

Altibase Application Development

Stored Procedures Manual

Release 6.1.1

April 23, 2012



Altibase Application Development Stored Procedures Manual

Release 6.1.1

Copyright © 2001~2012 Altibase Corporation. All rights reserved.

This manual contains proprietary information of Altibase® Corporation; it is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright, patent and other intellectual property law. Reverse engineering of the software is prohibited.

All trademarks, registered or otherwise, are the property of their respective owners.

Altibase Corporation

10F, Daerung PostTower II, 182-13,

Guro-dong Guro-gu Seoul, 152-847, Korea

Telephone: +82-2-2082-1000 Fax: 82-2-2082-1099

E-mail: support@altibase.com [www: http://www.altibase.com](http://www.altibase.com)

Contents

Preface	i
About This Manual	ii
Audience.....	ii
Software Environment.....	ii
Organization.....	ii
Documentation Conventions	iii
Sample Schema.....	vi
Related Reading	vi
Online Manuals	vi
Altibase Welcomes Your Comments	vi
1. Introduction	1
1.1 Overview	2
1.1.1 Definition	2
1.1.2 The Kinds of Stored Objects.....	2
1.1.3 Features.....	2
1.2 Structure of Stored Procedures.....	5
1.3 Considerations when using Stored Procedures	6
1.3.1 Transaction Management.....	6
1.3.2 Limitations.....	6
1.3.3 Related Meta Tables.....	6
2. SQL Statements for Managing Stored Procedures.....	7
2.1 Overview	8
2.1.1 The SQL Statements that are used with Stored Procedures	8
2.1.2 Data Types	9
2.2 CREATE PROCEDURE	10
2.2.1 Syntax.....	10
2.2.2 Purpose.....	10
2.2.3 Example.....	12
2.2.4 Limitation.....	15
2.3 ALTER PROCEDURE	16
2.3.1 Syntax.....	16
2.3.2 Purpose.....	16
2.3.3 Example.....	16
2.4 DROP PROCEDURE	18
2.4.1 Syntax.....	18
2.4.2 Purpose.....	18
2.4.3 Example.....	18
2.5 EXECUTE	19
2.5.1 Syntax.....	19
2.5.2 Purpose.....	19
2.5.3 Example.....	19
2.6 CREATE FUNCTION	21
2.6.1 Syntax.....	21
2.6.2 Purpose.....	21
2.6.3 Example.....	22
2.6.4 Limitations.....	23
2.7 ALTER FUNCTION.....	24
2.7.1 Syntax.....	24
2.7.2 Purpose.....	24
2.7.3 Example.....	24
2.8 DROP FUNCTION	25
2.8.1 Syntax.....	25
2.8.2 Purpose.....	25
2.8.3 Example.....	25
3. Stored Procedure Blocks	27
3.1 Overview	28

3.1.1 Syntax	28
3.1.2 Declare Section	29
3.1.3 Block Body	29
3.1.4 Exception Handler Section	30
3.2 Declaring Local Variables	31
3.2.1 Syntax	31
3.2.2 Purpose	32
3.2.3 Examples	34
3.3 SELECT INTO	37
3.3.1 Syntax	37
3.3.2 Purpose	37
3.3.3 Example	37
3.4 Assignment Statements	42
3.4.1 Syntax	42
3.4.2 Purpose	42
3.4.3 Examples	42
3.5 LABEL	44
3.5.1 Purpose	44
3.5.2 Limitations	44
3.6 PRINT	46
3.6.1 Syntax	46
3.6.2 Purpose	46
3.6.3 Examples	46
3.7 RETURN	49
3.7.1 Syntax	49
3.7.2 Purpose	49
3.7.3 Example	49
4. Control Flow Statements	51
4.1 Overview	52
4.1.1 Syntax	52
4.2 IF	53
4.2.1 Syntax	53
4.2.2 Purpose	53
4.2.3 Examples	54
4.3 CASE	57
4.3.1 Syntax	57
4.3.2 Purpose	57
4.3.3 Example	58
4.4 LOOP	60
4.4.1 Syntax	60
4.4.2 Purpose	60
4.4.3 Example	60
4.5 WHILE LOOP	62
4.5.1 Syntax	62
4.5.2 Purpose	62
4.5.3 Example	62
4.6 FOR LOOP	64
4.6.1 Syntax	64
4.6.2 Purpose	64
4.6.3 Example	65
4.7 EXIT	69
4.7.1 Syntax	69
4.7.2 Purpose	69
4.7.3 Example	70
4.8 CONTINUE	72
4.8.1 Syntax	72
4.8.2 Purpose	72
4.8.3 Example	72
4.9 GOTO	74

4.9.1 Syntax.....	74
4.9.2 Purpose.....	74
4.9.3 Limitations.....	74
4.10 NULL.....	77
4.10.1 Syntax	77
4.10.2 Purpose	77
4.10.3 Example	77
5. Using Cursors	79
5.1 Overview	80
5.1.1 Declaring a Cursor	80
5.1.2 Cursor Management Using OPEN, FETCH, and CLOSE.....	80
5.1.3 Cursor Management Using a Cursor FOR LOOP	81
5.2 CURSOR.....	82
5.2.1 Syntax.....	82
5.2.2 Purpose.....	82
5.2.3 Example	83
5.3 OPEN	85
5.3.1 Syntax.....	85
5.3.2 Purpose.....	85
5.3.3 Examples	86
5.4 FETCH.....	88
5.4.1 Syntax.....	88
5.4.2 Purpose.....	88
5.4.3 Example	89
5.5 CLOSE.....	90
5.5.1 Syntax.....	90
5.5.2 Purpose.....	90
5.5.3 Example	90
5.6 Cursor FOR LOOP	91
5.6.1 Syntax.....	91
5.6.2 Purpose.....	91
5.6.3 Example	92
5.7 Cursor Attributes.....	93
5.7.1 Syntax.....	93
5.7.2 Purpose.....	93
5.7.3 Examples	95
6. User-Defined Types.....	99
6.1 Overview	100
6.1.1 RECORD Types.....	100
6.1.2 Associative Arrays	100
6.1.3 REF CURSOR (Cursor Variable).....	100
6.2 Defining a User-Defined Type.....	101
6.2.1 Syntax.....	101
6.2.2 Examples	102
6.3 Functions for Use with Associative Arrays.....	103
6.3.1 Syntax.....	103
6.3.2 Purpose.....	103
6.3.3 Examples	104
6.4 Using RECORD Type Variables and Associative Array Variables.....	107
6.4.1 Compatibility between User-Defined Types	107
6.4.2 RECORD Type Variable Example	108
6.4.3 Associative Array Examples	108
6.5 REF CURSOR.....	110
7. Typesets	115
7.1 Overview	116
7.1.1 Features	116
7.1.2 Structure.....	116
7.2 CREATE TYPESET	118
7.2.1 Syntax.....	118

7.2.2 Prerequisites	118
7.2.3 Description	118
7.2.4 Examples	118
7.3 DROP TYPESET	120
7.3.1 Syntax	120
7.3.2 Prerequisites	120
7.3.3 Description	120
7.3.4 Example	120
8. Dynamic SQL	121
8.1 Overview	122
8.1.1 Executing Dynamic SQL	122
8.1.2 Features	123
8.2 EXECUTE IMMEDIATE	124
8.2.1 Syntax	124
8.2.2 Description	124
8.2.3 Example	125
8.2.4 Restrictions	125
8.3 OPEN FOR	127
8.3.1 Syntax	127
8.3.2 Description	127
8.3.3 Example	127
9. Exception Handlers	129
9.1 Overview	130
9.1.1 Types	130
9.1.2 Declaring an Exception	131
9.1.3 Raising an Exception	131
9.1.4 The Exception Handler	131
9.2 EXCEPTION	132
9.2.1 Syntax	132
9.2.2 Purpose	132
9.2.3 Example	132
9.3 RAISE	133
9.3.1 Syntax	133
9.3.2 Purpose	133
9.3.3 Example	133
9.4 RAISE_APPLICATION_ERROR	135
9.4.1 Syntax	135
9.4.2 Parameter	135
9.4.3 Purpose	135
9.4.4 Example	135
9.5 User-defined Exceptions	136
9.5.1 User-defined Exception Codes	136
9.5.2 System-defined Exception Codes	137
9.6 SQLCODE and SQLERRM	139
9.7 The Exception Handler	141
9.7.1 Syntax	141
9.7.2 Purpose	141
9.7.3 Example	142
10. Built-in Functions and Stored Procedures	145
10.1 File Control	146
10.1.1 Managing Directories	146
10.1.2 File Control	147
10.1.3 FCLOSE	150
10.1.4 FCLOSE_ALL	151
10.1.5 FCOPY	152
10.1.6 FFLUSH	154
10.1.7 FOPEN	155
10.1.8 FREMOVE	156
10.1.9 FRENAME	157

10.1.10 GET_LINE	159
10.1.11 IS_OPEN	160
10.1.12 NEW_LINE	161
10.1.13 PUT	162
10.1.14 PUT_LINE.....	163
10.1.15 Handling File Control-Related Exceptions	165
10.1.16 File Control Examples	166
10.2 DataPort.....	169
10.2.1 Using DataPort	169
10.2.2 DataPort Stored Procedures and Functions	172
10.2.3 EXPORT_TO_FILE	172
10.2.4 EXPORT_PARTITION_TO_FILE	174
10.2.5 EXPORT_USER_TABLES	175
10.2.6 IMPORT_FROM_FILE.....	177
10.2.7 CLEAR_DP.....	179
10.2.8 RESUME_DP.....	179
10.2.9 REMOVE_DP	180
10.2.10 DataPort Exceptions	181
10.2.11 Examples	182
10.3 DBMS Stat	186
10.3.1 Overview.....	186
10.3.2 DBMS Stat Stored Procedures.....	186
10.3.3 Notes.....	186
10.3.4 GATHER_SYSTEM_STATS.....	186
10.3.5 GATHER_DATABASE_STATS	187
10.3.6 GATHER_TABLE_STATS	188
10.3.7 GATHER_INDEX_STATS	189
10.4 Miscellaneous Functions	191
10.4.1 REMOVE_XID	191
10.4.2 SLEEP	191
AppendixA. Examples.....	193
Stored Procedure Examples.....	193
Example 1	193
Example 2	194
Example 3	196
Example 4	197
Example 5	198
File Control Example	199
Result	200

Preface

About This Manual

This manual explains how to use stored procedures with an ALTIBASE® HDB™.

Audience

This manual has been prepared for the following ALTIBASE HDB users:

- Database administrators
- Application developers
- Programmers

It is recommended that those reading this manual possess the following background knowledge:

- Basic knowledge in the use of computers, operating systems, and operating system utilities
- Experience in using relational databases and an understanding of database concepts
- Computer programming experience

Software Environment

This manual has been prepared assuming that ALTIBASE HDB 5.5.1 will be used as the database server.

Organization

This manual is organized as follows:

- [Chapter1: Introduction](#)

This chapter explains the concept and structure of stored procedures and sets forth some cautions that must be kept in mind when using them.

- [Chapter2: SQL Statements for Managing Stored Procedures](#)

This chapter explains the SQL statements that are used to manage stored procedures.

- [Chapter3: Stored Procedure Blocks](#)

This chapter explains the concept of stored procedure blocks, how to define local variables within the body of stored procedures, and which statements can be used in stored procedures.

- [Chapter4: Control Flow Statements](#)

This chapter explains the control flow statements that can be used to author a procedural program within the body of a stored procedure.

- [Chapter5: Using Cursors](#)

This chapter explains cursor-related statements, which are used to define and control cursors so that multiple records returned by a SELECT statement can be processed within a stored procedure.

- [Chapter6: User-Defined Types](#)

This chapter explains how to define and use records and associative arrays, which are user-defined types that can be used within stored procedures and functions.

- [Chapter7: Typesets](#)

This chapter explains how to define and use user-defined typesets.

- [Chapter8: Dynamic SQL](#)

This chapter explains dynamic SQL, which enables queries to be created and executed as desired by the user at runtime.

- [Chapter9: Exception Handlers](#)

This chapter explains the exception handler, which handles exceptions when an error occurs while a stored procedure is being executed.

- [Chapter10: Built-in Functions and Stored Procedures](#)

This chapter explains various built-in functions, including DataPort, which allows data to be moved between systems, and functions that enable stored procedures to read from and write to text files in the host's disk file system.

- [Appendix A. Examples](#)

This appendix explains the schema and sample program examples used in this manual.

Documentation Conventions





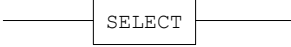
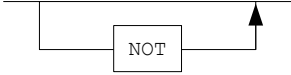
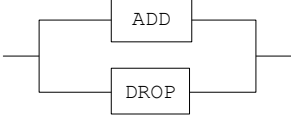
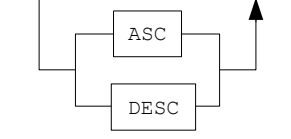
This section describes the conventions used in this manual. Understanding these conventions will make it easier to find information in this manual and in the other manuals in the series.

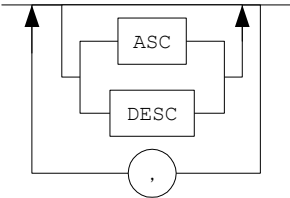
There are two sets of conventions:

- Syntax Diagram Conventions
- Sample Code Conventions

Syntax Diagram Conventions

In this manual, the syntax of commands is described using diagrams composed of the following elements:

Element	Description
	Indicates the start of a command. If a syntactic element starts with an arrow, it is not a complete command.
	Indicates that the command continues to the next line. If a syntactic element ends with this symbol, it is not a complete command.
	Indicates that the command continues from the previous line. If a syntactic element starts with this symbol, it is not a complete command.
	Indicates the end of a statement.
	Indicates a mandatory element.
	Indicates an optional element.
	Indicates a mandatory element comprised of options. One, and only one, option must be specified.
	Indicates an optional element comprised of options.

Element	Description
	Indicates an optional element in which multiple elements may be specified. A comma must precede all but the first element.

Sample Code Conventions

The code examples explain SQL statements, stored procedures, iSQL statements, and other command line syntax.

The following table describes the printing conventions used in the code examples.

Convention	Meaning	Example
[]	Indicates an optional item.	VARCHAR [(size)] [[FIXED] VARIABLE]
{ }	Indicates a mandatory field for which one or more items must be selected.	{ ENABLE DISABLE COMPILE }
	A delimiter between optional or mandatory arguments.	{ ENABLE DISABLE COMPILE } [ENABLE DISABLE COMPILE]
. . . .	Indicates that the previous argument is repeated, or that sample code has been omitted.	iSQL> select e_lastname from employees; E_LASTNAME ----- Moon Davenport Kobain 20 rows selected.
Other symbols	Symbols other than those shown above are part of the actual code.	EXEC :p1 := 1; acc NUMBER(11,2);
Italics	Statement elements in italics indicate variables and special values specified by the user.	SELECT * FROM table_name; CONNECT userID/password;
Lower Case Letters	Indicate program elements set by the user, such as table names, column names, file names, etc.	SELECT e_lastname FROM employees;

Convention	Meaning	Example
Upper Case Letters	Keywords and all elements provided by the system appear in upper case.	DESC SYSTEM_.SYS_INDICES_;

Sample Schema

Some of the examples in this manual are based on sample tables, including the *employees*, *departments* and *orders* tables. These tables can be created using the schema.sql file in the \$ALTIBASE_HOME/sample/APRE/schema directory. For complete information on the sample schema, please refer to the *ALTIBASE HDB General Reference*.

Related Reading

For additional technical information, please refer to the following manuals:

- ALTIBASE HDB Installation Guide
- ALTIBASE HDB Getting Started
- ALTIBASE HDB SQL Reference
- ALTIBASE HDB iSQL User's Manual
- ALTIBASE HDB Error Message Reference

Online Manuals

Online versions of our manuals (PDF or HTML) are available from the Altibase Download Center (<http://atc.altibase.com/>).

Altibase Welcomes Your Comments

Please feel free to send us your comments and suggestions regarding this manual. Your comments and suggestions are important to us, and may be used to improve future versions of the manual.

When you send your feedback, please make sure to include the following information:

- The name and version of the manual that you are using
- Any comments that you have about the manual
- Your full name, address, and phone number

Write to us at the following e-mail address: support@altibase.com

For immediate assistance with technical issues, please contact the Altibase Customer Support Center.

We always appreciate your comments and suggestions.

1 Introduction

1.1 Overview

1.1.1 Definition

A stored procedure is a kind of database object that consists of SQL statements, control statements, assignment statements, exception handlers, etc. Stored procedures are created in advance, compiled, and stored in a database, ready for execution. In that state, stored procedures can be simultaneously accessed by multiple SQL statements.

The term “stored procedure” is sometimes used to refer to stored procedures and stored functions collectively. Stored procedures and stored functions differ only in that stored functions return a value to the calling application, whereas stored procedures do not.

Stored procedures and stored functions can be created using the CREATE PROCEDURE and CREATE FUNCTION statements, respectively. For more information about these statements, please refer to the explanations of the CREATE PROCEDURE and CREATE FUNCTION statements in [Chapter2: SQL Statements for Managing Stored Procedures](#) of this manual.

1.1.2 The Kinds of Stored Objects

1.1.2.1 Stored Procedures

Stored procedures are called, either by SQL statements or by other stored procedures, using IN parameters, OUT parameters, or IN/-OUT parameters. When a stored procedure is called, procedural statements defined in the body of the procedure are executed. A stored procedure has no return value, but can still pass values to the client or calling routine via OUT or IN/OUT parameters. However, because a stored procedure has no return value, it cannot be used as an operand in an expression in an SQL statement.

1.1.2.2 Stored Functions

A stored function is the same as a stored procedure with the exception that it has a return value, and thus can be used as an operand in an expression in an SQL statement.

1.1.2.3 Typesets

A typeset is a set of user-defined types that can be used in stored procedures. They are chiefly used for passing user-defined types between procedures in the form of parameters and return values.

For more information about typesets, please refer to [Chapter7: Typesets](#).

1.1.3 Features

1.1.3.1 Procedural Programming using SQL

The ALTIBASE HDB PSM (Persistent Stored Module) provides control flow statements and exception handlers so that procedural programming can be conducted using SQL statements.

1.1.3.2 Performance

When a client sequentially executes multiple SQL statements, it must send each SQL statement individually and wait for the result before sending the next statement. This increases the amount of time and expense that is required for communication between the server and client. In contrast, a program that is authored such that it uses stored procedures needs to communicate with the server only one time in order to execute multiple SQL statements, because the client only needs to call one stored procedure comprising several SQL statements.

Therefore, using stored procedures reduces communication expenses, and additionally reduces the burden associated with type conversion when different data types are used on the server and client applications.

1.1.3.3 Modularity

All of the SQL operations required to conduct one business action can be gathered together and modularized in the form of a single stored procedure.

1.1.3.4 Easily Maintained Source Code

Because stored procedures reside in the database server, when business logic changes, only the stored procedures need to be changed; there is no need to update client programs distributed among multiple machines.

1.1.3.5 Sharing and Productivity

Stored procedures are stored in the database, which means that one user can execute another user's stored procedures, as long as s/he has been granted suitable access privileges. Moreover, because stored procedures can be called from within other stored procedures, when the need arises for a new business process that is based on an existing business process, the stored procedure for the new business process has only to call the stored procedure for the existing business process, thereby eliminating redundancy and increasing productivity.

1.1.3.6 Integration with SQL

The conditions that are used in the WHERE clause of a SELECT statement can be used as conditions in control flow statements in stored procedures without change. This means that SQL-style functions that are not originally supported for use as conditions in control flow statements in host languages such as C/C++ can now be used. Furthermore, built-in functions that are supported in SQL statements can be used without change in stored procedures.

1.1.3.7 Error Handling in SQL

Because exception handlers are provided for use with stored procedures, appropriate action can be immediately taken on the server in response to errors that occur during the execution of SQL statements.

1.1.3.8 Persistent Storage

Stored procedures are database objects, and thus are permanently stored in the database until explicitly dropped by a user. This means that business logic that supports business practices is also

1.1 Overview

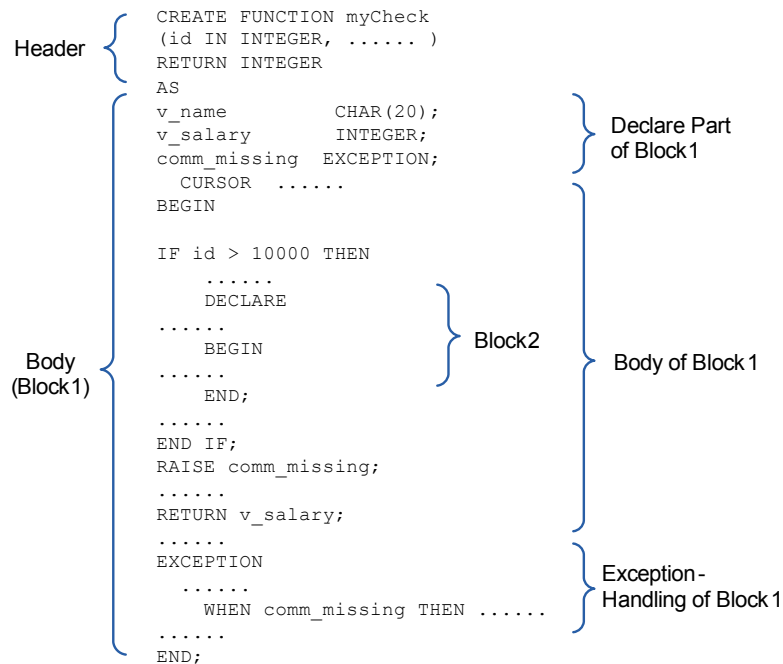
permanently preserved in the database.

1.2 Structure of Stored Procedures

Stored procedures are a kind of block-structured language. The body of one stored procedure typically comprises several logical blocks.

A stored procedure comprises a header and a body. The body of a stored procedure is one large block that comprises a declare section, the actual body of the procedure, and an exception-handling section. The main block can have multiple sub-blocks.

The following is an example illustrating the structure of a stored procedure:



Block2 is a sub-block of Block1 and can have a structure just like that of Block1, including a `DECLARE` section, body and an exception-handling section.

A control flow statement is also a block, in that it has an explicit beginning and ending.

1.3 Considerations when using Stored Procedures

1.3.1 Transaction Management

The transaction control commands that can be used in stored procedures are COMMIT and ROLLBACK statements. The use of these commands within a stored procedure can affect tasks that are being conducted outside of the stored procedure.

For example, assume that the following commands are executed in NON-AUTOCOMMIT mode:

```
iSQL> INSERT INTO t1 values (1);  
iSQL> INSERT INTO t1 values (2);  
iSQL> EXECUTE proc1;
```

Suppose that proc1 contains the commands “Insert Into t1 values (3)” and “ROLLBACK”. When it is executed, the statement within the procedure that inserted the value of 3 is not the only statement that will be rolled back. Additionally, the INSERT statements that were executed directly from iSQL and inserted the values of 1 and 2 will also be rolled back. That is, the two INSERT statements are handled as part of the same transaction as the statements within the stored procedure.

1.3.2 Limitations

It is impossible to execute the COMMIT or ROLLBACK commands while a cursor is open. That is, a transaction that uses cursors must contain an entire cursor control block, comprising OPEN, FETCH, and CLOSE statements.

Stored functions that are called from within SELECT statements cannot contain INSERT, UPDATE, or DELETE statements. Additionally, they cannot contain transaction control statements.

Stored functions that are called from within INSERT, UPDATE or DELETE statements cannot contain transaction control statements.

LOB type variables cannot be declared in the declare section of a stored procedure. Additionally, the %TYPE and %ROWTYPE attributes cannot be used to declare variables when the underlying column in the actual database object is a LOB type column.

Because LOB type variables cannot be declared within stored procedures, data in LOB type columns cannot be fetched using cursors. Therefore, LOB type columns cannot be referenced using cursor control statements.

1.3.3 Related Meta Tables

For information about the meta tables related to stored procedures, please refer to the Data Dictionary in the *General Reference*.

2 SQL Statements for Managing Stored Procedures

2.1 Overview

2.1.1 The SQL Statements that are used with Stored Procedures

This table lists the DDL statements that are used to create and manage stored procedures, stored functions, and typesets.

The descriptions of the CREATE TYPESET and DROP TYPESET statements can be found in [Chapter7: Typesets](#) of this manual.

Statement Type	Statement	Description
Creation	CREATE [OR REPLACE] PROCEDURE statement	This SQL statement is used to create a new stored procedure or change the definition of an existing stored procedure.
	CREATE [OR REPLACE] FUNCTION statement	This SQL statement is used to create a new stored function or change the definition of an existing stored function.
	CREATE [OR REPLACE] TYPESET statement	This SQL statement is used to create or alter a type-set.
Modification	ALTER PROCEDURE statement	This SQL statement is used to recompile a stored procedure to thus optimize the execution plan for the stored procedure. It is used when the definition of one or more objects referenced in the stored procedure has changed since the stored procedure was created, under which circumstances the current execution plan for the stored procedure is not optimal.
	ALTER FUNCTION statement	This SQL statement is used to recompile a stored function and optimize the execution plan for the stored function. It is used when the definition of one or more objects referenced in the stored function has changed since the stored function was created, under which circumstances the current execution plan for the stored function is not optimal.
Removal	DROP PROCEDURE statement	This SQL statement is used to remove a stored procedure.
	DROP FUNCTION statement	This SQL statement is used to remove a stored function.
	DROP TYPESET statement	This SQL statement is used to remove a TYPESET.

Statement Type	Statement	Description
Execution	EXECUTE statement	This SQL statement is used to EXECUTE a stored procedure or stored function.
	<i>function_name</i>	Stored functions can be referenced by name within SQL statements.

2.1.2 Data Types

The following data types are supported for use with stored procedures:

- Primitive Types
 - Basic Data Types

All data types that can be used in SQL statements are available for use in stored procedures. For more information, please refer to the chapter that describes data types in the *General Reference*.
 - BOOLEAN Type

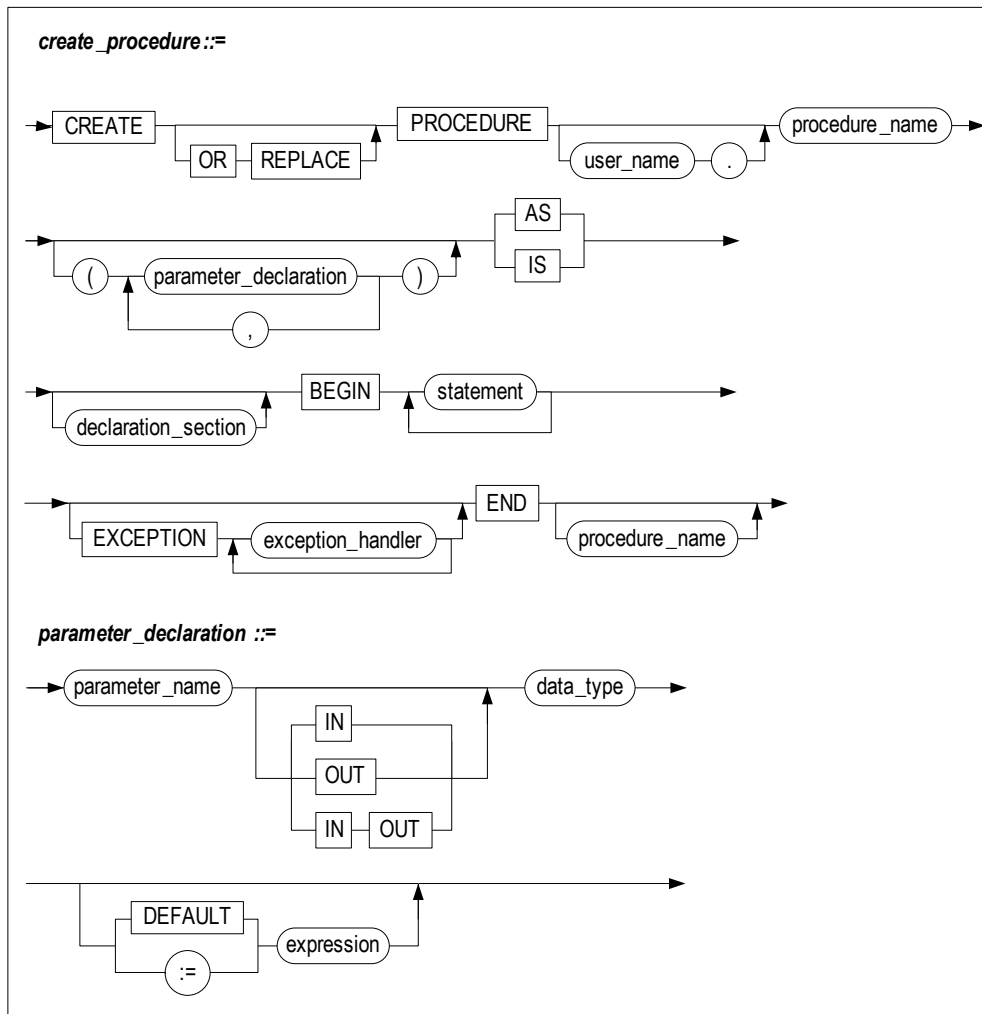
The BOOLEAN type can be used only in stored procedures. Its value is either TRUE or FALSE.
- FILE_TYPE

FILE_TYPE can be used only in stored procedures, and is used to control files in stored procedures. For more information, please refer to [10.1 File Control](#) in this manual.
- User-defined types

User-defined types can be used only in stored procedures. Records and associative arrays are supported for use as user-defined types. For more information, please refer to [Chapter6: User-Defined Types](#) in this manual.

2.2 CREATE PROCEDURE

2.2.1 Syntax



2.2.2 Purpose

This statement creates a new stored procedure, or replaces an existing stored procedure with a new stored procedure.

2.2.2.1 parameter_declaration

parameter_declaration can be omitted. When *parameter_declaration* is provided, the name, data type, and parameter type must all be specified.

A parameter can be defined as one of the following three types:

- IN: an input parameter for which the value is specified when the procedure is called
- OUT: an output parameter, which returns an output value after the procedure has executed
- IN/OUT: an input/output parameter, for which the value is specified when calling the procedure, and which also returns an output value, typically after some operations are performed thereon

If the parameter type is omitted, IN is the default type.

When a stored procedure is executed, values are passed to the stored procedure using IN parameters, and the procedure returns values to the calling routine using OUT parameters.

An IN parameter is handled as a constant within a stored procedure. This means that a value cannot be assigned to an IN parameter within a stored procedure. Additionally, an IN parameter cannot be used in an INTO clause of a SELECT statement.

A parameter can have a default value. If no value is passed to a procedure for a parameter that has a default value, this default value will be used.

2.2.2.2 Declaration Section

Please refer to the section entitled [3.2 Declaring Local Variables](#) in Chapter 3 of this manual.

2.2.2.3 Data Types

Please refer to the section entitled [3.2 Declaring Local Variables](#) in Chapter 3 of this manual.

2.2.2.4 Exception Handlers

Please refer to [Chapter9: Exception Handlers](#) in this manual.

2.2.2.5 Executing the CREATE PROCEDURE Statement

A stored procedure creation statement can be written in advance in a text editor and pasted into iSQL, or can be entered line-by-line directly using iSQL.

Use a semicolon (“;”) at the end of SQL statements, stored procedure control flow statements, and blocks (“END”).

On the line following the last END statement, be sure to use a slash (“/”) to indicate the end of the procedure creation statement when using iSQL. The procedure creation statement is now ready for execution. When the CREATE PROCEDURE statement is executed, if there are no compile errors and the block is successfully compiled, the message “Create success” is output.

The elements that are used in the body of a stored procedure, namely blocks, control flow statements, cursors, and exception handlers, will be described individually in subsequent chapters.

Please refer to the following examples.

2.2 CREATE PROCEDURE

2.2.3 Example

2.2.3.1 Example 1 (Using IN parameters)

```
CREATE TABLE t1 (i1 INTEGER UNIQUE, i2 INTEGER, i3 INTEGER);
INSERT INTO t1 VALUES (1,1,1);
INSERT INTO t1 VALUES (2,2,2);
INSERT INTO t1 VALUES (3,3,3);
INSERT INTO t1 VALUES (4,4,4);
INSERT INTO t1 VALUES (5,5,5);
SELECT * FROM t1;
```

```
CREATE OR REPLACE PROCEDURE proc1
(p1 IN INTEGER, p2 IN INTEGER, p3 IN INTEGER)
AS
  v1 INTEGER;
  v2 t1.i2%type;
  v3 INTEGER;
BEGIN
  SELECT *
  INTO v1, v2, v3
  FROM t1
  WHERE i1 = p1 AND i2 = p2 AND i3 = p3;

  IF v1 = 1 AND v2 = 1 AND v3 = 1 THEN
    UPDATE t1 SET i2 = 7 WHERE i1 = v1;
  ELSIF v1 = 2 AND v2 = 2 AND v3 = 2 THEN
    UPDATE t1 SET i2 = 7 WHERE i1 = v1;
  ELSIF v1 = 3 AND v2 = 3 AND v3 = 3 THEN
    UPDATE t1 SET i2 = 7 WHERE i1 = v1;
  ELSIF v1 = 4 AND v2 = 4 AND v3 = 4 THEN
    UPDATE t1 SET i2 = 7 WHERE i1 = v1;
  ELSE
    DELETE FROM t1;
  END IF;
  INSERT INTO t1 VALUES (p1+10, p2+10, p3+10);
END;
```

```
/

iSQL> EXEC proc1 (2,2,2);
EXECUTE success.
iSQL> SELECT * FROM t1;
T1.I1      T1.I2      T1.I3
-----
1          1          1
3          3          3
4          4          4
5          5          5
2          7          2
12         12         12
6 rows selected.
```

2.2.3.2 Example 2 (using parameters with default values)

```
CREATE TABLE t1 (i1 INTEGER, i2 INTEGER, i3 INTEGER);

CREATE OR REPLACE PROCEDURE proc1
(p1 IN INTEGER DEFAULT 1, p2 IN INTEGER DEFAULT 1, p3 IN INTEGER DEFAULT 1)
AS
BEGIN
  INSERT INTO t1 VALUES (p1, p2, p3);
```

```

END;
/

EXEC proc1;
SELECT * FROM t1;
EXEC proc1(2);
SELECT * FROM t1;
EXEC proc1(3,3);
SELECT * FROM t1;
EXEC proc1(4,4,4);
iSQL> SELECT * FROM t1;
T1.I1      T1.I2      T1.I3
-----
1          1          1
2          1          1
3          3          1
4          4          4
4 rows selected.

```

2.2.3.3 Example 3

```

CREATE OR REPLACE PROCEDURE proc1
(emp_id INTEGER, amount NUMBER(10,2))
AS
BEGIN
  UPDATE employees SET salary = salary + amount
  WHERE eno = emp_id;
END;
/

iSQL> EXEC proc1(15, '250');
EXECUTE success.

iSQL> SELECT * FROM employees WHERE eno=15;
ENO          E_LASTNAME      E_FIRSTNAME
-----
EMP_JOB      EMP_TEL      DNO          SALARY      SEX
-----
BIRTH      JOIN_DATE    STATUS
-----
15          Davenport    Jason
webmaster    0119556884    1003         1250        M
901212      H
1 row selected.

```

2.2.3.4 Example 4 (Using OUT and IN/OUT parameters)

```

CREATE TABLE t4(i1 INTEGER, i2 INTEGER);
INSERT INTO t4 VALUES(1,1);
INSERT INTO t4 VALUES(1,1);
INSERT INTO t4 VALUES(1,1);
INSERT INTO t4 VALUES(1,1);
INSERT INTO t4 VALUES(1,1);

CREATE OR REPLACE PROCEDURE proc1(a1 OUT INTEGER, a2 IN OUT INTEGER)
AS
BEGIN
  SELECT COUNT(*) INTO a1 FROM t4 WHERE i2 = a2;
END;
/

iSQL> VAR t3 INTEGER;

```

2.2 CREATE PROCEDURE

```
iSQL> VAR t4 INTEGER;
iSQL> EXEC :t4 := 1;
EXECUTE success.
iSQL> EXEC proc1(:t3, :t4);
EXECUTE success.
iSQL> PRINT t3;
NAME          TYPE          VALUE
-----
T3            INTEGER        5
```

2.2.3.5 Example 5

```
CREATE OR REPLACE PROCEDURE proc1(p1 INTEGER, p2 IN OUT INTEGER, p3 OUT INTE-
GER)
AS
BEGIN
  p2 := p1;
  p3 := p1 + 100;
END;
/

iSQL> VAR v1 INTEGER;
iSQL> VAR v2 INTEGER;
iSQL> VAR v3 INTEGER;
iSQL> EXEC :v1 := 3;
EXECUTE success.
iSQL> EXEC proc1(:v1, :v2, :v3);
EXECUTE success.
iSQL> PRINT VAR;
[ HOST VARIABLE ]
-----
NAME          TYPE          VALUE
-----
V1            INTEGER        3
V2            INTEGER        3
V3            INTEGER        103
```

2.2.3.6 Example 6 (Using an IN/OUT parameter)

```
CREATE TABLE t3(i1 INTEGER);
INSERT INTO t3 VALUES(1);
INSERT INTO t3 VALUES(1);
INSERT INTO t3 VALUES(1);

CREATE OR REPLACE PROCEDURE proc1(a1 IN OUT INTEGER)
AS
BEGIN
  SELECT COUNT(*) INTO a1 FROM t3 WHERE i1 = a1;
END;
/

iSQL> VAR p1 INTEGER;
iSQL> EXEC :p1 := 1;
EXECUTE success.
iSQL> EXEC proc1(:p1);
EXECUTE success.
iSQL> PRINT p1;
NAME          TYPE          VALUE
-----
P1            INTEGER        3
```

2.2.3.7 Example 7

```
CREATE OR REPLACE PROCEDURE proc1(p1 INTEGER, p2 IN OUT INTEGER, p3 OUT INTE-
GER)
AS
BEGIN
  p2 := p1 + p2;
  p3 := p1 + 100;
END;
/
```

```
iSQL> VAR v1 INTEGER;
iSQL> VAR v2 INTEGER;
iSQL> VAR v3 INTEGER;
iSQL> EXEC :v1 := 3;
EXECUTE success.
iSQL> EXEC :v2 := 5;
EXECUTE success.
iSQL> EXEC proc1(:v1, :v2, :v3);
EXECUTE success.
iSQL> PRINT VAR;
[ HOST VARIABLE ]
```

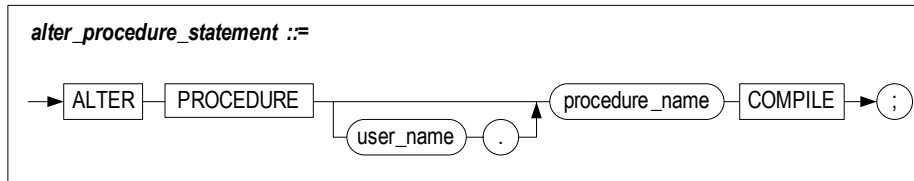
NAME	TYPE	VALUE
V1	INTEGER	3
V2	INTEGER	8
V3	INTEGER	103

2.2.4 Limitation

Parameters defined in CREATE PROCEDURE statements cannot be LOB type.

