database: "DBNAME" "SYSADM"  ""
#############################################################################
#
CREATE PROCEDURE ELSEIFS(IN con INT, OUT c1 INT)
LANGUAGE SQL
BEGIN
        IF con = 1 THEN
                SET c1 = 1;
                INSERT INTO TB_1 VALUES(C1);
        ELSEIF con = 2 THEN
                SET c1 = 2;
                INSERT INTO TB_1 VALUES(C1);
        ELSEIF con = 3 THEN
                SET c1 = 3;
                INSERT INTO TB_1 VALUES(C1);
        ELSE
                IF con = 5 THEN
                        SET c1 = 5;
                        INSERT INTO TB_1 VALUES(C1);
                ELSEIF con = 6 THEN
                        SET c1 = 6;
                        INSERT INTO TB_1 VALUES(C1);
```

```
                ELSE
                        SET c1 = 7;
                        INSERT INTO TB_1 VALUES(C1);
                END IF;
        END IF;
END;
```

Looping statements

Looping statements provide support for repeatedly executing some logic until a condition is met. The following looping statements are supported in SQL control statements:

- FOR
- LOOP
- WHILE
- REPEAT

The FOR statement is distinct from the others, because it is used to iterate over rows of a defined result set, whereas the others are using for iterating over a series of SQL statements until for each a condition is satisfied.

FOR STATEMENT IN SQL STORED PROCEDURES

FOR statements are a special type of looping statement, because they are used to iterate over rows in a defined read-only result set. When a FOR statement is executed a cursor is implicitly declared such that for each iteration of the FOR-loop the next row is the result set if fetched. Looping continues until there are no rows left in the result set.

The FOR statement simplifies the implementation of a cursor and makes it easy to retrieve a set of column values for a set of rows upon which logical operations can be performed.

The statement list within a FOR statement is repeated as long as the CURSUR which it fetch is not nil. statement_list consists of one or more statements. A FOR statement

can be labeled. end_label cannot be given unless begin_label also is present. If both are present, they must be the same.

➲ **Syntax: FOR statement syntax**

```
FOR [<for loop variable name> AS]
[<cursor name>[<cursor sensitivity>] CURSOR FOR]
<cursor specification>
DO
<sql statement list>
END FOR
```

For statement syntax supports the following four forms:

- FOR x AS select * from t1
- FOR x AS cur CURSOR FOR select * from t1
- FOR cur CURSOR FOR select * from t1
- FOR select * from t1

NOTE *There must be a* ***select*** *statement in the* ***For*** *statement. The queried table don't need existence when users create a SQL stored procedure.* ***X*** *represents scope of the reference variables. For example, you can use x.c1, x.c2 to represent results of querying table t1, if without x, you only can use c1, c2 instead of reference forms.* ***Cur*** *represents* ***for*** *statement will generate a cursor named cur, the cursor can't be used to reference variables, but can be used in the syntax concerning cursors, such as* **where current of xxxx**, FETCH *etc..*

➲ **Example 1: Calculate the sum of all the values in table t1, then insert them into table t2 in which c1 is the column name of table t1, so you can use both c1 and x.c1 forms to express c1 in the FOR syntax.**

```
CREATE TABLE t1 (c1 INT);
CREATE TABLE t2 (c1 INT);

CREATE PROCEDURE test1(OUT res INT)
LANGUAGE SQL
BEGIN
    SET res = 0;
    FOR x AS select * from t1
    DO
        SET res = res + c1;
        INSERT INTO t2 VALUES (x.c1);
    END FOR;
END;
```

➲ **Example 2: In this example, x is a loop variable name, used to reference variable old; cur is a cursor name, used in statements concerning cursors, for example, the update statement and so on.**

```
CREATE TABLE t1 (c1 INT);

CREATE PROCEDURE test2
LANGUAGE SQL
BEGIN
    FOR x AS cur CURSOR FOR select c1 as old from t1 for update
    DO
        update t1 set c1 = x.old + 100 where current of cur;
    END FOR;
END;
```

➲ **Example 3: In this example, there is no loop variable name, so users only can use variable old directly; cur is a cursor name, used in statements concerning cursors, for example, the update statement and so on.**

```
CREATE TABLE t1 (c1 INT);

CREATE PROCEDURE test3
LANGUAGE SQL
BEGIN
    FOR cur CURSOR FOR select c1 as old from t1 for update
    DO
        update t1 set c1 = old + 100 where current of cur;
    END FOR;
END;
```

➲ **Example 4: The function is same as above example, but the FOR statement does not use cursor or loop variable name, so x.c1 cannot be used to express.**

```
CREATE TABLE t1 (c1 INT);
CREATE TABLE t2 (c1 INT);

CREATE PROCEDURE test4(OUT res INT)
LANGUAGE SQL
BEGIN
    SET res = 0;
    FOR select * from t1
    DO
        SET res = res + c1;
        INSERT INTO t2 VALUES (c1);
    END FOR;
END;
```

LOOP STATEMENT IN SQL STORED PROCEDURES

The LOOP statement is a special type of looping statement, because has no terminating condition clause. It defines a series of statements that are executed

repeatedly until another piece of logic, generally a transfer of control statement, forces the flow of control to jump to some point outside of the loop.

The LOOP statement is useful when you have complicated logic in a loop which you might need to exit in more than one way, however it should be used with care to avoid instances of infinite loops.

If the LOOP statement is used alone without a transfer of control statement, the series of statements included in the loop will be executed indefinitely or until a database condition occurs that raises a condition handler that forces a change in the control flow or a condition occurs that is not handled that forces the return of the SQL stored procedure.

LOOP implements a simple loop construct, enabling repeated execution of the statement list, which consists of one or more statements. The statements within the loop are repeated until the loop is exited; usually this is accomplished with a LEAVE statement. A LOOP statement can be labeled. end_label cannot be given unless begin_label also is present. If both are present, they must be the same.

➲ **Syntax: LOOP statement syntax**

```
<loop_statement_main> ::=
LOOP <sp_statement_main> END LOOP
```

➲ **Example: The following is an example of an SQL stored procedure that contains a LOOP statement:**

```
CREATE PROCEDURE LOOP_IF (OUTPUT sum INTEGER)
LANGUAGE SQL
BEGIN
        DECLARE p_sum INTEGER;
        DECLARE P_SAL INTEGER;
        DECLARE CUR CURSOR FOR SELECT V1 FROM TB_1;
        SET p_sum = 0;
        SET p_sal = 0;
        OPEN CUR;
        FETCH FROM CUR INTO P_SAL;
        LOOP
```

```
                SET p_sum = p_sum + P_SAL;
                FETCH NEXT FROM CUR INTO P_SAL;
                IF P_SAL=NULL THEN
                        BREAK;
                END IF;
        END LOOP;
        CLOSE CUR;
        SET sum = p_sum;
END;
```

NOTE *The NULL in procedure will be translate as nil in LUA file*

WHILE STATEMENT IN SQL STORED PROCEDURES

The WHILE statement defines a set of statements to be executed until a condition that is evaluated at the beginning of the WHILE loop is false. The while-loop-condition (an expression) is evaluated before each iteration of the loop.

The statement list within a WHILE statement is repeated as long as the search_condition is true. statement_list consists of one or more statements. A WHILE statement can be labeled. end_label cannot be given unless begin_label also is present. If both are present, they must be the same.

➲ **Syntax: WHILE statement syntax**

```
<while_statement_main> ::=
WHILE <condition_name> DO <sp_statement_main> END WHILE
```

➲ **Example 1: The following is an example of an SQL stored procedure with a simple WHILE loop**

```
CREATE PROCEDURE WHILE_LOOP(OUT SUM INTEGER)
LANGUAGE SQL
BEGIN
        DECLARE P_SUM INTEGER;
        DECLARE P_SAL INTEGER;
        DECLARE CUR CURSOR FOR SELECT V1 FROM TB_1;
        SET P_SUM = 0;
```

```
        OPEN CUR;
        FETCH FROM CUR INTO P_SAL;
                WHILE (P_SAL!= NULL) DO
                        SET P_SUM = P_SUM + P_SAL;
                        FETCH NEXT FROM CUR INTO P_SAL;
                END WHILE;
        CLOSE CUR;
        SET SUM = P_SUM;
END;
```

➲ **Example 2: while statement usage**

```
#################################################################################
#
#       Module Name = SUM_WHILE.SP
#       Purpose  = Store Procedure testing program
#                       1. Test keyword "WHILE" in Store Procedure
#       Function = 1. create test table TB_1 and TB_2
#                          2. insert test data into TB_1 and TB_2
#                          3. select data from TB_1 TB_2 and use crusor
get data
#       Use Database : "DBNAME" "SYSADM"  ""
#                   table    : TB_1(V1 INTEGER) TB_2(V1 INTEGER)
#################################################################################
#
CREATE PROCEDURE SUM_WHILE(OUT SUM INTEGER)
LANGUAGE SQL
BEGIN
        DECLARE P_SUM INTEGER;
        DECLARE P_SAL INTEGER;
        DECLARE P_SAL1 INTEGER;
        DECLARE CUR CURSOR FOR SELECT  V1 FROM TB_1;
        DECLARE CUR1 CURSOR FOR SELECT  V1 FROM TB_2;
        SET P_SUM = 0;
        OPEN CUR;
        FETCH FROM CUR INTO P_SAL;
                WHILE(P_SAL != NULL)DO
```

```
                        OPEN CUR1;
                        FETCH FROM CUR1 INTO P_SAL1;
                                WHILE(P_SAL1 != NULL)DO
                                        SET P_SUM = P_SUM + P_SAL1;
                                        FETCH NEXT FROM CUR1 INTO P_SAL1;
                                END WHILE;
                        CLOSE CUR1;
                        FETCH NEXT FROM CUR INTO P_SAL;
                END WHILE;
        CLOSE CUR;
        SET SUM = P_SUM;
END;
The NULL in procedure will be translate as nil in lua file
```

REPEAT STATEMENT IN SQL STORED PROCEDURES

The REPEAT statement defines a set of statements to be executed until a condition that is evaluated at the end of the REPEAT loop is true. The repeat-loop-condition is evaluated at the completion of each iteration of the loop.

With a WHILE statement, the loop is not entered if the while-loop-condition is false at 1st pass. The REPEAT statement is useful alternative; however it is noteworthy that while-loop logic can be rewritten as a REPEAT statement.

The REPEAT and UNTIL statements can be used to create a loop that continues until some logical condition is met.

The REPEAT loop is somewhat easier to maintain because it is more obvious which conditions will cause the loop to terminate. The LEAVE statement in a simple loop could be anywhere, while the UNTIL statement is always associated with the END REPEAT clause at the very end of the loop. Furthermore, we don't need to specify a label for the REPEAT loop since the UNTIL condition is always specific to the current loop. However, we still recommend using labels with REPEAT loops to improve readability, especially if the loops are nested.

A REPEAT loop is always guaranteed to run at least once that is, the UNTIL condition is first evaluated after the first execution of the loop. For loops that should not run even once unless some condition is satisfied.

➲ **Example: The following is an SQL stored procedure that includes a REPEAT statement**