2.3 ALTER PROCEDURE

2.3.1 Syntax



2.3.2 Purpose

A stored procedure can access various database objects, such as tables, views, and sequences, and can also call other stored procedures and stored functions. After a procedure is created, if any of these objects are altered or changed, the stored procedure can enter what is known as an invalid state.

The reason for this is that the execution plans for the SQL statements in the stored procedure are built when the procedure is compiled, and may no longer be optimized, or even executable.

For example, suppose that an index that existed when a stored procedure was created is later deleted. In this case, because the execution plan for an SQL statement in the stored procedure used the index to access a table, it will become impossible to access the table using the stored procedure from the moment the index is deleted.

When an invalid procedure is called, it is automatically and immediately recompiled by the database. However, compiling at run time in this way can cause significant performance issues in some systems. Therefore, it is recommended that procedures be recompiled when they enter an invalid state.

The ALTER PROCEDURE statement is used to explicitly recompile a stored procedure under these circumstances.

2.3.3 Example

2.3.3.1 Example 1

```

CREATE TABLE t1 (i1 NUMBER, i2 VARCHAR(10), i3 DATE);

CREATE OR REPLACE PROCEDURE proc1
(p1 IN NUMBER, p2 IN VARCHAR(10), p3 IN DATE)
AS
BEGIN
    IF p1 > 0 then
        INSERT INTO t1 VALUES (p1, p2, p3);
    END IF;
END;
/

iSQL> EXECUTE proc1 (1, 'seoul', '20-JUN-2002');
  
```



```

EXECUTE success.
iSQL> EXECUTE proc1 (-3, 'daegu', '21-APR-2002');
EXECUTE success.
iSQL> SELECT * FROM t1;
T1.I1      T1.I2      T1.I3
-----
1          seoul      20-JUN-2002
1 row selected.

```

2.3.3.2 Example 2

```

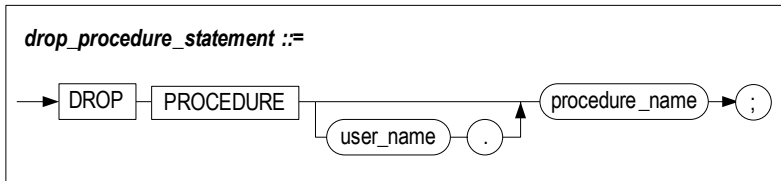
CREATE TABLE t1 (i1 NUMBER, i2 VARCHAR(10), i3 DATE DEFAULT SYSDATE);
ALTER PROCEDURE proc1 COMPILE;

iSQL> EXECUTE proc1 (2, 'incheon', SYSDATE);
EXECUTE success.
iSQL> SELECT * FROM t1;
T1.I1      T1.I2      T1.I3
-----
2          incheon    28-DEC-2010
1 row selected.

```

2.4 DROP PROCEDURE

2.4.1 Syntax



2.4.2 Purpose

This statement removes a stored procedure from the database.

Note that this statement will execute successfully even if there are other stored procedures or stored functions that reference the procedure to be dropped.

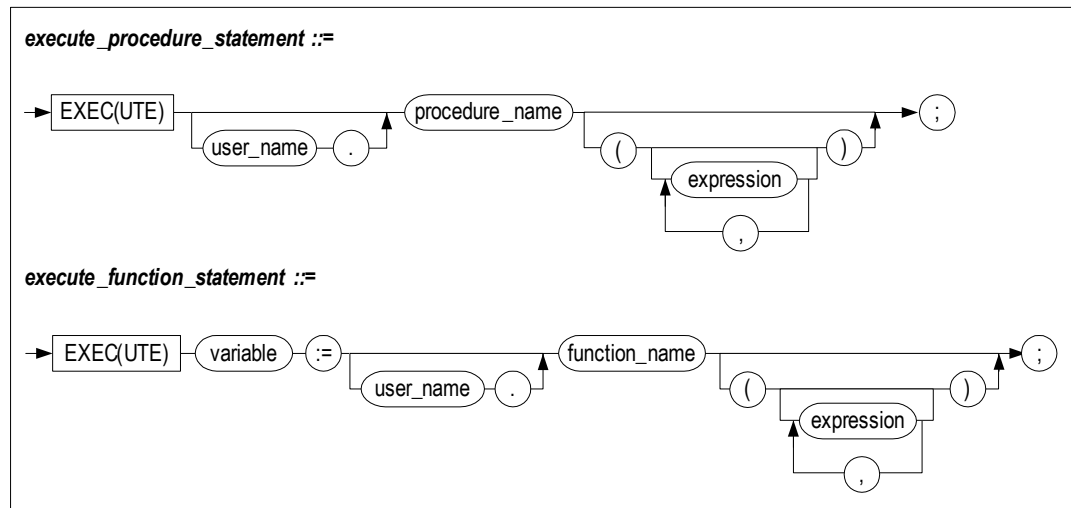
When a stored procedure or stored function attempts to call a stored procedure or stored function that has already been dropped, an error is returned.

2.4.3 Example

```
DROP PROCEDURE proc1;
```

2.5 EXECUTE

2.5.1 Syntax



2.5.2 Purpose

This statement is used to execute a stored procedure or stored function.

2.5.3 Example

```

CREATE OR REPLACE PROCEDURE procl(eid INTEGER, amount NUMBER(10,2))
AS
  current_salary NUMBER(10,2);
BEGIN
  SELECT salary
  INTO current_salary
  FROM employees
  WHERE eno = eid;

  UPDATE employees
  SET salary = salary + amount
  WHERE eno = eid;
END;
/

```

```

iSQL> SELECT * FROM employees WHERE eno=15;
ENO      E_LASTNAME      E_FIRSTNAME
-----

```

```

EMP_JOB      EMP_TEL      DNO      SALARY      SEX
-----
BIRTH      JOIN_DATE      STATUS
-----
15          Davenport      Jason
webmaster    0119556884      1003      1000      M
901212          H

```

2.5 EXECUTE

1 row selected.

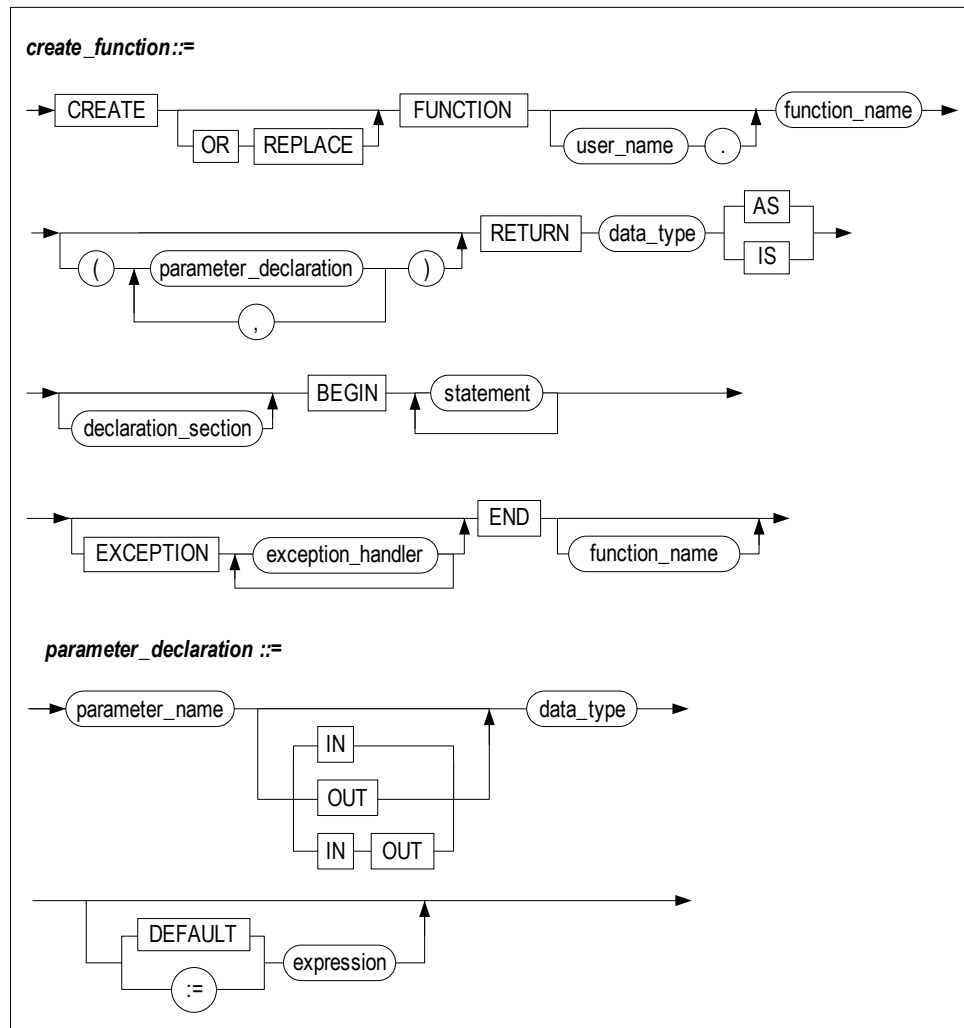
```
iSQL> EXEC proc1(15, 333);  
EXECUTE success.
```

```
iSQL> SELECT * FROM employees WHERE eno=15;  
ENO          E_LASTNAME          E_FIRSTNAME
```

```
-----  
EMP_JOB      EMP_TEL      DNO      SALARY      SEX  
-----  
BIRTH    JOIN_DATE    STATUS  
-----  
15      Davenport      Jason  
webmaster      0119556884      1003      1333      M  
901212      H  
1 row selected.
```

2.6 CREATE FUNCTION

2.6.1 Syntax



2.6.2 Purpose

This statement is used to create a new stored function or replace an existing function with a new function.

2.6.2.1 parameter_declaration

As with stored procedures, parameters in stored functions can be defined as IN, IN/OUT, or OUT parameters. The default type is IN.

For a detailed description of the parameter types and considerations when using parameters with stored functions, please refer to [2.2.2.1 parameter_declaration](#) in the explanation of the CREATE

2.6 CREATE FUNCTION

PROCEDURE statement.

2.6.2.2 RETURN data_type

Unlike stored procedures, stored functions return a single value after they are executed. Therefore, the data type of the return value must be specified.

2.6.2.3 Declaration Section

Please refer to the section entitled [3.2 Declaring Local Variables](#) in Chapter 3 of this manual.

2.6.2.4 Data Types

Please refer to the section entitled [3.2 Declaring Local Variables](#) in Chapter 3 of this manual.

2.6.2.5 Exception Handlers

Please refer to [Chapter9: Exception Handlers](#) in this manual.

2.6.2.6 Executing the CREATE FUNCTION Statement

Execution of the CREATE FUNCTION statement is similar to that for the CREATE PROCEDURE statement. Therefore, please refer to the corresponding description in [2.2.2.5 Executing the CREATE PROCEDURE Statement](#).

The following examples illustrate the use of the CREATE FUNCTION statement:

2.6.3 Example

```
CREATE TABLE t1(
  seq_no INTEGER,
  user_id VARCHAR(9),
  rate NUMBER,
  start_date DATE,
  end_date DATE);
INSERT INTO t1 VALUES(0, '000000500', 200.50, '23-May-2002', '23-Apr-2002');
INSERT INTO t1 VALUES(0, '000000501', 190, '23-Nov-2002', '23-Dec-2002');
INSERT INTO t1 VALUES(0, '000000523', 100, '12-Dec-2001', '12-Jan-2001');
INSERT INTO t1 VALUES(0, '000000532', 100, '11-Dec-2001', '11-Jan-2002');
INSERT INTO t1(seq_no, user_id, start_date, end_date) VALUES(0, '000000524',
  '30-Oct-2001', '30-Nov-2001');
INSERT INTO t1 VALUES(0, '000000524', 200.50, '30-Apr-2002', '30-May-2002');
INSERT INTO t1 VALUES(0, '000000524', 200.50, '30-Apr-2002', '30-May-2002');
INSERT INTO t1 VALUES(1, '000000524', 100, '30-Apr-2002', '30-May-2002');
INSERT INTO t1 VALUES(1, '000000524', 115.0, '19-Jan-2002', '19-Mar-2002');
INSERT INTO t1 VALUES(0, '000000502', 120.0, '27-Jan-2002', '27-Feb-2002');
INSERT INTO t1 VALUES(1, '000000504', 150.0, '26-Nov-2001', '26-Dec-2001');
```

```
iSQL> SELECT * FROM t1;
T1.SEQ_NO      T1.USER_ID      T1.RATE      T1.START_DATE
-----
T1.END_DATE
-----
0              000000500      200.5      2002/05/23 00:00:00
```

```

2002/04/23 00:00:00
0          000000501          190          2002/11/23 00:00:00
2002/12/23 00:00:00
0          000000523          100          2001/12/12 00:00:00
2001/01/12 00:00:00
0          000000532          100          2001/12/11 00:00:00
2002/01/11 00:00:00
0          000000524          200.5        2001/10/30 00:00:00
2001/11/30 00:00:00
0          000000524          200.5        2002/04/30 00:00:00
2002/05/30 00:00:00
0          000000524          200.5        2002/04/30 00:00:00
2002/05/30 00:00:00
1          000000524          100          2002/04/30 00:00:00
2002/05/30 00:00:00
1          000000524          115          2002/01/19 00:00:00
2002/03/19 00:00:00
0          000000502          120          2002/01/27 00:00:00
2002/02/27 00:00:00
1          000000504          150          2001/11/26 00:00:00
2001/12/26 00:00:00
11 rows selected.

```

```

CREATE OR REPLACE FUNCTION get_rate
(p1 IN CHAR(30), p2 IN CHAR(30), p3 IN VARCHAR(9))
RETURN NUMBER
AS
  v_rate NUMBER;
BEGIN
  SELECT NVL(SUM(rate), 0)
  INTO v_rate
  FROM (SELECT rate
        FROM t1
        WHERE start_date = TO_DATE(p1)
              AND end_date = TO_DATE(p2)
              AND user_id = '000000' || p3
              AND seq_no = 0);
  RETURN v_rate;
END;
/

```

```

iSQL> VAR res NUMBER;
iSQL> EXECUTE :res := get_rate('30-Apr-2002', '30-May-2002', '524');
EXECUTE success.
iSQL> PRINT res;
NAME      TYPE      VALUE
-----
RES       NUMBER    401

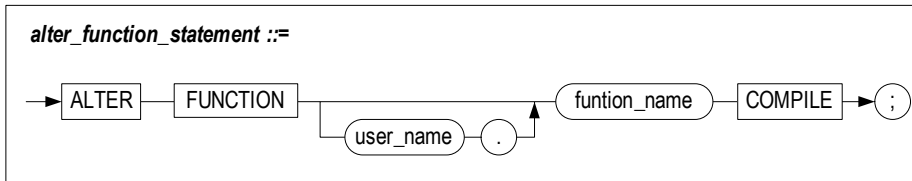
```

2.6.4 Limitations

As with CREATE PROCEDURE statements, parameters defined in CREATE FUNCTION statements cannot be LOB type. Additionally, the return value of a CREATE FUNCTION statement cannot be a LOB type value.

2.7 ALTER FUNCTION

2.7.1 Syntax



2.7.2 Purpose

As with a stored procedure, a stored function can enter what is known as an invalid state when one or more of the database objects that it references are changed after the function is created.

In such circumstances, the ALTER FUNCTION statement is used to explicitly recompile the stored function and create an execution plan that is valid and optimized.

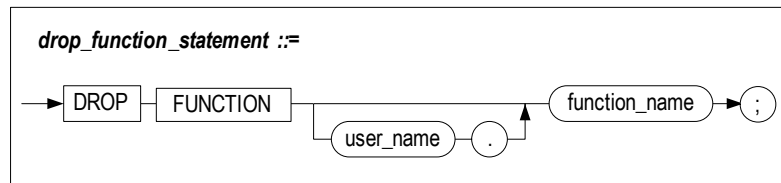
For a more detailed explanation, please refer to the corresponding description in [2.3.2 Purpose](#) in the explanation of the ALTER PROCEDURE statement.

2.7.3 Example

```
ALTER FUNCTION get_dept_name COMPILE;
```


2.8 DROP FUNCTION

2.8.1 Syntax



2.8.2 Purpose

This statement removes a stored function from the database.

Note that this statement will execute successfully even if there are other stored procedures or stored functions that reference the stored function to be dropped.

When a stored procedure or stored function attempts to call a stored procedure or stored function that has already been dropped, an error is returned.

2.8.3 Example

```
DROP FUNCTION get_dept_name;
```

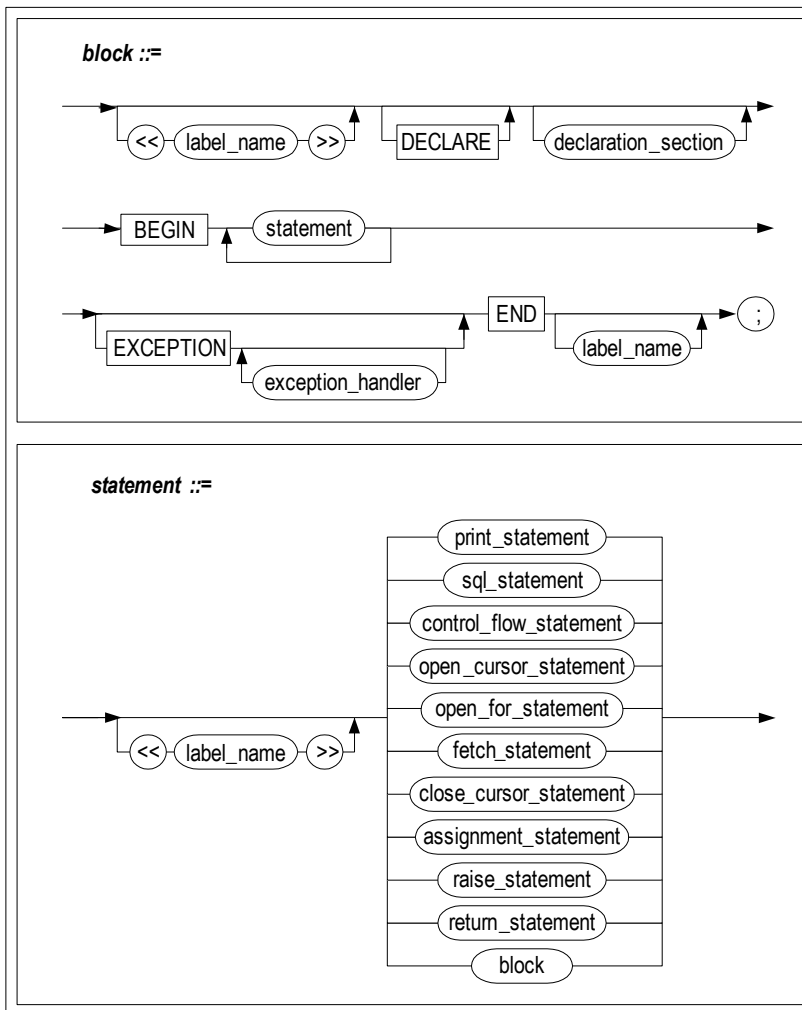
2.8 DROP FUNCTION

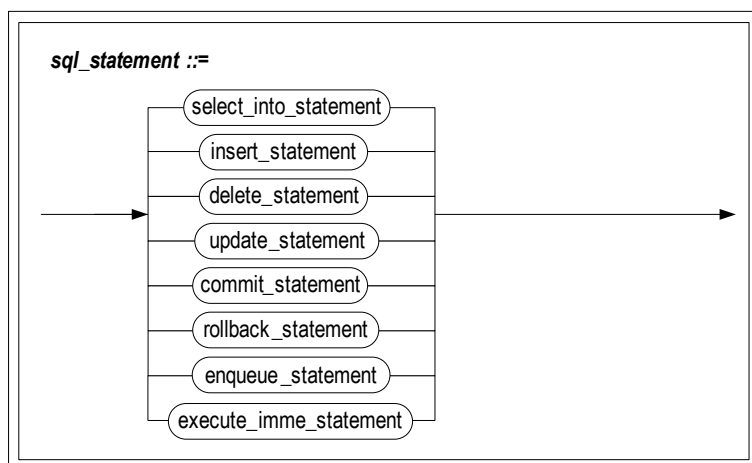
3 Stored Procedure Blocks

A stored procedure or function consists of one or more blocks. This chapter describes how to develop a procedural program within a stored procedure using blocks.

3.1 Overview

3.1.1 Syntax





A block can be broadly divided into a declaration section, a block body and an exception handler section.

A semicolon (";"), which indicates the end of a statement, is not used after the DECLARE, BEGIN or EXCEPTION statements, but must be placed after the END statement and other commands in stored procedures.

Comments can be used in stored procedures. To comment out all or part of a single line, place two hyphen characters ("--") at the beginning of the text to be commented out. To comment out multiple lines, place the C-style delimiters "/*" and "*/" around the text to be commented out.

In this chapter, the variable assignment statements, which can be used within the declaration section and block body, and the SELECT INTO, assignment statements, LABEL, PRINT and RETURN statements, which can be used only within the block body, will be described.

Information on the use of control flow statements, cursor-related statements and exception handlers in stored procedures can be found in subsequent chapters. For information on general SQL statements, please refer to the *SQL Reference*.

3.1.2 Declare Section

The declare section is delimited by the AS and BEGIN keywords for the main block, and by the DECLARE and BEGIN keywords for sub-blocks. Local variables, cursors, and user-defined exceptions for use in the current block are declared here.

In this chapter, only local variables will be described. Cursors and exception handlers will be described together with the related statements in [Chapter5: Using Cursors](#) and [Chapter9: Exception Handlers](#), respectively.

3.1.3 Block Body

The block body is the part between the BEGIN and END keywords. It contains SQL statements and control flow statements.

The following SQL statements and control flow statements can be used within the block body:

3.1 Overview

- DML statements: SELECT/INSERT/DELETE/UPDATE
- Transaction statements: COMMIT/ROLLBACK
- Control flow statements: IF, CASE, FOR, LOOP, WHILE, EXIT, CONTINUE, NULL
- Assignment statements
- Output statements: PRINT and PRINTLN
- Cursor statements: OPEN, FETCH, CLOSE, Cursor FOR LOOP
- Dynamic SQL statement: EXECUTE IMMEDIATE
- Exception handling statements: RAISE and RAISE_APPLICATION_ERROR

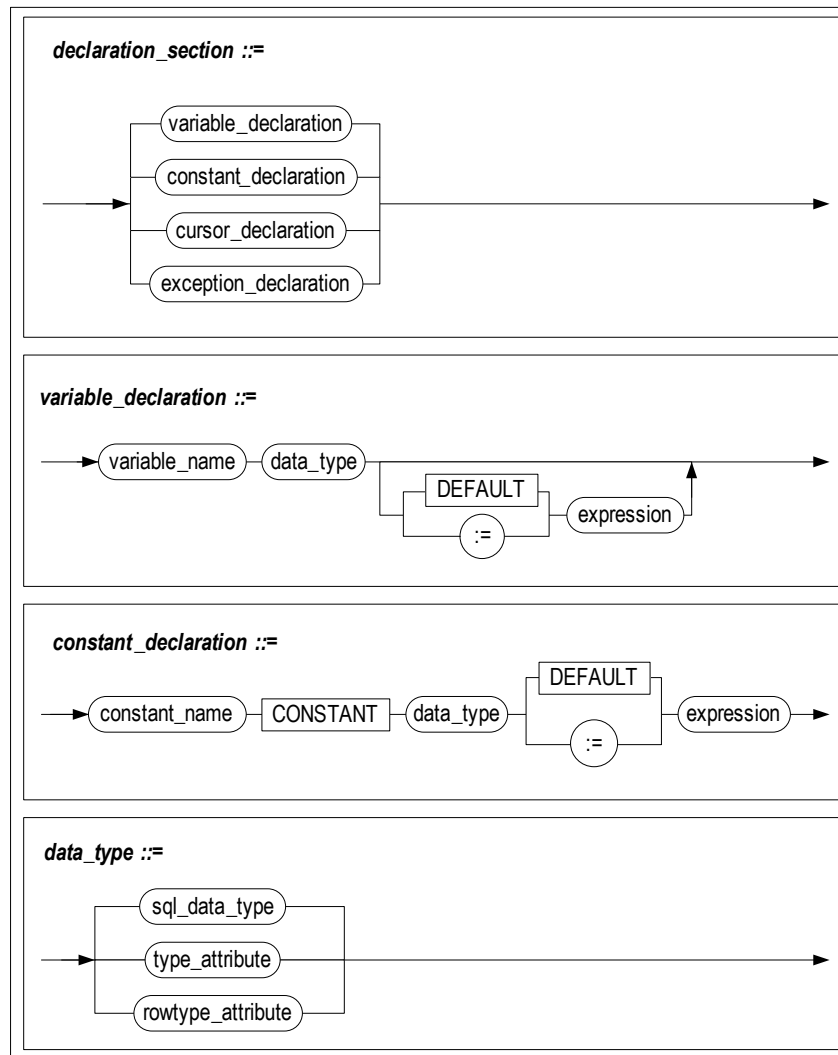
One advantage of stored procedures compared to SQL statements is that it is possible to nest blocks. Anywhere that commands can be used, commands can be formed into blocks, which can be nested.

3.1.4 Exception Handler Section

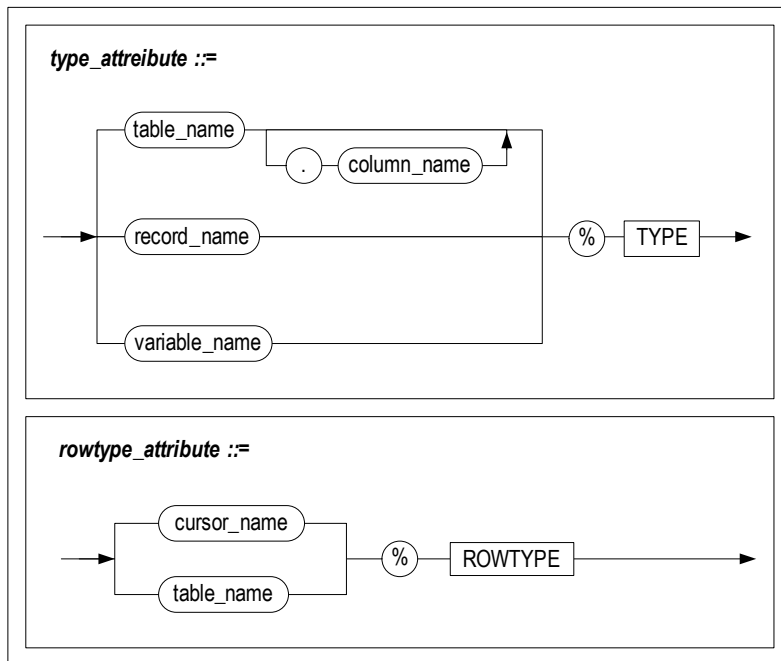
The exception handler section is delimited by the EXCEPTION and END keywords. It contains a routine for handling particular errors that may arise during execution of the stored procedure or function.

3.2 Declaring Local Variables

3.2.1 Syntax



3.2 Declaring Local Variables



3.2.2 Purpose

3.2.2.1 Variable Name

This is used to specify the name of a variable.

The name of the variable must be unique within the block in which it is declared.

If a column and a variable have the same name, any reference to this name in an SQL statement will be interpreted to mean the column. In the following example, both instances of *eno* are interpreted to mean the column name, with the undesirable result that all of the records in the *employees* table will be deleted.

```
DECLARE
eno INTEGER := 100;
BEGIN
DELETE FROM employees WHERE eno = eno;
```

To overcome this ambiguity, use a label to assign a name to the block as follows:

```
<<del_block>>
DECLARE
eno INTEGER := 100;
BEGIN
DELETE FROM employees WHERE eno = del_block.eno;
```

For more information about naming blocks, please refer to [3.5 LABEL](#) in this chapter.

3.2.2.2 Data Types

This is used to specify the data type of the variable. The following data types can be used within

stored procedures:

- All data types that can be used within SQL statements
- The BOOLEAN type
- Any type which is defined for a column or variable and is referenced using the %TYPE attribute
- A RECORD type, comprising multiple columns, referenced using the %ROWTYPE attribute
- User-defined types

The %TYPE and %ROWTYPE attributes obviate the necessity to change the code in stored procedures when table definitions change. That is, when the data type of a column in a table is changed, a variable defined using the %TYPE attribute will automatically take on the correct type, without any intervention. This helps realize data independence and lower maintenance expenses.

3.2.2.3 CONSTANT

This option is used when it is desired to use a particular variable as a constant, so that no other value can be assigned to it within the stored procedure. A variable defined in this way is read-only.

For example, when *max_val* is declared as shown below, it is handled as a constant having the value of 100, and no other value can be arbitrarily allocated thereto.

```
max_val CONSTANT integer := 100;
```

3.2.2.4 DEFAULT

This is used as follows to set an initial value for a variable when it is declared:

```
curr_val INTEGER DEFAULT 100;
count_val INTEGER := 0;
```

3.2.2.5 Cursor Declaration

Please refer to [5.2 CURSOR](#) in this manual.

3.2.2.6 Exception Declaration

Please refer to [9.2 EXCEPTION](#) in this manual.

3.2.2.7 Nested Blocks and Variable Scope

The scope of a variable specified in the DECLARE section of a block starts at the BEGIN statement and finishes at the END statement in the block in which it was declared.

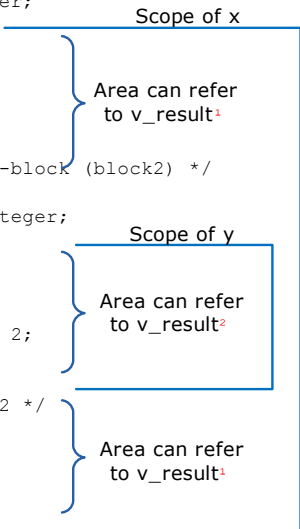
Suppose that *block2* is declared inside *block1*, and that variables having the same name, *v_result*, are declared within each block, as shown below. When *v_result* is referenced outside of *block2*, the reference is interpreted to mean the variable declared in *block1*, whereas when *v_result* is referenced inside *block2*, it is interpreted to mean the variable declared in *block2*.

Meanwhile, both the variable *x*, which was declared in *block1* (the outer block), and the variable *y*,

3.2 Declaring Local Variables

which was declared in *block2* (the inner block), can be referred to in the inner block, but only *x* can be referred to in the outer block.

```
/* start of block1 */
DECLARE
  v_result1 integer;
  x integer;
BEGIN
  ....
  v_result1 := 1;
  ....
  /* start of sub-block (block2) */
  DECLARE
    v_result2 integer;
    y number;
  BEGIN
    ...
    v_result2 := 2;
    ...
  END;
  /* end of block2 */
  ...
  v_result1 := 3;
  ...
END;
/* end of block1 */
```



3.2.2.8 Restrictions

The following are not supported when declaring variables:

- Variables defined within stored procedures cannot have NOT NULL constraints.
- Multiple variables cannot be declared at the same time. That is, statements such as the following are not possible:

```
i, j, k INTEGER;
```

3.2.3 Examples

3.2.3.1 Use of %TYPE

```
DECLARE
my_title books.title%TYPE;
```

In the above example, the variable *my_title* is declared such that it will have the same type as the *title* column in the *books* table.

3.2.3.2 Use of %ROWTYPE

```
DECLARE
dept_rec departments%ROWTYPE
```

In the above example, the variable *dept_rec*, which is a RECORD type variable, is declared such that it will be the same as the *departments* table or cursor.

3.2.3.3 Example 1

This example shows the declaration of constants and the use of the %ROWTYPE attribute.

```
CREATE TABLE t1(i1 INTEGER, i2 INTEGER);

CREATE OR REPLACE PROCEDURE proc1
AS
  v1 constant INTEGER := 1;
  v2 constant t1.i1%TYPE := 1;
BEGIN
  INSERT INTO t1 VALUES (v1,v2);
END;
/

EXEC proc1;
iSQL> SELECT * FROM t1;
T1.I1    T1.I2
-----
1         1
1 row selected.

--DROP TABLE t1;
CREATE TABLE t1 (i1 INTEGER, i2 INTEGER, i3 INTEGER);
INSERT INTO t1 VALUES(1,1,1);
CREATE OR REPLACE PROCEDURE proc1
AS
  r1 t1%ROWTYPE;
BEGIN
  INSERT INTO t1 VALUES(3,3,3);
  <<s>>
  DECLARE
    r1 t1%ROWTYPE;
  BEGIN
    SELECT i1, i2, i3 INTO s.r1.i1, s.r1.i2, s.r1.i3 FROM t1 WHERE i1 = 1;
    INSERT INTO t1 VALUES (s.r1.i1, s.r1.i2, s.r1.i3);
  END;
END;
/

iSQL> EXEC proc1;
EXECUTE success.
iSQL> SELECT * FROM t1;
T1.I1    T1.I2    T1.I3
-----
1         1         1
3         3         3
1         1         1
3 rows selected.
```

3.2.3.4 Example 2

This example also shows the use of the %ROWTYPE attribute.

```
CREATE TABLE emp(
  eno INTEGER,
  ename CHAR(10),
  emp_job CHAR(15),
  join_date DATE,
  salary NUMBER(10,2),
  dno BYTE(2));
```

3.2 Declaring Local Variables

```
CREATE TABLE emp401(
  eno INTEGER,
  ename CHAR(10),
  emp_job CHAR(15),
  join_date DATE,
  leave_date DATE,
  salary NUMBER(10,2),
  dno BYTE(2),
  fund NUMBER(10,2) DEFAULT 0);

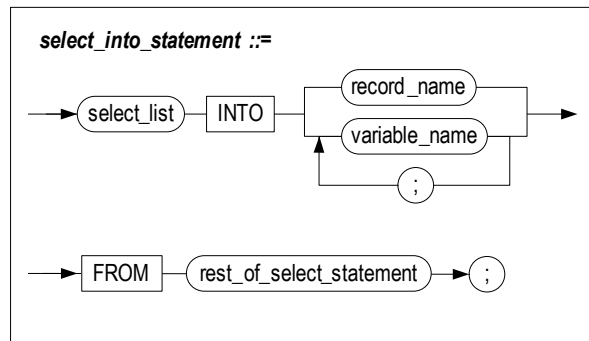
INSERT INTO emp VALUES (10, 'John Doe', 'engineer', '01-Jul-2010', 30000,
BYTE'D001');
INSERT INTO emp VALUES (20, 'Tom Jones', 'manager', '01-Nov-2009', 50000,
BYTE'C002');

CREATE OR REPLACE PROCEDURE proc1(p1 INTEGER)
AS
BEGIN
  DECLARE
    emp_rec emp%ROWTYPE;
  BEGIN
    SELECT * INTO emp_rec
    FROM emp
    WHERE eno = p1;
    INSERT INTO emp401(enno, ename, emp_job, join_date, leave_date, salary, dno)
    VALUES(emp_rec.enno, emp_rec.ename, emp_rec.emp_job, emp_rec.join_date, sys
      date, emp_rec.salary, emp_rec.dno);
  END;
END;
/

iSQL> EXEC proc1(10);
EXECUTE success.
iSQL> SELECT * FROM emp401;
EMP401.ENO          EMP401.ENAME      EMP401.EMP_JOB    EMP401.JOIN_DATE
-----
EMP401.LEAVE_DATE   EMP401.SALARY     EMP401.DNO        EMP401.FUND
-----
10                  John Doe          engineer          2010/07/01 00:00:00
2011/01/27 16:26:26 30000            D001              0
1 row selected.
```

3.3 SELECT INTO

3.3.1 Syntax



Because the syntax of *select_list* and *rest_of_select_statement* is the same as for a SELECT statement, please refer to the *SQL Reference* for more information on those elements.

3.3.2 Purpose

When a stored procedure includes a SELECT statement, the SELECT statement must contain an INTO clause.

A SELECT statement in a stored procedure or function must retrieve exactly one record. If the statement retrieves zero or multiple records, an error will be raised.

The number of columns in *select_list* in the SELECT clause must be the same as the number of *variable_name* in the INTO clause. Furthermore, the data types of corresponding columns and variables must be compatible. Similarly, when the %ROWTYPE attribute is used, the number of columns in the %ROWTYPE variable and the number of columns in *select_list* must be the same, and the data types of corresponding columns must be compatible.

When a standard exception occurs, the stored procedure raises an error. The NO_DATA_FOUND and TOO_MANY_ROW exceptions can be used to handle errors in the block's exception handler section. Please refer to [Chapter9: Exception Handlers](#) for more information about handling errors.

3.3.3 Example

3.3.3.1 Example 1

```

CREATE TABLE t1(i1 INTEGER, i2 INTEGER, i3 INTEGER);
INSERT INTO t1 VALUES(1,1,1);

CREATE OR REPLACE PROCEDURE proc1
AS
    v1 INTEGER;
    r1 t1%ROWTYPE;
BEGIN
    INSERT INTO t1 VALUES (3,3,3);

```

3.3 SELECT INTO

```
<<s>>
DECLARE
  v1 proc1.r1.i1%TYPE;
  r1 t1%ROWTYPE;
BEGIN
  SELECT i1,i2,i3
  INTO s.r1.i1, s.r1.i2, s.r1.i3
  FROM t1
  WHERE i1 = 1;

  INSERT INTO t1 VALUES(s.r1.i1, s.r1.i2, s.r1.i3);
END;
/

iSQL> EXEC proc1;
EXECUTE success.
iSQL> SELECT * FROM t1;
T1.I1      T1.I2      T1.I3
-----
1          1          1
3          3          3
1          1          1
3 rows selected.
```

3.3.3.2 Example 2

```
CREATE TABLE t1 (i1 INTEGER, i2 INTEGER, i3 INTEGER);
INSERT INTO t1 VALUES(100, 100, 100);

CREATE SEQUENCE seq1;

CREATE SEQUENCE seq2;

CREATE SEQUENCE seq3;
CREATE OR REPLACE PROCEDURE proc1
AS
BEGIN
  <<seq1>>
  DECLARE
    nextval INTEGER;
  BEGIN
    nextval := 10;
    INSERT INTO t1 VALUES (seq1.NEXTVAL,0,0);
  END;
END;
/
CREATE OR REPLACE PROCEDURE proc2
AS
BEGIN
  INSERT INTO t1 VALUES (seq1.NEXTVAL, seq2.NEXTVAL, seq3.NEXTVAL);
  INSERT INTO t1 VALUES (seq1.NEXTVAL, seq2.NEXTVAL, seq3.NEXTVAL);
  INSERT INTO t1 VALUES (seq1.NEXTVAL, seq2.NEXTVAL, seq3.NEXTVAL);
END;
/
CREATE OR REPLACE PROCEDURE proc3
AS
  v1 INTEGER;
  v2 INTEGER;
  v3 INTEGER;
BEGIN
  SELECT seq1.currval, seq2.NEXTVAL, seq3.NEXTVAL
  INTO v1, v2, v3 FROM t1 WHERE i1 = 100;
```

```

INSERT INTO t1 VALUES (v1, v2, v3);

SELECT seq1.currval, seq1.NEXTVAL, seq1.currval
INTO v1, v2, v3 FROM t1 WHERE i1 = 100;
INSERT INTO t1 VALUES (v1, v2, v3);

SELECT seq1.currval, seq2.NEXTVAL, seq3.NEXTVAL
INTO v1, v2, v3 FROM t1 WHERE i1 = 100;
INSERT INTO t1 VALUES (v1, v2, v3);
END;
/

EXEC proc1;
SELECT * FROM t1;
EXEC proc2;
SELECT * FROM t1;
EXEC proc3;
SELECT * FROM t1;
EXEC proc2;
SELECT * FROM t1;
EXEC proc3;

isQL> SELECT * FROM t1;
T1.I1 T1.I2 T1.I3
-----
100    100    100
10      0      0
1       1      1
2       2      2
3       3      3
3       4      4
4       4      4
4       5      5
5       6      6
6       7      7
7       8      8
7       9      9
8       8      8
8      10     10
14 rows selected.