```
CREATE PROCEDURE REPEATS
LANGUAGE SQL
BEGIN
        DECLARE V INTEGER DEFAULT 0;
          REPEAT
                INSERT INTO TB_1 VALUES(1,2,3,4);
                SET V = V+1;
        UNTIL V>10
        END REPEAT;
END;
```

Goto statements

Goto statements are used to skip some statements and jump to any suitable statement. However please use it gingerly. If the goto statement isn't used appropriately, the executing flow will be disordered. When the usage of goto statements is incorrect, the correlative error messages will be put into SQL stored procedure error message files (sp_name.msg).

➲ **Syntax: Goto statement syntax**

```
< goto statement syntax > ::=
goto <label name>;
```

<label name> : stored procedure statements;

Label names can be same with input/output variable names, declared variable names including common declare variables, cursor variables and condition variables.

➲ **Example1: The label name is same with input variable names**

```
create procedure gsp1(in p1 int, out p2 char(30))
language sql
```

```
begin
    set p2='more than or equal 10';
    if p1>=10 then goto p1;
end if;
  set p2=' less than 10';
  p1: set p1=1;
end;
```

➲ **Example2: The label name is same with common declared variable names**

```
create procedure gsp2(in p1 int, out p2 char(30))
language sql
begin
   declare v1 int;
set v1=1;
   set p2='more than or equal 10';
   if p1>=10 then goto v1;
   end if;
   set p2='less than 10';
   v1: set p1=v1;
end;
```

Users can define labels with the same name in different scopes. For labels, the begin/end block and loops that SQLSP supports will define a new scope.

➲ **Example1: Define label variables in different begin/end blocks with the same name.**

```
create procedure gsp5(out p1 int)
language sql
begin
  lab: set p1=1;
  begin
    lab: set p1=2;
  end;
end;
```

➲ **Example2: Define label variables in different loops with the same name.**

```
create procedure gsp6(in p1 int)
```

```
language sql
begin
while p1<5 do
lab: set p1=p1+1;
end while;
while p1<15 do
lab: set p1=p1+1;
end while;
end;
```

The label can be defined in front of goto statements or in back of goto statements.

➲ **Example1: The label is defined in front of goto statements.**

```
create procedure gsp10(in p1 int)
language sql
begin
 lab: set p1=p1+1;
 if p1<10 then goto lab;
end if;
end;
```

➲ **Example2: The label is defined in back of goto statements.**

```
create procedure gsp11(in p1 int, out p2 int)
language sql
begin
 if p1>10 then goto lab;
 end if;
 set p1= p1+1;
 lab: set p2=p1;
end;
```

The goto statement can be used in if statements, loop statements and handler actions of condition handlers.

➲ **Example1: The following is an goto statement used in if statements.**

```
create procedure gsp7(p1 int, out p2 int)
```

```
language sql
begin
  lab:set p1=p1+1;
  If p1<10 then goto lab;
  End if;
  set p2=p1;
end;
```

➲ **Example2: The following is an goto statement used in loop statements.**

```
create procedure gsp8(p1 int, out p2 int)
language sql
begin
  L1:begin
  L2:loop
        set p1=p1+1;
        L3:if p1>5 then goto L4;
        end if L3;
  end loop L2;
  L4:set p2=p1;
  end L1;
end;
```

➲ **Example3: The following is an goto statement used in handler actions of condition handlers.**

```
create procedure gsp9(out p1 int,out p2 int)
language sql
begin
 declare con1 condition for sqlstate 'HY019';
 declare continue handler for con1
 begin
     set p1=0;
     goto label1;
     set p1=1;
     label1: set p2=1;
end;
```

```
declare  i  int;
set i= 3147483647;
end;
```

When you use the goto statement, please note the following error cases:

➲ **Example1:**

The label used in the goto statement must be already defined. In the following example, the label used in the goto statement is not defined.

```
create procedure gsp12(in p1 int, out p2 int)
language sql
begin
  if p1>10 then goto lab;
  end if;
  set p1= p1+1;
  set p2=p1;
end;
```

➲ **Example2:**

The label name can not be repeated in the same scope. In the following example, the label name is repeated in the same scope.

```
create procedure gsp13(in p1 int, out p2 int)
language sql
begin
  if p1>10 then goto lab;
  end if;
  lab:set p1= p1+1;
  lab:set p2=p1;
end;
```

➲ **Example3: In the following example, the label is not in visible scopes.**

```
create procedure gsp14(in p1 int, out p2 int)
language sql
begin
  if p1>10 then goto lab;
```

```
  end if;
  while p1<15 do
  lab:set p1= p1+1;
    end while;
  set p2=p1;
end;
```

There are some restrictions on using the goto statement:

- Using goto statements in handle actions of condition handlers, users must put the handler action statement in a new begin/end block. At the same time, both the goto statement and label should be in the new block.
- Goto statements can't jump between begin blocks and end blocks.
- Goto statements can't jump among loops that SQL stored procedure supports. However in nested loops, goto statements can jump from the inner loop to the outer loop.
- In a loop, goto statements can't jump from the outside to the inside. This syntax is supported in DB2, but DBMaker does not support it.
- For begin/end blocks, goto statements can't jump from the outside to the inside in the block.
- Label can not be used in front of blank statement.

Return statements

With the return statement, SQL stored procedures can exit the procedure executing anywhere, and then an error message with user defined codes and message in dmSQL will be printed.

➲ **Syntax: Return statement syntax**

```
< return statement syntax > ::=
return <code>, <statue>
```

➲ **Example1: The following example shows a return statement in condition statements.**

```
dmSQL> @@create procedure ret_sp1(c1 int, out c2 int)
```

```
        2> language sql
        3> begin
        4> if c1 < 0 then
        5> return  -1, 'error';
        6> end if;
        7> set c2 = c1;
        8> end;@@
  dmSQL >  call ret_sp1(-1, ?);
   ERROR : [DBMaker] return user defined error code and message : -1,
error
```

➲ **Example2: The following example shows a return statement with goto statements.**

```
dmSQL> @@create procedure ret_sp2(c1 int, out c2 int)
        2> language sql
        3> begin
        4 > declare v1 int default 0 ;
        5> if c1 < 0 then
        6> set v1 = -1;
        7> goto LEXIT;
        8> elseif c1 > 0 then
        9> set v1 = 0;
        10> goto LEXIT;
        11> end if;
        12> set c2 = c1;
        13 > LEXIT :
        14> if v1 !=  0 then
        15> return -1, 'error';
        16> end if LEXIT;
        17> end;@@

dmSQL >  call ret_sp2(0, ?);
dmSQL >  call ret_sp2(-1, ?);
ERROR : [DBMaker] return user defined error code and message : -1, error
```

Return statements must be the follow format: return error_code, err_message, and they can be anywhere in SQL stored procedure.

Users can define error codes by any integer constant and error message by any string with single quotes . The error cods and error message are printed with user defined error code and message.

Transfer of control statement

Transfer of control statements are used to redirect the flow of control within an SQL stored procedure. This unconditional branching can be used to cause the flow of control to jump from one point to another point, which can either precede or follow the transfer of control statement. The supported transfer of control statements in SQL stored procedures are:

- ITERATE
- LEAVE

Transfer of control statements can be used anywhere within an SQL stored procedure, however ITERATE and LEAVE are generally used in conjunction with a LOOP statement or other looping statements.

ITERATE STATEMENT IN SQL STORED PROCEDURES

The ITERATE statement is used to cause the flow of control to return to the beginning of a labeled LOOP statement.

➲ **Example: The following is an example of an SQL stored procedure that contains an ITERATE statement**

```
CREATE PROCEDURE ITERATOR
LANGUAGE SQL
BEGIN
        DECLARE v_deptno CHAR(3);
        DECLARE v_deptname VARCHAR(29);
        DECLARE c1 CURSOR FOR SELECT deptno, deptname FROM department
ORDER BY deptno;
```

```
        OPEN c1;
        LOOP
                FETCH c1 INTO v_deptno, v_deptname;
                IF v_deptno = NULL THEN
                        LEAVE;
                ELSEIF v_deptno = 'D11' THEN
                           INSERT INTO department (deptno, deptname)
VALUES ( 'NEW', v_deptname);
                        ITERATE;
                ELSE
                        ITERATE;
                END IF;
        END LOOP;
        CLOSE c1;
END;
```

In the example, the ITERATE statement is used to return the flow of control to the LOOP statement defined with loop when a column value in a fetched row matches a certain value. The position of the ITERATE statement ensures that no values are inserted into the department table.

LEAVE STATEMENT IN SQL STORED PROCEDURES

The LEAVE statement transfers program control out of a loop or a compound statement. This statement can be embedded in an SQL stored procedure or dynamic compound statement. It is not an executable statement and cannot be dynamically prepared.

➲ **Example: The following is an example of an SQL stored procedure that contains an LEAVE statement**

```
CREATE PROCEDURE ITEA(OUT C1 INT)
LANGUAGE SQL
BEGIN
        DECLARE V1 INT;
        DECLARE CUR CURSOR FOR SELECT * FROM T3;
        OPEN CUR;
```