```

3.3.3.3 Example 3

```

CREATE TABLE t1(i1 INTEGER, i2 INTEGER, i3 INTEGER);

CREATE TABLE t2(i1 INTEGER, i2 INTEGER, i3 INTEGER);
INSERT INTO t1 VALUES (1,1,1);
INSERT INTO t1 VALUES (2,2,2);

CREATE OR REPLACE PROCEDURE proc1
AS
    v1 INTEGER;
    r1 t1%ROWTYPE;
BEGIN
    SELECT i1 INTO v1 FROM t1 WHERE i1 = 1;
    SELECT * INTO r1 FROM t1 WHERE i1 = 1;
    INSERT INTO t2 VALUES (v1, r1.i2, r1.i3);
    <<s>>
    DECLARE
        r1 t1%ROWTYPE;
    BEGIN
        SELECT i1, i2, i3 INTO s.r1.i1, s.r1.i2, s.r1.i3
        FROM t1 WHERE i1 = 2;
    
```

3.3 SELECT INTO

```
        INSERT INTO t2 VALUES (s.r1.i1, s.r1.i2, s.r1.i3);
    END;
END;
/

iSQL> EXEC proc1;
EXECUTE success.
iSQL> SELECT * FROM t2;
T2.I1      T2.I2      T2.I3
-----
1          1          1
2          2          2
2 rows selected.
```

3.3.3.4 Example 4

```
CREATE TABLE t3(i1 INTEGER);

CREATE OR REPLACE PROCEDURE proc1
AS
    max_qty orders.qty%TYPE;
BEGIN
    SELECT MAX(qty)
    INTO max_qty
    FROM orders;

    INSERT INTO t3 VALUES(max_qty);
END;
/

iSQL> exec proc1;
EXECUTE success
iSQL> SELECT * FROM t3;
T3.I1
-----
10000
1 row selected.
```

3.3.3.5 Example 5

```
CREATE TABLE delayed_processing(
    cno CHAR(14),
    order_date DATE);

CREATE OR REPLACE PROCEDURE proc1
AS
    de_cno CHAR(14);
    de_order_date DATE;
BEGIN
    INSERT INTO delayed_processing

    SELECT cno, order_date
    INTO de_cno, de_order_date
    FROM orders
    WHERE processing = 'D';

END;
/

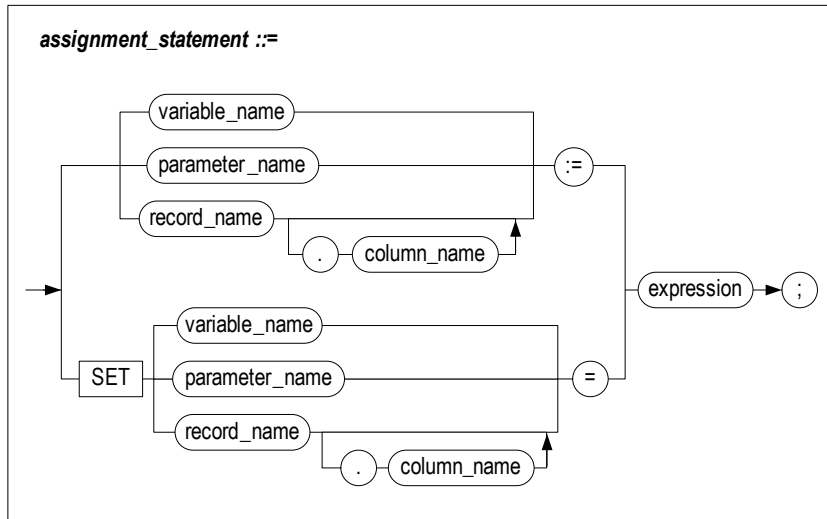
iSQL> EXEC proc1;
EXECUTE success.
iSQL> SELECT * FROM delayed_processing;
```



```
DELAYED_PROCESSING.CNO      DELAYED_PROCESSING.ORDER_DATE
-----
7610011000001              2000/11/29 00:00:00
7001011001001              2000/11/29 00:00:00
2 rows selected.
```

3.4 Assignment Statements

3.4.1 Syntax



3.4.2 Purpose

These statements are used to assign a value to a local variable or to an OUT or IN/OUT parameter.

Values can be assigned to variables and parameters using either of the two following statements:

- Using the assignment operator "::<="
 - `variable_name ::= value`
 - `parameter_name ::= value`
- Using the "SET" expression
 - `SET variable_name = value;`
 - `SET parameter_name = value;`

Refer to each of the individual values in a RECORD type variable that was declared using the %ROW-TYPE attribute in this way:

`record_variable_name.field_name`

3.4.3 Examples

3.4.3.1 Example 1

```

CREATE TABLE t1(i1 INTEGER, i2 INTEGER, i3 INTEGER);

CREATE OR REPLACE PROCEDURE proc1

```

```

AS
  i INTEGER;
BEGIN
  i := 5;

  WHILE i <= 10 LOOP
    INSERT INTO t1 VALUES (i, i+1, i+2);
    i := i + 1;
  END LOOP;

END;
/
iSQL> EXEC proc1;
EXECUTE success.
iSQL> SELECT * FROM t1;
T1.I1      T1.I2      T1.I3
-----
5           6           7
6           7           8
7           8           9
8           9          10
9          10          11
10         11          12
6 rows selected.

```

3.4.3.2 Example 2

```

CREATE TABLE t1(i1 INTEGER, i2 INTEGER, i3 INTEGER);

CREATE OR REPLACE FUNCTION plus20(p1 IN INTEGER)
RETURN INTEGER
AS
  v1 INTEGER;
BEGIN
  v1 := p1 + 20;
  RETURN v1;
END;
/

CREATE OR REPLACE PROCEDURE proc1
AS
  v1 INTEGER;
  in_arg INTEGER;
BEGIN
  in_arg := 80;
  v1 := plus20(in_arg);
  INSERT INTO t1 VALUES (v1, v1, v1);
END;
/

iSQL> EXEC proc1;
EXECUTE success.
iSQL> SELECT * FROM t1;
T1.I1      T1.I2      T1.I3
-----
100         100         100
1 row selected.

```

3.5 LABEL

The LABEL statement is used to name a particular point within a stored procedure. A label can be specified within a block using the delimiters shown below:

```
<< User_defined_label_name >>
```

3.5.1 Purpose

User-defined labels are used in the following three situations:

- To limit the scope of multiple variables having the same name, or to overcome ambiguity that occurs when a variable and a column have the same name
- To exit a nested loop
- For use with the GOTO statement

3.5.2 Limitations

1. The same label cannot be used more than once within the same block. In the following example, a compilation error will occur, because *LABEL1* appears twice within the same block:

```
CREATE OR REPLACE PROCEDURE PROC1
AS
    V1 INTEGER;
BEGIN
    << LABEL1>>
    V1 := 1;
    <<LABEL1>>
    V1 := V1 + 1;
    ...
```

2. In order to use labels to limit the scope of variables having the same name, the labels must be declared immediately before DECLARE statements. Note that it is possible to declare more than one label before a single DECLARE statement. In the following example, there are two references to the variable V1 denoted by (1):

```
CREATE OR REPLACE PROCEDURE PROC1
AS
    V1 INTEGER;
BEGIN
    <<LABEL1>> --- Declares a LABEL.
    <<LABEL2>>
    DECLARE
        V1 INTEGER; -- (1)
    BEGIN
        <<LABEL3>>
        DECLARE
            V1 INTEGER; -- (2)
        BEGIN
            LABEL1.V1 := 1; -- a reference to V1 indicated by (1)
            LABEL2.V1 := 2; -- also a reference to V1 indicated by (1)
            LABEL3.V1 := 3; -- a reference to V1 indicated by (2)
        END;
    END;
END;
```

/

In the following example, because the label declaration does not immediately precede the DECLARE statement, an error results:

```
CREATE OR REPLACE PROCEDURE PROC1
AS
  V1 INTEGER;
BEGIN
  <<LABEL1>>
  V1 := 1;
  DECLARE
    V1 INTEGER;
  BEGIN
    LABEL1.V1 := 1; --- ERROR.
```

3. Similarly, when using a label to exit nested loops, the label must be declared immediately before the loop. Note again that it is possible to declare more than one label before the loop.

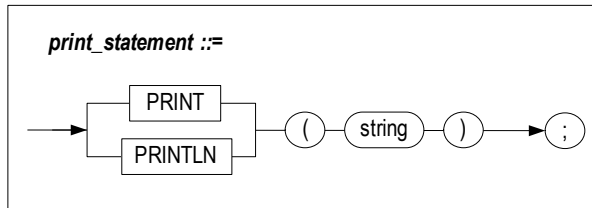
```
CREATE OR REPLACE PROCEDURE PROC1
AS
  V1 INTEGER;
BEGIN
  V1 := 0;
  <<LABEL1>>
  <<LABEL2>>
  FOR I IN 1 .. 10 LOOP
    V1 := V1 + 1;
    FOR I IN 1 .. 10 LOOP
      V1 := V1 + 1;
      EXIT LABEL1 WHEN V1 = 30;
    END LOOP;
  END LOOP;
END;
/
```

In the following example, one of the labels is not declared immediately before the loop. Because this label cannot be used to exit from the nested loops, an error is raised during the attempt to compile the stored procedure.

```
CREATE OR REPLACE PROCEDURE PROC1
AS
  V1 INTEGER;
BEGIN
  <<LABEL1>>
  V1 := 0;
  <<LABEL2>>
  FOR I IN 1 .. 10 LOOP
    V1 := V1 + 1;
    FOR I IN 1 .. 10 LOOP
      V1 := V1 + 1;
      EXIT LABEL1 WHEN V1 = 30; -- ERROR
    END LOOP;
  END LOOP;
END;
/
```

3.6 PRINT

3.6.1 Syntax



3.6.2 Purpose

The PRINT statement is used to output desired text to the calling client or routine. PRINT is a system procedure that is provided within ALTIBASE HDB, and is typically used for debugging and testing.

PRINTLN differs from PRINT only in that it outputs the appropriate newline sequence (“\r\n” in Windows, and “\n” in Unix) after the string. The owner of PRINT and PRINTLN is the SYSTEM_user. It is possible to specify the SYSTEM_user when using these routines, as follows:

```
SYSTEM_.PRINTLN('Hello World');
```

However, it is not necessary to specify the SYSTEM_user in this way, because a public synonym exists for these procedures.

3.6.2.1 string

This is the string to be output to the client.

As seen in Example 2 below, the double-vertical-bars concatenation operator ("||") can be used to combine the values of variables, query results, etc. with text to create a single line of text to be output to the client.

3.6.3 Examples

3.6.3.1 Example 1

```

CREATE OR REPLACE PROCEDURE proc1
AS
  v1 BIGINT;
BEGIN
  v1 := BIGINT'9223372036854775807';
  system_.println ('1');
  system_.println (v1);
  system_.println ('2');
END;
/

iSQL> EXEC proc1;
```

```

1
9223372036854775807
2
EXECUTE success.

```

3.6.3.2 Example 2

```

CREATE OR REPLACE PROCEDURE proc1
AS
    eno_count INTEGER;
BEGIN
    SELECT COUNT(eno) INTO eno_count FROM employees;
    println('The number of employees: ' || eno_count);
END;
/

iSQL> EXEC proc1;
The number of employees: 20
EXECUTE success.

```

3.6.3.3 Example 3

The following example illustrates how to use a loop with PRINT and PRINTLN to output formatted query results.

```

CREATE OR REPLACE PROCEDURE showProcedures
AS
    CURSOR c1 IS
        SELECT SYSTEM_.sys_procedures_.proc_name,
        decode(SYSTEM_.sys_procedures_.object_TYPE, 0, 'Procedure',1,'Function')
        FROM system_.sys_procedures_ ;

    v1 CHAR(40);
    v2 CHAR(20);
BEGIN
    OPEN c1;
    SYSTEM_.PRINTLN('-----');
    SYSTEM_.PRINT('Proc_Name');
    SYSTEM_.PRINTLN(' Procedure/Function');
    SYSTEM_.PRINTLN('-----');

    LOOP
        FETCH C1 INTO v1, v2;
        EXIT WHEN C1%NOTFOUND;
        PRINT(' ');
        PRINT(v1);
        PRINTLN(v2);
    END LOOP;

    PRINTLN('-----');
    CLOSE c1;
END;
/

iSQL> EXEC showProcedures;
-----
Proc_Name          Procedure/Function
-----
PRINT              Procedure
PRINTLN            Procedure
.

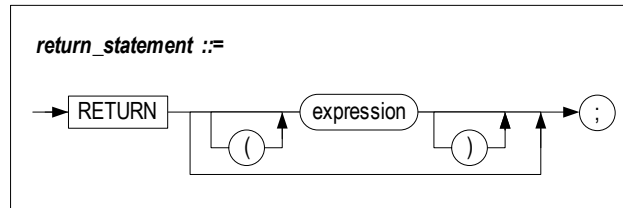
```

3.6 PRINT

```
.  
.  
SHOWPROCEDURES Procedure  
-----  
EXECUTE success.
```


3.7 RETURN

3.7.1 Syntax



3.7.2 Purpose

This statement is used to interrupt the execution of a stored procedure. When used with a stored function, it is additionally used to specify the return value.

Because stored procedures do not return values, an error will be raised in response to an attempt to compile a stored procedure that specifies a return value. In contrast, because a function must always return a value, it is necessary to specify a return value when creating a function.

3.7.2.1 expression

expression is used to specify the return value for a stored function. It is possible to perform operations in *expression*.

3.7.3 Example

3.7.3.1 Example 1

```
CREATE TABLE t1(i1 INTEGER, i2 INTEGER, i3 INTEGER);

INSERT INTO t1 VALUES(1,1,1);

CREATE OR REPLACE FUNCTION times_half(p1 IN INTEGER)
RETURN INTEGER
AS
BEGIN
  RETURN p1 / 2;
END;
/

iSQL> SELECT times_half(times_half(8)) FROM t1;
TIMES_HALF(TIMES_HALF(8))
-----
2
1 row selected.
```

3.7 RETURN

3.7.3.2 Example 2

```
CREATE TABLE t1(i1 INTEGER, i2 INTEGER, i3 INTEGER);

INSERT INTO t1 VALUES(1,1,1);
INSERT INTO t1 VALUES(10,10,10);
INSERT INTO t1 VALUES(100,100,100);

CREATE OR REPLACE FUNCTION max_all_val
RETURN INTEGER
AS
    v1 INTEGER;
BEGIN
    SELECT MAX(i1) INTO v1 FROM t1;
    RETURN v1;
END;
/

iSQL> SELECT max_all_val FROM t1;
MAX_ALL_VAL
-----
100
100
100
3 rows selected.
```

3.7.3.3 Example 3

```
CREATE TABLE t4(i1 INTEGER, i2 INTEGER);

INSERT INTO t4 VALUES(3, 0);
INSERT INTO t4 VALUES(2, 0);
INSERT INTO t4 VALUES(1, 0);
INSERT INTO t4 VALUES(0, 0);

CREATE OR REPLACE FUNCTION func_plus_10(p1 INTEGER)
RETURN INTEGER
AS
BEGIN
    RETURN p1+10;
END;
/

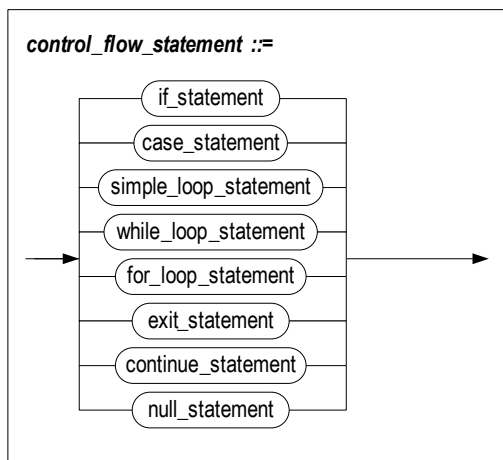
iSQL> SELECT func_plus_10(i1) FROM t4;
FUNC_PLUS_10(I1)
-----
13
12
11
10
4 rows selected.
```

4 Control Flow Statements

4.1 Overview

This chapter describes how to use control flow statements in a stored procedure body.

4.1.1 Syntax

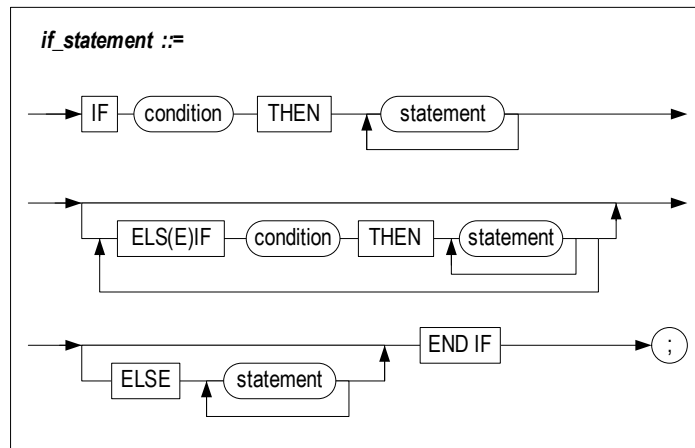


ALTIBASE HDB supports the use of the following control flow statements in stored procedures:

- The IF and CASE conditional statements
- The LOOP, WHILE and FOR loop constructs, which cause multiple statements to be repeatedly executed
- The EXIT and CONTINUE statements, which are used to control the iteration of loops
- The NULL statement, which indicates that nothing is to be executed
- The GOTO statement, which is used to transfer control to a particular point

4.2 IF

4.2.1 Syntax



4.2.2 Purpose

This is a conditional construct that determines where execution continues depending on whether or not a given condition is satisfied. The IF clause checks the condition and passes control to the THEN clause if the condition is true, or to the ELSE clause if the condition is false or NULL.

4.2.2.1 condition

All conditions that are available for use in the WHERE clause of SQL statements can be used here. For more information about the conditions that are supported in SQL, please refer to the *SQL Reference*.

4.2.2.2 ELS(E)IF

Use this clause to specify another condition to be checked when the previous IF condition is FALSE.

The spellings "ELSEIF" and "ELSIF" are both acceptable.

One IF clause can have multiple ELS(E)IF clauses. The ELS(E)IF clause is optional.

4.2.2.3 ELSE

This clause is used to specify what to do when all of the preceding IF and ELS(E)IF conditions are FALSE. One IF clause can have only one corresponding ELSE clause. The ELSE clause can be omitted.

4.2.2.4 Nested IF Constructs

IF constructs can be nested within other IF constructs. That is, one IF construct can be located within a series of statements that are executed depending on the outcome of another IF, ELS(E)IF, or ELSE

4.2 IF

clause. An END IF clause must be provided for every IF clause.

4.2.3 Examples

4.2.3.1 Example 1

```
CREATE OR REPLACE PROCEDURE proc1
AS
  CURSOR c1 IS SELECT eno, emp_job, salary FROM employees;
  emp_id employees.eno%TYPE;
  e_job employees.emp_job%TYPE;
  e_salary employees.salary%TYPE;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO emp_id, e_job, e_salary;
    EXIT WHEN c1%NOTFOUND;

    IF e_salary IS NULL THEN
      IF e_job = 'CEO' THEN
        e_salary := 5000;
      ELSIF e_job = 'manager' THEN
        e_salary := 4500;
      ELSIF e_job = 'engineer' THEN
        e_salary := 4300;
      ELSIF e_job = 'programmer' THEN
        e_salary := 4100;
      ELSE
        e_salary := 4000;
      END IF;

      UPDATE employees SET salary = e_salary WHERE eno = emp_id;
    END IF;
  END LOOP;
  CLOSE c1;
END;
/

iSQL> SELECT eno, emp_job FROM employees WHERE salary IS NULL;
ENO      EMP_JOB
-----
1        CEO
8        manager
20       sales rep
3 rows selected.

iSQL> EXEC proc1;
EXECUTE success.
iSQL> SELECT eno, emp_job, salary FROM employees
WHERE eno=1 OR eno=8 OR eno=20;
ENO      EMP_JOB      SALARY
-----
1        CEO          5000
8        manager      4500
20       sales rep     4000
3 rows selected.
```

4.2.3.2 Example 2

```
CREATE TABLE t1 (i1 VARCHAR(20), i2 NUMBER, i3 DATE);
```

```

CREATE TABLE t2 (i1 VARCHAR(20), i2 NUMBER, i3 DATE);

INSERT INTO t1 VALUES ('21-JUL-2001', 2, '01-JUL-2000');
INSERT INTO t2 VALUES (NULL,NULL,'01-FEB-1990');
INSERT INTO t2 VALUES (NULL,NULL,'02-FEB-1990');

CREATE OR REPLACE FUNCTION func2
(p1 IN DATE, p2 IN CHAR(30))
RETURN NUMBER
AS
BEGIN
  RETURN (TO_NUMBER(TO_CHAR(p1, 'dd')) + TO_NUMBER(p2));
END;
/
CREATE OR REPLACE FUNCTION func1
(p1 IN DATE, p2 IN DATE)
RETURN DATE
AS
BEGIN
  IF p1 >= p2 THEN
    RETURN add_months(p1, 3);
  ELSE
    RETURN add_months(p1, 4);
  END IF;
END;
/

CREATE OR REPLACE PROCEDURE proc1
AS
  v1 VARCHAR(20);
  v2 NUMBER;
  v3 DATE;
BEGIN
  SELECT i1, func2(TO_DATE(i1), TO_CHAR(i3, 'yyyy')), i3
  INTO v1,v2,v3 FROM t1 WHERE i2 = 2;
  INSERT INTO t2 VALUES (v1,v2,v3);

  IF v2 not in (2001, 2002, 2003) AND v1 = '21-JUL-2001' THEN
    UPDATE t2
      SET i1 = func1(v1, '17-JUL-2001'),
          i2 = nvl(i2, 10)
      WHERE i3 = '01-FEB-1990';

    UPDATE t2
      SET i1 = func1(v1, '27-JUL-2001'),
          i2 = nvl(i2, 10*2)
      WHERE i3 = '02-FEB-1990';
  END IF;
END;
/

iSQL> EXEC proc1;
EXECUTE success.
iSQL> SELECT * FROM t2;
T2.I1          T2.I2          T2.I3
-----
21-JUL-2001    2021          2000/07/01 00:00:00
21-OCT-01      10            1990/02/01 00:00:00
21-NOV-01      20            1990/02/02 00:00:00
3 rows selected.

```

4.2 IF

4.2.3.3 Example 3

```
CREATE TABLE payroll(
  eno INTEGER,
  bonus NUMBER(10, 2));

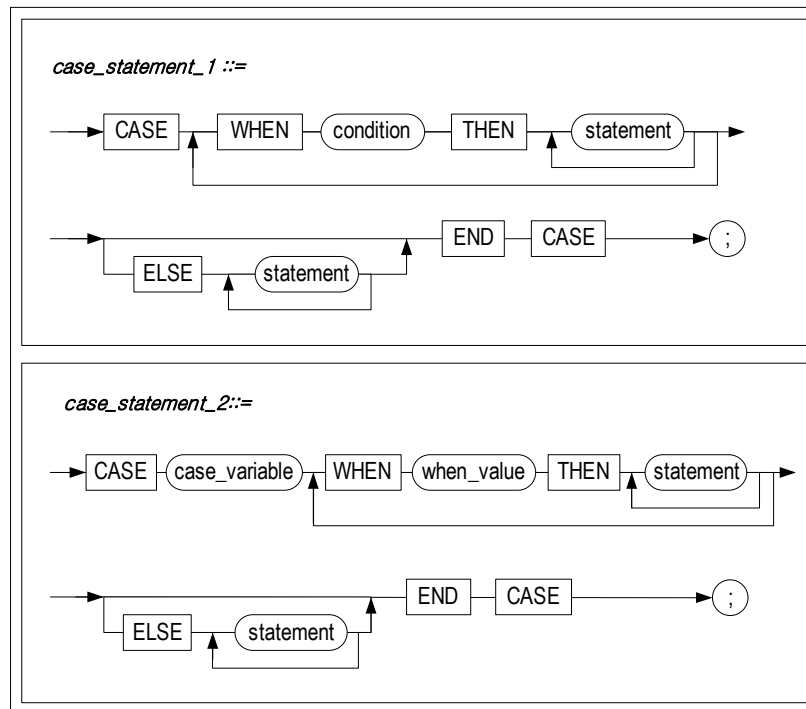
CREATE OR REPLACE PROCEDURE proc1
AS
BEGIN
  DECLARE
    CURSOR c1 IS
      SELECT DISTINCT(enos), SUM(qty) FROM orders GROUP BY eno;
    emp_id orders.eno%TYPE;
    sum_qty orders.qty%TYPE;
    bonus NUMBER(10, 2);
  BEGIN
    OPEN c1;
    IF c1%ISOPEN THEN
      LOOP
        FETCH c1 INTO emp_id, sum_qty;
        EXIT WHEN c1%NOTFOUND;
        IF sum_qty > 25000 THEN
          bonus := 1000;
        ELSIF sum_qty > 15000 THEN
          bonus := 500;
        ELSE
          bonus := 200;
        END IF;

        INSERT INTO payroll VALUES(emp_id, bonus);
      END LOOP;
    END IF;
  END;
END;
/

iSQL> EXEC proc1;
EXECUTE success.
iSQL> SELECT DISTINCT(enos), SUM(qty) sum FROM orders GROUP BY eno;
ENO      SUM
-----
12      17870
19      25350
20      13210
3 rows selected.
iSQL> SELECT * FROM payroll;
PAYROLL.ENO    PAYROLL.BONUS
-----
12              500
19             1000
20              200
3 rows selected.
```


4.3 CASE

4.3.1 Syntax



4.3.2 Purpose

CASE is a conditional construct that determines the flow of execution on the basis of the value of some variable. Its functionality is similar to that of the IF statement, however, it is more easily legible.

As can be seen in the above diagram, the CASE statement can have one of two forms:

- *case_statement_1*: The first is used to execute a desired statement or series of statements when the specified condition is satisfied.
- *case_statement_2*: The second is used to execute a desired statement or series of statements when the variable has the specified value.

Note that both methods cannot be used together within a single CASE construct.

If none of the conditions specified in the case construct are satisfied, then the statements following the ELSE clause are executed. If the ELSE clause is omitted and none of the conditions are satisfied, then nothing is executed.

4.3.2.1 condition

This is used to specify the condition to check. It has the same form as the condition in the WHERE

4.3 CASE

clause of a SELECT SQL statement.

4.3.2.2 case_variable

This is used to specify the name of the variable that is checked to determine procedural flow within the stored procedure.

4.3.2.3 when_value

This is the value with which *case_variable* is compared. If they are the same, the statement or statements following the THEN statement will be executed.

4.3.2.4 ELSE

If none of the WHEN conditions are satisfied in the case of *case_statement_1*, or if *case_variable* does not match any *when_value* in the case of *case_statement_2*, the statements following the ELSE clause will be executed. The ELSE clause can be omitted, and only one ELSE clause can be specified for one CASE construct. If there is no ELSE clause and none of the conditions are satisfied, no statement will be executed.

4.3.3 Example

4.3.3.1 Example 1

```
CREATE OR REPLACE PROCEDURE proc1
AS
  CURSOR c1 IS SELECT eno, emp_job, salary FROM employees;
  emp_id employees.eno%TYPE;
  e_job employees.emp_job%TYPE;
  e_salary employees.salary%TYPE;
BEGIN
  OPEN c1;

  LOOP
    FETCH c1 INTO emp_id, e_job, e_salary;
    EXIT WHEN c1%NOTFOUND;

    IF e_salary IS NULL THEN
      CASE
        WHEN e_job = 'CEO' THEN e_salary := 5000;
        WHEN e_job = 'manager' THEN e_salary := 4500;
        WHEN e_job = 'engineer' THEN e_salary := 4300;
        WHEN e_job = 'programmer' THEN e_salary := 4100;
        ELSE e_salary := 4000000;
      END CASE;
      UPDATE employees SET salary = e_salary WHERE eno = emp_id;
    END IF;

  END LOOP;

  CLOSE c1;
END;
/

iSQL> EXEC proc1;
```

```
EXECUTE success.
isQL> SELECT eno, emp_job, salary FROM employees
WHERE eno=1 OR eno=8 OR eno=20;
ENO      EMP_JOB      SALARY
-----
1        CEO          5000
8        manager      4500
20       sales rep    4000
3 rows selected.
```

4.3.3.2 Example 2

```
@schema.sql
```

```
CREATE OR REPLACE PROCEDURE proc1
AS
  CURSOR c1 IS SELECT eno, emp_job, salary FROM employees;
  emp_id employees.eno%TYPE;
  e_job employees.emp_job%TYPE;
  e_salary employees.salary%TYPE;
BEGIN
  OPEN c1;

  LOOP
    FETCH c1 INTO emp_id, e_job, e_salary;
    EXIT WHEN c1%NOTFOUND;

    IF e_salary IS NULL THEN
      CASE e_job
        WHEN 'CEO' THEN e_salary := 5000;
        WHEN 'manager' THEN e_salary := 4500;
        WHEN 'engineer' THEN e_salary := 4300;
        WHEN 'programmer' THEN e_salary := 4100;
        ELSE e_salary := 4000;
      END CASE;
      UPDATE employees SET salary = e_salary WHERE eno = emp_id;
    END IF;

  END LOOP;

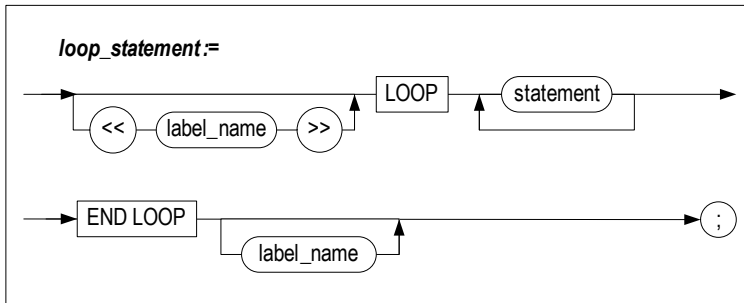
  CLOSE c1;
END;
/
```

```
isQL> SELECT eno, emp_job FROM employees WHERE salary IS NULL;
ENO      EMP_JOB
-----
1        CEO
8        manager
20       sales rep
3 rows selected.
isQL> EXEC proc1;
EXECUTE success.
isQL> SELECT eno, emp_job, salary FROM employees WHERE eno=1 OR eno=8 OR
eno=20;
ENO      EMP_JOB      SALARY
-----
1        CEO          5000
8        manager      4500
20       sales rep    4000
3 rows selected.
```

4.4 LOOP

4.4 LOOP

4.4.1 Syntax



4.4.2 Purpose

The LOOP construct is used to repeatedly execute a desired statement or series of statements without using a particular condition to control execution.

Bear in mind that using the LOOP construct without an EXIT statement or some other way of exiting the loop can create an infinite loop, which can cause system problems.

4.4.3 Example

```
CREATE TABLE item(id INTEGER, counter NUMBER(2));
```

```
CREATE OR REPLACE PROCEDURE proc1
```

```
AS
```

```
BEGIN
```

```
  DECLARE
```

```
    v_id item.id%TYPE := 501;
```

```
    v_counter NUMBER(2) := 1;
```

```
  BEGIN
```

```
    LOOP
```

```
      INSERT INTO item VALUES(v_id, v_counter);
```

```
      v_counter := v_counter + 1;
```

```
      EXIT WHEN v_counter > 10;
```

```
    END LOOP;
```

```
  END;
```

```
END;
```

```
/
```

```
iSQL> EXEC proc1;
```

```
EXECUTE success.
```

```
iSQL> SELECT * FROM item;
```

```
ITEM.ID    ITEM.COUNTER
```

```
-----
```

```
501         1
```

```
501         2
```

```
...
```

```
501         9
```

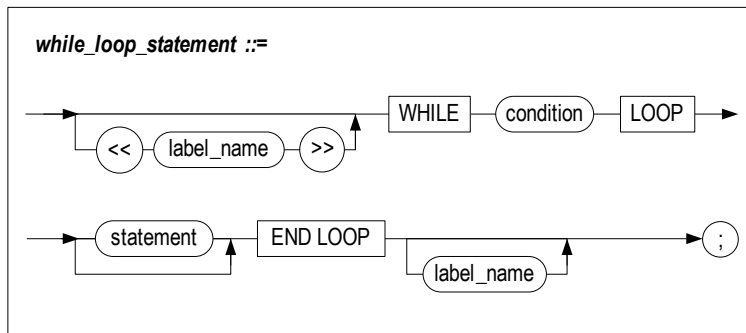
```
501        10
```

```
10 rows selected.
```

4.5 WHILE LOOP

4.5 WHILE LOOP

4.5.1 Syntax



4.5.2 Purpose

The WHILE LOOP construct iterates the statements in the loop body as long as the condition remains true. If this condition is not true the first time it is executed, the statements in the loop will not be executed even once, and control will pass to the statement following the loop.

4.5.2.1 condition

This conditional expression is repeatedly evaluated to determine whether to iterate through the loop.

All conditions that are available for use in the WHERE clause of SQL statements can be used here. For more information about the conditions that are supported in SQL, please refer to the *SQL Reference*.

4.5.3 Example

```
CREATE TABLE t1(i1 INTEGER, i2 INTEGER, i3 INTEGER);

CREATE OR REPLACE PROCEDURE proc1
AS
  v1 INTEGER;
BEGIN
  v1 := 1;

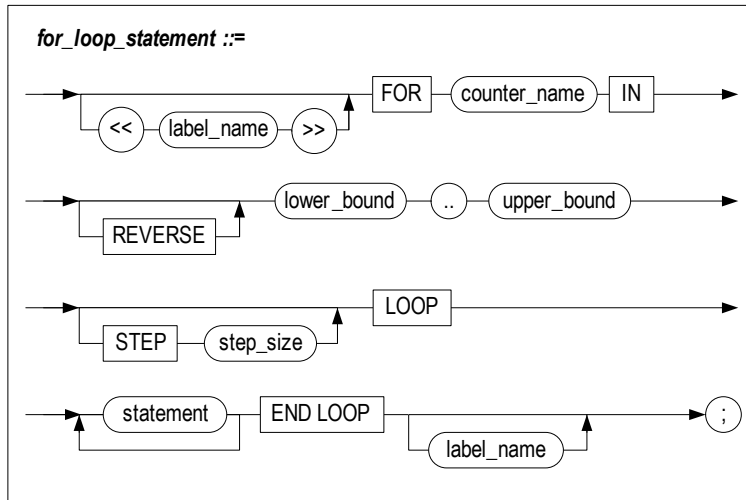
  WHILE v1 < 3 LOOP
    v1 := v1 + 1;
    INSERT INTO t1 VALUES (v1, v1, v1);
    IF v1 = 2 THEN
      CONTINUE;
    END IF;
  END LOOP;

END;
/
iSQL> EXEC proc1;
```

```
EXECUTE success.  
isQL> SELECT * FROM t1;  
T1.I1    T1.I2    T1.I3  
-----  
2         2         2  
3         3         3  
2 rows selected.
```

4.6 FOR LOOP

4.6.1 Syntax



4.6.2 Purpose

The FOR LOOP construct is used to repeatedly execute a desired statement or series of statements a predetermined number of times. The range is specified using two periods ("**..**"), and is only evaluated once, before entering the FOR loop. If the lower and higher bounds are set to the same value, the loop body is iterated only one time.

4.6.2.1 counter_name

This loop construct uses an integer variable that increases or decreases to a fixed final value. This variable does not need to be expressly declared. The scope of this variable is limited to the statements between the **LOOP** and **END LOOP** clauses. No other value can be assigned to this variable.

4.6.2.2 REVERSE

This statement is optionally used to specify that the counter is to decrease from *upper_bound* to *lower_bound*.

4.6.2.3 lower_bound

This is the minimum value that the counter can have. It must take the form of an integer, or an expression that is compatible with the **INTEGER** type.

lower_bound can be a local variable. Note however that the value of the variable is determined and stored only once, at the beginning of the first iteration of the FOR loop. This means that subsequently changing the value of this local variable during execution of the FOR loop will have no effect on the number of iterations.

If *lower_bound* is a non-integer number, it is rounded to the nearest integer.

4.6.2.4 upper_bound

This is the maximum value that the counter can have. Like *lower_bound*, it must take the form of an integer, or an expression that is compatible with the INTEGER type. If it is a non-integer number, it is rounded to the nearest integer.

If the value of *upper_bound* is lower than that of *lower_bound* upon first execution of the FOR statement, no error is raised; the entire FOR loop is skipped, and control is passed to the following statement.

As with *lower_bound*, *upper_bound* can be a local variable, but as the value of the variable is determined and stored only at the beginning of the first iteration of the FOR loop, subsequently changing the value of this local variable will have no effect on the number of iterations.

4.6.2.5 step_size

step_size is used to set the amount by which the value of the counter is incremented or decremented. If it is omitted, 1 is the default value.

Note that *step_size* cannot be set to a value less than 1. Additionally, if it is a non-integer number, it is rounded to the nearest integer.

4.6.3 Example

```
--#####
--FOR LOOP
--#####
```

4.6.3.1 Example 1

```
CREATE TABLE t6(i1 INTEGER, sum INTEGER);

CREATE OR REPLACE PROCEDURE proc1
AS
  v1 INTEGER;
  sum INTEGER := 0;
BEGIN
  FOR i IN 1 .. 50 LOOP
    v1 := 2 * i - 1;
    sum := sum + v1;
    INSERT INTO t6 VALUES(v1, sum);
  END LOOP;
END;
/

iSQL> EXEC proc1;
EXECUTE success.
iSQL> SELECT * FROM t6;
T6.I1      T6.SUM
-----
1          1
```

4.6 FOR LOOP

```
3          4
5          9
...
97         2401
99         2500
50 rows selected.
```

4.6.3.2 Example 2

```
CREATE OR REPLACE PROCEDURE proc1
AS
  eno_count INTEGER;
BEGIN
  SELECT COUNT(eno) INTO eno_count FROM employees;
  FOR i IN 1 .. eno_count LOOP
    UPDATE employees SET salary = salary * 1.2 WHERE eno = i;
  END LOOP;
END;
/

isQL> SELECT eno, salary FROM employees WHERE eno in (11,12,13);
ENO      SALARY
-----
11       2750
12       1890
13       980
3 rows selected.

isQL> EXEC proc1;
EXECUTE success.
isQL> SELECT eno, salary FROM employees WHERE eno IN (11,12,13);
ENO      SALARY
-----
11       3300
12       2268
13       1176
3 rows selected.
```

4.6.3.3 Example 3

```
CREATE TABLE t1(i1 INTEGER, i2 INTEGER, i3 INTEGER);
CREATE OR REPLACE PROCEDURE proc1
AS
BEGIN
  <<a>>
  INSERT INTO t1 VALUES (1,1,1);
  IF 1 = 1 THEN
    NULL;
  END IF;
  <<b>>
  FOR v1 IN 1 .. 3 LOOP
    <<c>>
    FOR v1 IN 1 .. 3 LOOP
      INSERT INTO t1 VALUES (b.v1, b.v1, c.v1);
    END LOOP;
  END LOOP;
END;
/

isQL> EXEC proc1;
EXECUTE success.
isQL> SELECT * FROM t1;
```

```

T1.I1      T1.I2      T1.I3
-----
1          1          1
1          1          1
1          1          2
1          1          3
2          2          1
2          2          2
2          2          3
3          3          1
3          3          2
3          3          3
10 rows selected.