```
        FETCH CUR INTO V1;
        LOOP
        IF V1 = 2 THEN
                SET C1 = 1;
                FETCH NEXT FROM CUR INTO V1;
                ITERATE;
        ELSEIF V1 != NULL THEN
                FETCH NEXT FROM CUR INTO V1;
                ITERATE;
        ELSE
                LEAVE;
        END IF;
        END LOOP;
        CLOSE CUR;
END;
```

Labels and SQL stored procedure compound statements

Labels can optionally be used to name any control statement in an SQL stored procedure, including compound statements and loops. By referencing labels in other statements you can force the flow of execution to jump out of a compound statement or loop or additionally to jump to the beginning of a compound statement or loop. Labels can be referenced by the ITERATE, and LEAVE statements.

Optionally you can supply a corresponding label for the END of a compound statement. If an ending label is supplied, it must be same as the label used at its beginning.

Each label must be unique within the body of an SQL stored procedure.

Labels can also be used to avoid ambiguity if a variable with the same name has been declared in more than one compound statement in the stored procedure. A label can be used to qualify the name of an SQL variable.

➲ **Syntax: The following is the syntax of compound statements**

```
[<label name> COLON] BEGIN [<any statement list>] END [<label name>]
```

➲ **Example:**

```
CREATE PROCEDURE test3
LANGUAGE SQL
L1: BEGIN
    ...
    L2: BEGIN
        ...
    END L2;
END L1;
```

NOTE *<label name> must appear accordingly in compound statements. In addition, compound statements can be used in nested forms.*

➲ **Example: The following is an example of an SQL stored procedure with a simple label**

```
CREATE PROCEDURE LABEL_1(OUT C1 char(20))
LANGUAGE SQL
BEGIN
        DECLARE V1 INT;
        DECLARE CUR CURSOR FOR SELECT * FROM T3;
        OPEN CUR;
        FETCH CUR INTO V1;
        label1: LOOP
        IF V1 = 2 THEN
                SET C1 = 'OK';
                FETCH NEXT FROM CUR INTO V1;
                ITERATE label1;
        ELSEIF V1 != NULL THEN
                FETCH NEXT FROM CUR INTO V1;
                ITERATE label1;
        ELSE
                LEAVE label1;
        END IF;
        END LOOP label1;
        CLOSE CUR;
```

```
END;
```

➲ **Example: The following is an example of an SQL stored procedure with leave label to leave begin/end block**

```
CREATE PROCEDURE LEAVE_2(IN V1 INT, OUT V2 INT)
LANGUAGE SQL
BEGIN
lable1: LOOP
    IF V1 < 0 THEN
      SET V2 = -1;
      LEAVE lable1;
    END IF;

    lable2: BEGIN
      IF V1 > 100 THEN
        SET V2 = -2;
        LEAVE lable1;
      END IF;

      lable3: BEGIN
        IF V1 > 50 THEN
           SET V2 = -3;
           LEAVE lable1;
        END IF;
        SET V1 = V1+1;
      END lable3;

    END lable2;

  END LOOP lable1;
END;
```

Scope checking of common variables

The common variable is defined by base data type, such as INT, CHAR, FLOAT and so on, which are supported by SQL stored procedures. Every variable has its available scope. The variable defined in the previous level can be used in the next level, but the variable defined in the next level cannot be used in the previous level.

In SQL stored procedures, to indicate a new level, you can enclose compound statements by using the BEGIN…END statement.

➲ **Example 1:**

```
CREATE PROCEDURE test4(OUT res1 INT, OUT res2 INT, OUT res3 INT)
LANGUAGE SQL
L1: BEGIN
    DECLARE c1, c2 INT;
    SET c1 = 1; #set a value in level L1
    L2: BEGIN
        DECLARE c1 INT;
        SET c1 = 2;
        SET c2 = 3;
        SET res3 = c1; #C1 is redefined in level L2, so the value in
level L1 is shielded, res3 = 2.
    END L2;

    SET res1 = c1; #When the assignment statements in level L2 are end,
c1 will regains the value in level L1, so res1 = 1.
    SET res2 = c2; #C2 is only defined in L1, and the changes in level
L2 also will affect the level L1, so res2 = 3.

END L1;
```

NOTE *FOR statement also is a new level.*

➲ **Example 2:**

```
CREATE PROCEDURE test5(OUT res INT)
LANGUAGE SQL
BEGIN
    DECLARE c1 INT;
    SET c1 = 10;
    SET res = 0;
    FOR select * from t1
    DO
        SET res = res + c1;
        INSERT INTO t2 VALUES (c1);
    END FOR;

    #As the influence scope is limited in the FOR statement, so the
value of c1 still is 10 when the FOR statement is end.
END;
```

SQLCODE and SQLSTATE variables in SQL stored procedures

To perform error handling or to help you debug your SQL stored procedures, you might find it useful to test the value of the SQLCODE or SQLSTATE values, return these values as output parameters or insert these values into a table to provide basic tracing support.

DBMaker implicitly sets these variables whenever a statement is executed. If a statement raises a condition for which a handle exists, the values of the SQLSTATE and SQLCODE variables are available at the beginning of the handler execution. However, the variables are reset as soon as the first statement in the handler is executed. Therefore, it is common practice to copy the values of SQLSTATE and SQLCODE into local variables in the first statement of the handler. In the following example, a CONTINUE handler for any condition is used to copy the SQLCODE variable into another variable named retcode. The variable retcode can then be used in the executable statements to control procedural logic, or pass the value back as an output parameter.

When you need to check result run the statement, please use the SQLCODE variable firstly, Then you can use SQLSTATE variable to check detailed error message, So you can understand the execution states of the statement accurately.

```
BEGIN
DECLARE retcode INTEGER DEFAULT 0;
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION, SQLWARNING, NOT FOUND SET
retcode = SQLCODE;
END
```

SQLCODE VARIABLE DEFINITION IN SQL STORED PROCEDURES

The return value of the SQLCODE variable in SQL stored procedures is defined as follows:

= 0 success: indicate the running result is success.

= 1 warning: indicate the running result is not normal

= 100 no data found: indicate no data is found in the running result

< 0 negative DBMaker native code

SQLSTATE VARIABLE DEFINITION IN SQL STORED PROCEDURES

Currently, the information SQLSTATE variable included in SQL stored procedure conform to the ODBC 3.0 standard definition. please refer to the *Error and Message Reference User's Guide*.

Condition handlers in SQL stored procedures

Condition handlers in SQL stored procedures: Condition handlers determine the behavior of your SQL stored procedure when a condition occurs. You can declare one or more condition handlers in your SQL stored procedure for general conditions, named conditions, or specific SQLSTATE values.

If a statement in your SQL stored procedure raises an SQLWARNING or NOT FOUND condition, and you have declared a handler for the respective condition, DBMaker passes control to the corresponding handler. If you have not declared a handler for such a condition, DBMaker passes control to the next statement in the SQL stored procedure body. If the SQLCODE and SQLSTATE variables have been declared, they will contain the corresponding values for the condition.

If a statement in your SQL stored procedure raises an SQLEXCEPTION condition, and you declared a handler for the specific SQLSTATE or the SQLEXCEPTION condition, DBMaker passes control to that handler. If the SQLSTATE and SQLCODE variables have been declared, their values after the successful execution of a handler will be '00000' and 0 respectively.

If a statement in your SQL stored procedure raises an SQLEXCEPTION condition, and you have not declared a handler for the specific SQLSTATE or the SQLEXCEPTION condition, DBMaker terminates the SQL stored procedure and returns to the caller.

5.8 Returning result sets from SQL stored procedures

In SQL stored procedures, cursors can be used to do more than iterate through rows of a result set. They can also be used to return result sets to the calling program.

To return a result set from an SQL stored procedure, you must:

1. DECLARE the cursor using the WITH RETURN clause
2. Open the cursor in the SQL stored procedure
3. Keep the cursor open for the client application - do not close it

➲ **Example 1: The following is an example of an SQL stored procedure with a simple return**

```
CREATE PROCEDURE call_ret
LANGUAGE SQL
BEGIN
DECLARE cur CURSOR WITH RETURN FOR select * from tb;
OPEN cur;
END;
```

➲ **Example 2:**

```
#############################################################################
####
#       Module Name = RETU.SP
#       Purpose  = Store Procedure testing program
#                  1. Test keyword "RETURN" in Store Procedure
#       Function = 1. decalar r1~r14 for pass parameter
#                  2. create table tb_1 and insert data into tb_1
#                  3. use cursor get data from tb_1
#       Use Database : DBNAME
#       table : TB_1(V1 int,V2 smallint,V3 INT,V4 FLOAT,V5 DOUBLE,
#                    V6 DECIMAL(20,4), V7 BINARY(10),V8 CHAR(20),
#                    V9 VARCHAR(20),V10 NCHAR(40),
#                    V11 NVARCHAR(40),V12 DATE,V13 TIME,V14 TIMESTAMP)
```

```
#########################################################################
####
CREATE PROCEDURE RETU
LANGUAGE SQL
BEGIN
        DECLARE cur CURSOR WITH RETURN FOR select * from tb_1;
        OPEN CUR ;
END;
```

The following is the result of call RETU:

```
dmSQL> call retu;
V1  V2  V3  V4  V5   V6   V7  V8  V9  V10  V11   V12 V13 V14
=== === === === === ==== ==== === ========== ====== === === ====
  1   2   3 *00 *00 2.33 626* ch* va* 6e0063006* 6e0076006* 20* 11* 200*
  1   2   3 *00 *00 2.33 626* ch* va* 6e0063006* 6e0076006* 20* 11* 200*
  1   2   3 *00 *00 2.33 626* 汉* va* 6e0063006* 6e0076006* 20* 11* 200*
  1   2   3 *00 *00 2.33 626* 汉* va* 6e0063006* 6e0076006* 20* 11* 200*
  1   2   3 *00 *00 2.33 626* 汉* va* 6e0063006* 6e0076006* 20* 11* 200*
  1   2   3 *00 *00 2.33 626* ch* 汉* 6e0063006* 6e0076006* 20* 11* 200*
  1   2   3 *00 *00 2.33 626* ch* va* 6e0063006* 6e0076006* 20* 11* 200*
  1   2   3 *00 *00 2.33 626* ch* va* 6e0063006* ba00ba00d* 20* 11* 200*
  1   2   3 *00 *00 2.33 626* ch* va* 6e0063006* 6e0076006* 20* 11* 200*
  1   2   3 *00 *00 2.33 626* ch* va* ba00ba00d* 6e0076006* 20* 11* 200*
  1   2   3 *00 *00 2.33 626* ch* va* 6e0063006* 6e0076006* 20* 11* 200*
11 rows selected
```

5.9 Return status of SQL stored procedure

Status code reflects stored procedure's status, success or not. and user cannot define status code in stored procedure.

Statues code:

-1: the stored procedure execute error

0: the stored procedure execute OK

1: the stored procedure excute have warning

If you want to return status of stored procedure, you should need add 'RETURN STATUS' before LANGUAGE SQL,

➲ **Example1: The following is an example of an SQL stored procedure with return status.**