```

```

--#####
--reverse
--#####

```

```

CREATE TABLE t6(i1 INTEGER, sum INTEGER);

CREATE OR REPLACE PROCEDURE procl
AS
    sum INTEGER := 0;
BEGIN
    FOR i IN reverse 1 .. 100 LOOP
        sum := sum + i;
        INSERT INTO t6 VALUES(i, sum);
    END LOOP;
END;
/

```

```

iSQL> EXEC procl;
EXECUTE success.
iSQL> SELECT * FROM t6;
T6.I1      T6.SUM
-----
100        100
99         199
98         297
...
3          5047
2          5049
1          5050
100 rows selected.

```

```

--#####
--step
--#####

```

```

CREATE TABLE t6(i1 INTEGER, sum INTEGER);

CREATE OR REPLACE PROCEDURE procl
AS
    sum INTEGER := 0;
BEGIN
    FOR i IN 1 .. 100 STEP 2 LOOP
        sum := sum + i;
        INSERT INTO t6 VALUES(i, sum);
    END LOOP;
END;
/

iSQL> EXEC procl;

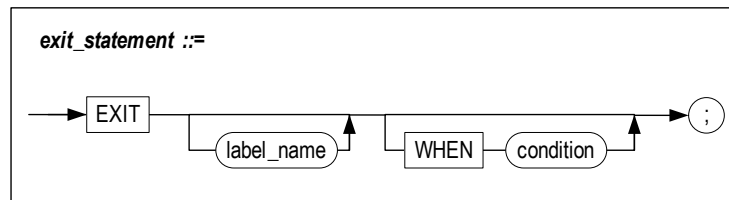
```

4.6 FOR LOOP

```
EXECUTE success.  
iSQL> SELECT * FROM t6;  
T6.I1      T6.SUM  
-----  
1          1  
3          4  
5          9  
...  
97         2401  
99         2500  
50 rows selected.
```

4.7 EXIT

4.7.1 Syntax



4.7.2 Purpose

The EXIT statement is used to terminate the iteration of a loop. If *label_name* is specified, iteration of the loop specified using *label_name* is terminated. If *label_name* is not specified, iteration of the innermost loop is terminated.

If the EXIT statement is used anywhere other than inside a loop, an error will occur.

```

<<outer>>
LOOP
  ...
  LOOP
    ...
    EXIT outer WHEN ... -- EXIT both LOOPS
  END LOOP;
  ...
END LOOP outer;

EXIT WHEN count > 100;

IF count > 100 THEN
  EXIT;
END IF;

```

The EXIT statement can be used inside any of the following LOOP statements:

- LOOP
- WHILE LOOP
- FOR LOOP
- CURSOR FOR LOOP

4.7.2.1 label_name

To exit a loop other than the innermost loop, define a label immediately before the corresponding loop, and specify the name here.

4.7 EXIT

4.7.2.2 WHEN condition

A conditional expression can be specified in the WHEN clause, to make it possible to exit the loop only when a certain condition is satisfied. All conditions that are available for use in the WHERE clause of a SELECT statement can be used in this expression. For more information about the conditions that are supported in ALTIBASE HDB, please refer to the *ALTIBASE HDB SQL Reference*.

When an EXIT statement is encountered, if the condition specified in the WHEN clause is true, iteration of the innermost loop (or the loop identified using the label) terminates, and control is passed to the next statement.

Using EXIT WHEN is akin to using a simple IF construct. The following are logically identical:

```
EXIT WHEN count > 100;
```

```
IF count > 100 THEN
  EXIT;
END IF;
```

4.7.3 Example

```
CREATE TABLE stock(
  gno BYTE(5) primary key,
  stock INTEGER,
  price numeric(10,2));

CREATE OR REPLACE PROCEDURE proc1
AS
  CURSOR c1 IS SELECT gno, stock, price FROM goods;
  rec1 c1%ROWTYPE;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO rec1;
    IF c1%FOUND THEN
      IF rec1.stock > 0 AND rec1.stock < 1000 THEN
        INSERT INTO stock VALUES(rec1.gno, rec1.stock, rec1.price);
      END IF;
    ELSIF c1%NOTFOUND THEN
      EXIT;
    END IF;
  END LOOP;
  CLOSE c1;
END;
/
iSQL> EXEC proc1;
EXECUTE success.
iSQL> SELECT * FROM stock;
STOCK.GNO      STOCK.STOCK      STOCK.PRICE
-----
A111100002      100              98000
B111100001      780              35800
D111100003      650              45100
E111100001      900              2290.54
E111100006      900              2338.62
5 rows selected.

--#####
--EXIT WHEN
--#####
```

```

CREATE OR REPLACE PROCEDURE procl
AS
  CURSOR c1 IS SELECT gno, stock, price FROM goods;
  rec1 c1%ROWTYPE;
BEGIN
  OPEN c1;
  IF c1%ISOPEN THEN
    LOOP
      FETCH c1 INTO rec1;
      EXIT WHEN c1%NOTFOUND;
      IF rec1.stock > 0 AND rec1.stock < 1000 THEN
        INSERT INTO stock VALUES(rec1.gno, rec1.stock, rec1.price);
      END IF;
    END LOOP;
  END IF;
  CLOSE c1;
END;
/

```

```

iSQL> EXEC procl;
EXECUTE success.
iSQL> SELECT * FROM stock;

```

STOCK.GNO	STOCK.STOCK	STOCK.PRICE
A111100002	100	98000
B111100001	780	35800
D111100003	650	45100
E111100001	900	2290.54
E111100006	900	2338.62

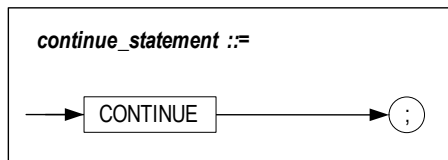
```

5 rows selected.

```

4.8 CONTINUE

4.8.1 Syntax



4.8.2 Purpose

The CONTINUE statement causes subsequent statements in the loop in which it is found to be ignored, and passes control to the beginning of the loop. That is, it terminates the current iteration of the loop.

The CONTINUE statement can be used inside any of the following loop statements:

- LOOP
- WHILE LOOP
- FOR LOOP
- CURSOR FOR LOOP

If the CONTINUE statement is used anywhere other than inside a loop, an error will occur.

4.8.3 Example

```

CREATE TABLE t8(i1 INTEGER, mathpower INTEGER default 0);

INSERT INTO t8(i1) VALUES(7);
INSERT INTO t8(i1) VALUES(3);
INSERT INTO t8(i1) VALUES(20);
INSERT INTO t8(i1) VALUES(15);
INSERT INTO t8(i1) VALUES(6);
INSERT INTO t8(i1) VALUES(1);
INSERT INTO t8(i1) VALUES(9);

CREATE OR REPLACE PROCEDURE proc1
AS
BEGIN
  DECLARE
    CURSOR c1 IS SELECT i1 FROM t8;
    rec c1%ROWTYPE;
  BEGIN
    OPEN c1;
    LOOP
      FETCH c1 INTO rec;
      EXIT WHEN c1%NOTFOUND;
    
```



```

    IF power(rec.i1, rec.i1) > 50000 THEN
        continue;
    ELSE
        UPDATE t8 SET mathpower = power(rec.i1, rec.i1)
        WHERE i1 = rec.i1;
    END IF;
END LOOP;
CLOSE c1;
END;
END;
/

```

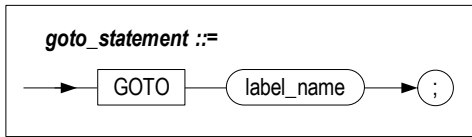
```

iSQL> EXEC procl;
EXECUTE success.
iSQL> SELECT * FROM t8;
T8.I1      T8.MATHPOWER
-----
7          0
20         0
15         0
9          0
3          27
6         46656
1          1
7 rows selected.

```

4.9 GOTO

4.9.1 Syntax



4.9.2 Purpose

This statement passes control to the specified label.

4.9.2.1 label_name

This is the name of the label to which control will be transferred.

4.9.3 Limitations

The use of the GOTO statement is limited as follows:

1. When used within an IF or CASE block, it cannot be used to transfer control from one of the alternative execution paths, that is, one of the statement blocks preceded by a THEN, ELS(E)IF, ELSE or WHEN statement, to another. If this is attempted, an error will occur when attempting to compile the procedure, as seen below:

```

CREATE OR REPLACE PROCEDURE PROC1
AS
    V1 INTEGER;
BEGIN
    V1 := 1;
    IF V1 = 1 THEN
        GOTO LABEL1;
    ELSE
        <<LABEL1>>
        PRINTLN(V1);
    END IF;
END;
/
[ERR-3120F : Illegal GOTO statement.
In PROC1
0007 :    GOTO LABEL1;
          ^      ^
]

```

2. It cannot be used to transfer control from an external block to an internal block. This limitation applies to all BEGIN/END blocks and all loop constructs.

```

CREATE OR REPLACE PROCEDURE PROC1
AS
    V1 INTEGER;
BEGIN

```

```

V1 := 1;
DECLARE
  V2 INTEGER;
BEGIN
  <<LABEL1>>
  V2 := 1;
END;
GOTO LABEL1;
END;
/
[ERR-3120F : Illegal GOTO statement.
In PROC1
0012 : GOTO LABEL1;
      ^   ^
]

```

3. It cannot be used to pass control from within an exception handler to another location within the block to which the exception handler pertains. Therefore, in the following example, an error is returned.

- **Example 1**

```

CREATE OR REPLACE PROCEDURE PROC1
AS
  E1 EXCEPTION;
BEGIN
  RAISE E1;
  <<LABEL1>>
  PRINTLN('END');
  EXCEPTION
    WHEN E1 THEN
      GOTO LABEL1;
END;
/
[ERR-3120F : Illegal GOTO statement.
In PROC1
0010 : GOTO LABEL1;
      ^   ^
]

```

- However, it is acceptable to use a GOTO statement to pass control from an exception handler in one block to the body of an outer block. In the following example, before the value of *V1* reaches 5, four exceptions occur. After that, execution terminates normally.

```

CREATE OR REPLACE PROCEDURE PROC1
AS
  E1 EXCEPTION;
  V1 INTEGER;
BEGIN
  V1 := 1;
  <<LABEL1>>
  V1 := V1 + 1;
  PRINTLN('BLOCK1');
  BEGIN
    PRINTLN('BLOCK2');
    PRINTLN(V1);
    IF V1 = 5 THEN
      PRINTLN('goto label2 ' || v1);
      GOTO LABEL2;
    ELSE
      RAISE E1;
    END IF;
  EXCEPTION
    WHEN E1 THEN
      PRINTLN('goto label1 ' || v1);

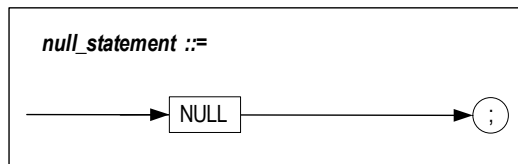
```

4.9 GOTO

```
        GOTO LABEL1;
    END;
    <<LABEL2>>
    PRINTLN('BLOCK1 AFTER BLOCK2');
END;
/
iSQL> EXEC PROC1;
BLOCK1
BLOCK2
2
goto label1 2
BLOCK1
BLOCK2
3
goto label1 3
BLOCK1
BLOCK2
4
goto label1 4
BLOCK1
BLOCK2
5
goto label2 5
BLOCK1 AFTER BLOCK2
Execute success.
```

4.10 NULL

4.10.1 Syntax



4.10.2 Purpose

The NULL statement does nothing. It is used to expressly pass control to the next statement.

4.10.3 Example

```

CREATE OR REPLACE PROCEDURE bonus (amount NUMBER(10,2))
AS
    CURSOR c1 IS SELECT eno, sum(qty) FROM orders group by eno;
    order_eno orders.eno%TYPE;
    order_qty orders.qty%TYPE;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO order_eno, order_qty;
        EXIT WHEN c1%NOTFOUND;
        IF order_qty > 20000 THEN
            UPDATE employees SET salary = salary + amount
            WHERE eno = order_eno;
        ELSE
            NULL;
        END IF;
    END LOOP;
    CLOSE c1;
END;
/

```

```

iSQL> SELECT e.eno, salary, sum(qty)
FROM employees e, orders o
WHERE e.eno = o.eno
group by e.eno, salary;
ENO      SALARY      SUM(QTY)
-----
12      1890      17870
19      1800      25350
20              13210
3 rows selected.
iSQL> EXEC bonus(75);
EXECUTE success.
iSQL> SELECT eno, salary FROM employees WHERE eno = 19;
ENO      SALARY
-----
19      1875
1 row selected.

```


5 Using Cursors

This chapter describes how to manage and use cursors.

5.1 Overview

There are two ways of reading table records within a stored procedure: using the SELECT INTO statement, or using a cursor.

The SELECT INTO statement can be used to read only a single record. If more than one record is returned by a SELECT INTO statement, an error will be raised. Therefore, in situations where it can be expected that more than one record will be returned, it is necessary to use a cursor.

5.1.1 Declaring a Cursor

A cursor must be explicitly declared in the declare section of a stored procedure block, along with the SELECT statement with which it is used. After it has been declared, a cursor can be managed in one of the following two ways:

- [Cursor Management Using OPEN, FETCH, and CLOSE](#)
- [Cursor Management Using a Cursor FOR LOOP](#)

5.1.2 Cursor Management Using OPEN, FETCH, and CLOSE

A cursor can be controlled in the block body using the OPEN, FETCH, and CLOSE statements. The OPEN statement is used to initialize the cursor. The FETCH statement is then executed repeatedly to retrieve rows. Finally, the cursor is released using the CLOSE statement.

5.1.2.1 OPEN

This statement is used to initialize all of the resources that are necessary in order to use a cursor. If user-defined parameters were specified when the cursor was defined, they are passed to the cursor using the OPEN statement.

5.1.2.2 FETCH

The FETCH statement is used to retrieve one record at a time from the set of results that satisfy the cursor's SELECT statement and store it in one or more variables. Each column can be stored in a separate variable, or the entire row can be stored in a RECORD type variable, typically declared using %ROWTYPE, having the same number and type of fields as the retrieved record.

For an explanation of RECORD type variables, please refer to the following chapter: [Chapter6: User-Defined Types](#)

5.1.2.3 CLOSE

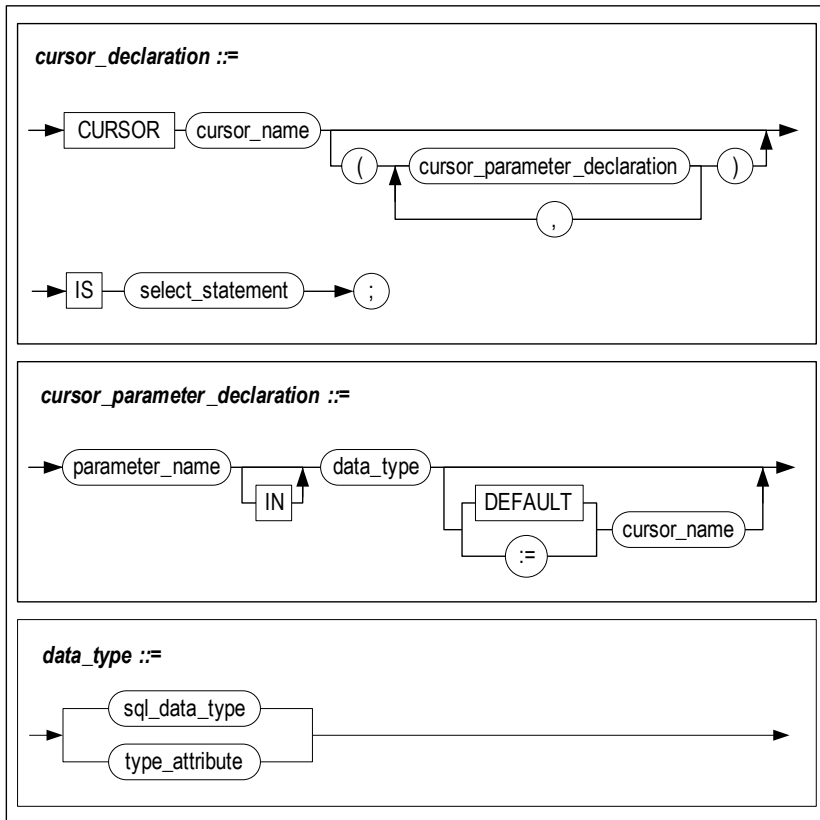
This is the step of releasing the resources allocated to a cursor that is no longer in use. A cursor must be closed before the procedure or function in which the cursor was declared is terminated.

5.1.3 Cursor Management Using a Cursor FOR LOOP

This is a kind of loop that executes all of the OPEN, FETCH, and CLOSE statements. Iteration of the loop continues until there are no more records left to process. This statement is convenient to use because it obviates the need to use explicit OPEN and CLOSE statements.

5.2 CURSOR

5.2.1 Syntax



5.2.2 Purpose

The CURSOR statement is used to declare a cursor. It must specify the name of the cursor and the SELECT statement that the cursor uses to retrieve records.

5.2.2.1 cursor_name

This is the name of the cursor, which is referenced in the OPEN, FETCH, CLOSE, and Cursor FOR LOOP statements.

5.2.2.2 cursor_parameter_declaration

In cases where it is necessary to use parameters with a cursor's SELECT statement, they can be defined for the cursor in the same way that they are for stored procedures.

The following limitations apply to the use of parameters with cursors:

- Cursor parameters can be used only within SELECT statements.
- The use of %ROWTYPE is not supported.
- Cursor parameters cannot be OUT or IN/OUT parameters.

A value is assigned to a cursor parameter using an OPEN CURSOR or CURSOR FOR statement. This value is used when the cursor's SELECT statement is executed.

```
DECLARE
CURSOR c1 IS
  SELECT eno, e_lastname, e_firstname, emp_job, salary
  FROM emp
  WHERE sal > 2000;
CURSOR c2
  (low INTEGER DEFAULT 0,
   high INTEGER DEFAULT 99) IS
  SELECT .....
```

5.2.2.3 data_type

Please refer to the [3.2 Declaring Local Variables](#) in Chapter 3 of this manual.

5.2.3 Example

```
CREATE TABLE highsal
(enno INTEGER, e_firstname CHAR(20), e_lastname CHAR(20), salary NUM-
BER(10,2));
```

```
CREATE OR REPLACE PROCEDURE proc1
AS
BEGIN
  DECLARE
    CURSOR c1 IS
      SELECT eno, e_firstname, e_lastname, salary FROM employees
      WHERE salary IS not NULL
      ORDER BY salary desc;
    emp_first CHAR(20);
    emp_last CHAR(20);
    emp_no INTEGER;
    emp_sal NUMBER(10,2);
  BEGIN
    OPEN c1;
    FOR i IN 1 .. 5 LOOP
      FETCH c1 INTO emp_no, emp_first, emp_last, emp_sal;
      EXIT WHEN c1%NOTFOUND;
      INSERT INTO highsal VALUES(emp_no, emp_first, emp_last, emp_sal);
    END LOOP;
    CLOSE c1;
  END;
END;
```

```
iSQL> EXEC proc1;
EXECUTE success.
iSQL> SELECT * FROM highsal;
```

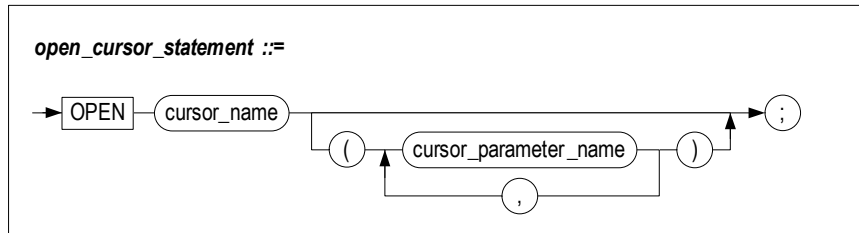
ENO	E_FIRSTNAME	E_LASTNAME	SALARY
10	Elizabeth	Bae	4000
11	Zhen	Liu	2750

5.2 CURSOR

5	Farhad	Ghorbani	2500
16	Wei-Wei	Chen	2300
14	Yuu	Miura	2003
5 rows selected.			

5.3 OPEN

5.3.1 Syntax



5.3.2 Purpose

This statement is used to initialize a cursor, execute the query, and determine the result set, so that data can be retrieved using the FETCH statement.

When this statement is executed, the system will allocate all resources required to use the cursor.

If an attempt is made to open a cursor that is already open, a `CURSOR_ALREADY_OPEN` error will be raised.

5.3.2.1 cursor_name

This is the name of the cursor to open.

A cursor having this name must have been declared in the declare section of the current block or an outer block.

5.3.2.2 cursor_parameter_name

Parameters can be optionally specified for a cursor. These parameters can be used in the associated query in place of constants or local variables.

If the cursor has parameters, they are declared as shown below.

```

DECLARE
  CURSOR c1(pname VARCHAR(40), pno INTEGER) IS
    SELECT empno, ename, job, sal
    FROM emp
    WHERE ename = pname;
BEGIN
  OPEN c1;
  .....
END;
  
```

The OPEN statement is used as follows when parameter values are passed to a cursor.

```

OPEN c1(emp_name, 100);
OPEN c1('Mary Mason', 100);
OPEN c1(emp_name, dept_no);
  
```

5.3 OPEN

5.3.3 Examples

5.3.3.1 Example 1

```
CREATE TABLE mgr
(mgr_eno INTEGER, mgr_first CHAR(20), mgr_last CHAR(20), mgr_dno SMALLINT);

CREATE OR REPLACE PROCEDURE proc1
AS
BEGIN
  DECLARE
    CURSOR emp_cur IS
      SELECT eno, e_firstname, e_lastname, dno FROM employees
      WHERE emp_job = 'manager';
    emp_no employees.eno%TYPE;
    emp_first employees.e_firstname%TYPE;
    emp_last employees.e_lastname%TYPE;
    emp_dno employees.dno%TYPE;
  BEGIN
    OPEN emp_cur;
    LOOP
      FETCH emp_cur INTO emp_no, emp_first, emp_last, emp_dno;
      EXIT WHEN emp_cur%NOTFOUND;
      INSERT INTO mgr VALUES (emp_no, emp_first, emp_last, emp_dno);
    END LOOP;
    CLOSE emp_cur;
  END;
END;
/

iSQL> EXEC proc1;
Execute success.
iSQL> select * from mgr;
MGR.MGR_ENO MGR.MGR_FIRST          MGR.MGR_LAST          MGR.MGR_DNO
-----
7           Gottlieb             Fleischer             4002
8           Xiong                Wang                  4001
16          Wei-Wei              Chen                  1001
3 rows selected.
```

5.3.3.2 Example 2

```
CREATE TABLE t1(i1 INTEGER, i2 INTEGER, i3 INTEGER);

CREATE TABLE t2(i1 INTEGER, i2 INTEGER, i3 INTEGER);

INSERT INTO t1 VALUES(1,1,1);
INSERT INTO t1 VALUES(2,2,2);
INSERT INTO t1 VALUES(30,30,30);
INSERT INTO t1 VALUES(50,50,50);

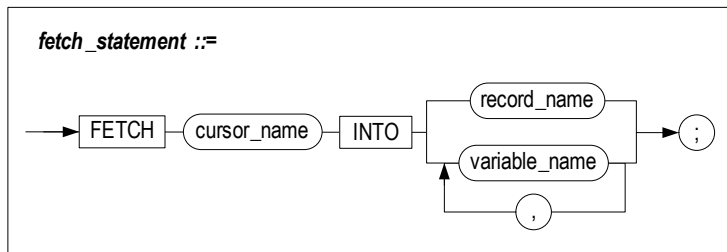
CREATE OR REPLACE PROCEDURE proc1
AS
  CURSOR c1(k1 INTEGER, k2 INTEGER, k3 INTEGER) IS
    SELECT * FROM t1
    WHERE i1 <= k1 AND i2 <= k2 AND i3 <= k3;
BEGIN
  FOR rec1 IN c1(2,2,2) LOOP
    INSERT INTO t2 VALUES (rec1.i1, rec1.i2, rec1.i3);
  END LOOP;
END;
```

```
/

iSQL> SELECT * FROM t2;
T2.I1      T2.I2      T2.I3
-----
No rows selected.
iSQL> EXEC procl;
EXECUTE success.
iSQL> SELECT * FROM t2;
T2.I1      T2.I2      T2.I3
-----
1          1          1
2          2          2
2 rows selected.
```

5.4 FETCH

5.4.1 Syntax



5.4.2 Purpose

This statement is used to obtain one row from an open cursor and store the value(s) in the variable(s) specified in the INTO clause of the SELECT statement.

A list of variables that match the column types specified in the cursor's SELECT statement is specified. Alternatively, the name of a RECORD type variable is specified, and the row retrieved from the cursor is saved in the RECORD type variable.

The following restrictions govern the use of RECORD type variables with the FETCH statement:

- Only one RECORD type variable can be used to store one retrieved row.
- It must be possible to save all of the columns retrieved by the SELECT statement into the RECORD type variable.
- RECORD type variables cannot be combined with regular variables.

If an attempt is made to fetch results from a cursor that is not open, an INVALID_CURSOR error will occur.

5.4.2.1 cursor_name

This is the name of the cursor to use to fetch records. A cursor having this name must have been declared in the declare section of the current block or an outer block.

5.4.2.2 record_name

This is used to specify the name of the RECORD type variable into which the cursor's SELECT statement retrieves records. The RECORD type variable that is used must have the same number of columns as the SELECT statement's select list, and the column types must be compatible and specified in the corresponding order.

When retrieving all columns from a table, it is convenient to declare a RECORD type variable using the %ROWTYPE attribute for the table from which the records are to be retrieved.

5.4.2.3 variable_name

This is the name of the variable into which a value will be stored. The number of such variables must be the same as the number of columns specified in the cursor's SELECT statement. Furthermore, the order of the variables must be set such that their types correspond with the respective types of the columns in the select list.

```
LOOP
  FETCH c1 INTO my_name, my_empno, my_deptno;
  EXIT WHEN c1%NOTFOUND;
END LOOP;
```

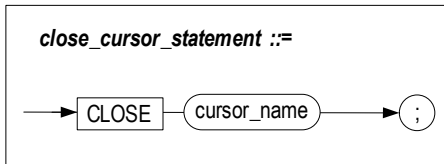
5.4.3 Example

```
CREATE TABLE emp_temp(eno INTEGER, e_firstname CHAR(20), e_lastname
CHAR(20));
CREATE OR REPLACE PROCEDURE proc1
AS
BEGIN
  DECLARE
    CURSOR c1 IS SELECT eno, e_firstname, e_lastname FROM employees;
    emp_rec c1%ROWTYPE;
  BEGIN
    OPEN c1;
    LOOP
      FETCH c1 INTO emp_rec;
      EXIT WHEN c1%NOTFOUND;
      INSERT INTO emp_temp
        VALUES(emp_rec.eno, emp_rec.e_firstname, emp_rec.e_lastname);
    END LOOP;
    CLOSE c1;
  END;
END;
/
```

```
iSQL> select eno, e_firstname, e_lastname from emp_temp;
ENO          E_FIRSTNAME          E_LASTNAME
-----
1            Chan-seung          Moon
2            Susan              Davenport
3            Ken                Kobain
.
.
.
18           John              Huxley
19           Alvar             Marquez
20           William           Blake
20 rows selected.
```

5.5 CLOSE

5.5.1 Syntax



5.5.2 Purpose

This statement is used to close an open cursor and free all associated resources.

A cursor that has already been closed can be reopened using the `OPEN` statement. If an attempt is made to close a cursor that is already closed, a `INVALID_CURSOR` error will be raised.

If the user doesn't expressly close a cursor using this statement, the cursor is automatically closed upon exiting the block in which the cursor was declared. However, it is recommended that the user use this statement to expressly close a cursor immediately after s/he has finished using it in order to return all associated resources to the system as early as possible.

5.5.2.1 cursor_name

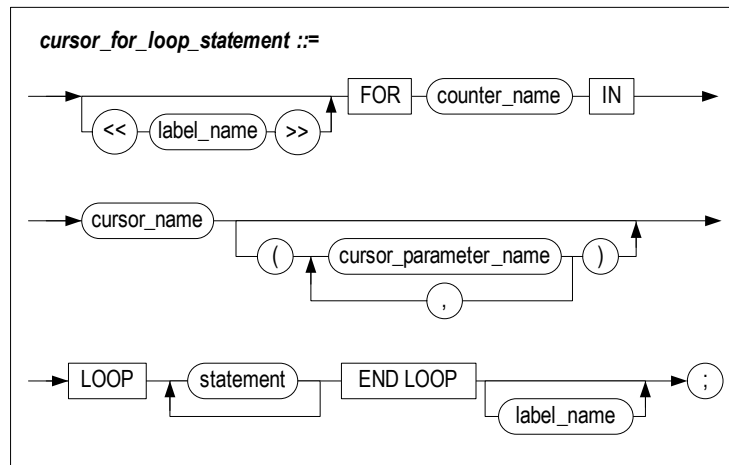
This is the name of the cursor to close.

5.5.3 Example

```
CLOSE c1;
```

5.6 Cursor FOR LOOP

5.6.1 Syntax



5.6.2 Purpose

The Cursor FOR LOOP construct automatically opens a cursor, fetches results, and closes the cursor.

A cursor FOR LOOP uses a cursor declared in the declare section of the block, and returns one of the rows retrieved by the query every time the loop iterates. The current record is saved in a RECORD type variable that can be accessed from within the loop.

5.6.2.1 *label_name*

This is used to specify a label for the loop, which will be necessary in order to designate the loop in an EXIT or CONTINUE statement.

5.6.2.2 *counter_name*

This is used to specify the name of the RECORD type variable in which one row that was fetched using the cursor will be stored. This variable does not need to be declared in the declare section of the block, because it is automatically created such that the number of columns and the types of the columns match those of the fetched rows.

A variable created in this way is referenced using the syntax "*counter_name.column_name*". When referencing a variable in this way, *column_name* is the name of a column in the select list of the cursor's SELECT statement. Therefore, when an expression is used in the select list, an alias must be specified for the expression in the select list in order to allow the expression to be referenced in this way.

5.6 Cursor FOR LOOP

5.6.2.3 cursor_name

This is used to specify the name of the cursor to use in the loop. This cursor must have been declared in the declare section of the current block or an outer block.

5.6.2.4 cursor_parameter_name

Please refer to [5.3.2.2 cursor_parameter_name](#) in this chapter.

5.6.3 Example

```
CREATE TABLE emp_temp(eno INTEGER, e_firstname CHAR(20), e_lastname
CHAR(20));
CREATE OR REPLACE PROCEDURE proc1
AS
BEGIN
  DECLARE
    CURSOR c1 IS SELECT eno, e_firstname, e_lastname FROM employees;
    emp_rec c1%ROWTYPE;
  BEGIN
    OPEN c1;
    LOOP
      FETCH c1 INTO emp_rec;
      EXIT WHEN c1%NOTFOUND;
      INSERT INTO emp_temp
        VALUES(emp_rec.eno, emp_rec.e_firstname, emp_rec.e_lastname);
    END LOOP;
    CLOSE c1;
  END;
END;
/
```

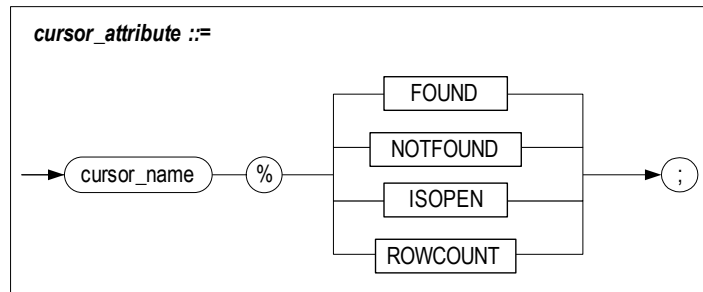
```
iSQL> SELECT * FROM emp_temp;
```

ENO	E_FIRSTNAME	E_LASTNAME
1	Chan-seung	Moon
2	Susan	Davenport
3	Ken	Kobain
.		
.		
18	John	Huxley
19	Alvar	Marquez
20	William	Blake

20 rows selected.

5.7 Cursor Attributes

5.7.1 Syntax



5.7.2 Purpose

Cursor attributes are user-accessible. With the exception of ROWCOUNT, which returns an integer, cursor attributes are Boolean type expressions that provide information about the state of a cursor.

Based on the current state of the cursor, the value of each attribute can be TRUE or FALSE.

The user can check the values of attributes of cursors declared using the DECLARE statement, and can additionally check those of implicit cursors declared within the system. Implicit cursors exist for DELETE, UPDATE, and INSERT statements, as well as for the SELECT INTO statement, which returns one record. They contain the attribute values for the cursor pertaining to the most recently executed SQL statement.

5.7.2.1 %FOUND

This attribute indicates whether any rows satisfying the condition in the cursor's SELECT statement have been found. Note however that the value of %FOUND is always FALSE in the following cases, regardless of whether or not any rows that satisfy the condition actually exist:

- A cursor that has not been opened
- A cursor for which a FETCH statement has never been executed
- A cursor that has been closed

For implicit cursors, if one or more records are affected by the execution of a DELETE, UPDATE or INSERT statement, or if a SELECT INTO statement returns at least one record, the value of %FOUND for the associated cursor is TRUE.

However, if a SELECT INTO statement returns two or more records, the TOO_MANY_ROWS exception will occur before it is possible to check the value of the %FOUND attribute. Such a case should be handled as an exception rather than by referring to the %FOUND cursor attribute.

The value of the %FOUND attribute can be checked as follows:

```
DELETE FROM emp;
```

5.7 Cursor Attributes

```
IF SQL%FOUND THEN -- delete succeeded
  INSERT INTO emp VALUES ( ..... );
  .....
END IF;
```

5.7.2.2 %NOTFOUND

This attribute is also used to check whether any rows that satisfy the condition in the cursor's SELECT statement have been found. It always has the opposite value of %FOUND.

If no records are affected by the result of execution of a DELETE, UPDATE or INSERT statement, or if a SELECT INTO statement does not return any records, the value of the %NOTFOUND attribute for the associated implicit cursor is TRUE.

However, if a SELECT INTO statement returns no records, the NO_DATA_FOUND exception will occur before it is possible to check the value of the %NOTFOUND attribute. Such a case should be handled as an exception rather than by referring to the %NOTFOUND cursor attribute.

The value of the %NOTFOUND attribute can be checked as follows:

```
DELETE FROM emp;
IF SQL%NOTFOUND THEN
  .....
END IF;
```

5.7.2.3 %ISOPEN

The %ISOPEN attribute is used to check whether the cursor is open. If the cursor is closed, this value will be FALSE.

The value of the %ISOPEN attribute can be checked as follows:

```
OPEN c1; -- CURSOR OPEN
IF c1%ISOPEN THEN
  .....
END IF;
```

5.7.2.4 %ROWCOUNT

%ROWCOUNT indicates how many rows have been fetched by the cursor at the present point in time.

Note that %ROWCOUNT does not indicate the number of records that satisfy the conditions in the cursor's SELECT statement. Rather, it increases by 1 whenever one row is fetched. If not even one row has been fetched, the value of %ROWCOUNT will be zero.

If this attribute is checked before a cursor is opened, or after it has been closed, an INVALID_CURSOR error will be returned.

```
DELETE FROM emp;
IF SQL%ROWCOUNT > 10 THEN
  .....
END IF;
```

5.7.3 Examples

5.7.3.1 Example 1

```

CREATE TABLE t1(i1 INTEGER, i2 INTEGER, i3 INTEGER);

CREATE TABLE t3(i1 INTEGER);
INSERT INTO t1 VALUES(2,2,2);

CREATE OR REPLACE PROCEDURE proc1
AS
    v1 INTEGER;
BEGIN
    SELECT i1 INTO v1 FROM t1 WHERE i1 = 2;
    IF SQL%found THEN
        INSERT INTO t1 SELECT * FROM t1;

        v1 := SQL%ROWCOUNT;
        INSERT INTO t3 VALUES(v1);
    END IF;
END;
/

iSQL> EXEC proc1;
EXECUTE success.
iSQL> SELECT * FROM t3;
T3.I1
-----
1
1 row selected.

```

5.7.3.2 Example 2

```

CREATE TABLE t1(i1 INTEGER, i2 INTEGER, i3 INTEGER);

CREATE TABLE t2(i1 INTEGER, i2 INTEGER, i3 INTEGER);
CREATE TABLE t3(i1 INTEGER);
INSERT INTO t1 VALUES(1,1,1);
INSERT INTO t1 VALUES(1,1,1);
INSERT INTO t1 VALUES(1,1,1);

CREATE OR REPLACE PROCEDURE proc1
AS
    CURSOR c1 IS SELECT * FROM t1;
    v1 INTEGER;
    v2 INTEGER;
    v3 INTEGER;
BEGIN
    OPEN c1;

    IF c1%ISOPEN THEN
        LOOP
            FETCH c1 INTO v1, v2, v3;
            IF c1%FOUND THEN
                INSERT INTO t2 VALUES (v1, v2, v3);
            ELSIF c1%NOTFOUND THEN
                EXIT;
            END IF;
        END LOOP;
    END IF;

```

5.7 Cursor Attributes

```
v1 := c1%ROWCOUNT;
INSERT INTO t3 VALUES (v1);
CLOSE c1;
END;
/

iSQL> EXEC proc1;
EXECUTE success.
iSQL> SELECT * FROM t1;
T1.I1      T1.I2      T1.I3
-----
1          1          1
1          1          1
1          1          1
3 rows selected.
iSQL> SELECT * FROM t2;
T2.I1      T2.I2      T2.I3
-----
1          1          1
1          1          1
1          1          1
3 rows selected.
iSQL> SELECT * FROM t3;
T3.I1
-----
3
1 row selected.
```

5.7.3.3 Example 3

```
CREATE TABLE emp_temp(eno INTEGER, e_firstname CHAR(20), e_lastname
CHAR(20));
```

```
CREATE OR REPLACE PROCEDURE proc1
AS
BEGIN
  DECLARE
    CURSOR c1 IS SELECT eno, e_firstname, e_lastname FROM employees;
    emp_rec c1%ROWTYPE;
  BEGIN
    OPEN c1;

    LOOP
      FETCH c1 INTO emp_rec;
      EXIT WHEN c1%ROWCOUNT > 10 OR c1%NOTFOUND;
      INSERT INTO emp_temp
        VALUES(emp_rec.eno, emp_rec.e_firstname, emp_rec.e_lastname);
    END LOOP;

    CLOSE c1;
  END;
END;
/
```

```
iSQL> EXEC proc1;
EXECUTE success.
iSQL> SELECT * FROM emp_temp;
ENO      E_FIRSTNAME      E_LASTNAME
-----
1        Chan-seung      Moon
2        Susan          Davenport
3        Ken            Kobain
4        Aaron          Foster
```


5	Farhad	Ghorbani
6	Ryu	Momoi
7	Gottlieb	Fleischer
8	Xiong	Wang
9	Curtis	Diaz
10	Elizabeth	Bae

10 rows selected.

6 User-Defined Types

In this chapter, the user-defined types that can be used with stored procedures and functions will be described.

6.1 Overview

RECORD types and associative arrays, the user-defined types provided for use with stored procedures, make it possible to organize data into logical units for processing. They can also be used as parameters or return values when stored procedures and functions call other stored procedures and functions. Note however that values that have user-defined types cannot be passed to clients.

6.1.1 RECORD Types

A RECORD type is a user-defined type that consists of a set of columns. It can be used to configure data of different types into logical units for processing. For example, different data types corresponding to "Name", "Salary" and "Department" can be combined into a single data type called "Employee", which is easy to process. A RECORD type defined in a block is local in scope; that is, it is available only in the block in which the type is defined.

For information on defining RECORD types, please refer to [6.2 Defining a User-Defined Type](#). Aside from the difference in how they are declared, the use of a RECORD type variable declared using the %ROWTYPE keyword is the same as for other RECORD type variables.

6.1.2 Associative Arrays

An associative array is similar to a hash table. An associative array is a set of key-value pairs. The keys are unique indexes that are used to locate the associated values with the syntax *variable_name[index]*. The data type of *index* can be either VARCHAR or INTEGER. It can be used to combine data items of the same type into a single data item for processing, regardless of the amount of data. For example, suppose that it is desired to process the data pertaining to employees having employee numbers from 1 to 100. These 100 data items can be processed using an associative array.

For information on defining associative arrays, please refer to [6.2 Defining a User-Defined Type](#).

Square brackets ("[]") are used to access the elements in an associative array variable, as shown below:

```
Ex.) V1[ 1 ] := 1;
```

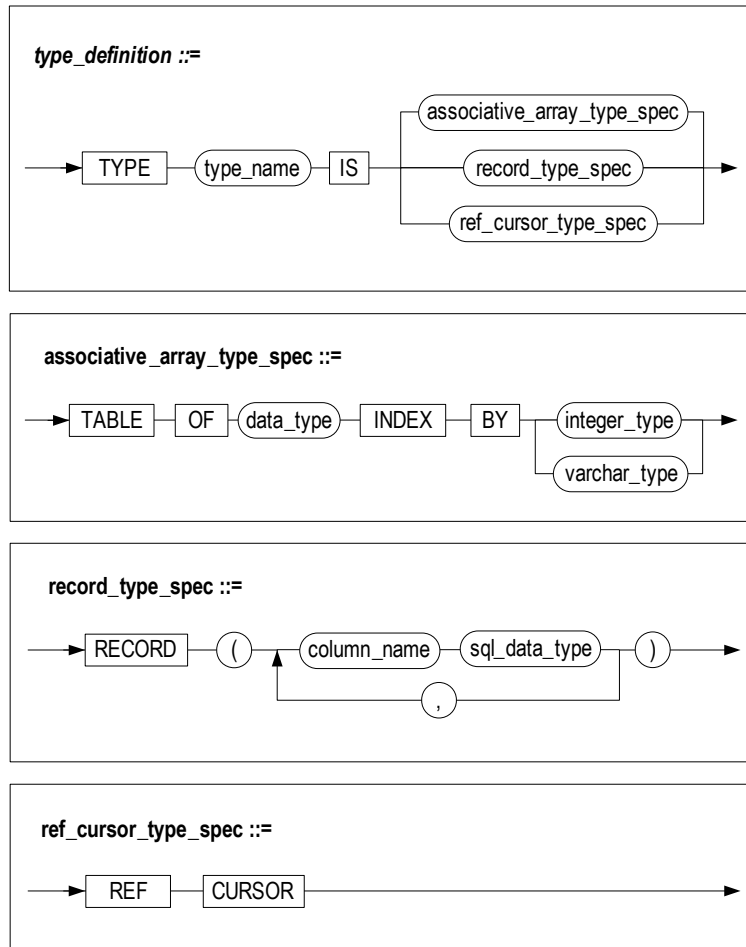
6.1.3 REF CURSOR (Cursor Variable)

A cursor variable is intended for use with dynamic SQL statements that are expected to return multiple records. A cursor variable is more flexible than a regular cursor (i.e. an explicit cursor) because it is not associated with a particular query. Cursor variables can be passed as parameters between stored procedures and functions, and can even be passed to clients.

The difference between a cursor variable and a regular cursor is that a cursor variable, while it is open, can refer to different queries, while a regular cursor can only refer to the query with which it was declared.

6.2 Defining a User-Defined Type

6.2.1 Syntax



6.2.1.1 type_name

The name of the user-defined type is specified here.

6.2.1.2 associative_array_type_spec

This defines an associative array of the data type specified using *data_type*. Note that *data_type* cannot be another associative array. That is, an array of arrays cannot be created.

6.2.1.3 record_type_spec

This defines a RECORD type that consists of multiple columns, each having its own *sql_data_type*. *sql_data_type* can be any data type that is available for use in SQL statements. Note that *sql_data_type* cannot be an associative array or another RECORD type.

6.2 Defining a User-Defined Type

6.2.1.4 ref_cursor_type_spec

This defines a REF CURSOR type.

6.2.2 Examples

6.2.2.1 Example 1

Define a RECORD type called *employee* that comprises the *name* (VARCHAR(20)), *department* (INTEGER) and *salary* (NUMBER(8)) elements.

```
DECLARE
TYPE employee IS RECORD( name VARCHAR(20),
    dept INTEGER,
    salary NUMBER(8));
....
BEGIN
....
```

6.2.2.2 Example 2

Define an associative array called “namelist” that uses the VARCHAR type for its elements and has an INTEGER type index.

```
DECLARE
TYPE namelist IS TABLE OF VARCHAR(20)
    INDEX BY INTEGER;
....
BEGIN
....
```

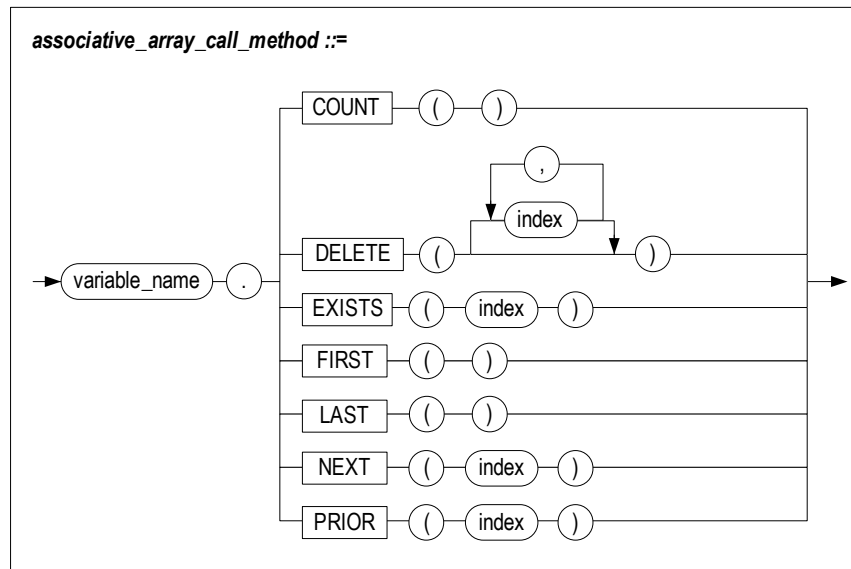
6.2.2.3 Example 3

Define an associative array called *employeeelist* that uses the *employee* user-defined record type for its elements and has a VARCHAR type index.

```
DECLARE
TYPE employee IS RECORD( name VARCHAR(20),
    dept INTEGER,
    salary NUMBER(8));
TYPE employeeelist IS TABLE OF employee
    INDEX BY VARCHAR(10);
....
BEGIN
....
```

6.3 Functions for Use with Associative Arrays

6.3.1 Syntax



6.3.2 Purpose

Various functions are provided for manipulating associative array elements. Unlike SQL functions, parentheses “()” cannot be omitted when using these functions.

6.3.2.1 COUNT

This returns the number of elements in an associative array.

6.3.2.2 DELETE

DELETE() removes all elements and returns the number of elements that were removed.

DELETE(n) removes the element whose index is n, and returns the number of element(s) that were removed, i.e. 0 or 1.

DELETE(m, n) removes all elements whose indexes are in the range from m to n inclusive and returns the number of the elements that were removed. Note that if the value of m is greater than the value of n, then no elements will be removed. If they are the same, then only that element will be removed.

6.3.2.3 EXISTS

EXISTS(n) checks whether the element whose index is n exists. Returns the boolean value TRUE if it exists, or FALSE if it does not.

6.3 Functions for Use with Associative Arrays

6.3.2.4 FIRST

For an array indexed by integers, FIRST returns the smallest index number. For an array indexed by strings, FIRST returns the lowest key value. If the array does not contain any elements, it returns NULL.

6.3.2.5 LAST

For an array indexed by integers, LAST returns the largest index number. For an array indexed by strings, LAST returns the highest key value. If the array does not contain any elements, it returns NULL.

6.3.2.6 NEXT

NEXT(n) returns the index number that follows index n. For associative arrays with VARCHAR keys, NEXT returns the next key value. The binary values of the characters in the string determine the order. If there is no index at this position, it returns NULL.

6.3.2.7 PRIOR

PRIOR(n) returns the index number that precedes index n. For associative arrays with VARCHAR keys, PRIOR returns the preceding key value. The binary values of the characters in the string determine the order. If there is no index at this position, it returns NULL.

6.3.3 Examples

6.3.3.1 Example 1

Delete elements from the associative array variable "V1".

```
CREATE OR REPLACE PROCEDURE PROC1 (
P1 IN VARCHAR(10),
P2 IN VARCHAR(10) )
AS
TYPE MY_ARR IS TABLE OF INTEGER
INDEX BY VARCHAR(10);
V1 MY_ARR;
V2 INTEGER;
BEGIN
V1['FSDGADS'] := 1;
V1['AA'] := 2;
V1['7G65'] := 3;
V1['N887K'] := 4;
V1['KU'] := 5;
V1['34'] := 6;

PRINTLN( 'V1 COUNT IS : ' || V1.COUNT() );

V2 := V1.DELETE(P1, P2);
PRINTLN( 'DELETED COUNT IS : ' || V2);
PRINTLN( 'V1 COUNT IS : ' || V1.COUNT() );
END;
/
```


The result of execution:

```
EXEC PROC1('005T34', 'BC35'); -- The elements whose indexes fall in this
range are V1['34'], V1['7G65'] and V1['AA']. Three elements will be deleted.
```

```
V1 COUNT IS : 6
DELETED COUNT IS : 3
V1 COUNT IS : 3
Execute success.
```

6.3.3.2 Example 2

Output the elements in the associative array variable “V1” in ascending and descending order.

```
CREATE OR REPLACE PROCEDURE PROC1
AS
  TYPE MY_ARR1 IS TABLE OF INTEGER INDEX BY INTEGER;
  V1 MY_ARR1;
  V1_IDX INTEGER;
BEGIN

  V1[435754] := 1;
  V1[95464] := 2;
  V1[38] := 3;
  V1[57334] := 4;
  V1[138] := 5;
  V1[85462] := 6;

  PRINTLN( 'ASCENDING ORDER V1' );

  V1_IDX := V1.FIRST();

  LOOP
    IF V1_IDX IS NULL
    THEN
      EXIT;
    ELSE
      PRINTLN( 'V1_IDX IS : ' || V1_IDX || ' VALUE IS : ' || V1[V1_IDX] );
      V1_IDX := V1.NEXT(V1_IDX);
    END IF;
  END LOOP;

  PRINTLN( 'DESCENDING ORDER V1' );

  V1_IDX := V1.LAST();

  LOOP
    IF V1_IDX IS NULL
    THEN
      EXIT;
    ELSE
      PRINTLN( 'V1_IDX IS : ' || V1_IDX || ' VALUE IS : ' || V1[V1_IDX] );
      V1_IDX := V1.PRIOR(V1_IDX);
    END IF;
  END LOOP;
END;
/
```

The result of execution:

```
EXEC PROC1;
ASCENDING ORDER V1
V1_IDX IS : 38 VALUE IS : 3
```

6.3 Functions for Use with Associative Arrays

```
V1 IDX IS : 138 VALUE IS : 5
V1 IDX IS : 57334 VALUE IS : 4
V1 IDX IS : 85462 VALUE IS : 6
V1 IDX IS : 95464 VALUE IS : 2
V1 IDX IS : 435754 VALUE IS : 1
DESCENDING ORDER V1
V1 IDX IS : 435754 VALUE IS : 1
V1 IDX IS : 95464 VALUE IS : 2
V1 IDX IS : 85462 VALUE IS : 6
V1 IDX IS : 57334 VALUE IS : 4
V1 IDX IS : 138 VALUE IS : 5
V1 IDX IS : 38 VALUE IS : 3
Execute success.
```

6.4 Using RECORD Type Variables and Associative Array Variables

This section outlines the rules governing the use of user-defined types in stored procedures, with reference to examples. For information on using user-defined types as parameters and return values, please refer to [Chapter 7: Typesets](#).

6.4.1 Compatibility between User-Defined Types

```
L_VALUE := R_VALUE;
```

The inter-compatibility between user-defined types when used in assignment statements such as that shown above is set forth in the following table:

Type of L_VALUE	Type of R_VALUE	Compatibility
RECORD Type	RECORD Type	Only RECORD type variables that are the same user-defined type (i.e. that have the same type name) are compatible. Two different record types are not inter-compatible even when they have the same internal structure.
RECORD Type	%ROWTYPE	Compatible, as long as they comprise the same number and type of columns.
%ROWTYPE	RECORD Type	Compatible, as long as they comprise the same number and type of columns.
Associative Array	Associative Array	Only associative array variables that are the same user-defined type (i.e. that have the same type name) are compatible.

In the following example, the last assignment statement fails even though the two user-defined types have the same internal structure.

6.4.1.1 Example 1

Assigning values to RECORD type variables.

```
CREATE OR REPLACE PROCEDURE PROC1
AS
TYPE emp_rec_type1 IS RECORD ( -- define a RECORD type, emp_rec_type
  e_lastname VARCHAR(20),
  e_firstname VARCHAR(20),
  emp_job VARCHAR(10),
  salary NUMBER(8) );

TYPE emp_rec_type2 IS RECORD (
  e_lastname VARCHAR(20),
  e_firstname VARCHAR(20),
```

6.4 Using RECORD Type Variables and Associative Array Variables

```
emp_job VARCHAR(10),
salary NUMBER(8) );
v_emp1 emp_rec_type1; -- variable of emp_rec_type1 type
v_emp2 emp_rec_type2; -- variable of emp_rec_type2 type

BEGIN
  v_emp1.e_lastname := 'Smith';
  v_emp1.e_firstname := 'Robert';
  v_emp1.emp_job := 'RND1069';
  v_emp1.salary := '10000';

  v_emp2 := v_emp1;
```

Even though the two variables have the same structure, the assignment operation fails because they refer to different user-defined types. However, assignment operations between individual elements whose types match, as shown below, will be successful:

```
v_emp2.e_lastname := v_emp1.e_lastname;
```

6.4.2 RECORD Type Variable Example

6.4.2.1 Example 1

Create a RECORD type for storing the name, salary and department of employees.

```
CREATE OR REPLACE PROCEDURE PROC1
AS
TYPE emp_rec_type IS RECORD (
    e_lastname VARCHAR(20),
    e_firstname VARCHAR(20),
    emp_job VARCHAR(10),
    salary NUMBER(8) );

v_emp emp_rec_type;

BEGIN
  v_emp.e_lastname := 'Smith';
  v_emp.e_firstname := 'Robert';
  v_emp.emp_job := 'RND1069';
  v_emp.salary := '10000000';

  PRINTLN('NAME : ' || v_emp.e_firstname || ' ' || v_emp.e_lastname || ' ' ||
    'JOB ID : ' || v_emp.emp_job || ' ' ||
    'SALARY : ' || v_emp.salary );

END;
/
```

6.4.3 Associative Array Examples

6.4.3.1 Example 1

Output the last names of all employees whose ID numbers range from 1 to 100 inclusive.

```
CREATE OR REPLACE PROCEDURE PROC1
AS
TYPE emp_array_type IS TABLE OF VARCHAR(20)
INDEX BY INTEGER;
```

```
v_emp emp_array_type;

BEGIN

FOR I IN 1 .. 100 LOOP
  SELECT e_lastname INTO v_emp[I] FROM employees
    WHERE eno = I;
END LOOP;

FOR I IN v_emp.FIRST() .. v_emp.LAST() LOOP
  PRINTLN( v_emp[I] );
END LOOP;
END;
/
```

6.4.3.2 Example 2

Output the name, salary and department of all employees whose ID numbers range from 1 to 100 inclusive.

```
CREATE OR REPLACE PROCEDURE PROC1
AS
TYPE emp_rec_type IS RECORD (
  first_name VARCHAR(20),
  last_name VARCHAR(20),
  emp_job VARCHAR(15),
  salary NUMBER(8) );
TYPE emp_array_type IS TABLE OF emp_rec_type
  INDEX BY INTEGER;

v_emp emp_array_type;

BEGIN

FOR I IN 1 .. 100 LOOP
  SELECT e_firstname, e_lastname, emp_job, salary INTO v_emp[I]
    FROM employees
    WHERE eno = I;
END LOOP;

FOR I IN v_emp.FIRST() .. v_emp.LAST() LOOP
  PRINTLN( v_emp[I].first_name||' '||
v_emp[I].last_name||' '||
v_emp[I].emp_job||' '||
v_emp[I].salary );
END LOOP;
END;
/
```

6.5 REF CURSOR

A stored procedure can pass a result set, resulting from execution of a SQL statement, to a client using a cursor variable (REF CURSOR).

Opening a cursor variable with the OPEN FOR statement and then passing the cursor to a client using an OUT parameter makes it possible for the client to access the result set. If multiple cursors are sent, the client can access multiple result sets. Except for the fact that the OPEN FOR statement is used to open a cursor variable, the use of cursor-related statements is the same as for regular cursors.

A cursor variable can only be passed as an OUT or IN/OUT parameter of a stored procedure. It cannot be returned with the RETURN statement.

In order for a client to be able to fetch a result set, a cursor variable must be open when it is passed from the stored procedure to the client. In other words, if the cursor is closed when it is passed, it will be impossible to fetch the result set.

When an UPDATE or INSERT statement is executed inside a stored procedure, the number of affected records (the affected row count) is not passed to the client.

The way that the client receives the result set using the cursor variable varies depending on the type of client. Using a cursor variable to pass the result set to a client is possible only in ODBC and JDBC. It is not possible in embedded SQL (Precompiler, APRE).

Examples

Create a stored procedure that uses a REF CURSOR.

1. Create *emp* and *staff* tables, and insert values into them.

```
create table emp (eno integer, ename char(20), dno integer);
create table staff (name char(20), dept char(20), job char(20), salary
integer);

insert into emp values (10, 'Dulgi Papa', 100);
insert into emp values (20, 'Fred Babbage' , 200);
insert into emp values (30, 'okasa' , 300);

insert into staff values ('Dulgi Papa' , '100' , 'father', 100);
insert into staff values ('Simon Blackwater' , '200' , 'engineer' , 200);
insert into staff values ('Ji-hyung Moon', '300', '', 0);
```

2. Create the user-defined type *MY_CUR*, which is a REF CURSOR, and create a typeset called *MY_TYPE* containing type *MY_CUR*.

```
create typeset MY_TYPE
as
  type MY_CUR is REF CURSOR;
end;
/
```

3. Create the stored procedure *PROC1*, which has two OUT parameters, P1 and P2, of type *MY_CUR*, and one IN parameter, SAL, of type INTEGER.

```
create or replace procedure PROC1 (P1 out MY_TYPE.MY_CUR, P2 out
MY_TYPE.MY_CUR, SAL in INTEGER)
as
```

```

    sql_stmt VARCHAR2(200);
begin
    sql_stmt := 'SELECT name,dept,job FROM staff WHERE salary > ?';
    OPEN P1 FOR 'SELECT eno, ename, dno FROM emp';
    OPEN P2 FOR sql_stmt USING SAL;
end;
/

```

4. After connecting to the database, execute procedure *PROC1*.

```

SQLRETURN execute_proc()
{
    SQLCHAR errMsg[MSG_LEN];
    char sql[1000];
    SQLHSTMT stmt = SQL_NULL_HSTMT;

    int sal;
    int sal_len;
    int eno;
    int eno_len;
    int dno;
    int dno_len;
    SQLCHAR ename[ENAME_LEN+1];
    SQLCHAR name[NAME_LEN+1];
    SQLCHAR dept[DEPT_LEN+1];
    SQLCHAR job[JOB_LEN+1];

    int job_ind;

    SQLRETURN rc = SQL_SUCCESS;

    if (SQL_ERROR == SQLAllocStmt(dbc, &stmt))
    {
        printf("SQLAllocStmt error!!\n");
        return SQL_ERROR;
    }

    /* Prepare SQL statements for execution. */
    sprintf(sql, "EXEC proc1(?)");
    if ( SQLPrepare(stmt, (SQLCHAR *)sql, SQL_NTS) == SQL_ERROR )
    {
        printf("ERROR: prepare stmt\n");
    }
    else
    {
        printf("SUCCESS: prepare stmt\n");
    }

    /* Specify sal as 100. */
    sal = 100;

    /* Bind sal as a parameter into SQL statements. */
    if ( SQLBindParameter( stmt,
                           1,
                           SQL_PARAM_INPUT,
                           SQL_C_SLONG,
                           SQL_INTEGER,
                           0,
                           0,
                           &sal,
                           0,
                           NULL) == SQL_ERROR )
    {
        printf("ERROR: Bind Parameter\n");
    }
}

```

6.5 REF CURSOR

```
    }
    else
    {
        printf("SUCCESS: 1 Bind Parameter\n");
    }

    /* Execute the SQL statements (execute procedure PROC1). The procedure
    passes the results of 'SELECT eno, ename, dno FROM emp' and those of
    'SELECT name,dept,job FROM staff WHERE salary > ?'(USING SAL) using OUT
    parameters, p1 and p2, to the client. */
    if (SQL_ERROR == SQLExecute(stmt))
    {
        printf("ERROR: Execute Procedure\n");
    }

    /* Store the results of 'SELECT eno, ename, dno FROM emp' in variables
    (eno, ename and dno). */
    if (SQL_ERROR == SQLBindCol(stmt, 1, SQL_C_SLONG,
                                &eno, 0, (long *)&eno_len))
    {
        printf("ERROR: Bind 1 Column\n");
    }
    if (SQL_ERROR == SQLBindCol(stmt, 2, SQL_C_CHAR,
                                ename, sizeof(ename), NULL))
    {
        printf("ERROR: Bind 2 Column\n");
    }
    if (SQL_ERROR == SQLBindCol(stmt, 3, SQL_C_SLONG,
                                &dno, 0, (long *)&dno_len))
    {
        printf("ERROR: Bind 3 Column\n");
    }

    /* Retrieve the results and then display them on the screen while they
    exist in P1. */
    while (SQL_SUCCESS == rc)
    {
        rc = SQLFetch(stmt);
        if (SQL_SUCCESS == rc)
        {
            printf("Result Set 1 : %d,%s,%d\n", eno, ename, dno);
        }
        else
        {
            if (SQL_NO_DATA == rc)
            {
                break;
            }
            else
            {
                printf("ERROR: SQLFetch [%d]\n", rc);
                execute_err(dbc, stmt, sql);
                break;
            }
        }
    }

    /* Move to the next result (P2). */
    rc = SQLMoreResults(stmt);
    if (SQL_ERROR == rc)
    {
        printf("ERROR: SQLMoreResults\n");
    }
    else
    {

```



```

/* Store the results of 'SELECT name,dept,job FROM staff WHERE salary
> ?'(USING SAL) in variables(name, dept and job). */
if (SQL_ERROR == SQLBindCol(stmt, 1, SQL_C_CHAR,
                             name, sizeof(name), NULL))
{
    printf("ERROR: Bind 1 Column\n");
}
if (SQL_ERROR == SQLBindCol(stmt, 2, SQL_C_CHAR,
                             dept, sizeof(dept), NULL))
{
    printf("ERROR: Bind 2 Column\n");
}
if (SQL_ERROR == SQLBindCol(stmt, 3, SQL_C_CHAR,
                             job, sizeof(job), (long *)&job_ind))
{
    printf("ERROR: Bind 3 Column\n");
}

/* Retrieve the results and then display them on the screen while
they exist in P2. */
while (SQL_SUCCESS == rc)
{
    rc = SQLFetch(stmt);
    if (SQL_SUCCESS == rc)
    {
        if( job_ind == -1 )
            printf("Result Set 2 : %s,%s,NULL\n", name, dept);
        else
            printf("Result Set 2 : %s,%s,%s\n", name, dept, job);
    }
    else
    {
        if (SQL_NO_DATA == rc)
        {
            break;
        }
        else
        {
            printf("ERROR: SQLFetch [%d]\n", rc);
            execute_err(dbc, stmt, sql);
            break;
        }
    }
}

if (SQL_ERROR == SQLFreeStmt( stmt, SQL_DROP ))
{
    printf("sql free stmt error\n");
}
}

```


7 Typesets

This chapter describes how to define and use typesets.

7.1 Overview

A typeset is a database object that allows the user-defined types used in stored procedures to be stored and managed in one place.

7.1.1 Features

7.1.1.1 Sharing User-Defined Types

When a typeset is used, all user-defined types can be managed in one place. This means that it is not necessary to repeatedly declare user-defined types having identical structures in respective stored procedures.

7.1.1.2 Use of User-Defined Types as Parameters or Return Values

Types belonging to the same typeset can be passed as parameters or return values between different procedures. Note however that individual types cannot be passed to clients without using a REF CURSOR.

7.1.1.3 Integration of Data Types in Logical Units

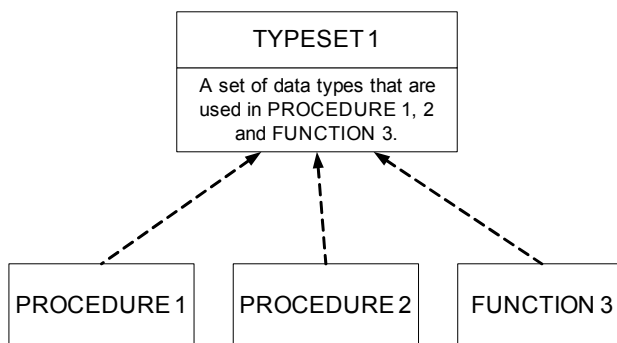
Typesets can be used to integrate data types into logical units for easier management within stored procedures and stored functions.

7.1.1.4 Passing Result Sets to Client Applications

A result set returned by an SQL statement that is executed within a stored procedure can be passed to a client using a REF CURSOR type variable in a typeset.

7.1.2 Structure

As shown in the following diagram and sample code, using a typeset allows user-defined types to be shared and managed by different procedures, facilitating data transfer.



7.1.2.1 TYPESET 1

The *emp_rec_type* and *emp_arr_type* types are defined within *typeset_1*.

```
CREATE TYPESET typeset_1
AS
TYPE emp_rec_type IS RECORD (
    name VARCHAR(20),
    job_id VARCHAR(10),
    salary NUMBER(8) );
TYPE emp_arr_type IS TABLE OF emp_rec_type
INDEX BY INTEGER;
END;
/
```

7.1.2.2 PROCEDURE 1

procedure_1 calls *procedure_2* using *emp_arr_type* as an OUT parameter.

```
CREATE PROCEDURE procedure_1
AS
    V1 typeset_1.emp_arr_type;
BEGIN
    procedure_2( V1 );
    PRINTLN(V1[1].name);
    PRINTLN(V1[1].job_id);
    PRINTLN(V1[1].salary);
END;
/
```

7.1.2.3 PROCEDURE 2

procedure_2 assigns the value returned by *function_3* to its OUT parameter.

```
CREATE PROCEDURE procedure_2
( P1 OUT typeset_1.emp_arr_type )
AS
V1 typeset_1.emp_rec_type;
BEGIN
V1 := function_3();
P1[1] := V1;
END;
/
```

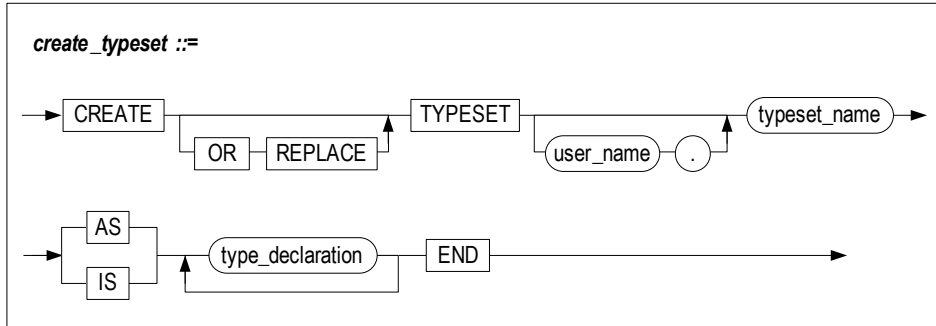
7.1.2.4 FUNCTION 3

function_3 returns a value whose type is *typeset_1.emp_rec_type*.

```
CREATE FUNCTION function_3
RETURN typeset_1.emp_rec_type
AS
    V1 typeset_1.emp_rec_type;
BEGIN
    V1.name := 'Smith';
    V1.job_id := 1010;
    V1.salary := 200;
    RETURN V1;
END;
/
```

7.2 CREATE TYPESET

7.2.1 Syntax



7.2.2 Prerequisites

Only the SYS user and users having the CREATE PROCEDURE or CREATE ANY PROCEDURE system privilege can execute the CREATE TYPESET statement.

7.2.3 Description

This statement defines a user-defined typeset for use in a stored procedure or stored function. The individual types defined in a typeset can also be used as stored procedure INPUT/OUTPUT parameters.

7.2.3.1 user_name

This is used to specify the name of the owner of the typeset to be created. If it is omitted, Altı will create the typeset in the schema of the user who is connected via the current session.

7.2.3.2 typeset_name

This is used to specify the name of the typeset.

7.2.3.3 type_declaration

Please refer to the [6.2 Defining a User-Defined Type](#) in Chapter 6, User-Defined Types.

7.2.4 Examples

7.2.4.1 Example 1

Create a typeset named *my_typeset*.

```

CREATE TYPESET my_typeset
AS
TYPE emp_rec_type IS RECORD(
    name VARCHAR(20), id INTEGER );
TYPE emp_arr_type IS TABLE OF emp_rec_type
    INDEX BY INTEGER;
END;
/

```

7.2.4.2 Example 2

Create a procedure *my_proc1*, which uses *my_typeset*.

```

CREATE PROCEDURE my_proc1
AS
    V1 my_typeset.emp_rec_type;
    V2 my_typeset.emp_arr_type;
BEGIN
    V1.name := 'jejeong';
    V1.id := 10761;
    V2[1] := V1;

    V1.name := 'ehkim';
    V1.id := 11385;
    V2[2] := V1;

    V1.name := 'mslee';
    V1.id := 13693;
    V2[3] := V1;

    PRINTLN('NAME : ' || V2[1].name || ' ID : ' || V2[1].id );
    PRINTLN('NAME : ' || V2[2].name || ' ID : ' || V2[2].id );
    PRINTLN('NAME : ' || V2[3].name || ' ID : ' || V2[3].id );
END;
/

```

The Results

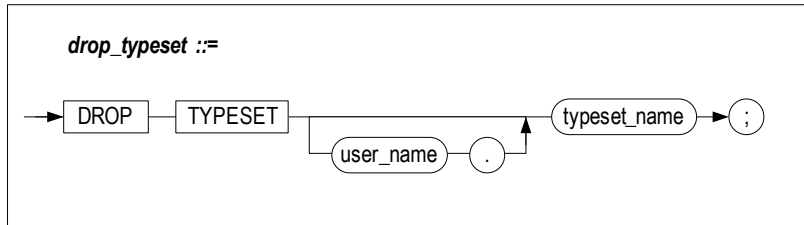
```

iSQL> exec my_proc1;
NAME : jejeong ID : 10761
NAME : ehkim ID : 11385
NAME : mslee ID : 13693
Execute success.

```

7.3 DROP TYPESET

7.3.1 Syntax



7.3.2 Prerequisites

Only the SYS user, the owner of the typeset to be dropped, and users having the DROP ANY PROCEDURE system privilege can execute the DROP TYPESET statement.

7.3.3 Description

This statement is used to remove the specified typeset. Once the typeset has been removed, any stored procedures that use the typeset will be invalid.

7.3.3.1 user_name

This is used to specify the name of the owner of the typeset to be removed. If it is omitted, ALTIBASE HDB will assume that the typeset to be removed is in the schema of the user who is connected via the current session.

7.3.3.2 typeset_name

This specifies the name of the typeset to remove.

7.3.4 Example

Remove a typeset named *my_typeset*.

```
DROP TYPESET my_typeset;
```


8 Dynamic SQL

This chapter describes how to use dynamic SQL in stored procedures and functions.

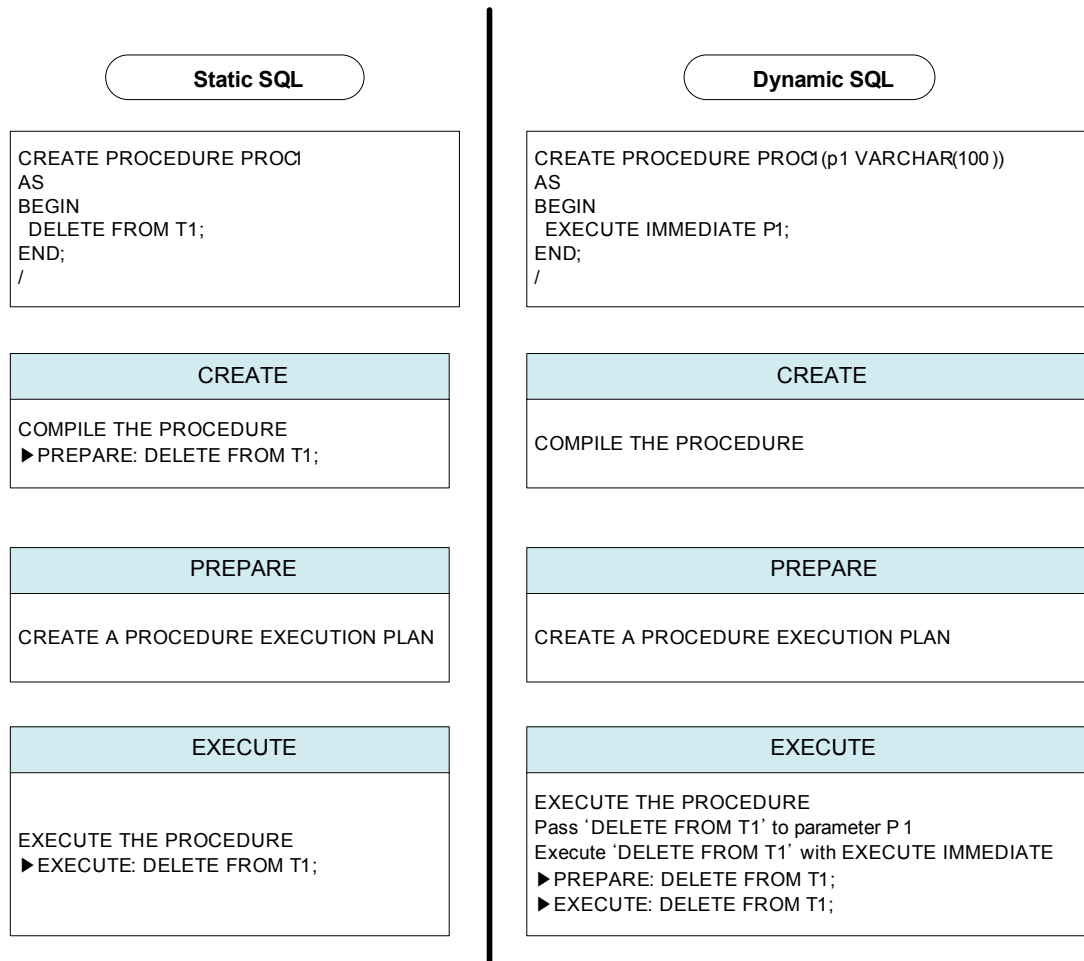
8.1 Overview

With dynamic SQL, the user can create queries as desired at runtime and then execute them. In static execution, which is the standard way to execute SQL statements in stored procedures, an execution plan for all SQL statements in a stored procedure is created in advance when the stored procedure is compiled. Using dynamic SQL is the only way to execute SQL statements that did not exist when the stored procedure was compiled.

8.1.1 Executing Dynamic SQL

The following diagram compares the tasks involved in executing static vs. dynamic SQL statements in stored procedures.

Figure 8-1 Execution of Static SQL vs. Dynamic SQL



In Figure 8-1, the stored procedure on the left processes the 'DELETE FROM T1' statement statically, whereas the stored procedure on the right uses the EXECUTE IMMEDIATE statement to process the same DELETE statement dynamically at runtime.

In the former, an execution plan for the DELETE statement is prepared in advance, at the time that the procedure is created. This execution plan is followed when the SQL statement is executed at run-

time. In contrast, in the latter, the execution plan for the DELETE statement is not prepared when the stored procedure is compiled. Rather, it is both prepared and executed at runtime. Because the execution plan is prepared and executed at runtime, the SQL statement that is actually executed at runtime can be changed.

8.1.2 Features

The advantage of dynamic SQL is that it allows the user to freely change SQL statements as desired during runtime. Furthermore, the user can execute almost any type of SQL statement, as long as it is supported by the DBMS.

Dynamic SQL is useful in the following cases:

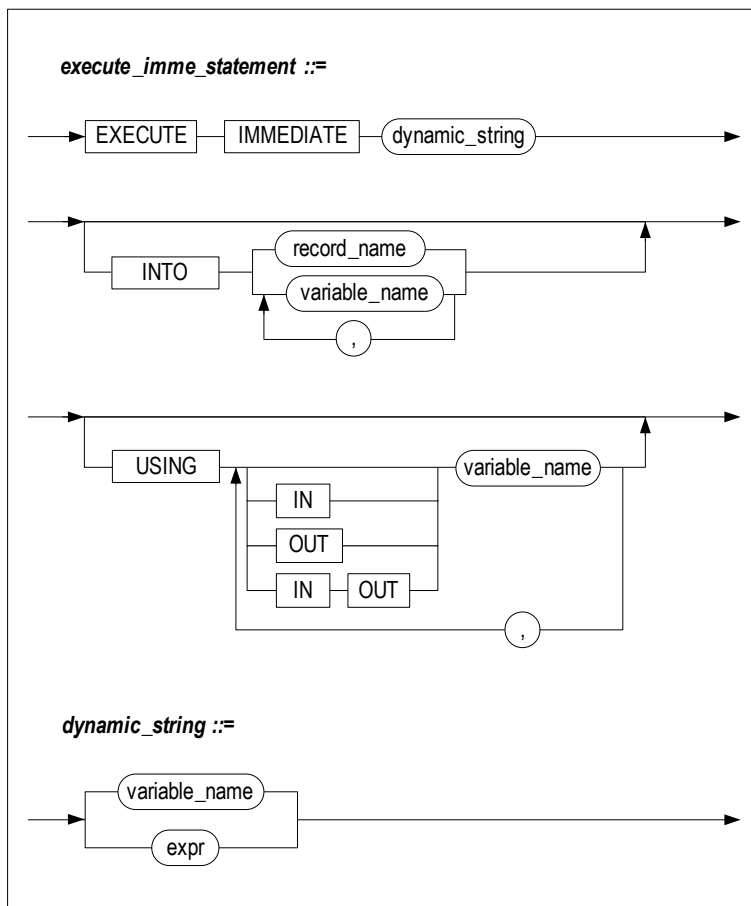
- When the name of the table to be queried can vary during runtime
- When it is appropriate to change a query hint depending on the circumstances, or when it is necessary to change a conditional operator for a condition clause
- When SQL statements that are used in stored procedures and functions need to be optimized frequently due to the frequent execution of DDL and DML statements
- When it is necessary to frequently execute SQL statements for which the execution cost exceeds the optimization cost.
- When it is desired to create versatile, reusable stored procedures

However, in some situations, using dynamic SQL may realize lower performance than using static SQL. This is attributable to the high cost of creating and deleting statements and binding variables to them. Although the use of dynamic SQL statements permits greater flexibility when designing applications, it may result in reduced performance.