```
CREATE PROCEDURE ret_status RETURN STATUS
LANGUAGE SQL
BEGIN

END;
```

5.10 Dynamic SQL Stored Procedure

EXECUTE IMMEDIATE statement

You can dynamically prepare and execute a EXECUTE IMMEDIATE statement.

➲ **Syntax: EXECUTE IMMEDIATE statement syntax**

```
<execute immediate statement main> ::=
EXECUTE IMMEDIATE <SQL statement variable>
```

The declared type of <SQL statement variable> shall be character string.

➲ **Example:**

```
CREATE PROCEDURE dym_test1
LANGUAGE SQL
BEGIN
        DECLARE str char(128);
        SET str= 'insert into t1 values(5)';
        EXECUTE IMMEDIATE str;
END;
```

PREPARE statement

Prepare a statement for execution.

➲ **Syntax: PREPARE statement syntax**

```
<prepare statement main> ::=
PREPARE <SQL statement name> FROM <SQL statement variable>
```

➲ **Example:**

```
CREATE PROCEDURE dym_test2
LANGUAGE SQL
BEGIN
        DECLARE str char(128);
        SET str= 'insert into t1 values(5)';

        PREPARE stmt FROM str;
        EXECUTE stmt;
        DEALLOCATE PREPARE stmt;
END;
```

EXECUTE statement

Associate input SQL parameters and output targets with a prepared statement and execute the statement.

➲ **Syntax: EXECUTE statement syntax**

```
<execute statement> ::=
EXECUTE <SQL statement name> [ <result using clause> ] [ <parameter
using clause> ]
<result using clause> ::= INTO <into argument> [ { <comma> <into
argument> }... ]
<parameter using clause> ::= USING <using argument> [ { <comma> <using
argument> }... ]
<into argument> ::= SQL SP variable
<using argument> ::= SQL SP variable
```

➲ **Example 1:**

```
CREATE PROCEDURE dym_test3(IN val INT)
LANGUAGE SQL
BEGIN
        DECLARE str char(128);
        SET str= 'insert into t1 values(?)';

        PREPARE stmt FROM str;
        EXECUTE stmt USING val;
        DEALLOCATE PREPARE stmt;
END;
```

➲ **Example 2:**

```
CREATE PROCEDURE dym_test4(IN val INT, OUT co INT)
LANGUAGE SQL
BEGIN
        DECLARE str char(128);
        SET str= 'select COUNT(*) from t1 where c1=?';

        PREPARE stmt FROM str;
        EXECUTE stmt INTO co USING val;
        DEALLOCATE PREPARE stmt;
END;
```

DEALLOCATE PREPARE statement

Deallocate SQL-statements that have been prepared with a prepare statement.

➲ **Syntax: DEALLOCATE PREPARE statement syntax**

```
<deallocate prepared statement> ::= DEALLOCATE PREPARE <SQL statement
name>
```

➲ **Example :**

```
CREATE PROCEDURE dym_test2
LANGUAGE SQL
```

```
BEGIN
        DECLARE str char(128);
        SET str= 'insert into t1 values(5)';

        PREPARE stmt FROM str;
        EXECUTE stmt;
        DEALLOCATE PREPARE stmt;
END;
```

Dynamic declare cursor

Declare a cursor to be associated with a statement name, which may in turn be associated with a cursor.

➲ **Syntax: Dynamic declare cursor syntax**

```
<dynamic declare cursor> ::= DECLARE <dynamic cursor name> [[NO] SCROLL]
CURSOR [WITH RETURN] FOR <prepare statement name>
```

Dynamic open cursor

Associate input dynamic parameters with a cursor and open the cursor.

➲ **Syntax: Dynamic open cursor syntax**

```
<dynamic open statement> ::= OPEN <dynamic cursor name> [ <input using
clause> ]
```

➲ **Example 1: dynamic cursor**

```
CREATE PROCEDURE dym_test5(OUT sum INT)
LANGUAGE SQL
BEGIN
        DECLARE val INT;
        DECLARE str char(128);
        DECLARE CONTINUE HANDLER FOR NOT FOUND;

        SET str= 'select * from t1';
        SET sum = 0;
```

```
        PREPARE stmt FROM str;
        DECLARE cur CURSOR FOR stmt;

        OPEN cur;
        WHILE SQLCODE = 0 DO
                        SET sum = sum + val;
                        FETCH cur INTO val;
                END WHILE;
        CLOSE cur;

        DEALLOCATE PREPARE stmt;
END;
```

➲ **Example 2: dynamic cursor, have input parameter**

```
CREATE PROCEDURE dym_test6(IN cd INT, OUT sum INT)
LANGUAGE SQL
BEGIN
        DECLARE val INT;
        DECLARE str char(128);
        DECLARE CONTINUE HANDLER FOR NOT FOUND;

        SET str= 'select * from t1 where c1=?';
        SET sum = 0;

        PREPARE stmt FROM str;
        DECLARE cur CURSOR FOR stmt;

        OPEN cur USING cd;
        WHILE SQLCODE = 0 DO
                        SET sum = sum + val;
                        FETCH cur INTO val;
                        END WHILE;
        CLOSE cur;
```

```
        DEALLOCATE PREPARE stmt;
END;
```

➲ **Example 3: return result set of dynamic cursor**

```
CREATE PROCEDURE dym_test7(IN cd INT)
LANGUAGE SQL
BEGIN
        DECLARE str char(128);
        SET str= 'select * from t1 where c1=?';

        PREPARE stmt FROM str;
        DECLARE cur CURSOR WITH RETURN FOR stmt;

        OPEN cur USING cd;
END;
```

5.11 Temp stored procedure

Temp SP is temporary SQL Stored Procedure There are two kinds of Temp SP, one is local temp stored procedure, and the other is global. Please note that the temp stored procedure can't be created from file .Local Temp SP can be called by the connection which create it, and can't be called by other connections, but other connections can create a local temp SP with the same name. Global temp sp is public to all connections. Other conections can call it without granted execute privilege. When the connection creating the global temp sp disconnect, other connections can't call it any more..

If you create a temp stored procedure in multi-user mode , the temp stored procedure will be dropped when the connection is disconnected; if you create temp procedure in signal user mode, it can be dropped when database is started. When you disconnect the connection which create some temp stored procedures, the temp stored procedure will not be dropped immediately, and it will be dropped in ten minutes. User can

replace temp sp created by him, When database shutdown or database crash, clear all temp sp when database restart.

- Permanent sp and global temp sp can't have the same name.
- Different connections can create local temp procedures with the same name.
- Temp sp can be replaced by permanent sp with the same name. But permanent sp can't be replaced by temp sp.
- If SYSADM create a permanent sp named XX, other user also can create any type sp named XX.

➲ **Syntax::**

```
CREATE [OR REPLACE] [GLOBAL\LOCAL] TEMP PROCEDURE <sp_name>
[RETURNS STATUS]
LANGUAGE SQL
BEGIN
<sp_body>
END;
```

➲ **Example1: create a Temp stored procedure tsp1 and call it**

```
dmSQL> set block delimiter @@;
dmSQL> connect to test sysadm;
dmSQL> @@
                2> /* default is local temp sp*/
                3> create temp procedure tsp1
                4> language sql
                5> begin
                6> end;
                7> @@
```

➲ **Example2: create temp stored procedure tsp2 with out parameter,and call it**

```
dmSQL> @@
                2> /* create local temp sp*/
                3> create local temp procedure tsp2(out c1 int)
                4> language sql
```

```
                5> begin
                6> set c1 = 1;
                7> end;
                8> @@
dmSQL> call tsp2(?);
```

➲ **Example3: create a local temp stored procedure with the same name tsp2，and call it**

```
dmSQL> use 2;
dmSQL> connect to test sysadm;
dmSQL> @@
                2> /* create local temp sp with the same name */
                3> create local temp procedure tsp2(out c1 int)
                4> language sql
                5> begin
                6> set c1 = 2;
                7> end;
                8> @@
dmSQL> call tsp2(?);
```

➲ **Example4: create a global temp stored procedure tsp3**

```
dmSQL> @@
                2> /* create global temp sp by sysadm */
                3> create global temp procedure tsp3
                4> language sql
                5> begin
                6> end;
                7> @@
```

➲ **Example5: Other user create a global temp stored procedure with the same name tsp3**

```
dmSQL> use 3;
dmSQL> connect to test xu;
dmSQL> @@
                2> /* create global temp sp with the same name by sysadm
*/
                3> create global temp procedure tsp3
                4> language sql
```

```
5> begin
6> end;
7> @@
```

5.12 Data Processing

This section introduces SQL stored procedure data processing using some basic examples.

Create an empty SQL stored procedure

➲ **Example : create an empty SQL stored procedure**

```
CREATE PROCEDURE SQLSP
LANGUAGE SQL
BEGIN
END;
```

INSERT statement

The following example shows using the insert statement in SQL stored procedure.

➲ **Example : creating an SQL stored procedure, complete the table insert data manipulate**

```
################################################################################
####
#       Module Name = INPUTS.SP
#       Purpose  = Store Procedure testing program
#              1. Test INPUT parameter store procedure
#              2. Test all type that can use in Store Procedure
#       Function = 1. create table INPUTS
#       Use Database : "DBNAME" "SYSADM"  ""
#       table : INPUTS(V1 int,V2 BIGINT,V3 smallint,V4 INT,V5 FLOAT,
#                      V6 DOUBLE, V7 DECIMAL(20,4),V8 BINARY(20),
#                       V9 CHAR(20),V10 VARCHAR(20), V11 NCHAR(40),
#                       V12 NVARCHAR(40),V13 DATE,V14 TIME,V15
                TIMESTAMP,V16 REAL)
```