8.2 EXECUTE IMMEDIATE

This statement is used to dynamically execute a DDL, DCL or DML statement, including a SELECT query that returns a single record.

8.2.1 Syntax



8.2.2 Description

8.2.2.1 dynamic_string

This is the string containing the query statement to be executed.

8.2.2.2 INTO

The optional INTO clause indicates the variables in which to store the retrieved result set, in the same manner as a `SELECT ... INTO` statement.

8.2.2.3 USING

The optional USING clause is used to specify parameters to bind to the SQL statement at runtime. The parameters are bound to the statement at the positions indicated by question marks (“?”) in the order in which they appear. IN, OUT and IN/OUT parameters can all be specified.

8.2.3 Example

The following is an example of the use of dynamic SQL to execute a DML statement.

```
CREATE PROCEDURE fire_emp(v_emp_id INTEGER)
AS
BEGIN
    EXECUTE IMMEDIATE
        'DELETE FROM employees WHERE emp_id = ?'
        USING v_emp_id;
END;
/

CREATE PROCEDURE insert_table (
    table_name VARCHAR(100),
    dept_no NUMBER,
    dept_name VARCHAR(100),
    location VARCHAR(100))
AS
    stmt VARCHAR2(200);
BEGIN
    stmt := 'INSERT INTO ' || table_name || ' values (?, ?, ?)';
    EXECUTE IMMEDIATE stmt
        USING dept_no, dept_name, location;
END;
/
```

The syntax “EXECUTE IMMEDIATE dynamic_string” is used to execute a query in Direct-Execute mode. The variables that follow USING are binding parameters. In addition to DML statements, DDL and DCL statements can also be executed using EXECUTE IMMEDIATE.

8.2.4 Restrictions

The following SQL statements are supported for execution as dynamic SQL in stored procedures:

- DML
SELECT, INSERT, UPDATE, DELETE, MOVE, LOCK TABLE, ENQUEUE
- DDL
CREATE, ALTER, DROP
- DCL
ALTER SYSTEM, ALTER SESSION, COMMIT, ROLLBACK

The following statements are not supported for use with dynamic SQL:

- Statements that can only be executed from iSQL, for example:

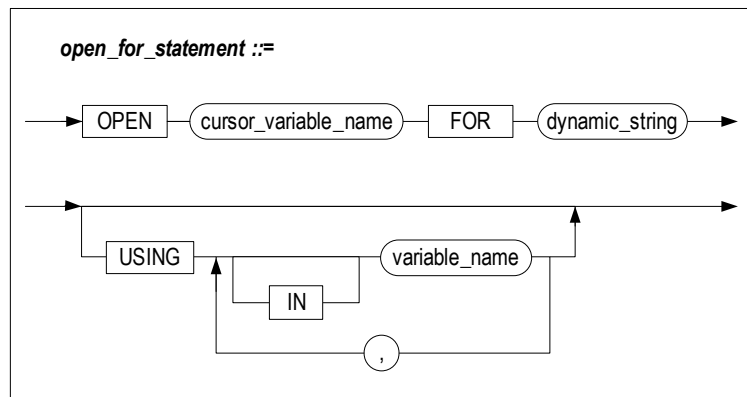
8.2 EXECUTE IMMEDIATE

- SELECT * FROM tab;
 - DESC
 - SET TIMING
 - SET AUTOCOMMIT
- CONNECT
- DISCONNECT
- DEQUEUE

8.3 OPEN FOR

This statement is used to initialize a cursor variable (REF CURSOR), execute the query, and determine the result set, so that data can be retrieved using the FETCH statement or can be passed to a client using stored procedure parameters. The USING clause is used to bind parameters.

8.3.1 Syntax



8.3.2 Description

8.3.2.1 cursor_variable_name

This is used to specify the name of a REF CURSOR-type cursor variable.

8.3.2.2 dynamic_string

dynamic_string is the query to be executed. This can only be a SELECT statement in the form of a string.

8.3.2.3 USING

The optional USING clause is used to specify parameters to bind to the SQL statement at runtime. The parameters are bound to the statement at the positions indicated by question marks ("?",) in the order in which they appear.

8.3.3 Example

The following example illustrates how to open a cursor variable within a stored procedure in order to fetch multiple rows resulting from the execution of a dynamic SQL statement.

For information on how to fetch a result set using an open cursor variable in a client program, please refer to the *Precompiler User's Manual*, *ODBC Reference* and *API User's Manual*.

8.3 OPEN FOR

```
CREATE OR REPLACE PROCEDURE fetch_employee
AS
    TYPE MY_CUR IS REF CURSOR;
    emp_cv MY_CUR;
    emp_rec employees%ROWTYPE;
    stmt VARCHAR2(200);
    v_job VARCHAR2(10) := 'webmaster';
BEGIN
    stmt := 'SELECT * FROM employees WHERE emp_job = ?';
    OPEN emp_cv FOR stmt USING v_job;
    LOOP
        FETCH emp_cv INTO emp_rec;
        EXIT WHEN emp_cv%NOTFOUND;
        PRINTLN(' [Name]: ' || emp_rec.e_firstname || emp_rec.e_lastname ||
            ' [Job Id]: ' || emp_rec.emp_job);
    END LOOP;
    CLOSE emp_cv;
END;
/
```


9 Exception Handlers

9.1 Overview

Exceptions that occur while a stored procedure is executing can be managed by appropriately declaring exceptions and managing them using exception handlers.

9.1.1 Types

Two kinds of exceptions can occur within stored procedures in ALTIBASE HDB:

- System-defined exceptions
- User-defined exceptions

9.1.1.1 System-Defined Exceptions

System-defined exceptions are already defined within the system, and thus do not need to be declared in the DECLARE section of a stored procedure or block.

Some of the system-defined exceptions that can occur within stored procedures are as follows:

Exception Name	Cause
CURSOR_ALREADY_OPEN	This exception is raised when an attempt is made to open a cursor that is already open without first closing it. In the case of a Cursor FOR LOOP, because the cursor is implicitly opened, this exception will be raised if an attempt is made to explicitly open the cursor using the OPEN statement within the loop.
DUP_VAL_ON_INDEX	This exception is raised when an attempt is made to insert a duplicate value into a column designated as a unique index.
INVALID_CURSOR	This exception is raised when the operation cannot be completed with the cursor in its current state, such as when an attempt is made to use a cursor that is not open to perform a FETCH or CLOSE operation.
NO_DATA_FOUND	This exception is raised when no records are returned by a SELECT statement.
TOO_MANY_ROWS	A SELECT INTO statement can return only one row. This occurs when more than one row is returned.

9.1.1.2 User-defined Exceptions

User-defined exceptions are expressly declared by the user and intentionally raised using the RAISE statement.

An example is shown below:

```
DECLARE
  comm_missing EXCEPTION;  -- DECLARE user defined EXCEPTION
BEGIN
```

```

.....
RAISE comm_missing; -- raising EXCEPTION
.....
EXCEPTION
WHEN comm_missing THEN .....

```

If a user-defined exception has the same name as a system-defined exception, the user-defined exception will take precedence over the system-defined exception.

9.1.2 Declaring an Exception

The names of system-defined exceptions are defined inside the system, so there is no need to explicitly declare them.

In contrast, user-defined exceptions must be explicitly declared in the DECLARE section of a block or stored procedure.

9.1.3 Raising an Exception

There is no need to explicitly raise system-defined exceptions. If a system-defined exception occurs during the execution of a stored procedure, whether an exception handler exists for the system-defined exception is checked. If such an exception handler exists, control is automatically diverted to the exception handler, and the tasks defined therein are undertaken.

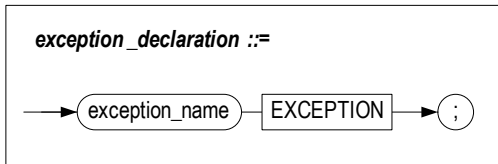
In contrast, user-defined exceptions must be explicitly raised in a stored procedure. User-defined exceptions are raised during the execution of a stored procedure using the RAISE statement.

9.1.4 The Exception Handler

The tasks to perform in the event of a system-defined or user-defined exception are defined here.

9.2 EXCEPTION

9.2.1 Syntax



9.2.2 Purpose

To define the user-defined exception.

9.2.2.1 exception_name

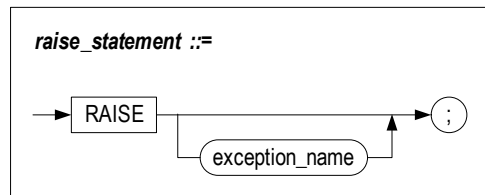
The scope of an exception is from the BEGIN statement to the END statement of the block in which it is declared. The name of the exception must be unique within the block.

9.2.3 Example

```
DECLARE
  error_1 EXCEPTION;
  error_2 EXCEPTION;
  error_3 EXCEPTION;
```

9.3 RAISE

9.3.1 Syntax



9.3.2 Purpose

This statement is used to expressly raise an exception and pass control to the routine defined for the corresponding exception handler.

9.3.2.1 exception_name

The name of the exception to raise is specified here.

exception_name must be either the name of an exception declared in the declare section of the block or a system-defined exception.

If the exception specified here has not been declared, it will be impossible to compile the stored procedure. If the exception has been declared but no corresponding exception handler exists in the exception handler section, execution of the stored procedure will stop and an “unhandled exception” error will be returned.

User exceptions having the same name can be declared in inner and outer blocks. To avoid ambiguity in such cases, label each block and then reference the appropriate exception by specifying the label before the exception name in the RAISE statement.

An exception declared for an outer block can be raised within the handler for an exception declared for an inner block.

An exception name can be omitted only when the RAISE statement is used in the exception handler section, in which case it raises the exception that occurred previously.

9.3.3 Example

9.3.3.1 Example 1

In the following example, the `VALUE_ERROR` exception is handled in the exception handler, and the same exception is raised from the exception handler.

```

CREATE OR REPLACE PROCEDURE PROC1
AS
BEGIN

```

9.3 RAISE

```
        RAISE VALUE_ERROR;
    EXCEPTION
        WHEN VALUE_ERROR THEN
            PRINTLN('VALUE ERROR CATCHED. BUT RE-RAISE. ');
            RAISE;
END;
/

iSQL> EXEC PROC1;
VALUE ERROR CATCHED. BUT RE-RAISE.
[ERR-3116F : Value error
0004 :   RAISE VALUE_ERROR;
      ^         ^
]
```

9.3.3.2 Example 2

In the following example, the exception raised from PROC1 of Example 1 is handled.

```
CREATE OR REPLACE PROCEDURE PROC2
AS
BEGIN
    PROC1;
    EXCEPTION
        WHEN OTHERS THEN
            PRINTLN('EXCEPTION FROM PROC1 CATCHED. ');
            PRINTLN('SQLCODE : ' || SQLCODE);
END;
/

iSQL> EXEC PROC2;
VALUE ERROR CATCHED. BUT RE-RAISE.
EXCEPTION FROM PROC1 CATCHED.
SQLCODE : 201071
Execute success.
```

9.4 RAISE_APPLICATION_ERROR

The use of up to 1001 user-defined error codes, specifically the error codes ranging from 990000 to 991000, is supported.

9.4.1 Syntax

```
RAISE_APPLICATION_ERROR (
    errcode INTEGER,
    errmsg VARCHAR(2047) );
```

9.4.2 Parameter

Name	Input/Output	Data Type	Description
errcode	IN	INTEGER	User-defined Error Code (in the range from 990000 to 991000)
errmsg	IN	VARCHAR	User-defined Error Message Text

9.4.3 Purpose

This procedure is used to raise an exception having a user-defined error code and message.

9.4.4 Example

The following example shows how to raise user-defined errors. Note that in iSQL, error codes are displayed as hexadecimal values.

```
CREATE OR REPLACE PROCEDURE PROC1
AS
BEGIN
    RAISE_APPLICATION_ERROR( 990000,
    'This is my error msg. ' );
END;
/
iSQL> exec proc1;
[ERR-F1B30 : This is my error msg.
0004 : RAISE_APPLICATION_ERROR( 990000,
0005 : 'This is my error msg.' );
```

9.5 User-defined Exceptions

There are two kinds of situations in which the user might want to use the RAISE statement to generate an exception:

- To handle a user-defined exception
- To handle a system-defined exception

9.5.1 User-defined Exception Codes

When handling user-defined exceptions, the error code is always 201232, which can be verified by checking the value of SQLCODE.

```
CREATE OR REPLACE PROCEDURE PROC1
AS
    E1 EXCEPTION;
BEGIN
    RAISE E1;
EXCEPTION
    WHEN E1 THEN
        PRINTLN('SQLCODE: ' || SQLCODE); -- output error code
        PRINTLN('SQLERRM: ' || SQLERRM); -- output error message
END;
/

iSQL> EXEC PROC1;
SQLCODE: 201232
SQLERRM: User-Defined Exception.
Execute success.
```

If the exception is not handled as a user-defined exception in the exception handler section, the following error occurs. This message means that there is no user-defined exception handler for the exception.

```
CREATE OR REPLACE PROCEDURE PROC1
AS
    E1 EXCEPTION;
BEGIN
    RAISE E1;
END;
/

iSQL> EXEC PROC1;
[ERR-31157 : Unhandled exception : E1]
```

The following error code is always output (in either decimal or hexadecimal form) for user-defined exceptions:

Exception Name	Error Code (integer)	Error Code (hexadecimal)	Error Section
	201232	31210	qpERR_ABORT_QSX_USER_DEFINED_EXCEPTION

9.5.2 System-defined Exception Codes

When a system-defined exception occurs, the corresponding system-defined error code is returned, as shown below:

```
CREATE OR REPLACE PROCEDURE PROC1
AS
BEGIN
    RAISE NO_DATA_FOUND;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        PRINTLN('SQLCODE: ' || SQLCODE); -- output error code
        PRINTLN('SQLERRM: ' || SQLERRM); -- output error message
END;
/
iSQL> EXEC PROC1;
SQLCODE: 201066
SQLERRM: No data found.
0004 : RAISE NO_DATA_FOUND;
Execute success.
```

Even if no error handler is specifically provided for a system-defined exception, its predefined error code is still returned. (Note again that this is shown as a hexadecimal number within iSQL.)

```
CREATE OR REPLACE PROCEDURE PROC1
AS
BEGIN
    RAISE NO_DATA_FOUND;
END;
/

iSQL> EXEC PROC1;
[ERR-3116A : No data found.
0004 : RAISE NO_DATA_FOUND;
```

For convenience, the most commonly used system-defined exception codes are listed in the following table. For information on the cause of each exception, please refer to [9.1.1.1 System-Defined Exceptions](#).

Exception Name	Error Code (integer)	Error Code (hexadecimal)	Error Section
"CURSOR_ALREADY_OPEN"	201062	31166	qpERR_ABORT_QSX_CURSOR_ALREADY_OPEN
"DUP_VAL_ON_INDEX"	201063	31167	qpERR_ABORT_QSX_DUP_VAL_ON_INDEX
"INVALID_CURSOR"	201064	31168	qpERR_ABORT_QSX_INVALID_CURSOR
"NO_DATA_FOUND"	201066	3116A	qpERR_ABORT_QSX_NO_DATA_FOUND
"TOO_MANY_ROWS"	201070	3116E	qpERR_ABORT_QSX_TOO_MANY_ROWS

9.5 User-defined Exceptions

Exception Name	Error Code (integer)	Error Code (hexadecimal)	Error Section
"INVALID_PATH"	201237	31215	qpERR_ABORT_QSX_FILE_INVALID_PATH
"INVALID_MODE"	201235	31213	qpERR_ABORT_QSX_INVALID_FILE_OPEN_MODE
"INVALID_FILEHANDLE"	201238	31216	qpERR_ABORT_QSX_FILE_INVALID_FILEHANDLE
"INVALID_OPERATION"	201239	31217	qpERR_ABORT_QSX_FILE_INVALID_OPERATION
"READ_ERROR"	201242	3121A	qpERR_ABORT_QSX_FILE_READ_ERROR
"WRITE_ERROR"	201243	3121B	qpERR_ABORT_QSX_FILE_WRITE_ERROR
"ACCESS_DENIED"	201236	31214	qpERR_ABORT_QSX_DIRECTORY_ACCESS_DENIED
"DELETE_FAILED"	201240	31218	qpERR_ABORT_QSX_FILE_DELETE_FAILED
"RENAME_FAILED"	201241	31219	qpERR_ABORT_QSX_FILE_RENAME_FAILED

For the complete list of all error codes, please refer to the *Error Message Reference*.

9.6 SQLCODE and SQLERRM

SQLCODE and SQLERRM are used in an exception handler to obtain the error code and message for an exception that occurs during the execution of an SQL statement so that the exception can be responded to in a suitable manner.

The contents of SQLCODE and SQLERRM are set in the following cases:

- When an error occurs during the execution of a stored procedure
- When a user-defined exception occurs
- When a system-defined exception occurs
- When RAISE_APPLICATION_ERROR is used to raise a user-defined error (code and message)
- When another exception is raised within an exception handler

In all of the above cases, the current contents of SQLCODE and SQLERRM are replaced with the error code and message corresponding to the newly raised exception.

Additionally, after an exception handler is executed without raising another exception, the contents of SQLCODE and SQLERRM are restored to the state before the exception occurred, in the manner of a LIFO (last in, first out) stack.

Therefore, once the contents of SQLCODE and SQLERRM have been set in response to an exception, they will remain unchanged until control is passed to an outer block of the stored procedure, regardless of whether the exception is handled within the block in which it was raised.

The following is an example:

```
CREATE OR REPLACE PROCEDURE PROC1
AS
BEGIN
  BEGIN
    RAISE NO_DATA_FOUND;
  EXCEPTION
    WHEN OTHERS THEN BEGIN
      PRINTLN('1SQLCODE : ' || SQLCODE);
      PRINTLN('1SQLERRM : ' || SQLERRM);
      RAISE VALUE_ERROR;
    EXCEPTION
      WHEN OTHERS THEN
        PRINTLN('2SQLCODE : ' || SQLCODE);
        PRINTLN('2SQLERRM : ' || SQLERRM);
      END;
    PRINTLN('3SQLCODE : ' || SQLCODE);
    PRINTLN('3SQLERRM : ' || SQLERRM);
  END;
  PRINTLN('4SQLCODE : ' || SQLCODE);
  PRINTLN('4SQLERRM : ' || SQLERRM);
END;
```

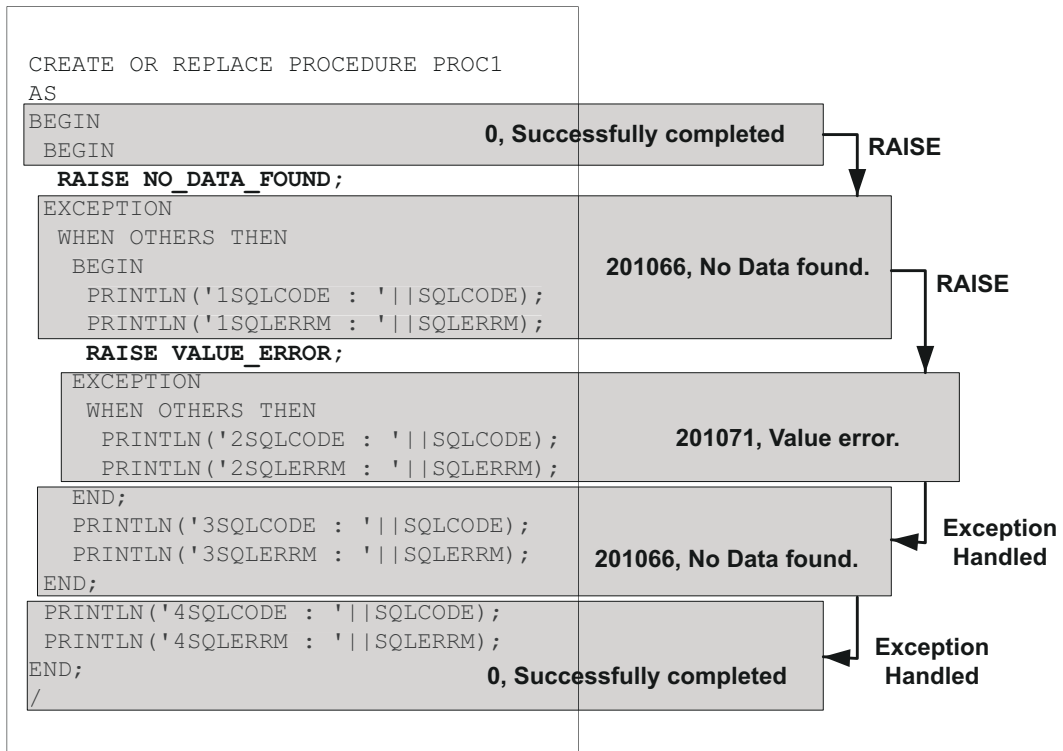
The output of the above example is as follows:

```
1SQLCODE : 201066
1SQLERRM : No data found.
0005 :      RAISE NO_DATA_FOUND;
```

9.6 SQLCODE and SQLERRM

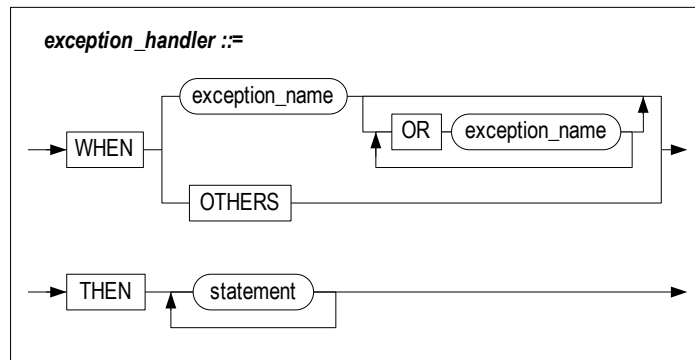
```
2SQLCODE : 201071
2SQLERRM : Value error
0011 :      RAISE VALUE_ERROR;
      ^
3SQLCODE : 201066
3SQLERRM : No data found.
0005 :      RAISE NO_DATA_FOUND;
      ^
4SQLCODE : 0
4SQLERRM : Successfully completed
Execute success.
```

The scope of SQLCODE and SQLERRM in the above example is illustrated in the following figure:



9.7 The Exception Handler

9.7.1 Syntax



9.7.2 Purpose

Exception handlers are used to specify the actions to take in response to exceptions. When an exception occurs, ALTIBASE HDB looks for an exception handler to which to pass control. The rules that are followed when looking for an exception handler are as follows:

1. Starting with the current block and progressing successively outwards to blocks that contain the current block, ALTIBASE HDB looks for a handler for the exception. During this process, if an OTHERS exception handler is found in any block, that OTHERS handler will be used to handle the exception.
2. If no exception handler is found even in the outermost block, an "Unhandled Exception" error is raised, and execution of the stored procedure or function stops immediately.

SQLCODE and SQLERRM can be used in an exception handler to check which kind of error occurred and return the related error message. In other words, SQLCODE returns the ALTIBASE HDB error number and SQLERRM returns the corresponding error message. SQLCODE and SQLERRM cannot be directly used in SQL statements. Instead, assign their values to local variables and use these variables within SQL statements.

9.7.2.1 exception_name

This is used to specify the name of the system-defined or user-defined exception to handle.

Multiple exceptions to be handled in the same way can be combined using "OR" and processed using the same routine.

9.7.2.2 OTHERS

If an exception that is not handled by any other exception handlers is raised, it will ultimately be handled by the OTHERS routine if present.

9.7.3 Example

9.7.3.1 Example 1

```
CREATE TABLE t1(i1 INTEGER, i2 INTEGER, i3 INTEGER);

CREATE TABLE t2(i1 INTEGER, i2 INTEGER, i3 INTEGER);
INSERT INTO t1 VALUES(1,1,1);
INSERT INTO t1 VALUES(2,2,2);

CREATE OR REPLACE PROCEDURE proc1
AS
BEGIN
  DECLARE
    CURSOR c1 IS SELECT * FROM t1;
    v1 INTEGER;
    v2 INTEGER;
    v3 INTEGER;
  BEGIN
    -- OPEN c1;

    FETCH c1 INTO v1, v2, v3;
    INSERT INTO t2 VALUES (v1, v2, v3);

    CLOSE c1;

  EXCEPTION
    WHEN INVALID_CURSOR THEN
      INSERT INTO t2 VALUES (-999, -999, -999);

  END;
END;
/
iSQL> EXEC proc1;
EXECUTE success.
iSQL> SELECT * FROM t2;
T2.I1      T2.I2      T2.I3
-----
-999      -999      -999
1 row selected.
```

9.7.3.2 Example 2

```
CREATE TABLE t1(i1 INTEGER, i2 INTEGER, i3 INTEGER);

CREATE OR REPLACE PROCEDURE proc1(p1 IN INTEGER)
AS
  v1 INTEGER;
  err1 EXCEPTION;
BEGIN
  IF p1 < 0 THEN
    RAISE err1;
  END IF;

  SELECT i1 INTO v1 FROM t1;

EXCEPTION
  WHEN NO_DATA_FOUND OR TOO_MANY_ROWS THEN
    INSERT INTO t1 VALUES(1,1,1);
  WHEN OTHERS THEN
```

```

INSERT INTO t1 VALUES(0,0,0);

END;
/

iSQL> EXEC proc1(1);
EXECUTE success.
iSQL> SELECT * FROM t1;
T1.I1      T1.I2      T1.I3
-----
1          1          1
1 row selected.
iSQL> EXEC proc1(-8);
EXECUTE success.
iSQL> SELECT * FROM t1;
T1.I1      T1.I2      T1.I3
-----
1          1          1
0          0          0
2 rows selected.

```

9.7.3.3 Example 3

```

CREATE TABLE t1(i1 INTEGER NOT NULL);

CREATE OR REPLACE PROCEDURE proc1
AS
    code INTEGER;
    errm VARCHAR(200);
BEGIN
    INSERT INTO t1 VALUES(NULL);
EXCEPTION
    WHEN OTHERS THEN

-- SQLCODE error code is stored in the variable "code".
    code := SQLCODE;

-- SQLERRM error message is stored in the variable "errm"
    errm := SUBSTRING(SQLERRM, 1, 200);
    system_.println('SQLCODE : ' || code);
    system_.println('SQLERRM : ' || errm);
END;
/

iSQL> EXEC proc1;
SQLCODE : 822558860
SQLERRM : Unable to insert or update the NULL value to a NOT NULL column
0006 : INSERT INTO T1 VALUES(NULL);
      ^                               ^

EXECUTE success.

```


10 Built-in Functions and Stored Procedures

ALTIBASE HDB provides a variety of built-in stored procedures and functions, including file control functions and the migration-related stored procedures known collectively as DataPort. This chapter introduces these stored procedures and functions and describes how to use them.

This chapter contains the following topics:

- [File Control](#)
- [DataPort](#)
- [DBMS Stat](#)
- [Miscellaneous Functions](#)

10.1 File Control

The file control functionality of stored procedures enables users to read from and write to text files in the file system. This functionality allows users to perform a wide variety of tasks, including maintaining their own logs, recording the results of tasks, and inserting data read from text files into database tables.

This functionality is described in detail in this section.

10.1.1 Managing Directories

In order for stored procedures to be able to create and manage text files, it is first necessary to use DML to create a directory object that corresponds to the actual directory in which the files are to be saved.

This section describes how to create, maintain and remove directory objects using SQL.

10.1.1.1 Creating a Directory Object

The CREATE DIRECTORY statement is used to create a database object corresponding to each directory in which it is desired to store and maintain files.

When the CREATE DIRECTORY statement is executed, information about the directory is recorded in the SYS_DIRECTORIES_ meta table. However, this statement does not actually create the physical directory in the file system. Therefore, the user must first manually perform the additional tasks of creating the physical directory and granting suitable permissions for the directory.

In the CREATE DIRECTORY statement, the user must specify the name and the absolute path of the directory to be accessed by the database.

Consider the following example.

First, a physical directory named *alti_dir1* is created in the */home/altibase/altibase_home/psm_msg* directory.

```
$ mkdir /home/altibase/altibase_home/psm_msg/alti_dir1
```

Then, a corresponding directory object is created within the database to make it possible to manipulate the files in the *alti_dir1* directory.

```
iSQL> CREATE DIRECTORY alti_dir1 as '/home/altibase/altibase_home/psm_msg/
alti_dir1';
Create success.
```

10.1.1.2 Changing a Directory Object

It is possible to use the CREATE OR REPLACE DIRECTORY statement to change the absolute path to which an existing directory object refers:

```
iSQL> create or replace directory alti_dir1 as '/home/altibase/altibase_home/
psm_result';
Create success.
```

The effect of the above statement will vary depending on whether the *alti_dir1* directory object

already exists in the database. If a directory object having that name already exists, the path to which it refers will be changed to the one specified. If the *alti_dir1* directory object does not exist in the database, it will be created.

10.1.1.3 Dropping a Directory Object

Directory objects can be removed from the database using the DROP DIRECTORY statement.

Note that the DROP DIRECTORY statement merely removes the directory object from the database. It does not actually delete the physical directory from the file system.

Therefore, the user must manually delete unnecessary directories and files from the file system using operating system commands.

The following example shows the use of the DROP DIRECTORY statement to remove a directory object from the database.

```
iSQL> DROP DIRECTORY alti_dir1;
Drop success.
```

10.1.2 File Control

10.1.2.1 FILE_TYPE

To enable stored procedures to control files, ALTIBASE HDB support a data type called "FILE_TYPE".

FILE_TYPE contains file identifiers and other information; however, this information is not directly accessible by users. Local variables having the FILE_TYPE data type can be used within stored procedures as parameters for file control-related system stored procedures and stored functions.

Example

The following is an example showing the declaration of a FILE_TYPE variable:

```
CREATE OR REPLACE PROCEDURE WRITE_T1
AS
    V1          FILE_TYPE;
    ID          INTEGER;
    NAME        VARCHAR(40);
BEGIN
    ...
END;
/
```

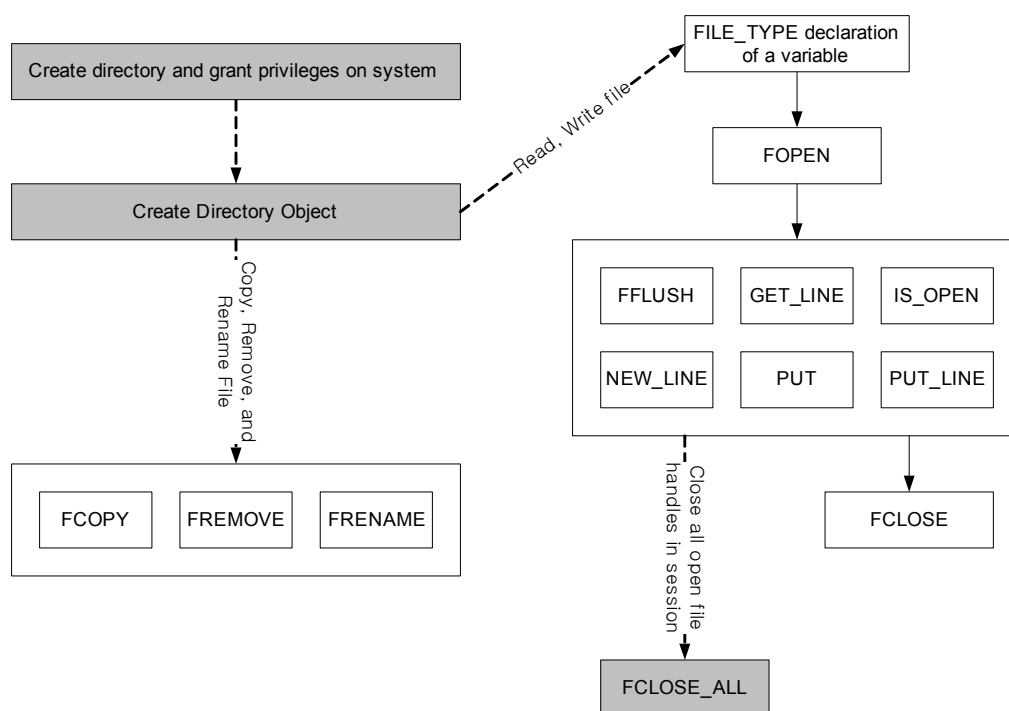
10.1.2.2 System-Provided Stored Procedures and Stored Functions for Handling Files

ALTIBASE HDB provides the following 12 system stored procedures and stored functions for managing files:

Name	Description
FCLOSE	Closes an open file
FCLOSE_ALL	Closes all files that were opened in the current session
FCOPY	Copies a file
FFLUSH	Physically writes data to a file
FOPEN	Opens a file for reading or writing
FREMOVE	Removes a file
FRENAME	Renames a file
GET_LINE	Reads one line from a file
IS_OPEN	Checks whether a file is open
NEW_LINE	Outputs an OS-specific carriage return character or sequence
PUT	Writes a string of text to a file
PUT_LINE	Writes a line of text, followed by a carriage return character or sequence, to a file (=PUT+NEW_LINE)

The system stored procedures and functions listed above are automatically created in the system when the CREATE DATABASE statement is executed. Additionally, PUBLIC synonyms are defined for these procedures and functions so that any user can use them to handle files within stored procedures.

The process of managing files using system procedures and functions is illustrated in the following figure:



10.1.2.3 Limitations

The following may cause errors during the execution of file control-related system stored procedures and stored functions:

Directory name

When using a file control function, the directory parameter must be specified in upper-case letters, and must be the name of a directory object that was created using the CREATE DIRECTORY statement.

For example,

```
CREATE DIRECTORY alti_dir '...';
```

After creating a directory object as shown above, use a statement like the following in the stored procedure:

```
file = FOPEN( 'ALTI_DIR', 'a.txt', 'r' );
```

Even if the name of the directory object was specified in lower-case letters, the names of all objects are stored in upper-case letters in the database. Therefore when specifying the name of a directory object as a parameter for a system procedure or function, it is necessary to use upper-case letters.

File open mode

When opening a file, the file open mode must be specified using lower-case letters ("r", "w" or "a.")

The following is an example:

10.1 File Control

```
file = FOPEN( 'ALTI_DIR', 'a.txt', 'r' );
```

The length of one line of text

The maximum length of one line of text within a file cannot exceed 32767 bytes. An error will occur if this maximum length is exceeded.

File data types

Users cannot read or arbitrarily change the value of a FILE_TYPE variable. FILE_TYPE variables can be used only as parameters for system stored procedures and stored functions.

File Control-Related System Stored Procedures and Stored Functions

The system stored procedures and stored functions provided to manage files may generate exceptions other than system exceptions.

For example, when there is not enough disk space, or when there are not enough file handles, system stored procedures and functions will raise unforeseeable errors such as INVALID_OPERATION.

If an invalid parameter is passed to a file control-related system stored procedure or stored function, a VALUE_ERROR exception will occur.

10.1.3 FCLOSE

This stored procedure closes and reinitializes a file handle.

10.1.3.1 Syntax

```
FCLOSE ( file IN OUT FILE_TYPE );
```

10.1.3.2 Parameters

Name	Input/Output	Data type	Description
file	IN OUT	FILE_TYPE	The file handle

10.1.3.3 Return Value

Because it is a stored procedure, no result value is returned.

10.1.3.4 Exceptions

This stored procedure never raises an error, even when it is executed on a file handle that is already closed.

10.1.3.5 Example

After executing FOPEN and performing actions on files, FCLOSE is called to close the file handle, as shown below:

```
CREATE OR REPLACE PROCEDURE PROC1
AS
    V1 FILE_TYPE;
    V2 VARCHAR(1024);
BEGIN
    V1 := FOPEN( 'ALTI_DIR', 'schema.sql', 'r' );
    GET_LINE( V1, V2, 100 );
    PRINTLN(V2);
    FCLOSE(V1);
END;
/
```

10.1.4 FCLOSE_ALL

This stored procedure closes all of the file handles that were opened in the current session. It is commonly used within exception handlers to ensure that files are closed properly even when exceptions are raised within stored procedures.

10.1.4.1 Syntax

```
FCLOSE_ALL;
```

10.1.4.2 Parameters

This stored procedure has no parameters.

10.1.4.3 Return Value

Because it is a stored procedure, no result value is returned.

10.1.4.4 Exceptions

This stored procedure never raises an error.

10.1.4.5 Example

The following example shows the use of FCLOSE_ALL to close all opened file handles when handling an exception.

```
CREATE OR REPLACE PROCEDURE PROC1
AS
    V1 FILE_TYPE;
    V2 VARCHAR(1024);
BEGIN
    V1 := FOPEN( 'ALTI_DIR', 'schema.sql', 'r' );
    GET_LINE( V1, V2, 100 );
    PRINTLN(V2);
    FCLOSE(V1);
EXCEPTION
```

10.1 File Control

```
WHEN READ_ERROR THEN
  PRINTLN('READ ERROR!!!');
  FCLOSE_ALL;
END;
/
```

10.1.5 FCOPY

This stored procedure is used to copy individual lines of text from one file to another. If the destination file does not exist in the specified destination directory, it is created, and the specified contents are copied from the source file to the new file. If the destination file already exists, the contents of the existing file are replaced with the specified contents from the source file.

10.1.5.1 Syntax

```
FCOPY ( location IN VARCHAR,
        filename IN VARCHAR,
        dest_dir IN VARCHAR,
        dest_file IN VARCHAR,
        start_line IN INTEGER DEFAULT 1,
        end_line IN INTEGER DEFAULT NULL);
```

10.1.5.2 Parameters

Name	Input/Output	Data type	Description
location	IN	VARCHAR	The directory object corresponding to the path in which the source file is located
filename	IN	VARCHAR	The name of the source file
dest_dir	IN	VARCHAR	The directory object corresponding to the path in which the destination file is located
dest_file	IN	VARCHAR	The name of the destination file
start_line	IN	INTEGER	The first line to copy Default: 1
end_line	IN	INTEGER	The last line to copy. Copies to the end of the file if set to NULL or not specified. Default: NULL

10.1.5.3 Return Value

Because it is a stored procedure, no result value is returned.

10.1.5.4 Exceptions

FCOPY can raise the following system-defined exceptions.