```
##########################################################################
####
CREATE PROCEDURE INPUTS(INPUT V1 int, INPUT V2 BIGINT,
                        INPUT V3 smallint, INPUT V4 INT,
                         INPUT V5 FLOAT,INPUT V6 DOUBLE,
                         INPUT V7 DECIMAL(20,4),INPUT V8 BINARY(20),
                         INPUT V9 CHAR(20),INPUT V10 VARCHAR(20),
                         INPUT V11 NCHAR(40),INPUT V12 NVARCHAR(40),
                         INPUT V13 DATE,INPUT V14 TIME,
                         INPUT V15 TIMESTAMP,INPUT V16 REAL)
LANGUAGE SQL
BEGIN
        INSERT INTO INPUTS VALUES(V1,V2,V3,V4,V5,V6,V7,V8,
V9,V10,V11,V12,V13,V14,V15,V16);
END;
```

Select statement

The following example shows using the select statement in an SQL stored procedure.

➲ **Example :**

```
##########################################################################
####
#       Module Name = OUTPUTS.SP
#       Purpose  = Store Procedure testing program
#               1. Test OUTPUT parameter store procedure
#               2. Test all type which can use in Store Procedure
#       Function = 1. create table OUTPUTS and insert data into OUTPUTS
#       2. use cursor get data from OUTPUTS and pass data  to OUTPUT
parameter
#       Use Database : DBNAME
#       table : OUTPUTS(V1 int, V2 BIGINT, V3 smallint,V4 INT,
#                        V5 double,V6 DOUBLE, V7 DECIMAL(20,4),
#                       V8 BINARY(10),V9 CHAR(20), V10 VARCHAR(20),
#                       V11 NCHAR(40),V12 NVARCHAR(40),
#                        V13 DATE,V14 TIME,V15 TIMESTAMP,V16 REAL)
```

```
##########################################################################
####

CREATE PROCEDURE OUTPUTS(OUTPUT V1 int, OUTPUT V2 BIGINT,
                         OUTPUT V3 smallint, OUTPUT V4 INT,
                          OUTPUT V5 FLOAT,OUTPUT V6 DOUBLE,
                         OUTPUT V7 DECIMAL(8,4),OUTPUT V8 BINARY(20),
                         OUTPUT V9 CHAR(20),OUTPUT V10 VARCHAR(20),
                         OUTPUT V11 NCHAR(40),
                         OUTPUT V12 NVARCHAR(40),OUTPUT V13 DATE,
                         OUTPUT V14 TIME,OUTPUT V15 TIMESTAMP,OUTPUT V16
REAL)
LANGUAGE SQL
BEGIN
        DECLARE CUR CURSOR FOR select * from OUTPUTS;

        OPEN CUR;
        FETCH FROM CUR INTO
V1,V2,V3,V4,V5,V6,V7,V8,V9,V10,V11,V12,V13,V14,V15,V16;
        CLOSE CUR;
END;
```

Create statement

The following example demonstrates how to use the create statement in SQL stored procedure.

➲ **Example 1: Using SQL stored procedure to create table**

```
CREATE PROCEDURE CRETB
LANGUAGE SQL
BEGIN

         CREATE TABLE TB_1(V1 int, V2 BIGINT, V3 smallint,V4 INT,
                          V5 FLOAT,V6 DOUBLE,V7 DECIMAL(8,2),V8 CHAR(20),
                           V9 CHAR(20),V10 VARCHAR(20),V11 CHAR(40),
                           V12 VARCHAR(40),V13 DATE,V14 TIME,
```

```
                                V15 TIMESTAMP,V16 REAL);
END;
```

➲ **Example 2: Using SQL stored procedure to create view**

```
CREATE PROCEDURE CREVE
LANGUAGE SQL
BEGIN
        CREATE VIEW VE_1 AS SELECT * FROM TB_1;
END;
```

➲ **Example 3: Using SQL stored procedure to create user-defined date type**

```
CREATE PROCEDURE CREDM
LANGUAGE SQL
BEGIN
        CREATE DOMAIN TYP_1 VARCHAR(35);
END;
```

➲ **Example 4: Using SQL stored procedure to create index**

```
CREATE PROCEDURE CREIND
LANGUAGE SQL
BEGIN
        CREATE UNIQUE INDEX IND_1 ON TB_1(V1);
END;
```

Drop statement

The following example demonstrates how to use the drop statement in SQL stored procedure.

➲ **Example 1: Using SQL stored procedure to drop table**

```
CREATE PROCEDURE DRPTB(IN V1 CHAR(20),OUT WORNING CHAR(40))
LANGUAGE SQL
BEGIN
        IF V1 = 'DROP TABLE' THEN
                DROP TABLE TB_1;
                SET WORNING ='NORMAL! TABLE TB_1 DROPED';
```

```
        ELSEIF V1 = 'CREATE TABLE' THEN
                CALL CRETB;
                SET WORNING ='NORMAL! CREATE A NEW TABLE NAMED TB_1';
        ELSE
                SET WORNING = 'YOU INPUT WRONG PARAMETER!';
        END IF;
END;
```

➲ **Example 2: Using SQL stored procedure to drop view**

```
CREATE PROCEDURE DRPVE(IN V1 CHAR(20),OUT WORNING CHAR(40))
LANGUAGE SQL
BEGIN
        IF V1 = 'DROP VIEW VE_1' THEN
                DROP VIEW VE_1;
                SET WORNING ='NORMAL! VIEW VE_1 DROPED';
        ELSEIF V1 = 'CREATE VIEW' THEN
                CALL CREVE;
                SET WORNING ='NORMAL! CREATE VIEW AS SELECT V1 FROM
TB_1';
        ELSE
                SET WORNING = 'YOU INPUT WRONG PARAMETER!';
        END IF;
END;
```

Tracking SQL stored procedure execution

Trace functionality to help users trace the execution of SQL stored procedures for debugging. Turn on and use the TRACE function to place variables for tracing and print messages. After the SQL stored procedure executes, all trace information will be written to a file named **_sptrace.log** in the DBMaker bin directory .

➲ **Syntax: TRACE statement syntax**

```
<TRACE statement main> ::=
TRACE [ON <trace_file> | OFF | <trace_detail>]
```

The following is TRACE usage form:

```
TRACE('V1=', V1);
TRACE('V2=', V2, 'V3=', V3);
```

Expression must use single quotes “ ’ ” and support all data types, but the BINARY, NCHAR, NVARCHAR data types are shown in hexadecimal format.

You can open and use the TRACE function to set tracking variables and output results. When the SQL stored procedure execution, all trace information will be stored in spusr.log file, this file is placed in the DBMaker client’s dmconfig.ini file **DB_SPlog** defined directory.

➲ **Example:**

```
CREATE PROCEDURE INPUTS(INPUT V1 int, INPUT V2 BIGINT,
                        INPUT V3 smallint,INPUT V4 INT,
                        INPUT V5 FLOAT,INPUT V6 DOUBLE,
                        INPUT V7 DECIMAL(20,4),
                        INPUT V8 BINARY(20),INPUT V9 CHAR(20),
                        INPUT V10 VARCHAR(20),INPUT V11 NCHAR(40),
                        INPUT V12 NVARCHAR(40),INPUT V13 DATE,
                        INPUT V14 TIME, INPUT V15 TIMESTAMP,INPUT V16
REAL)
LANGUAGE SQL
BEGIN
        TRACE ON;
        TRACE(‘V1=‘, V1);
        TRACE(‘V2=‘, V2);
        TRACE(‘V3=‘, V3);

        TRACE(‘V4=‘, V4);
        TRACE(‘V5=‘, V5);

        TRACE(‘V8=‘, V8);
        TRACE(‘V9=‘, V9);

        TRACE(‘V12=‘, V12);
        TRACE(‘V13=‘, V13);
```

```
        TRACE('V14=', V14);
        TRACE('V15=', V15);
        TRACE('V16=', V16);
        TRACE OFF;
                INSERT INTO INPUTS
VALUES(V1,V2,V3,V4,V5,V6,V7,V8,V9,V10,V11,V12,V13,V14,V15,V16);
END;
```

After call input(1,7396,2,3,4,5,6,'binary','char','varchar','NCHAR','NVARCHAR','2008-01-01','11:11:11','2008-01-01 11:11:11',' 123.456') _sptrace.log adds log as follows

```
INPUTS 47: Begin trace ====>
INPUTS 48: V1=1
INPUTS 49: V2=7396
INPUTS 50: V3=2
INPUTS 51: V4=3
INPUTS 52: V5=4
INPUTS 53: V8=62696e617279
INPUTS 54: V9=char
INPUTS 55: V12=4e00560041005200430048004100 5200
INPUTS 56: V13=2008-01-01
INPUTS 57: V14=11:11:11
INPUTS 58: V15=2008-01-01 11:11:11.000
INPUTS 59: V16=123.4560012817383
```

6 Working with SQL Stored Procedures

Developing SQL stored procedures is similar to developing other types of stored procedures. Development of SQL stored procedures covers all of the steps required from the design stage to the deployment stage.

Here, we introduce the use of SQL stored procedures from the following chapters.

- Creating SQL stored procedures
- Executing SQL stored procedures
- Dropping SQL stored procedures

To assist you in developing SQL stored procedures, several examples of SQL stored procedures are available for reference.

6.1 Creating SQL Stored Procedures

Creating SQL stored procedures requires an understanding of your requirements, SQL stored procedure features, how to use the SQL features, and knowledge of any restrictions that might impede your design.

Creating SQL stored procedures is similar to creating any database object in that it consists of executing a SQL statement. SQL stored procedures are created by executing the CREATE PROCEDURE statement which can be done using graphical user interface tools (JDBA Tool)or by directly executing the statement from the DBMaker Command Line Tool (dmSQL Tool).

Create SQL stored procedure from file

First, write the SQL stored procedure and save it to a file, then use DBMaker tools like dmSQL to store this SQL stored procedure in the database.

NOTE *DBMaker SQL stored procedure can only by call an external file (*.sp) to create, can not directly write it in the dmSQL command line tool.*

➲ **Syntax: create stored procedure syntax**

```
<CREATE SQL PROCEDURE syntax> ::=
CREATE PROCEDURE FROM <file_name>
```

➲ **Example: to create an SQL stored procedure by reference**

```
dmSQL> CREATE PROCEDURE FROM 'CRETB.SP';
dmSQL> CREATE PROCEDURE FROM '.\SPDIR\CRETB.SP';
dmSQL> CREATE PROCEDURE FROM 'D:\DATABASE\SPDIR\CRETB.SP';
```

The above examples show how to create SQL stored procedure from file using dmSQL tool.

Create SQL stored procedure in script

In DBMaker 5.3, users can create SQLSP not only from files, but also in dmSQL directly Users can call and drop the SQLSP which owned to himself, and execute SQLSP which granted privilege.

SQLSP contains more than one SQL statements, and each statement is end of ';'. So dmSQL must support block delimiter. Block delimiter can be a combination of a-z, A-Z, @, %, ^, and contains two characters at least and seven characters at most, but can't be set/block/delimiter. We don't forbid user to set block delimiter as other keywords

in SQL statement (create, table…), but we suggest user use non-alphanumeric sign like @, % and ^. In block delimiter, ';' doesn't denote end of the input. In addition, users must set block delimiter before writing SQLSP in dmSQL, otherwise, it will return error.

➲ **Syntax: create stored procedure syntax**

No special SQL syntax requirements and it is same as writing SQLSP in files.

```
CREATE [OR REPLACE] PROCEDURE <sp_name>
[RETURN STATUS]
BEGIN
        <sp_body>
END;
```

➲ **Example: create stored procedure syntax in script**

```
dmSQL> set block delimiter @@;
dmSQL> @@
               2> create procedure sp_in_script2
               3> language sql
               4> begin
               5> insert into t1 values(1);
               6> end;
               7> @@
dmSQL> set block delimiter;
```

NOTE *# is not comment character in dmSQL, because table name maybe contains '#', we support --, // for line comment and /**/ for block comment. We also advise users not to use '#' for comment character in SQLSP file, though '#' is comment when users create SQLSP from files.*

NOTE *If the command of creating procedure is more than 4K, dmSQL don't save it as history command, so user can't use history command.*

Using ODBC API

ODBC API supports creating permanent and temp sp, users can write AP with ODBC to support creating any type of the SQL Procedure. For example,

➲ **Example: to create stored procedure syntax with ODBC API**

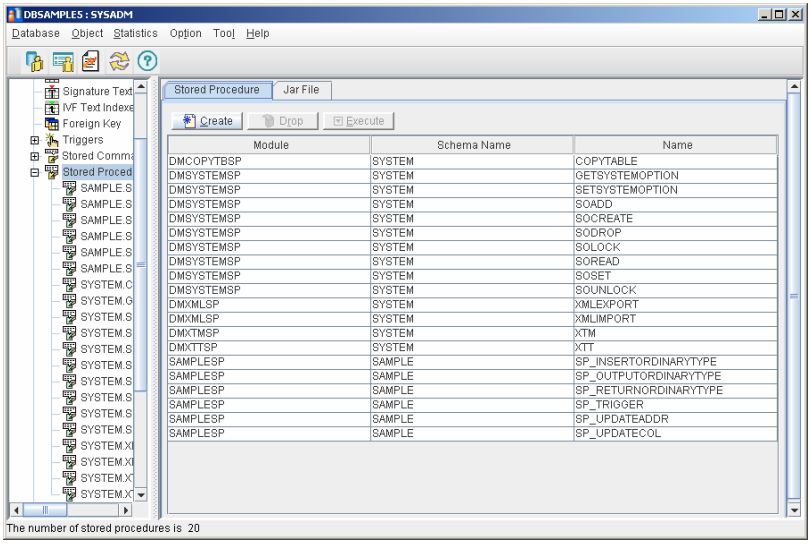
```
#define CRELTSP "create temp procedure tsp1 language sql begin end;"
#define CREGTSP "create global temp procedure tsp2 language sql begin
end;"


int main()
{
                …
                SQLExecDriect(hstmt, CRELTSP, SQL_NTS);


                SQLPrepare(hstmt, CREGTSP, SQL_NTS);
SQLExecute (hstmt);
                …
}
```

Using JDBA Tool

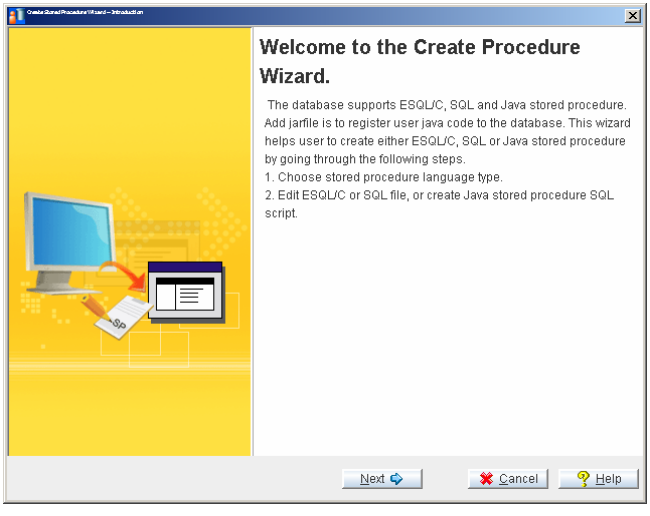
DBMaker provides three languages for creating stored procedures: SQL, ESQL/C and Java. The following is an illustrator for creating a stored procedure using SQL language.

➲ **Creating a SQL stored procedure:**

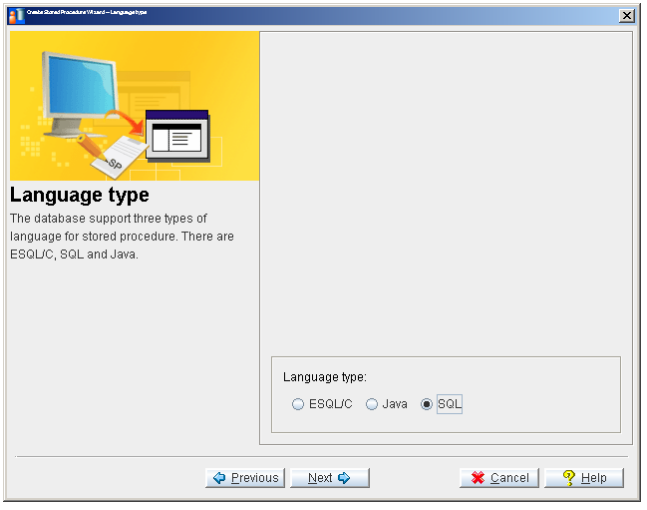
1. Click the object **Stored Procedure** in the tree. The **Stored Procedures** page appears.



2. Click the **Create** button. The Introduction window of the **Create Stored Procedure** wizard appears.

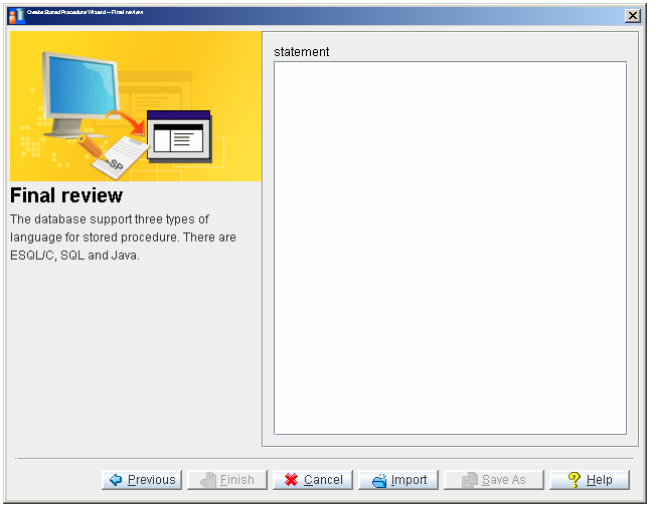


3. Click the **Next** button. The **Language Type** wizard appears.



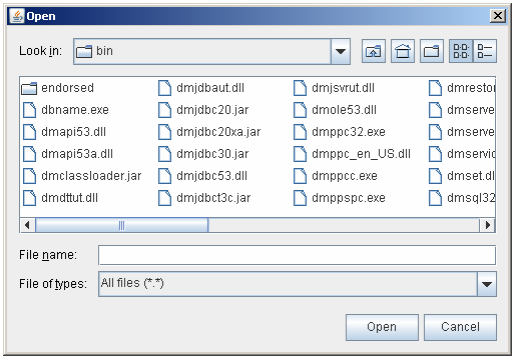
4. Select SQL language to use for writing the stored procedure by clicking the **SQL** radio button.

5. Click the **Next** button. The **Final Review** window appears. Input the SQL statement or click the **Import button** to import a SQL statement from a file.



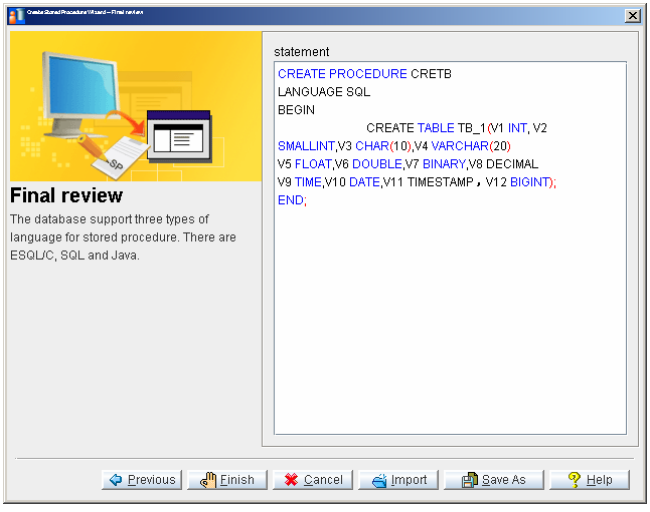
6. Click the **Import** button. The **Open** window appears. Import files from any source, including the SPDIR directory of other databases on the server or

network drives. Enter the path in the **File name** field, or browse the directory tree until the desired path is found.

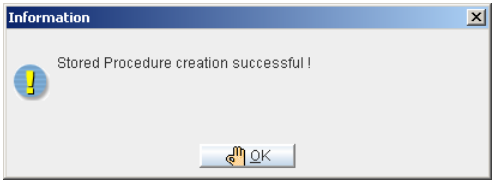


7. Click the **Open** button.

8. The **Final review** window reappears as in the example below if the imported file contains properly formatted (ASCII) text, or if you choose to manually enter the code. Click the **Save As** button to save the stored procedure to another location, or click **Finish** to compile and save the stored procedure in the database.



9. If the SQL stored procedure compiles correctly, the following message appears.



10. Click the **OK** button.

6.2 Executing SQL Stored Procedures

SQL stored procedures are executed by executing the CALL statement which can be done using graphical user interface tools (JDBA Tool)or by directly executing the statement from the DBMaker Command Line Tool (dmSQL Tool).

Executing SQL stored procedures syntax

The CALL statement calls a procedure. This statement can be embedded in an application program or issued through the use of dynamic SQL statements. It is an executable statement that can be dynamically prepared.

You can invoke a stored procedure in dmSQL tool, or using a trigger action. The user must have the privileges required to execute the Call PROCEDURE statement for an SQL stored procedure.

➲ **Syntax: CALL SQL stored procedure syntax**