INVALID_PATH

ACCESS_DENIED

INVALID_OPERATION

READ_ERROR

WRITE_ERROR

For a detailed explanation of how to handle exceptions, please refer to [10.1.15 Handling File Control-Related Exceptions](#) in this chapter.

10.1.5.5 Example

In the following example, the entire contents of a.txt are copied to b.txt.

```
iSQL> EXEC FCOPY( 'ALTI_DIR', 'a.txt', 'ALTI_DIR', 'b.txt' );
EXECUTE success.
```

```
$ cat a.txt
1-ABCDEFG
2-ABCDEFG
3-ABCDEFG
4-ABCDEFG
5-ABCDEFG
6-ABCDEFG
7-ABCDEFG
8-ABCDEFG
9-ABCDEFG
10-ABCDEFG
```

```
$ cat b.txt
1-ABCDEFG
2-ABCDEFG
3-ABCDEFG
4-ABCDEFG
5-ABCDEFG
6-ABCDEFG
7-ABCDEFG
8-ABCDEFG
9-ABCDEFG
10-ABCDEFG
```

In the following example, only the specified lines are copied from a.txt to b.txt.

```
iSQL> EXEC FCOPY( 'ALTI_DIR', 'a.txt', 'ALTI_DIR2', 'b.txt', 4, 9 );
EXECUTE success.
```

```
$ cat a.txt
1-ABCDEFG
2-ABCDEFG
3-ABCDEFG
4-ABCDEFG
5-ABCDEFG
6-ABCDEFG
7-ABCDEFG
8-ABCDEFG
9-ABCDEFG
```

10.1 File Control

```
10-ABCDEFG
```

```
$ cat b.txt
```

```
4-ABCDEFG
```

```
5-ABCDEFG
```

```
6-ABCDEFG
```

```
7-ABCDEFG
```

```
8-ABCDEFG
```

```
9-ABCDEFG
```

10.1.6 FFLUSH

A stored procedure that physically writes data on the file.

10.1.6.1 Syntax

```
FFLUSH ( file IN FILE_TYPE );
```

10.1.6.2 Parameters

Name	Input/Output	Data type	Description
file	IN	FILE_TYPE	A file handle

10.1.6.3 Return Value

Because it is a stored procedure, no result value is returned.

10.1.6.4 Exceptions

FFLUSH can raise the following system-defined exceptions.

```
INVALID_FILEHANDLE
```

```
WRITE_ERROR
```

For a detailed explanation of how to handle exceptions, please refer to [10.1.15 Handling File Control-Related Exceptions](#) in this chapter.

10.1.6.5 Example

In the following example, all of the data in column I1 of table T1 are written to a file at one time. The last PUT_LINE parameter, *autoflush*, is set to FALSE. This prevents the data from being flushed to the file every time PUT_LINE is called. Instead, the data are flushed at the end using FFLUSH.

```
CREATE OR REPLACE PROCEDURE PROC1
AS
  V1 FILE_TYPE;
  R2 T1%ROWTYPE;
  CURSOR C1 IS SELECT I1 FROM T1;
BEGIN
```

```

V1 := FOPEN( 'ALTI_DIR', 'a.txt', 'w' );
FOR R2 IN C1 LOOP
    PUT_LINE( V1, R2.I1, FALSE );
END LOOP;
FFLUSH(V1);
FCLOSE(V1);
EXCEPTION
    WHEN INVALID_PATH THEN
        PRINTLN('CANNOT OPEN FILE.');
```

```

    WHEN NO_DATA_FOUND THEN
        PRINTLN('NO DATA FOUND.');
```

```

        FCLOSE( V1 );
END;
/
```

10.1.7 FOPEN

This stored function opens a file and returns a file handle.

10.1.7.1 Syntax

```

FILE_TYPE variable :=
    FOPEN (
        location IN VARCHAR,
        filename IN VARCHAR,
        open_mode IN VARCHAR );
```

10.1.7.2 Parameters

Name	Input/Output	Data type	Description
location	IN	VARCHAR	The directory object corresponding to the path in which the file is located
filename	IN	VARCHAR	The name of the file to open
open_mode	IN	VARCHAR	Can be set to one of the following three options: r: Read w: Write a: Append Only one option can be specified for <i>open_mode</i> . That is, combinations of two (or more) options, such as “rw” and “wa”, cannot be used. Additionally, this argument must be specified using lower-case letters.

10.1.7.3 Return Value

When this function is executed successfully, it returns a file handler of which the data type is FILE_TYPE (an opened file handle).

10.1 File Control

10.1.7.4 Exceptions

FOPEN can raise the following system-defined exceptions.

INVALID_PATH

ACCESS_DENIED

INVALID_OPERATION

INVALID_MODE

For a detailed explanation of how to handle exceptions, please refer to [10.1.15 Handling File Control-Related Exceptions](#) in this chapter.

10.1.7.5 Example

The following example shows that before a file can be read from or written to, it is first necessary to open the file in the appropriate mode using FOPEN:

```
CREATE OR REPLACE PROCEDURE PROC1
AS
    V1 FILE_TYPE;
    V2 VARCHAR(1024);
BEGIN
    V1 := FOPEN( 'ALTI_DIR', 'schema.sql', 'r' );
    GET_LINE( V1, V2, 100 );
    PRINTLN(V2);
    FCLOSE(V1);
END;
/
```

10.1.8 FREMOVE

This stored procedure deletes the specified file.

10.1.8.1 Syntax

```
FREMOVE (
    location IN VARCHAR,
    filename IN VARCHAR);
```

10.1.8.2 Parameters

Name	Input/Output	Data type	Description
location	IN	VARCHAR	The directory object corresponding to the path in which the file is located
filename	IN	VARCHAR	The file name

10.1.8.3 Return Value

As it is a stored procedure, no result value is returned.

10.1.8.4 Exceptions

FREMOVE can raise the following system-defined exceptions.

INVALID_PATH

ACCESS_DENIED

DELETE_FAILED

For a detailed explanation of how to handle exceptions, please refer to [10.1.15 Handling File Control-Related Exceptions](#) in this chapter.

10.1.8.5 Example

The following example shows how to use FREMOVE to delete files:

```
--## Current directory information
$ ls
a.txt b.txt schema.sql

--## FREMOVE Execution
iSQL> EXEC FREMOVE('ALTI_DIR', 'b.txt');
EXECUTE success.

--# Directory information after the stored procedure is executed
$ ls
a.txt    schema.sql
```

10.1.9 FRENAME

This stored procedure is used to change the name of a file or move the file to a different location. Its functionality is similar to that of the Unix `mv` command.

10.1.9.1 Syntax

```
FRENAME (
    location IN VARCHAR,
    filename IN VARCHAR,
    dest_dir IN VARCHAR,
    dest_file IN VARCHAR,
    overwrite IN BOOLEAN DEFAULT FALSE );
```

10.1 File Control

10.1.9.2 Parameters

Name	Input/Output	Data type	Description
location	IN	VARCHAR	The directory object corresponding to the original location of the file
filename	IN	VARCHAR	The original name of the file
dest_dir	IN	VARCHAR	The directory object corresponding to the directory to which the file is to be moved
dest_file	IN	VARCHAR	The new name for the file
overwrite	IN	BOOLEAN	If a file having the new name or location already exists, indicates whether to overwrite the existing file. TRUE: overwrite the file FALSE: do not overwrite the file. Default: FALSE

10.1.9.3 Return Value

Because it is a stored procedure, no result value is returned.

10.1.9.4 Exceptions

RENAME can raise the following system-defined exceptions.

INVALID_PATH

ACCESS_DENIED

RENAME_FAILED

For a detailed explanation of how to handle exceptions, please refer to [10.1.15 Handling File Control-Related Exceptions](#) in this chapter.

10.1.9.5 Example

The following example shows how to change the name of a file from “a.txt” to “result.txt”.

```
--## Current directory information
$ ls
a.txt    schema.sql

--## RENAME execution
iSQL> EXEC RENAME('ALTI_DIR','a.txt','ALTI_DIR','result.txt',TRUE);
EXECUTE success.
```

```
--# Directory information after the stored procedure is executed
$ ls
result.txt    schema.sql
```

10.1.10 GET_LINE

This stored procedure reads one line from the specified file.

10.1.10.1 Syntax

```
GET_LINE (
    file IN FILE_TYPE,
    buffer OUT VARCHAR,
    len IN INTEGER DEFAULT NULL);
```

10.1.10.2 Parameters

Name	Input/Output	Data type	Description
file	IN	FILE_TYPE	A file handle
buffer	OUT	VARCHAR	A buffer to store one line read from the file
len	IN	INTEGER	The maximum number of bytes to read from one line of the file. If this value is not specified, a maximum of 1024 bytes will be read from each line. Default: 1024)

10.1.10.3 Return Value

As it is a stored procedure, no result value is returned.

10.1.10.4 Exceptions

GET_LINE can raise the following system-defined exceptions.

NO_DATA_FOUND

READ_ERROR

INVALID_FILEHANDLE

For a detailed explanation of how to handle exceptions, please refer to [10.1.15 Handling File Control-Related Exceptions](#) in this chapter.

10.1 File Control

10.1.10.5 Example

In the following example, 100 bytes are read from one line of the file.

```
CREATE OR REPLACE PROCEDURE PROC1
AS
    V1 FILE_TYPE;
    V2 VARCHAR(1024);
BEGIN
    V1 := FOPEN( 'ALTI_DIR', 'schema.sql', 'r' );
    GET_LINE( V1, V2, 100 );
    PRINTLN(V2);
    FCLOSE(V1);
END;
/

iSQL> EXEC PROC1;
create table t1 (i1 integer, i2 integer, i3 integer);

EXECUTE success.
```

10.1.11 IS_OPEN

This stored function checks whether or not the specified file is open.

10.1.11.1 Syntax

```
BOOLEAN variable :=
    IS_OPEN ( file IN FILE_TYPE );
```

10.1.11.2 Parameters

Name	Input/Output	Data type	Description
file	IN	FILE_TYPE	A file handle

10.1.11.3 Return Value

The return type is BOOLEAN. This stored function returns TRUE if the specified file is open and FALSE if the file is not open.

10.1.11.4 Exceptions

If the specified file handle is open, TRUE is returned. In all other circumstances, FALSE is returned. Therefore, this function never returns an error.

10.1.11.5 Example

The following example shows how to check whether a file handle is open.


```

CREATE OR REPLACE PROCEDURE PROC1
AS
    V1 FILE_TYPE;
BEGIN
    IF IS_OPEN(V1) = FALSE THEN
        PRINTLN('V1 IS NOT OPENED.');
```

```

    ELSE
        PRINTLN('V1 IS OPENED.');
```

```

    END IF;
    V1 := FOPEN( 'ALTI_DIR', 'a.txt', 'w' );
    PRINTLN('FOPEN FUNCTION CALLED.');
```

```

    IF IS_OPEN(V1) = FALSE THEN
        PRINTLN('V1 IS NOT OPENED.');
```

```

    ELSE
        PRINTLN('V1 IS OPENED.');
```

```

    END IF;
    FCLOSE( V1 );
    PRINTLN('FCLOSE FUNCTION CALLED.');
```

```

    IF IS_OPEN(V1) = FALSE THEN
        PRINTLN('V1 IS NOT OPENED.');
```

```

    ELSE
        PRINTLN('V1 IS OPENED.');
```

```

    END IF;
END;
/
```

10.1.12 NEW_LINE

This store procedure writes an OS-specific carriage return character or sequence the specified number of times in the file.

10.1.12.1 Syntax

```

NEW_LINE (
    file IN FILE_TYPE,
    lines IN INTEGER DEFAULT 1 );
```

10.1.12.2 Parameters

Name	Input/Output	Data type	Description
file	IN	FILE_TYPE	A file handle
lines	IN	INTEGER	The number of lines to write in the file. Default: 1

10.1.12.3 Return Value

As it is a stored procedure, no result value is returned.

10.1 File Control

10.1.12.4 Exceptions

NEW_LINE can raise the following system-defined exceptions.

INVALID_FILEHANDLE

WRITE_ERROR

For a detailed explanation of how to handle exceptions, please refer to [10.1.15 Handling File Control-Related Exceptions](#) in this chapter.

10.1.12.5 Example

The following example shows the use of NEW_LINE to insert blank lines in a file:

```
CREATE OR REPLACE PROCEDURE PROC1
AS
  V1 FILE_TYPE;
BEGIN
  V1 := FOPEN( 'ALTI_DIR', 'a.txt', 'w' );
  PUT_LINE( V1, 'REPORT', TRUE );
  NEW_LINE( V1, 3 );
  PUT_LINE( V1, '-----', TRUE );
  FCLOSE( V1 );
END;
/
EXEC proc1;

--## a.txt file after the above-described stored procedure is executed
$ cat a.txt
REPORT

-----
$
```

10.1.13 PUT

This stored procedure is used to write a character string to a file.

10.1.13.1 Syntax

```
PUT (
  file IN FILE_TYPE,
  buffer IN VARCHAR);
```

10.1.13.2 Parameters

Name	Input/Output	Data type	Description
file	IN	FILE_TYPE	A file handle

Name	Input/Output	Data type	Description
buffer	IN	VARCHAR	A buffer in which to store the character string to be written to the file

10.1.13.3 Return Value

As it is a stored procedure, no result value is returned.

10.1.13.4 Exceptions

PUT can raise the following system-defined exceptions:

INVALID_FILEHANDLE

WRITE_ERROR

For a detailed explanation of how to handle exceptions, please refer to [10.1.15 Handling File Control-Related Exceptions](#) in this chapter.

10.1.13.5 Example

The following example shows how to write text to a file:

```
CREATE OR REPLACE PROCEDURE PROC1
AS
    V1 FILE_TYPE;
BEGIN
    V1 := FOPEN( 'ALTI_DIR', 'a.txt', 'w' );
    PUT( V1, 'REPORT' );
    PUT( V1, '-->' );
    PUT_LINE( V1, 'SUCCESS', TRUE );
    FCLOSE( V1 );
END;
/
EXEC proc1;

--## a.txt file result after the above-described stored procedure is executed
$ cat a.txt
REPORT-->SUCCESS
$
```

10.1.14 PUT_LINE

This stored procedure writes one line of text, including a carriage return character or sequence, to a file.

10.1.14.1 Syntax

```
PUT_LINE (
    file IN FILE_TYPE,
    buffer IN VARCHAR,
```

10.1 File Control

```
autoflush IN BOOLEAN DEFAULT FALSE);
```

10.1.14.2 Parameters

Name	Input/Output	Data type	Description
file	IN	FILE_TYPE	A file handle
buffer	IN	VARCHAR	A buffer containing the line of text to be written to the file
autoflush	IN	BOOLEAN	Whether to flush to the file automatically Default: FALSE

autoflush

The autoflush parameter specifies whether to write physically to the file automatically every time the function is called. The default is FALSE, meaning that the data being written to a file are cached.

10.1.14.3 Return Value

As it is a stored procedure, no result value is returned.

10.1.14.4 Exceptions

PUT_LINE can raise the following system-defined exceptions.

INVALID_FILEHANDLE

WRITE_ERROR

For a detailed explanation of how to handle exceptions, please refer to [10.1.15 Handling File Control-Related Exceptions](#) in this chapter.

10.1.14.5 Example

The following example shows how to write a line of text to a file:

```
CREATE OR REPLACE PROCEDURE PROC1
AS
  V1 FILE_TYPE;
BEGIN
  V1 := FOPEN('ALTI_DIR', 'a.txt', 'w');
  PUT_LINE(V1, '1-ABCDEFGH');
  PUT_LINE(V1, '2-ABCDEFGH');
  PUT_LINE(V1, '3-ABCDEFGH');
  PUT_LINE(V1, '4-ABCDEFGH');
  PUT_LINE(V1, '5-ABCDEFGH');
  PUT_LINE(V1, '6-ABCDEFGH');
  PUT_LINE(V1, '7-ABCDEFGH');
  PUT_LINE(V1, '8-ABCDEFGH');
  PUT_LINE(V1, '9-ABCDEFGH');
```

```

    PUT_LINE(V1, '10-ABCDEFG');
    FCLOSE(V1);
END;
/
EXEC proc1;

```

After the above stored procedure is performed, the contents of the file will be as follows:

```

$ cat a.txt
1-ABCDEFG
2-ABCDEFG
3-ABCDEFG
4-ABCDEFG
5-ABCDEFG
6-ABCDEFG
7-ABCDEFG
8-ABCDEFG
9-ABCDEFG
10-ABCDEFG

```

10.1.15 Handling File Control-Related Exceptions

The following is an explanation of some considerations to keep in mind when handling file control-related exceptions that may occur during the execution of a stored procedure or function.

The exceptions that may occur during the execution of file control-related stored procedures and functions are set forth in the following table. These exceptions can be handled using user-defined exception handlers, just like other system-defined exceptions.

Exception Name	Description
INVALID_PATH	The specified directory object does not exist (i.e. the specified object is not a directory object created using the CREATE DIRECTORY statement).
INVALID_MODE	The file open mode is not valid. (A value other than "r", "w", or "a" was specified for the file open mode.)
INVALID_FILEHANDLE	The file handle is invalid. (The specified file could not be opened.)
INVALID_OPERATION	The actual directory or file does not exist in the file system, or else access to the file system was denied.
READ_ERROR	There is no opened file to read from, access to the file has been denied.
WRITE_ERROR	There is no open file to write to, write access to the file has been denied, or the file was not opened in write mode.
ACCESS_DENIED	The user was denied access to the directory object. Sufficient privileges must be granted to the user using the GRANT statement.
DELETE_FAILED	The file to be deleted does not exist, or access to the file has been denied.

10.1 File Control

Exception Name	Description
RENAME_FAILED	A file having the specified name already exists and the overwrite option was not specified, or another file system error has occurred.

10.1.16 File Control Examples

10.1.16.1 Example 1

The following procedure takes the directory and file names as input parameters, opens the corresponding file, and reads and echoes the contents of the file. The directory or file name may not be properly specified, or the file might be empty. Therefore, this stored procedure includes respective exception handlers for the INVALID_PATH and NO_DATA_FOUND system-defined exceptions.

```
--# CREATE VERIFY PROCEDURE
CREATE OR REPLACE PROCEDURE PROC2
( PATH VARCHAR(40), FILE VARCHAR(40) )
AS
  V1 FILE_TYPE;
  V2 VARCHAR(100);
BEGIN
  V1 := FOPEN( PATH, FILE, 'r' );
  LOOP
    GET_LINE( V1, V2, 100 );
    PRINT( V2 );
  END LOOP;
EXCEPTION
  WHEN INVALID_PATH THEN
    PRINTLN('CANNOT OPEN FILE. ');
  WHEN NO_DATA_FOUND THEN
    PRINTLN('NO DATA FOUND. ');
  FCLOSE( V1 );
END;
/
```

10.1.16.2 Example 2

The following example shows how to write the contents of the table to a file or read from a file.

Create a user and assign suitable privileges to the user.

```
CONNECT SYS/MANAGER;
CREATE USER MHJEONG IDENTIFIED BY MHJEONG;
GRANT CREATE ANY DIRECTORY TO MHJEONG;
GRANT DROP ANY DIRECTORY TO MHJEONG;
```

Create and populate a table and create a directory object.

```
CONNECT MHJEONG/MHJEONG;
CREATE TABLE T1( ID INTEGER, NAME VARCHAR(40) );
INSERT INTO T1 VALUES( 1, 'JAKIM' );
INSERT INTO T1 VALUES( 2, 'PEH' );
INSERT INTO T1 VALUES( 3, 'KUMDORY' );
INSERT INTO T1 VALUES( 4, 'KHSHIM' );
INSERT INTO T1 VALUES( 5, 'LEEKMO' );
INSERT INTO T1 VALUES( 6, 'MHJEONG' );
CREATE DIRECTORY MYDIR AS '/home1/mhjeong';
```

Create a procedure that reads all of the records in table *T1* and writes the data to *t1.txt*.

```
CREATE OR REPLACE PROCEDURE WRITE_T1
AS
    V1 FILE_TYPE;
    ID INTEGER;
    NAME VARCHAR(40);
BEGIN
    DECLARE
        CURSOR T1_CUR IS
            SELECT * FROM T1;
    BEGIN
        OPEN T1_CUR;
        V1 := FOPEN( 'MYDIR', 't1.txt', 'w' );
        LOOP
            FETCH T1_CUR INTO ID, NAME;
            EXIT WHEN T1_CUR%NOTFOUND;
            PUT_LINE( V1, 'ID : ' || ID || ' NAME : ' || NAME );
        END LOOP;
        CLOSE T1_CUR;
        FCLOSE( V1 );
    END;
END;
/
```

Create a procedure that reads the contents of *t1.txt* and displays the contents on the screen.

```
CREATE OR REPLACE PROCEDURE READ_T1
AS
    BUFFER VARCHAR(200);
    V1 FILE_TYPE;
BEGIN
    V1 := FOPEN( 'MYDIR', 't1.txt', 'r' );
    LOOP
        GET_LINE( V1, BUFFER, 200 );
        PRINT( BUFFER );
    END LOOP;
    FCLOSE( V1 );
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        FCLOSE( V1 );
END;
/
```

10.1.16.3 Result

When the stored procedures created above are executed, the following result will be displayed:

```
iSQL> exec write_t1;
EXECUTE success.
iSQL> exec read_t1;
ID : 1 NAME : JAKIM
ID : 2 NAME : PEH
ID : 3 NAME : KUMDORY
ID : 4 NAME : KHSHIM
ID : 5 NAME : LEEKMO
ID : 6 NAME : MHJEONG
EXECUTE success.
```

The following file will be visible in the corresponding directory in the file system.

```
$ cd /home1/mhjeong
$ cat t1.txt
```

10.1 File Control

```
ID : 1 NAME : JAKIM
ID : 2 NAME : PEH
ID : 3 NAME : KUMDORY
ID : 4 NAME : KHSHIM
ID : 5 NAME : LEEKMO
ID : 6 NAME : MHJEONG
```


10.2 DataPort

DataPort is an ALTIBASE HDB feature that is intended for use when moving the data in tables from one Altibase database to another. This functionality is provided in the form of multiple system-defined stored procedures that are used for this purpose.

This section contains the following topics:

- [Using DataPort](#)
- [DataPort Stored Procedures and Functions](#)
- [DataPort Exceptions](#)
- [Examples](#)

10.2.1 Using DataPort

10.2.1.1 Overview

Using DataPort for database migration tasks realizes the following benefits:

- Better performance is realized when migrating from one Altibase database to another.
- The source and destination databases can reside on different kinds of hardware and operating systems, and can have different character sets.
- All of the data types that are supported for use in Altibase databases are also supported by DataPort.
- A previously paused import job can be restarted without duplicating or omitting any data from the original job.

10.2.1.2 Related Meta Table

- `SYS_DATA_PORTS_`

This meta table contains information about the export and import jobs that are either underway or have been completed. Query this table to check the status of a particular job. The `RESUME_DP` procedure also checks this table when restarting a stopped job. For more information, please refer to the ALTIBASE HDB *General Reference*.

10.2.1.3 Related Properties

The following DataPort-related properties can be set in `altibase.properties`:

- `DATAPORT_FILE_DIRECTORY`
- `DATAPORT_IMPORT_COMMIT_UNIT`
- `DATAPORT_IMPORT_STATEMENT_UNIT`

10.2 DataPort

For more information, please refer to the ALTIBASE HDB *General Reference*.

10.2.1.4 Exceptions related to FILE Input/Output

The DataPort stored procedures and functions can raise exceptions. For example, if a hardware or OS error, such as a Disk I/O error, occurs, the procedure will raise an `Invalid Operation` exception.

If an exception occurs while exporting data, there will be no effect on the source database. The file to which the data are being exported will be treated as a file that has zero (0) rows.

If an exception occurs while importing data, most of the rows that have been processed will be inserted into the destination table. In this case, the number of rows that will be inserted depends on the values set for the `DATAPORT_IMPORT_COMMIT_UNIT` and `DATAPORT_IMPORT_STATEMENT_UNIT` properties.

Setting these properties to smaller values increases the number of rows that are still inserted in the event of an exception, but entails a drop in performance. In contrast, setting these properties to larger values improves performance, but fewer rows are successfully inserted if an exception occurs.

The status of DataPort jobs can be monitored by querying the `SYS_DATA_PORTS` meta table.

10.2.1.5 Character Sets and DataPort

DataPort stored procedures export and import data without performing any data conversion. If the character set of the source database is different from the character set of the destination database, the desired results may not be achieved, or the import job may fail during validation. In such cases, it will be necessary to convert the character set of the data to be imported so that it matches that of the destination database using the `convdp` utility. For more information about this utility, please refer to the ALTIBASE HDB *Utilities Manual*.

10.2.1.6 Naming Conventions Used When Exporting and Importing Files

The naming conventions described in this section apply to the `IMPORT_FROM_FILE` stored procedure when importing data from files, to the `EXPORT_TO_FILE`, `EXPORT_PARTITION_TO_FILE`, and `EXPORT_USER_TABLES` stored procedures when exporting data to files.

These settings are also used by the `convdp` utility when searching for the file on which to perform the conversion and determining the name of the output file.

Importing Data from Files

If the user has specified a file name, data files are searched for in the following order:

1. the name specified by the user
2. the name specified by the user + ".dpf"
3. the name specified by the user + "_0.dpf"

If the specified file cannot be found after searching for all three of the above-described name variations, an error is raised.

If the user has not specified a file name, data files are searched for in the following order. Here, "*table-*

name" and *ownername*" refer respectively to the name of the table into which the data are to be imported and the table's owner.

1. *ownername_tablename*
2. *ownername_tablename* + ".dpf"
3. *ownername_tablename* + "_0.dpf"

If the specified file cannot be found after searching for all three of the above-described name variations, an error is raised.

Exporting Data To Files

If the user has specified a file name, then "_0.dpf" is appended to the file name.

If no file name is specified, the file will be given an automatically generated name having the format *ownername_tablename_0.dpf*, where *tablename*" and *ownername*" refer respectively to the table from which the data are exported and the owner of the table.

Note that in both cases, if a file having the same name already exists, it will be overwritten. Additionally, if the *split* parameter is set, and the number of records is greater than the value of the split parameter, then multiple files will be created. In this case, the file names will end with "_1.dpf", "_2.dpf", etc.

User-Specified Filename Extensions

Note that for both import and export operations, filename extensions on user-specified file names are not treated as extensions. That is, ".dpf" or "_0.dpf" is appended to the end of a user-specified filename, regardless of any filename extension (including the ".dpf" extension) that the name may already have. For example, if the user specifies "a.dpf" as the name of an export file, the file will be saved as "a.dpf_0.dpf".

10.2.1.7 Restrictions

A DataPort stored procedure can only import data into existing tables. In other words, DataPort cannot be used to create tables.

All attributes for all columns, such as their type, precision, scale, position in the table, etc. must be the same for both the source table and the destination table.

Tables that contain encrypted columns cannot be exported or imported. Before importing or exporting a table that has an encrypted column, it is first necessary to either remove the encryption from the encrypted column or use the CREATE TABLE AS SELECT (CTAS) statement to create a table that has no encrypted columns but contains the same data as the original table, and then export the data from the new table.

If the destination table has any triggers or indexes, if any of its columns have any constraints, such as FOREIGN KEY or NOT NULL constraints, or if it is a replication target table, it will be impossible to import data into the table. Before importing data, all such constraints and objects will have to be removed from the table, after which it will be possible to import the data.

10.2.2 DataPort Stored Procedures and Functions

The stored procedures and functions that comprise DataPort are listed in the following table. Only the SYS user can execute these procedures.

Name	Description
EXPORT_TO_FILE	Exports all of the data in a table to one or more files
EXPORT_PARTITION_TO_FILE	Exports the data in one of the partitions of the specified table to one or more files
EXPORT_USER_TABLES	Exports all of the tables belonging to a user to one or more files
IMPORT_FROM_FILE	Imports data from a DataPort file into a destination table
CLEAR_DP	Removes all records pertaining to completed jobs from the SYS_DATA_PORTS_ meta table
RESUME_DP	Restarts a stopped import job
REMOVE_DP	Forcibly removes a specific job from the SYS_DATA_PORTS_ meta table

10.2.3 EXPORT_TO_FILE

This procedure downloads data from the specified table and writes the data to DataPort files located in the specified directory. These DataPort files can be later imported into another database using the IMPORT_FROM_FILE procedure.

10.2.3.1 Syntax

```
EXPORT_TO_FILE (
    owner      IN VARCHAR,
    tablename  IN VARCHAR,
    filename   IN VARCHAR DEFAULT NULL,
    jobname    IN VARCHAR DEFAULT NULL,
    split      IN BIGINT  DEFAULT NULL,
    directory  IN VARCHAR DEFAULT NULL);
```

10.2.3.2 Parameters

Name	Input/Output	Data type	Description
<i>owner</i>	IN	VARCHAR	The name of the user who owns the source table
<i>tablename</i>	IN	VARCHAR	The name of the source table. It must be owned by <i>owner</i> .

Name	Input/Output	Data type	Description
<i>filename</i>	IN	VARCHAR	The name of the file to which the data will be exported. Please refer to 10.2.1.6 Naming Conventions Used When Exporting and Importing Files for complete information on how the file name is determined.
<i>jobname</i>	IN	VARCHAR	The name of the job. If no name is specified, the job will be given an automatically generated name having the following format: <owner>_<tablename>.
<i>split</i>	IN	BIGINT	The maximum number of rows that are allowed in one file. If the number of exported rows reaches this maximum number, an additional file will be created. If <i>split</i> is not specified, or if it is set to NULL, there will be no limit on the number of rows that can be written to one file. That is, all data are written to the same file.
<i>directory</i>	IN	VARCHAR	The name of the directory in which the file specified by <i>filename</i> is to be saved. The default directory is the directory specified in the DATAPORT_FILE_DIRECTORY property.

10.2.3.3 Return Value

Because it is a stored procedure, no result value is returned.

10.2.3.4 Exceptions

EXPORT_TO_FILE can raise the following system-defined exceptions.

- RUNNING_JOB
- TOO_LONG_DATAPORT_DIRECTORY_PATH
- SUSPENDED_JOB

For more information on the system-related exceptions that can be raised by DataPort, please refer to [10.2.10 DataPort Exceptions](#) in this chapter.

10.2.3.5 Example

The following example shows how to export the *TD* table, which belongs to the *SYS* user. Because the *filename*, *jobname*, *split* and *directory* arguments are not specified, the exported data will be writ-

10.2 DataPort

ten to the SYS_TD_0 .dpf file in the default directory.

```
iSQL> EXEC EXPORT_TO_FILE( 'SYS', 'TD' );  
Export - SYS_TD 10 record(s).  
Execute success.
```

10.2.4 EXPORT_PARTITION_TO_FILE

This procedure downloads data from the specified partition of the specified table and writes the data to DataPort files located in the specified directory. Later, these DataPort files can be imported into another database using the IMPORT_FROM_FILE procedure.

10.2.4.1 Syntax

```
EXPORT_PARTITION_TO_FILE (  
    owner          IN VARCHAR,  
    tablename      IN VARCHAR,  
    partitionname  IN VARCHAR,  
    filename       IN VARCHAR DEFAULT NULL,  
    jobname        IN VARCHAR DEFAULT NULL,  
    split          IN BIGINT  DEFAULT NULL,  
    directory      IN VARCHAR DEFAULT NULL);
```

10.2.4.2 Parameters

Name	Input/Output	Data type	Description
<i>owner</i>	IN	VARCHAR	The name of the user who owns the source table
<i>tablename</i>	IN	VARCHAR	The name of the source table. It must belong to <i>owner</i> .
<i>partitionname</i>	IN	VARCHAR	The name of a partition. It must be the name of a partition in the specified table.
<i>filename</i>	IN	VARCHAR	The name of the file to which the data will be exported. Please refer to 10.2.1.6 Naming Conventions Used When Exporting and Importing Files for complete information on how the file name is determined.
<i>jobname</i>	IN	VARCHAR	The name of the job. If no name is specified, the job will be given an automatically generated name having the following format: <i><owner>_<tablename></i> .

Name	Input/Output	Data type	Description
<i>split</i>	IN	BIGINT	The maximum number of rows that are allowed in one file. If the number of exported rows reaches this maximum number, an additional file will be created. If <i>split</i> is not specified, or if it is set to NULL, there will be no limit on the number of rows that can be written to one file. That is, all data will be written to the same file.
<i>directory</i>	IN	VARCHAR	The name of the directory in which the file specified by <i>filename</i> is to be saved. The default directory is the directory specified in the <code>DATAPORT_FILE_DIRECTORY</code> property.

10.2.4.3 Return Value

Because it is a stored procedure, no result value is returned.

10.2.4.4 Exceptions

EXPORT_PARTITION_TO_FILE can raise the following system-defined exceptions.

- RUNNING_JOB
- TOO_LONG_DATAPORT_DIRECTORY_PATH
- SUSPENDED_JOB

For more information on the system-related exceptions that can be raised by DataPort, please refer to [10.2.10 DataPort Exceptions](#) in this chapter.

10.2.4.5 Example

The following example shows how to export partition *L1* of table *TESTL*, which belongs to the *SYS* user. Because the *filename*, *jobname*, *split* and *directory* arguments are not specified, the exported data will be written to the `SYS_TESTL_L1_0.dpf` file in the default directory.

```
iSQL> EXEC EXPORT_PARTITION_TO_FILE('SYS', 'TESTL', 'L1' );
Export - SYS_TESTL_L1 2 record(s).
Execute success.
```

10.2.5 EXPORT_USER_TABLES

This procedure downloads all of the data from all of the tables that belong to the specified user and writes the data to DataPort files located in the specified directory.

10.2 DataPort

10.2.5.1 Syntax

```
EXPORT_USER_TABLES (  
    owner      IN VARCHAR,  
    split      IN BIGINT  DEFAULT NULL,  
    directory  IN VARCHAR DEFAULT NULL);
```

10.2.5.2 Parameters

Name	Input/Output	Data type	Description
<i>owner</i>	IN	VARCHAR	The name of the user whose tables are to be exported. All of the tables belonging to the specified user will be exported.
<i>split</i>	IN	BIGINT	The maximum number of rows that are allowed in one file. If the number of exported rows reaches this maximum number, an additional file will be created. If <i>split</i> is not specified, or if it is set to NULL, there will be no limit on the number of rows that can be written to one file. That is, all of the data in each table will be written to the same file.
<i>directory</i>	IN	VARCHAR	The name of the directory in which to save the resultant files. The default directory is the directory specified in the DATAPORT_FILE_DIRECTORY property.

10.2.5.3 Return Value

Because it is a stored procedure, no result value is returned.

10.2.5.4 Exception

EXPORT_USER_TABLES can raise the following system-defined exception.

- TOO_LONG_DATAPORT_DIRECTORY_PATH

For more information on the system-related exceptions that can be raised by DataPort, please refer to [10.2.10 DataPort Exceptions](#) in this chapter.

10.2.5.5 Example

The following example shows how to export all tables belonging to the SYS user. Tables TD and TESTL, which belong to the SYS user, will be written to the following files, which are located in the

default directory: SYS_TD_0.dpf and SYS_TESTL_0.dpf

```
iSQL> EXEC EXPORT_USER_TABLES( 'SYS' );
USER_NAME: SYS
TABLE_NAME: TD
Export - SYS_TD 10 record(s).
TABLE_NAME: TESTL
Export - SYS_TESTL 3 record(s).
Execute success.
```

10.2.6 IMPORT_FROM_FILE

This procedure loads data from the export file into the destination table.

10.2.6.1 Syntax

```
IMPORT_FROM_FILE (
    owner      IN VARCHAR,
    tablename  IN VARCHAR,
    filename   IN VARCHAR DEFAULT NULL,
    jobname    IN VARCHAR DEFAULT NULL,
    firstrow   IN BIGINT  DEFAULT NULL,
    lastrow    IN BIGINT  DEFAULT NULL,
    directory  IN VARCHAR DEFAULT NULL);
```

10.2.6.2 Parameters

Name	Input/Output	Data type	Description
<i>owner</i>	IN	VARCHAR	The name of the user who owns the destination table.
<i>tablename</i>	IN	VARCHAR	The name of the destination table. It must belong to the user specified by <i>owner</i> .
<i>filename</i>	IN	VARCHAR	The name of the file from which the data are to be imported. Please refer to 10.2.1.6 Naming Conventions Used When Exporting and Importing Files for complete information on how the file name is determined.
<i>jobname</i>	IN	VARCHAR	The name of the job. If no name is specified, the job will be given an automatically generated name having the following format: <i><owner>_<tablename></i> .
<i>firstrow</i>	IN	BIGINT	The first row to be imported (the default is the first row in the file).

10.2 DataPort

Name	Input/Output	Data type	Description
<i>lastrow</i>	IN	BIGINT	The last row to be imported (the default is the last row in the file).
<i>directory</i>	IN	VARCHAR	The name of the directory in which <i>filename</i> is located. The default is the directory specified using the DATAPORT_FILE_DIRECTORY property.

10.2.6.3 Return Value

Because it is a stored procedure, no result value is returned.

10.2.6.4 Exceptions

IMPORT_FROM_FILE can raise the following system-defined exceptions:

- RUNNING_JOB
- TOO_LONG_DATAPORT_DIRECTORY_PATH
- SUSPENDED_JOB
- CORRUPTED_BLOCK
- INVALID_COLUMN_SIZE
- CORRUPTED_HEADER
- DATA_PORT_INTERNAL_ERROR
- VERSION_NOT_SUPPORTED
- CANT_OPEN_FILE
- INCOMPATIBLE_DATA_TYPE
- TABLE_HAS_TRIGGERS
- TABLE_HAS_CONSTRAINTS
- TABLE_HAS_REPLICATION
- TABLE_HAS_INDEX
- TABLE_HAS_FOREIGN_KEY
- NOT_SAME_DATABASE_CHARSET
- NOT_SAME_NATIONAL_CHARSET

For more information on the system-related exceptions that can be raised by DataPort, please refer

to [10.2.10 DataPort Exceptions](#) in this chapter.

10.2.6.5 Example

The following example shows how to import data from one of the following files: TDFILE, TDFILE.dpf or TDFILE_0.dpf, which are located in the default directory, to table *TD*, which belongs to the *SYS* user.

```
iSQL> EXEC IMPORT_FROM_FILE( 'SYS', 'TD', 'TDFILE' );
SYS_TD 10 record(s)
Import - SYS_TD 10 record(s).
```

10.2.7 CLEAR_DP

This procedure removes all records pertaining to completed import and export jobs from the *SYS_DATA_PORTS_meta* table.

10.2.7.1 Syntax

```
CLEAR_DP ( ) ;
```

10.2.7.2 Return Value

Because it is a stored procedure, no result value is returned.

10.2.7.3 Exceptions

This procedure never returns an error.

10.2.8 RESUME_DP

This procedure restarts an import job that is currently stopped.

10.2.8.1 Syntax

```
RESUME_DP ( jobname IN VARCHAR ) ;
```

10.2.8.2 Parameters

Name	Input/Output	Data type	Description
<i>jobname</i>	IN	VARCHAR	The name of the job to be restarted.

10.2.8.3 Return Value

Because it is a stored procedure, no result value is returned.

10.2 DataPort

10.2.8.4 Exception

RESUME_DP can raise the following system-defined exception.

- INVALID_JOBNAME

For more information on the system-related exceptions that can be raised by DataPort, please refer to [10.2.10 DataPort Exceptions](#) in this chapter.

10.2.8.5 Example

The following example shows how to restart a job that was stopped due to a lack of disk space or a file I/O error.

```
iSQL> EXEC IMPORT_FROM_FILE( 'SYS', 'TD' );
TM 230 record(s)
[ERR-11123: The tablespace does not have enough free space ( TBS Name:TBS1
).]
iSQL> ALTER TABLESPACE TBS1 ADD DATAFILE 'TBS2.TBS' SIZE 50M;
ALTER success.
iSQL> EXEC RESUME_TE( 'SYS_TD' );
TM 320 record(s)
Import - TM 320 record(s).
Execute success.
```

10.2.9 REMOVE_DP

This procedure forcibly removes a job from the list of jobs in the SYS_DATA_PORTS__meta table, even if the job is suspended.

10.2.9.1 Syntax

```
REMOVE_DP (jobname IN VARCHAR);
```

10.2.9.2 Parameters

Name	Input/Output	Data type	Description
<i>jobname</i>	IN	VARCHAR	The name of the job to be removed.

10.2.9.3 Return Value

Because it is a stored procedure, no result value is returned.

10.2.9.4 Exceptions

REMOVE_DP can raise the following system-defined exceptions.

- INVALID_JOBNAME

- `RUNNING_JOB`

For more information on the system-related exceptions that can be raised by DataPort, please refer to [10.2.10 DataPort Exceptions](#) in this chapter.

10.2.10 DataPort Exceptions

Some system-defined exceptions are specific to DataPort, and are thus only raised by DataPort stored procedures. These exceptions are listed in the following table:

Name	Description
<code>CANT_OPEN_FILE</code>	The specified file could not be found.
<code>CORRUPTED_BLOCK</code>	The n^{th} block in the DataPort file is corrupted.
<code>CORRUPTED_HEADER</code>	The DataPort file header is corrupted.
<code>DATA_PORT_INTERNAL_ERROR</code>	There are one or more errors in the DataPort file.
<code>INCOMPATIBLE_DATA_TYPE</code>	The DataPort file and the destination table contain non-matching column data types.
<code>INVALID_COLUMN_SIZE</code>	The input data column size exceeds the maximum column size.
<code>INVALID_JOBNAME</code>	A job having that name could not be found.
<code>NOT_SAME_DATABASE_CHARSET</code>	The character sets for the database and the DataPort file are not the same.
<code>NOT_SAME_NATIONAL_CHARSET</code>	The national character sets for the destination database and the DataPort file are not the same.
<code>RUNNING_JOB</code>	The specified job already exists.
<code>SUSPENDED_JOB</code>	The specified job name is the same as the name of a suspended job.
<code>TABLE_HAS_CONSTRAINTS</code>	The data could not be imported because one or more constraints have been defined for the table.
<code>TABLE_HAS_FOREIGN_KEY</code>	The data could not be imported because one or more foreign keys are present.
<code>TABLE_HAS_INDEX</code>	The data could not be imported because one or more indexes exist for the table.
<code>TABLE_HAS_REPLICATION</code>	The data could not be imported because the table is a replication target table.
<code>TABLE_HAS_TRIGGERS</code>	The data could not be imported because one or more triggers have been defined for the table.
<code>TOO_LONG_DATAPORT_DIRECTORY_PATH</code>	The length of the specified directory name is longer than 1024 bytes.

Name	Description
VERSION_NOT_SUPPORTED	The version of the DataPort file is not supported.

10.2.11 Examples

- Normal Execution

```

iSQL> CREATE TABLE TM ( I1 INTEGER, I2 CHAR(1000) );
CREATE success.
iSQL> CREATE TABLE TD ( I1 INTEGER, I2 CHAR(1000) );
CREATE success.

iSQL> INSERT INTO TM VALUES( '1', '1' );
1 row inserted.
iSQL> INSERT INTO TM VALUES( '2', '2' );
1 row inserted.
iSQL> INSERT INTO TM VALUES( '3', '3' );
1 row inserted.
iSQL> INSERT INTO TM VALUES( '4', '4' );
1 row inserted.
iSQL> INSERT INTO TM VALUES( '5', '5' );
1 row inserted.
iSQL> INSERT INTO TM VALUES( '6', '6' );
1 row inserted.
iSQL> INSERT INTO TM VALUES( '7', '7' );
1 row inserted.
iSQL> INSERT INTO TM VALUES( '8', '8' );
1 row inserted.
iSQL> INSERT INTO TM VALUES( '9', '9' );
1 row inserted.
iSQL> INSERT INTO TM VALUES( '10','10' );
1 row inserted.
iSQL> INSERT INTO TM SELECT * FROM TM;
10 rows inserted.
iSQL> INSERT INTO TM SELECT * FROM TM;
20 rows inserted.
iSQL> INSERT INTO TM SELECT * FROM TM;
40 rows inserted.
iSQL> INSERT INTO TM SELECT * FROM TM;
80 rows inserted.
iSQL> INSERT INTO TM SELECT * FROM TM;
160 rows inserted.

--# Export_to_file( <Table Owner>, <Table Name>, <File Name> );
iSQL> EXEC EXPORT_TO_FILE( 'SYS', 'TM', 'TMFILE' );
Export - SYS_TM 320 record(s).
Execute success.

--# Copy the generated file, TMFILE_0.dpf,
--# to the default directory, ${ALTIBASE_HOME}/dbs, on the target
machine.

--# Import_from_file( <Table Owner>, <Table Name>, <File Name> );
iSQL> EXEC IMPORT_FROM_FILE( 'SYS', 'TD', 'TMFILE' );
SYS_TD 50 record(s)
SYS_TD 100 record(s)
SYS_TD 150 record(s)
SYS_TD 200 record(s)
SYS_TD 250 record(s)

```

```
SYS_TD 300 record(s)
Import - SYS_TD 320 record(s).
Execute success.
```

```
--# Clear_the meta table
iSQL> EXEC CLEAR_DP();
REMOVE: SYS_TD
REMOVE: SYS_TM
CLEAR DATA_PORTS
Execute success.
```

- When an exception occurs while importing data

```
iSQL> CREATE TABLE TM ( I1 INTEGER, I2 CHAR(32000) );
CREATE success.
```

```
iSQL> CREATE TABLESPACE TBS1 DATAFILE 'TBS1.TBS' SIZE 10M;
CREATE success.
iSQL> CREATE TABLE TD ( I1 INTEGER, I2 CHAR(32000) ) TABLESPACE TBS1;
CREATE success.
```

```
iSQL> INSERT INTO TM VALUES( '1', '1' );
1 row inserted.
iSQL> INSERT INTO TM VALUES( '2', '2' );
1 row inserted.
iSQL> INSERT INTO TM VALUES( '3', '3' );
1 row inserted.
iSQL> INSERT INTO TM VALUES( '4', '4' );
1 row inserted.
iSQL> INSERT INTO TM VALUES( '5', '5' );
1 row inserted.
iSQL> INSERT INTO TM VALUES( '6', '6' );
1 row inserted.
iSQL> INSERT INTO TM VALUES( '7', '7' );
1 row inserted.
iSQL> INSERT INTO TM VALUES( '8', '8' );
1 row inserted.
iSQL> INSERT INTO TM VALUES( '9', '9' );
1 row inserted.
iSQL> INSERT INTO TM VALUES( '10','10' );
1 row inserted.
iSQL> INSERT INTO TM SELECT * FROM TM;
10 rows inserted.
iSQL> INSERT INTO TM SELECT * FROM TM;
20 rows inserted.
iSQL> INSERT INTO TM SELECT * FROM TM;
40 rows inserted.
iSQL> INSERT INTO TM SELECT * FROM TM;
80 rows inserted.
iSQL> INSERT INTO TM SELECT * FROM TM;
160 rows inserted.
```

```
--# Export_to_file( <Table Owner>, <Table Name>, <File Name>, <Job Name>
);
iSQL> EXEC EXPORT_TO_FILE( 'SYS', 'TM', 'TMFILE', 'SYS_TM_EXPORT' );
Export - SYS_TM_EXPORT 320 record(s).
Execute success.
```

```
--# In the following example,
--# if there is insufficient space on the device, the operation is rolled
back in units of 50 rows and
--# after abnormal shutdown of a server, the operation is rolled back in
units of 500(=50*10) rows.
```

10.2 DataPort

```
iSQL> ALTER SYSTEM SET DATAPORT_IMPORT_COMMIT_UNIT = 10;

ALTER success.
iSQL> ALTER SYSTEM SET DATAPORT_IMPORT_STATEMENT_UNIT = 50;
ALTER success.

--# Copy the generated file, TMFILE_0.dpf,
--# to the default directory, ${ALTIBASE_HOME}/dbs, on the target
machine.

--# Import_from_file( <Table Owner>, <Table Name>, <File Name>, <Job
Name> );
iSQL> EXEC IMPORT_FROM_FILE( 'SYS', 'TD', 'TMFILE', 'SYS_TM_IMPORT' );
SYS_TM_IMPORT 50 record(s)
SYS_TM_IMPORT 100 record(s)
SYS_TM_IMPORT 150 record(s)
SYS_TM_IMPORT 200 record(s)
[ERR-11123 : The tablespace does not have enough free space ( TBS Name
:TBS1 ).]

iSQL> ALTER TABLESPACE TBS1 ADD DATAFILE 'TBS2.TBS' SIZE 10M;
ALTER success.

--# Resume_dp( <Job Name> );
iSQL> EXEC RESUME_DP( 'SYS_TM_IMPORT' );
SYS_TM_IMPORT 250 record(s)
SYS_TM_IMPORT 300 record(s)
Import - SYS_TM_IMPORT 320 record(s).
Execute success.

--# Clear_DP
iSQL> EXEC CLEAR_DP();
REMOVE: SYS_TM_EXPORT
REMOVE: SYS_TM_IMPORT
CLEAR DATA_PORTS
Execute success.
```

- **Exporting one partition**

```
iSQL> CREATE TABLE TM ( I1 INTEGER, I2 CHAR(1000) );
CREATE success.
iSQL> CREATE TABLE TD ( I1 INTEGER, I2 CHAR(1000) )
PARTITION BY HASH ( I1 )
(
    PARTITION H1,
    PARTITION H2,
    PARTITION H3
) TABLESPACE SYS_TBS_DISK_DATA;
CREATE success.

iSQL> INSERT INTO TD VALUES( '1', '1' );
1 row inserted.
iSQL> INSERT INTO TD VALUES( '2', '2' );
1 row inserted.
iSQL> INSERT INTO TD VALUES( '3', '3' );
1 row inserted.
iSQL> INSERT INTO TD VALUES( '4', '4' );
1 row inserted.
iSQL> INSERT INTO TD VALUES( '5', '5' );
1 row inserted.
iSQL> INSERT INTO TD VALUES( '6', '6' );
1 row inserted.
iSQL> INSERT INTO TD VALUES( '7', '7' );
1 row inserted.
```



```

iSQL> INSERT INTO TD VALUES( '8', '8' );
1 row inserted.
iSQL> INSERT INTO TD VALUES( '9', '9' );
1 row inserted.
iSQL> INSERT INTO TD VALUES( '10','10' );
1 row inserted.

--# Export_partition_to_file( <Table Owner>, <Table Name>, <Partition
Name>, <File Name>, <Job Name> );
iSQL> EXEC EXPORT_PARTITION_TO_FILE( 'SYS', 'TD', 'H1',
'SYS_TD_PARTITION_H1', 'EXPORT_PARTITION_H1' );
Export - EXPORT_PARTITION_H1 2 record(s).
Execute success.

--# Copy the generated file, TMFILE_0.dpf,
--# to the default directory, ${ALTIBASE_HOME}/dbs, on the target
machine.

--# Import_from_file( <Table Owner>, <Table Name>, <File Name>, <Job
Name> );
iSQL> EXEC IMPORT_FROM_FILE( 'SYS', 'TM', 'SYS_TD_PARTITION_H1' );
Import - SYS_TM 2 record(s).
Execute success.

```

10.3 DBMS Stat

DBMS Stat is an ALTIBASE HDB feature that is intended for use when collecting statistics about the database. This functionality is provided in the form of multiple system-defined stored procedures that are used for this purpose.

10.3.1 Overview

If the statistics that are automatically collected in the ALTIBASE HDB server are not correct, it will be impossible for the query optimizer to create optimized execution plans. Therefore, in such cases, it will be necessary to collect statistics manually.

10.3.2 DBMS Stat Stored Procedures

The stored procedures that comprise DBMS Stat are listed in the following table. These procedures can be used to gather statistics and reconstruct execution plans. Only the SYS user can execute these procedures

Name	Description
GATHER_SYSTEM_STATS	Gathers statistics about the database system
GATHER_DATABASE_STATS	Gathers statistics about all of the tables in the database
GATHER_TABLE_STATS	Gathers statistics about a particular table
GATHER_INDEX_STATS	Gathers statistics about a particular index

10.3.3 Notes

- The process of gathering statistics imposes an additional workload on the ALTIBASE HDB server.
- The statistics that are gathered in this way should be considered approximate.
- After statistics are gathered, ALTIBASE HDB reconstructs the execution plans for all queries that reference any of the objects for which the statistical information was gathered. During this process, the performance of the ALTIBASE HDB server can suffer somewhat.

10.3.4 GATHER_SYSTEM_STATS

This procedure gathers statistics about the database system.

10.3.4.1 Syntax

```
GATHER_SYSTEM_STATS ( );
```

10.3.4.2 Return Value

Because it is a stored procedure, no result value is returned.

10.3.4.3 Example

```
iSQL> EXEC GATHER_SYSTEM_STATS();
Execute success.
```

10.3.5 GATHER_DATABASE_STATS

This procedure gathers statistics about all of the tables that exist in the database.

10.3.5.1 Syntax

```
GATHER_DATABASE_STATS (
    estimate_percent    IN FLOAT    DEFAULT NULL,
    degree              IN INTEGER  DEFAULT NULL,
    no_invalidate       IN BOOLEAN  DEFAULT FALSE );
```

10.3.5.2 Parameters

Name	Input/Output	Data type	Description
<i>estimate_percent</i>	IN	FLOAT	This is the ratio of the amount of data to be sampled (for the purpose of estimating statistics) to the total amount of data available for the target object. It can be set anywhere in the range from 0 (zero) to 1. If <i>estimate_percent</i> is not specified, or if it is set to NULL, this value is automatically determined depending on the size of the object.
<i>degree</i>	IN	INTEGER	This is the number of threads that work in parallel to gather statistics. If <i>degree</i> is not specified, the default value is 1.
<i>no_invalidate</i>	IN	BOOLEAN	This determines whether to reconstruct the execution plans for queries pertaining to the objects for which statistics are gathered. Setting <i>no_invalidate</i> to TRUE disables reconstruction of the execution plans. If set to FALSE, which is the default value, the execution plans are reconstructed.

10.3 DBMS Stat

10.3.5.3 Return Value

Because it is a stored procedure, no result value is returned.

10.3.5.4 Example

```
isQL> EXEC GATHER_DATABASE_STATS();  
SYSTEM_.SYS_TABLES_  
SYSTEM_.SYS_COLUMNS_  
SYSTEM_.SYS_DATABASE_  
SYSTEM_.SYS_USERS_  
.  
.  
.  
Execute success.
```

10.3.6 GATHER_TABLE_STATS

This procedure gathers statistics about a specific table and the indexes that are defined on the basis of that table.

10.3.6.1 Syntax

```
GATHER_TABLE_STATS (  
    owner          IN VARCHAR,  
    tablename      IN VARCHAR,  
    partname       IN VARCHAR DEFAULT NULL,  
    estimate_percent IN FLOAT   DEFAULT NULL,  
    degree         IN INTEGER  DEFAULT NULL,  
    no_invalidate  IN BOOLEAN  DEFAULT FALSE );
```

10.3.6.2 Parameters

Name	Input/Output	Data type	Description
<i>owner</i>	IN	VARCHAR	The name of the user who owns the table for which statistics are gathered
<i>tablename</i>	IN	VARCHAR	The name of the table for which statistics are gathered
<i>partname</i>	IN	VARCHAR	The name of a partition. If specified, only statistics pertaining to that partition are gathered. If not specified, it defaults to NULL, and statistics about all of the partitions in the specified table are gathered

Name	Input/Output	Data type	Description
<i>estimate_percent</i>	IN	FLOAT	This is the ratio of the amount of data to be sampled (for the purpose of estimating statistics) to the total amount of data available for the target object. It can be set anywhere in the range from 0 (zero) to 1. If <i>estimate_percent</i> is not specified, or if it is set to NULL, this value is automatically determined depending on the size of the object.
<i>degree</i>	IN	INTEGER	This is the number of threads that work in parallel to gather statistics. If <i>degree</i> is not specified, the default value is 1.
<i>no_invalidate</i>	IN	BOOLEAN	This determines whether to reconstruct the execution plans for queries pertaining to the objects for which statistics are gathered. Setting <i>no_invalidate</i> to TRUE disables reconstruction of the execution plans. If set to FALSE, which is the default value, the execution plans are reconstructed.

10.3.6.3 Return Value

Because it is a stored procedure, no result value is returned.

10.3.6.4 Example

```
isQL> EXEC GATHER_TABLE_STATS( 'SYS', 'T1' );
Execute success.
```

10.3.7 GATHER_INDEX_STATS

This procedure gathers statistics about a specific index.

10.3.7.1 Syntax

```
GATHER_INDEX_STATS (
    owner          IN VARCHAR,
    indexname      IN VARCHAR,
    estimate_percent IN FLOAT   DEFAULT NULL,
    degree         IN INTEGER  DEFAULT NULL,
    no_invalidate  IN BOOLEAN  DEFAULT FALSE );
```

10.3.7.2 Parameters

Name	Input/Output	Data type	Description
<i>owner</i>	IN	VARCHAR	The name of the user who owns the index for which statistics are gathered
<i>indexname</i>	IN	VARCHAR	The name of the index for which statistics are gathered
<i>estimate_percent</i>	IN	FLOAT	This is the ratio of the amount of data to be sampled (for the purpose of estimating statistics) to the total amount of data available for the target object. It can be set anywhere in the range from 0 (zero) to 1. If estimate_percent is not specified, or if it is set to NULL, this value is automatically determined depending on the size of the object.
<i>degree</i>	IN	INTEGER	This is the number of threads that work in parallel to gather statistics. If degree is not specified, the default value is 1.
<i>no_invalidate</i>	IN	BOOLEAN	This determines whether to reconstruct the execution plans for queries pertaining to the objects for which statistics are gathered. Setting no_invalidate to TRUE disables reconstruction of the execution plans. If set to FALSE, which is the default value, the execution plans are reconstructed.

10.3.7.3 Return Value

Because it is a stored procedure, no result value is returned.

10.3.7.4 Example

```
iSQL> EXEC GATHER_INDEX_STATS( 'SYS', 'T1_IDX' );
Execute success.
```

10.4 Miscellaneous Functions

10.4.1 REMOVE_XID

This procedure forcibly deletes old XID information which was heuristically rolled back or committed in an XA environment.

10.4.1.1 Syntax

```
REMOVE_XID (xidname IN VARCHAR);
```

10.4.1.2 Parameters

Name	Input/Output	Data type	Description
<i>xidname</i>	IN	VARCHAR	The name of the XID to be removed

10.4.1.3 Return Value

Because it is a stored procedure, no result value is returned.

10.4.1.4 Exceptions

REMOVE_XID can raise the following system-defined exceptions.

- NOT_EXIST_XID
- InvalidXaState

10.4.2 SLEEP

This procedure makes the session sleep for the number of seconds specified in the *seconds* argument.

10.4.2.1 Syntax

```
SLEEP (seconds IN INTEGER);
```

10.4.2.2 Parameters

Name	Input/Output	Data type	Description
<i>seconds</i>	IN	INTEGER	The amount of time to sleep, in seconds

10.4 Miscellaneous Functions

10.4.2.3 Return Value

Because it is a stored procedure, no result value is returned.

10.4.2.4 Exceptions

This stored procedure never raises an exception.

Appendix A. Examples

Stored Procedure Examples

Example 1

Create a stored procedure called *dumpReplScript*, which outputs a script for creating a replication object.

The tables to be replicated are the *employees* table and the *departments* table, the local server's IP address and port number are 192.168.1.12 and 35524, and the remote server's IP address and port number are 192.168.1.60 and 25524.

On the remote server:

```
iSQL> CREATE REPLICATION repl WITH '192.168.1.12',35524 FROM SYS.EMPLOYEES TO
SYS.EMPLOYEES, FROM SYS.DEPARTMENTS TO SYS.DEPARTMENTS;
Create success.
iSQL> ALTER REPLICATION repl START;
Alter success.
```

On the local server:

```
iSQL> CREATE REPLICATION repl WITH '192.168.1.60',25524 FROM SYS.EMPLOYEES TO
SYS.EMPLOYEES, FROM SYS.DEPARTMENTS TO SYS.DEPARTMENTS;
Create success.
iSQL> ALTER REPLICATION repl START;
Alter success.
```

```
iSQL> create or replace procedure dumpReplScript
(pl varchar(40))
as
  cursor c1 is
    select system_.sys_replications_.replication_name,
           system_.sys_repl_hosts_.host_ip,
           system_.sys_repl_hosts_.port_no,
           system_.SYS_REPLICATIONS_.ITEM_COUNT
    from system_.sys_replications_, system_.sys_repl_hosts_
    where system_.sys_replications_.replication_name =
system_.sys_repl_hosts_.replication_name
      and system_.sys_replications_.replication_name = UPPER(P1);
  r_name varchar(40);
  r_ip varchar(40);
  r_port varchar(20);
  r_item_cnt integer;
  r_local_user_name varchar(40);
  r_local_table_name varchar(40);
  r_remote_user_name varchar(40);
  r_remote_table_name varchar(40);
  cursor c2 is
    select system_.SYS_REPL_ITEMS_.LOCAL_USER_NAME,
           system_.SYS_REPL_ITEMS_.LOCAL_TABLE_NAME,
           system_.SYS_REPL_ITEMS_.REMOTE_USER_NAME,
           system_.SYS_REPL_ITEMS_.REMOTE_TABLE_NAME
```

Stored Procedure Examples

```
        from system_.sys_repl_items_
        where system_.SYS_REPL_ITEMS_.replication_name = r_name;
begin
    open c1;
    SYSTEM_.PRINTLN('-----');
    SYSTEM_.PRINTLN('');
    loop
        fetch C1 into r_name, r_ip, r_port, r_item_cnt;
        exit when C1%NOTFOUND;
        SYSTEM_.PRINT(' CREATE REPLICATION ');
        SYSTEM_.PRINT(r_name);
        SYSTEM_.PRINT(' WITH ');
        SYSTEM_.PRINT(r_ip);
        SYSTEM_.PRINT(' ',');
        SYSTEM_.PRINT(r_port);
        SYSTEM_.PRINTLN(' ');
        open c2;
        for i in 1 .. r_item_cnt loop
            fetch c2 into r_local_user_name,
                           r_local_table_name,
                           r_remote_user_name,
                           r_remote_table_name;
            SYSTEM_.PRINT(' FROM ');
            SYSTEM_.PRINT(r_local_user_name);
            SYSTEM_.PRINT(' ');
            SYSTEM_.PRINT(r_local_table_name);
            SYSTEM_.PRINT(' TO ');
            SYSTEM_.PRINT(r_remote_user_name);
            SYSTEM_.PRINT(' ');
            SYSTEM_.PRINT(r_remote_table_name);
            if i <> r_item_cnt then
                SYSTEM_.PRINTLN(' ');
            else
                SYSTEM_.PRINTLN(';');
            end if;
        end loop;
        close c2;
    end loop;
    close c1;
    SYSTEM_.PRINTLN('');
    SYSTEM_.PRINTLN('-----');
end;
/
```

The following is output by the *dumpReplScript* stored procedure on the local server.

```
iSQL> exec dumpReplScript('repl');
-----
CREATE REPLICATION REP1 WITH '192.168.1.60',25524
FROM SYS.DEPARTMENTS TO SYS.DEPARTMENTS,
FROM SYS.EMPLOYEES TO SYS.EMPLOYEES;
-----
EXECUTE success.
```

Example 2

Create a stored procedure called *showReplications*, which outputs the name and other information about replication objects.

```
create or replace procedure showReplications
as
    cursor c1 is select a.replication_name,
```

```

        b.host_ip, b.port_no,
        decode(a.is_started,1,'Running',0,'Not Running')
    from system.sys_replications_ a, system.sys_repl_hosts_ b
    where a.replication_name = b.replication_name;
r_name varchar(40);
r_ip varchar(40);
r_port varchar(20);
r_status varchar(20);
r_local_user_name varchar(40);
r_local_table_name varchar(40);
r_remote_user_name varchar(40);
r_remote_table_name varchar(40);
cursor c2 is select system.SYS_REPL_ITEMS.LOCAL_USER_NAME,
        system.SYS_REPL_ITEMS.LOCAL_TABLE_NAME,
        system.SYS_REPL_ITEMS.REMOTE_USER_NAME,
        system.SYS_REPL_ITEMS.REMOTE_TABLE_NAME
    from system.sys_repl_items_
    where system.SYS_REPL_ITEMS.replication_name = r_name;
begin
    open c1;
    SYSTEM_.PRINTLN('-----');
    SYSTEM_.PRINTLN(' Replication Info');
    SYSTEM_.PRINTLN('-----');
    SYSTEM_.PRINTLN(' Name IP Port Status');
    SYSTEM_.PRINTLN('-----');
    SYSTEM_.PRINTLN('');
    loop
        fetch C1 into r_name, r_ip, r_port, r_status;
        exit when C1%NOTFOUND;
        SYSTEM_.PRINT(' ');
        SYSTEM_.PRINT(r_name);
        SYSTEM_.PRINT(' ');
        SYSTEM_.PRINT(r_ip);
        SYSTEM_.PRINT(' ');
        SYSTEM_.PRINT(r_port);
        SYSTEM_.PRINT(' ');
        SYSTEM_.PRINTLN(r_status);
        SYSTEM_.PRINTLN('+++++++');
        SYSTEM_.PRINTLN(' Local Table Name Remote Table Name');
        SYSTEM_.PRINTLN('+++++++');
        open c2;
        loop
            fetch c2 into r_local_user_name, r_local_table_name,
r_remote_user_name, r_remote_table_name;
            exit when C2%NOTFOUND;
            SYSTEM_.PRINT(' ');
            SYSTEM_.PRINT(r_local_user_name);
            SYSTEM_.PRINT('.');
            SYSTEM_.PRINT(r_local_table_name);
            SYSTEM_.PRINT(' ');
            SYSTEM_.PRINT(r_remote_user_name);
            SYSTEM_.PRINT('.');
            SYSTEM_.PRINTLN(r_remote_table_name);
        end loop;
        close c2;
    end loop;
    close c1;
    SYSTEM_.PRINTLN('');
    SYSTEM_.PRINTLN('-----');
end;
/

```

The following is output by the *showReplications* stored procedure.

Stored Procedure Examples

```
iSQL> exec showReplications;
-----
Replication Info
-----
Name          IP          Port      Status
-----
REP1          192.168.1.60  25524     Running
+++++++
Local Table Name      Remote Table Name
+++++++
SYS.DEPARTMENTS      SYS.DEPARTMENTS
SYS.EMPLOYEES         SYS.EMPLOYEES
-----

EXECUTE success.
```

Example 3

Create a stored procedure called *showTables*, which outputs the names of all of a given user's tables.

```
create or replace procedure showTables(p1 in varchar(40))
as
  cursor c1 is select SYSTEM_.SYS_TABLES_.TABLE_NAME
    from SYSTEM_.SYS_TABLES_
   where SYSTEM_.SYS_TABLES_.USER_ID =
      (select SYSTEM_.SYS_USERS_.USER_ID
        from SYSTEM_.SYS_USERS_
       where SYSTEM_.SYS_USERS_.USER_NAME = upper(p1)
        AND system_.SYS_TABLES_.TABLE_TYPE = 'T');
  v1 CHAR(40);
begin
  open c1;
  SYSTEM_.PRINTLN('-----');
  SYSTEM_.PRINT(p1);
  SYSTEM_.PRINTLN(' Table');
  SYSTEM_.PRINTLN('-----');
  loop
    fetch C1 into v1;
    exit when C1%NOTFOUND;
    SYSTEM_.PRINT(' ');
    SYSTEM_.PRINTLN(v1);
  end loop;
  SYSTEM_.PRINTLN('-----');
  close c1;
end;
/
```

The following is output by the *showTables* stored procedure.

```
iSQL> exec showTables('SYS');
-----
SYS Table
-----
MANAGERS
CUSTOMERS
GOODS
EMPLOYEES
DEPARTMENTS
ORDERS
-----

EXECUTE success.
```

Example 4

Create a stored procedure called *showProcBody*, which outputs the contents of a desired stored procedure.

```
create or replace procedure showProcBody(p1 in varchar(40))
as
  cursor c1 is
    select system_.sys_proc_parse_.parse
    from system_.sys_proc_parse_
    where system_.sys_proc_parse_.proc_oid = (
      select SYSTEM_.sys_procedures_.proc_oid
      from system_.sys_procedures_
      where SYSTEM_.sys_procedures_.proc_name = upper(p1))
    order by system_.sys_proc_parse_.seq_no;
  v1 varchar(4000);
begin
  open c1;
  SYSTEM_.PRINTLN('-----');
  system_.print(p1);
  SYSTEM_.PRINTLN(' Procedure');
  SYSTEM_.PRINTLN('-----');
  SYSTEM_.PRINTLN('');
  loop
    fetch C1 into v1;
    exit when C1%NOTFOUND;
    SYSTEM_.PRINTLN(v1);
  end loop;
  close c1;
  SYSTEM_.PRINTLN('');
  SYSTEM_.PRINTLN('-----');
end;
/
```

The following is the result of querying the SYS_PROC_PARSE_ meta table, which contains the actual text of stored procedure creation statements.

```
iSQL> select system_.sys_proc_parse_.proc_oid, system_.sys_proc_parse_.parse
from system_.sys_proc_parse_
where system_.sys_proc_parse_.proc_oid = (
select SYSTEM_.sys_procedures_.proc_oid
from system_.sys_procedures_
where SYSTEM_.sys_procedures_.proc_name = upper('proc1'));
PROC_OID
-----
PARSE
-----
7695216

create or replace procedure PROC1
(P1 in NUMBER, P2 in VARCHAR(10), P3 in DATE)
as
begin
  if P1 >
7695216
  0 then
    insert into T1 values (P1, P2, P3);
    end if;
end
2 rows selected.
```

Stored Procedure Examples

The following is output by the *showProcBody* stored procedure.

```
iSQL> exec showProcBody('proc1');
-----
proc1 Procedure
-----
create or replace procedure PROC1
(P1 in NUMBER, P2 in VARCHAR(10), P3 in DATE)
as
begin
  if P1 >
0 then
    insert into T1 values (P1, P2, P3);
  end if;
end
-----
EXECUTE success.
```

Example 5

Create a stored procedure that uses a cursor variable. When this procedure is executed, a cursor variable is opened and used to read data via ODBC.

```
CREATE OR REPLACE TYPESET MY_TYPE
AS
  TYPE MY_CUR IS REF CURSOR;
END;
/
CREATE OR REPLACE PROCEDURE OPENCURSOR2
( P1 OUT MY_TYPE.MY_CUR, P2 IN INTEGER )
AS
BEGIN
  OPEN P1 FOR 'SELECT C1 FROM T1 WHERE C1 <= ?' USING P2;
END;
/
iSQL> EXEC OPENCURSOR2(4);
C1
-----
1.
2.
3.
4.
4 rows selected.

/* ODBC program */
...
SQLINTEGER c1;
SQLINTEGER param1;

/* allocate Statement handle */
if (SQL_ERROR == SQLAllocStmt(dbc, &stmt))
{
  printf("SQLAllocStmt error!!\n");
  return SQL_ERROR;
}

sprintf(query, "EXEC OPENCURSOR2(?)");

if (SQLPrepare(stmt, (SQLCHAR *) query, SQL_NTS) == SQL_ERROR)
{
  printf("ERROR: prepare stmt\n");
```

```

        execute_err(dbc, stmt, query);
        return SQL_ERROR;
    }

    if (SQLBindParameter(stmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
        SQL_INTEGER, 0, 0, &param1, 0, NULL) == SQL_ERROR)
    {
        printf("ERROR: Bind Parameter 1\n");
        execute_err(dbc, stmt, query);
        return SQL_ERROR;
    }

    param1 = 4;
    if (SQLExecute( stmt ) != SQL_SUCCESS)
    {
        execute_err(dbc, stmt, query);
        SQLFreeStmt(stmt, SQL_DROP);
        return SQL_ERROR;
    }

    if (SQL_ERROR ==
        SQLBindCol(stmt, 1, SQL_C_SLONG, &c1, 0, NULL))
    {
        printf("ERROR: Bind 1 Column\n");
    }

    while ( (rc = SQLFetch(stmt)) != SQL_NO_DATA)
    {
        if ( rc != SQL_SUCCESS )
        {
            execute_err(dbc, stmt, query);
            break;
        }
        printf(" Result Set : [ %d ] \n", c1 );
    }

    SQLFreeStmt(stmt, SQL_DROP);

    ....

$ refcursor
=====
Result Set : [ 1 ]
Result Set : [ 2 ]
Result Set : [ 3 ]
Result Set : [ 4 ]

```

File Control Example

Create a user and grant appropriate privileges to the user.

```

CONNECT SYS/MANAGER;
CREATE USER JEJEONG IDENTIFIED BY JEJEONG;
GRANT CREATE ANY DIRECTORY TO JEJEONG;
GRANT DROP ANY DIRECTORY TO JEJEONG;

```

Create a table and a directory object.

```

CONNECT JEJEONG/JEJEONG;
CREATE TABLE T1( ID INTEGER, NAME VARCHAR(40) );
INSERT INTO T1 VALUES( 1, 'JEJEONG');

```

File Control Example

```
INSERT INTO T1 VALUES( 2, 'EJPARK' );
INSERT INTO T1 VALUES( 3, 'WSKIM' );
INSERT INTO T1 VALUES( 4, 'KKSHIM' );
INSERT INTO T1 VALUES( 5, 'CSKIM' );
INSERT INTO T1 VALUES( 6, 'KDHONG' );
CREATE DIRECTORY MYDIR AS '/home/JEJEONG';
```

Create a stored procedure that reads all of the records from the table and writes them to the *t1.txt* file.

```
CREATE OR REPLACE PROCEDURE WRITE_T1
AS
    V1 FILE_TYPE;
    ID INTEGER;
    NAME VARCHAR(40);
BEGIN
    DECLARE
        CURSOR T1_CUR IS
            SELECT * FROM T1;
    BEGIN
        OPEN T1_CUR;
        V1 := FOPEN( 'MYDIR', 't1.txt', 'w' );
        LOOP
            FETCH T1_CUR INTO ID, NAME;
            EXIT WHEN T1_CUR%NOTFOUND;
            PUT_LINE( V1, 'ID : ' || ID || ' NAME : ' || NAME );
        END LOOP;
        CLOSE T1_CUR;
        FCLOSE(V1);
    END;
END;
/
```

Create a stored procedure that reads all of the records from the *t1.txt* file and outputs them to the screen.

```
CREATE OR REPLACE PROCEDURE READ_T1
AS
    BUFFER VARCHAR(200);
    V1 FILE_TYPE;
BEGIN
    V1 := FOPEN('MYDIR', 't1.txt', 'r' );
    LOOP
        GET_LINE( V1, BUFFER, 200 );
        PRINT( BUFFER );
    END LOOP;
    FCLOSE( V1 );
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        FCLOSE( V1 );
END;
/
```

Result

When the stored procedures created as described above are executed, the output is as follows:

```
iSQL> exec write_t1;
Execute success.
iSQL> exec read_t1;
ID : 1 NAME : JEJEONG
ID : 2 NAME : EJPARK
```



```
ID : 3 NAME : WSKIM
ID : 4 NAME : KKSHIM
ID : 5 NAME : CSKIM
ID : 6 NAME : KDHONG
Execute success.
```

The contents of the actual directory in the file system are as shown below:

```
$ cd /home/JEJEONG
$ cat t1.txt
ID : 1 NAME : JEJEONG
ID : 2 NAME : EJPARK
ID : 3 NAME : WSKIM
ID : 4 NAME : KKSHIM
ID : 5 NAME : CSKIM
ID : 6 NAME : KDHONG
```


Index

%ROWTYPE 34
%TYPE 34

A

ACCESS_DENIED 165
Alter function statement 24
Alter procedure statement 16
Assignment statements 42
Associative Arrays 100
 Functions 103

B

Block Body 29

C

Case Statement 57
Changing a directory object 146
CLEAR_DP 179
Continue Statement 72
Create function statement 21
Create procedure statement 10
Creating a directory object 146
Cursor Attributes 93
Cursor Management
 CLOSE 80, 90
 Cursor FOR LOOP 81, 91
 FETCH 80, 88
 OPEN 80, 85

D

Data Types 9
DataPort 169
DataPort Exceptions 181
DBMS Stat 186
Declare Section 29
Declaring a Cursor 80
 CURSOR 82
Declaring Local Variables 31
DELETE_FAILED 165
Drop function statement 25
Drop procedure statement 18
Dropping a directory object 147
dynamic SQL 122
 EXECUTE IMMEDIATE 124
 OPEN FOR 127
 Restrictions 125

E

exception declaration statement 132
Exception Handler 30
 declaring exception 131
 overview 130
 raising exception 131
Exception Handler statement 141
Execute statement 19
Exit Statement 69
EXPORT_PARTITION_TO_FILE 174
EXPORT_TO_FILE 172
EXPORT_USER_TABLES 175

F

FCLOSE 150
FCLOSE_ALL 151
FCOPY 152
FFLUSH 154
File Control-Related Exceptions 165
FILE_TYPE 147
FOPEN 155
FOR LOOP Statement 64
FREMOVE 156
FRENAME 157

G

GATHER_DATABASE_STATS 187
GATHER_INDEX_STATS 189
GATHER_SYSTEM_STATS 186
GATHER_TABLE_STATS 188
GET_LINE 159
GOTO statement 74

I

IF Statement 53
IMPORT_FROM_FILE 177
IN 11
INOUT 11
INVALID_FILEHANDLE 165
INVALID_OPERATION 165
INVALID_MODE 165
INVALID_PATH 165
IS_OPEN 160

L

LABEL statement 44
LOOP Statement 60

M

Managing directories 146

N

NEW_LINE 161

NULL Statement 77

O

OUT 11

P

parameter_declaration 10

PRINT statement 46

PUT 162

PUT_LINE 163

R

Raise Statement 133

RAISE_APPLICATION_ERROR 135

READ_ERROR 165

RECORD Types 100

REF CURSOR 110

REF CURSOR (Cursor Variable) 100

REMOVE_DP 180

REMOVE_XID 191

RENAME_FAILED 166

RESUME_DP 179

RETURN statement 49

S

SELECT INTO statement 37

SLEEP 191

SQL Statements 8

SQLCODE 139

SQLERRM 139

Structure of Stored Procedures 5

System-defined Exception Codes 137

T

Typeset 116

CREATE TYPESET 118

DROP TYPESET 120

U

User-defined Exception Codes 136

User-defined Exceptions 136

User-Defined Types

Defining 101

overview 100

User-defined Types

Compatibility 107

W

WHILE LOOP Statement 62

WRITE_ERROR 165