```
<CALL SQL stored procedure> ::=

CALL <procedure_name> [<variable_name> [ { <comma>
<variable_name> }... ]]
```

➲ **Example 1: call another SQL stored procedures**

```
CREATE  TABLE call_tb(c1 INT);

CREATE PROCEDURE case_test_1(IN inval INT, OUT outval1 INT, OUT outval2
INT)

BEGIN

                DECLARE cur CURSOR WITH RETURN FOR select * from call_tb;
```

```
                OPEN cur;
END;

CREATE PROCEDURE call1(IN inval INT, OUT outval1 INT, OUT outval2 INT)
LANGUAGE SQL
BEGIN
        CALL CASE_TEST_1(inval, outval1, outval2);
END;
```

➲ **Example 2: call other ESQL stored procedures**

```
EXEC SQL CREATE PROCEDURE ecret(INT ct output) RETURNS STATUS;
{
EXEC SQL BEGIN CODE SECTION;
        EXEC SQL SELECT count(*) FROM call_tb INTO :ct;
        exec sql RETURNS STATUS SQLCODE;
        EXEC SQL END CODE SECTION;
}

CREATE PROCEDURE call2(OUT st INT, OUT i2 INT)
LANGUAGE SQL
BEGIN
                SET st = CALL ecret(i2);
END;
```

➲ **Example 3: call JAVA stored procedure**

```
CREATE PROCEDURE CALLJAR
LANGUAGE SQL
BEGIN
        CALL INSERTS_3;        #  INSERTS_3 is a java sp
END;
```

➲ **Example 4: call other stored procedures as cursor**

```
CREATE PROCEDURE call3(OUT i1 INT, OUT i2 INT)
LANGUAGE SQL
BEGIN
```

```
                DECLARE cur CURSOR FOR call call_test;
                OPEN cur;
                FETCH FROM CUR INTO i1, i2;
                CLOSE cur;
END;
```

➲ **Example 5: call other stored procedures as result set**

```
CREATE PROCEDURE call4
LANGUAGE SQL
BEGIN
                DECLARE cur CURSOR WITH RETURN FOR call call_test;
                OPEN cur;
END;
```

➲ **Example 6: call stored procedures in the dmSQL command line tool**

```
dmSQL> create table tb_1(name char(20),phone char(20));
dmSQL> select * from tb_1;
        NAME                 PHONE
==================== ====================

0 rows selected
dmSQL> CREATE PROCEDURE FROM 'D:\SPDIR\INSERT_1.SP';
dmSQL> CALL INSERT_1;
dmSQL> SELECT * FROM TB_1;

        NAME                 PHONE
====================    ====================
JONTH                  1234567

1 rows selected
```

➲ **Example 7: call SQL stored procedure proc1 in standard ODBC program:**

```
dmSQL> create table odbc(c1 char(10),c2 char(10),c3 int);
dmSQL> insert into odbc values(?,?,?);
dmSQL/Val> 'linda','linda',1;
```

```
1 rows inserted
dmSQL/Val> end;

CREATE PROCEDURE proc1(in i1 char(10), in i2 char(10))
LANGUAGE SQL
BEGIN
               DECLARE cur CURSOR with return FOR select c3 from odbc
where c1=i1 and c2=i2;
               OPEN cur;
END;
SQLPrepare(cmdp,(UCHAR*)"call proc1(?, ?)", SQL_NTS);

SQLBindParameter(cmdp, 1, SQL_PARAM_INPUT_OUTPUT, SQL_C_CHAR, SQL_CHAR,
                 20, 0, n1, 20, NULL);

SQLBindParameter(cmdp, 2, SQL_PARAM_INPUT_OUTPUT, SQL_C_CHAR, SQL_CHAR,
                 20, 0, n2, 20, NULL);

SQLBindCol(cmdp, 1, SQL_C_LONG, &i, sizeof(long), NULL);
SQLExecute(cmdp); /* get n2 */

while ((rc=SQLFetch(cmdp))!=SQL_NO_DATA_FOUND) /* fetch result set */
```

Executing SQL stored Procedures by Trigger Action

Users can call a SQL stored procedure by trigger action: first you need create a table for example tb_2(name char(20),phone char(20)), then create a trigger for tb_2 as:

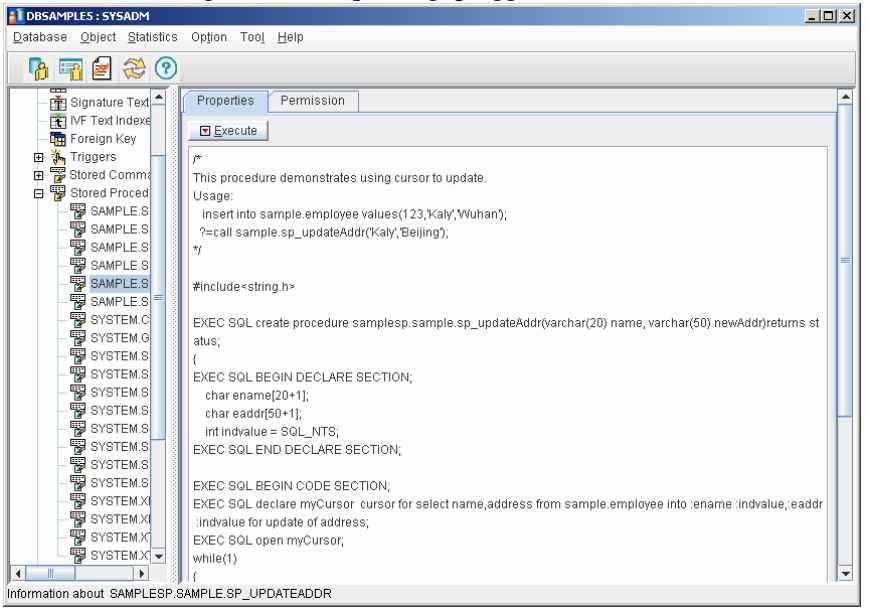
```
CREATE TRIGGER TRG_1 AFTER INSERT ON TB_2 FOR EACH ROW (CALL INSERT_1);
```

Using JDBA Tool

After creating a SQL stored procedure, you can execute it directly or in an application program with JDBA tool.

➲ **Executing a SQL stored procedure:**

1. Open the **Stored Procedure** node and select a SQL stored procedure. The SQL stored procedure **Properties** page appears.



NOTE *Double-clicking a SQL stored procedure in the right panel displays the same window.*

2. Click the **Execute** button. The result of the executed SQL stored procedure appears.

3. Click the **OK** button.

6.3 Dropping SQL Stored Procedures

You can drop a SQL stored procedure using JDBA tool or dmSQL command line tool.

Using dmSQL Tool

➲ **Syntax: dropping SQL stored procedure syntax**

```
<DROP SQL stored procedure> ::=
DROP PROCEDURE <procedure_name>
```

➲ **Example: Dropping SQL stored procedure in the dmSQL command line tool:**

```
dmSQL>  DROP PROCEDURE PROC1;
dmSQL>  DROP PROCEDURE USER1.PROC1;
```

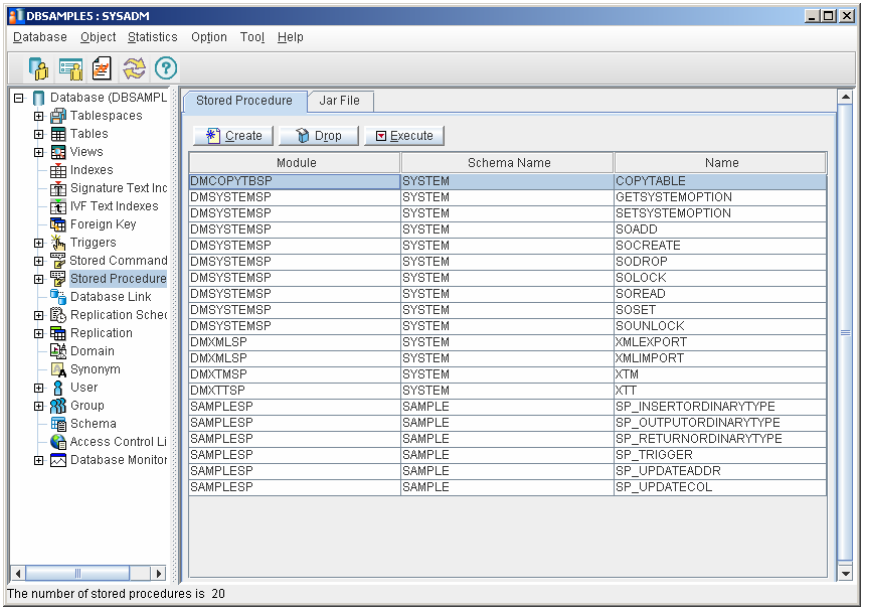
The first statement drops the stored procedure proc1,and the second statement drops the stored procedure user1.proc1.

Using JDBA Tool

If a SQL stored procedure is no longer required, it can be dropped using JDBA tool.

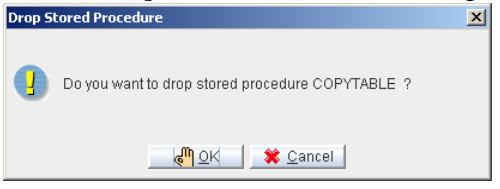
➲ **Example: Dropping a SQL stored procedure:**

1. Click the **Stored Procedure** object in the tree. All the stored procedures in the database appear.



2. Select a SQL stored procedure.

3. Click the **Drop** button. A confirmation dialog box appears.



4. Click the **OK** button to drop the stored procedure or click the **Cancel** button to terminate the dropping process.

6.4 Getting Information about SQL Stored Procedures

You can query the SYSPROCINFO and SYSPROCPARAM system tables to get the SQL stored procedure information by dmSQL tool.

➲ **Example :**

```
dmSQL> select * from sysprocinfo;
dmSQL> select * from SYSPROCPARAM;
```

6.5 Security management

Only the owner or a user with DBA or higher authority can initially execute a stored procedure. Other users can execute the procedure when the execution privilege has been granted to them or a group that the user is a member of. Only owner or a user with DBA or higher authority can grant EXECUTE PROCEDURE privilege on a stored procedure for other users.

➲ **Syntax: GRANT EXECUTE statement syntax**

```
<GRANT EXECUTE syntax> ::=
GRANT EXECUTE ON <COMMAND | PROCEDURE> <executable_name> TO
< <user_name> | <group_name> | <PUBLIC> > [ { <comma> < <user_name> |
<group_name> | <PUBLIC> > }... ]
```

The owner or a user with DBA or higher authority can also revoke execute privilege on a stored procedure for other users.

➲ **Syntax: REVOKE EXECUTE statement syntax**

```
<REVOKE EXECUTE syntax> ::=
REVOKE EXECUTE ON <COMMAND | PROCEDURE> <executable_name> FROM
<<user_name> | <group_name> | <PUBLIC>> [ { <comma> [<user_name> |
<group_name> | <PUBLIC>] }... ]
```

➲ **Example 1:**

user1 creates a stored procedure called **proc1** and grants the execute privilege to **user2** using dmSQL:

```
dmSQL> GRANT EXECUTE ON PROCEDURE proc1 TO user2;
```

➲ **Example 2:**

user1 creates a stored procedure called **proc1** and grants the execute privilege to **PUBLIC** using dmSQL:

```
dmSQL> GRANT EXECUTE ON PROCEDURE proc1 TO PUBLIC;
```

➲ **Example 3:**

user1 revokes the execute privilege from **user2** using dmSQL:

```
dmSQL> REVOKE EXECUTE ON PROCEDURE proc1 FROM user2;
```

➲ **Example 4:**

user1 revokes the execute privilege from **PUBLIC** using dmSQL:

```
dmSQL> REVOKE EXECUTE ON PROCEDURE proc1 FROM PUBLIC;
```

6.6 Configuration Settings for SQL Stored Procedures

When a stored procedure is created, a corresponding dynamic link library is built and stored on the server. By default, the library file is placed in the DBMaker server's working directory. The database administrator can set a preferred path to place the library files for stored procedures using the configuration keyword **DB_SPDir**.

The keyword **DB_SPLog** is used by client users to set the directory they prefer to receive error message files and trace log files, transmitted from the database server while creating or executing stored procedures.

➲ **Example 1:**

To set the default path of dynamic link library files for stored procedures to **/usr1/dbmaker/data/SP** add the following line in the **dmconfig.ini** file:

```
DB_SPDIR=/usr1/DBMaker/data/SP
```

➲ **Example 2:**

To set the stored procedure log file directory to **c:\usr\jerry\data\SP** add the following line in the **dmconfig.ini** file:

```
DB_SPLOG=c:\usr\jerry\data\SP
```

7 SQL Stored Procedures Migration

Stored procedures are easily migrated to other databases using the unload and load commands.

7.1 Unload\Load procedure

```
dmSQL>unload db to dbname;
```

UNLOAD [PROC | PROCEDURE]

```
dmSQL> unload procedure from call to 'd:\spdir\call';
```

After executing this command, the system generates two files named call.b0 and call.so in the d:\spdir\... directory. The call.b0 file stores BLOB data. The call.s0 file stores the script.

LOAD [PROC | PROCEDURE]

```
dmSQL> load procedure from 'd:\spdir\call';
```

After implement the above command, the system loads stored procedures from the specified external file.

8 Restriction on SQL Stored Procedures

There are some limitations in DBMaker SQL Stored Procedure, as follows:

1. If simple expressions contain NULL, the whole expression is NULL. If comparison expressions contain NULL, this comparison expression is false.

2. **0x** format express hexadecimal data format is not supported (0x format is a representation in C language), data format which SQL stored procedure supported can be a decimal and a exponential format.

   ➲ **Example 1: The following is not supported:**

   ```
   SET d1 = 0x1A;
   SET d2 = 0x124C;
   ```

   ➲ **Example 2: The following is supported:**

   ```
   SET d1 = 12.3;
   SET d2 = 1.2345E2;
   ```

3. The format of SQL stored procedure's name not support the *project_name.module_name* and the **MODULENAME** field in *user.SYSPROCINFO* table is NULL.

4. Using SET statement to assign character variable, if the length is over the definition of variable length, truncation will occur:

➲ **Example: If c1value is '1234';**

```
DECLARE c1 char(4);
SET c1 = '1234';
```

5. Using **SET** statement to assign variable will check the type of variables, different attribute types can not be assigned each other, that means users can not assign a character data to a numerical variable, and vice versa. For the assignment between the numerical variables, if necessary assign a decimal to a integer value, it will automatically rounding decimal part.

6. String can only enclosed with two single quotation marks (“”), and with the following conditions:

- If a string have two consecutive "'" , it means that one of the "'" is string content,

➲ **Example**

```
SET c1 = '12345''6789';  -- c1 = 12345'6789
```

- String can have any double quotation marks "" ", it's just a part of the string and will not be interpreted as a string delimiter.

➲ **Example**:

```
SET c1 = '12"34"56"78"9';  -- c1 = 12"34"56"78"9
```

- String can include backslash "\".

➲ **Example**:

```
SET c1 = '12345\6789';  -- c1 = 12345\6789
```

However, if the character is the comment symbol "#" after the backslash, it would be escape as an ordinary character "#".

➲ **Example**:

```
SET c1 = '12345\#6789';  -- c1 = 12345#6789
```

- String can include carriage return line feed.

➲ **Example**:

```
SET c1 = '12345
6789';
```

then

```
c1 = 12345
6789
```

- If the character is the backslash "\" before the carriage return line feed symbol, it denotes the carriage return line feed will be removed, and connect the upper and lower two lines at the same time.

➲ **Example**:

```
SET c1 = '12345\
6789';
```

then

```
c1 = 123456789
```

7. Support *SET n1, n2, …. = select c1, c2, … from t1* syntax, and only get the first record of *select* statement when performing. If the assigned variable number is not equal with the field number in the result set, it will accordance with the following rules:

There are n-variable numbers , m- result set field numbers , if n<=m, then it will get the result set from 1 to n, otherwise it will get all result set, and assign NULL to the extra variables. It needs to check the result set field type whether match the type of corresponding variable, if not match, it will occur error massage.

The most common way is use *count () function* to count.

➲ **Example:**

```
SET sum = select count(*) from t1;
```

8. The modulus operator use SQL function **MOD** () to run.

9. The **DB_SPLog** keyword in the dmconfig.ini file can only be used on the client side, but useless for the **TRACE** information of SQL Stored Procedure, TRACE function only accept the default path which is DBMaker implementation directory on the server side, and information will be written into the **_sptrace.log** file of this directory by default.

10. SQL stored procedure script can use any extension names, even without any extension names.

11. When table name or field name is the same as variable name which in SQL Stored Procedure statement, it needs to bracket the table name or field name with double quotation marks "" to distinguish the variable name.

➲ **Example:**

```
DECLARE c1 INT;
Select "c1" from t1 where "c1" > c1;
```

12. Support all the SQL commands except the following:

ABORT BACKUP

BEGIN BACKUP

BEGIN WORK

END BACKUP

13. If WARNING or ERROR messages are generated when user perform SQL Stored Procedure. By default, WARNING message will be ignored and continue to perform, but the final WARNING message will return to the user The ERROR

message will be discontinued and return to the user. If a WARNING and ERROR messages simultaneous occur, it will only return ERROR message.

14. Not support the following uncertain status:

➲ **Example: intc1, intMax, charc2, charc3 all are variables.**

```
INSERT INTO mytb(c1, c2) VALUES(intc1,
     CASE
        WHEN intc1 <= intMax THEN charc2
     ELSE charc3
END);
```

15. Not support the client syntax for example: *set client_char_set 'big5'*, user can use dynamic SQL to achieve the same syntax

16. The following is SQL Stored Procedure reserved keywords, the variable names can not be set to the following keywordsor the reserved keywords in the '*SQL Command and Function Reference"*. (may have intersection).

ADD, ALTER, AND, AS, ASENSITIVE, BEGIN, BIGINT, BINARY, BREAK, CALL, CASE, CHAR, CLOSE, COMMIT, CONDITION, CONNECT, CONT, CONTINUE, CREATE, CURSOR, DATE, DEALLOCATE, DECIMAL, DECLARE, DEFAULT, DELETE, DISCONNECT, DO, DOUBLE, DROP, DYNAMIC, ELSE, ELSEIF, END, EXECUTE, EXIT, FALSE, FETCH, FIRST, FLOAT, FOR, FOUND, FROM, GO, GOTO, GRANT,HANDLER, HOLD, IF, IMMEDIATE, IN, INOUT, INPUT, INSENSITIVE, INSERT, INT, INTEGER, INTO, IS, ITERATE, LANGUAGE, LAST, LEAVE, LOOP, NCHAR, NCLOB, NEXT, NO, NOT, NULL, NVARCHAR, OFF, ON, OPEN, OR, OUT, UTPUT, PREPARE, PRIOR, PROCEDURE, REPEAT, RESULT, RETURN, RETURNS, ROLLBACK,SCROLL, SELECT, SENSITIVE, SET, SETS, SHORT, SMALLINT, SQL, SQLCODE, SQLEXCEPTION, SQLSTATE, SQLWARNING, TATISTICS, STOP, THEN, TIME, TIMESTAMP, TO, TRACE, TRUE, UNTIL, UPDATE,

USING, VALUE, VARBINARY, VARBPTR, VARCHAR, VARCPTR, WHEN, WHILE, WITH, WITHOUT

17. You can use the backslash '\' in the SQL Stored Procedure, if it followed with carriage return line feed symbol, that denotes the backslash '\' will be removed and connect the upper and lower two lines at the same time.

18. INOUT parameters are not supported.