Forschungszentrum Karlsruhe
Technik und Umwelt

Wissenschaftliche Berichte
FZKA 5649

RODOS
Database Adapter

G. Xie, M. Rafat

Institut für Neutronenphysik und Reaktortechnik
Projekt Nukleare Sicherheitsforschung

November 1995

Forschungszentrum Karlsruhe
Technik und Umwelt
Wissenschaftliche Berichte
FZKA 5649

RODOS Database Adapter

Gang Xie*, Mamad Rafat**
Institut für Neutronenphysik und Reaktortechnik
Projekt Nukleare Sicherheitsforschung

*Institut of Nuclear Energy Technology
Tsinghua University, Peking (China)
**Fa. D.T.I. Dr. Trippe Ingenieurgesellschaft m.b.H., Karlsruhe

Forschungszentrum Karlsruhe GmbH, Karlsruhe
1995

This work has been performed with support of the European Commission, DGXI-A-1, under contract No. 94-PR-006

Forschungszentrum Karlsruhe GmbH
Postfach 3640, 76021 Karlsruhe

ISSN 0947-8620

Abstract:

Integrated data management is an essential aspect of many automatical information systems such as RODOS, a real-time on-line decision support system for nuclear emergency management.

In particular, the application software must provide access management to different commercial database systems.

This report presents the tools necessary for adapting embedded SQL-applications to both HP-ALLBASE/SQL and CA-Ingres/SQL databases.

The design of the database adapter and the concept of RODOS embedded SQL syntax are discussed by considering some of the most important features of SQL-functions and the identification of significant differences between SQL-implementations.

Finally fully part of the software developed and the administrator's and installation guides are described.

Anpassungssoftware für RODOS Datenbanken

Kurzfassung:

Integriertes Datenmangement ist ein wesentliches Merkmal automatischer Informationssysteme wie z.B. RODOS, einem Echtzeit On-line Entscheidungshilfesystem für den externen Notfallschutz nach kerntechnischen Unfällen. Insbesondere die Anwendungssoftware muß den Zugriff zu verschiedenen kommerziellen Datenbanksystemen ermöglichen.

Der vorliegende Bericht beschreibt die notwendigen Hilfsmittel zur Anpassung der SQL-Anwendungen an die Datenbanken HP-ALLBASE/SQL und CA-Ingres/SQL.

Das Design der Anpassungsprogramme und das Konzept der RODOS SQL-Syntax werden beschrieben unter Berücksichtigung der wichtigsten Eigenschaften der SQL-Funktionen und die wichtigsten Unterschiede zwischen SQL-Implementierungen werden identifiziert. Abschließend werden Teile der entwickelten Software sowie die Verwaltungs- und Installationsanleitungen ausführlich beschrieben.

Chapter 1 The Design and Development of Database Adapter

1.1 Adapter Overview

There have been some differences among the different SQL implementations. These differences cause difficulties for the application developers to make their applications transportable between different SQL implementations.

We have to face two kinds of problems. The first case is that you want to make your SQL application available for more than one SQL product. You have to develop a special version of the application for each of the considered SQL products. It will be very hard to maintenance such kind of application. In this case, the problem is how to help the application developers to work in such a way that they need to develop only one original version of the application which could be easily installed to work with any one of the current SQL implementations.

The second case is that you have to access to more than one database, which are created from different SQL products, in one SQL application. This case is likely more useful but more complicated. One of the important aspects is that the users must use an unique SQL syntax which could be dynamically translated to any one of the current SQL implementations. This problem has been solved partially in the first case. Another important aspect is that a global interface management must be designed to access the different databases.

The work related to first case has been done, during which a RODOS Embedded SQL (RE/SQL) syntax and a syntax translator have been developed. This report addresses the concepts developed to solve the problem mentioned in first case and how to use and administrate the software developed.

1.2 The Concept of RODOS-Database Adapter

The concept to solve the problem mentioned as first case is to develop a quasi standard SQL syntax. Because the quasi standard syntax could be different from the current SQL implementations in many syntax details, though these differences might be non-essential, we have to develop a translator, which could translate such a syntax to the syntax of any specified SQL implementations. The users would use this quasi standard SQL in a similar way as any one of the SQLs they used before. The only different is that the users have to use the syntax translator before they use the preprocessor of a SQL product. The Figure-1 can help to understand this concept. In this method, the major difficulty may arise from dynamic operations which include some embedded features beyond the SQL syntax, e.g. status checking, error handling, and using SQL communications Area and SQL Descriptions Area (SQLCA, SQLDA). Based on the investigation in ALLBARE/SQL and INGRES/SQL, the most important differences between them have been identified from these areas. Besides there exist differences in variables defined in SQLCA and SQLDA, the way these variables are used is also not the same. In order to solve these problems, some macros have been defined and some assistant functions have been developed, which would standardize the use of those parts of SQL features.

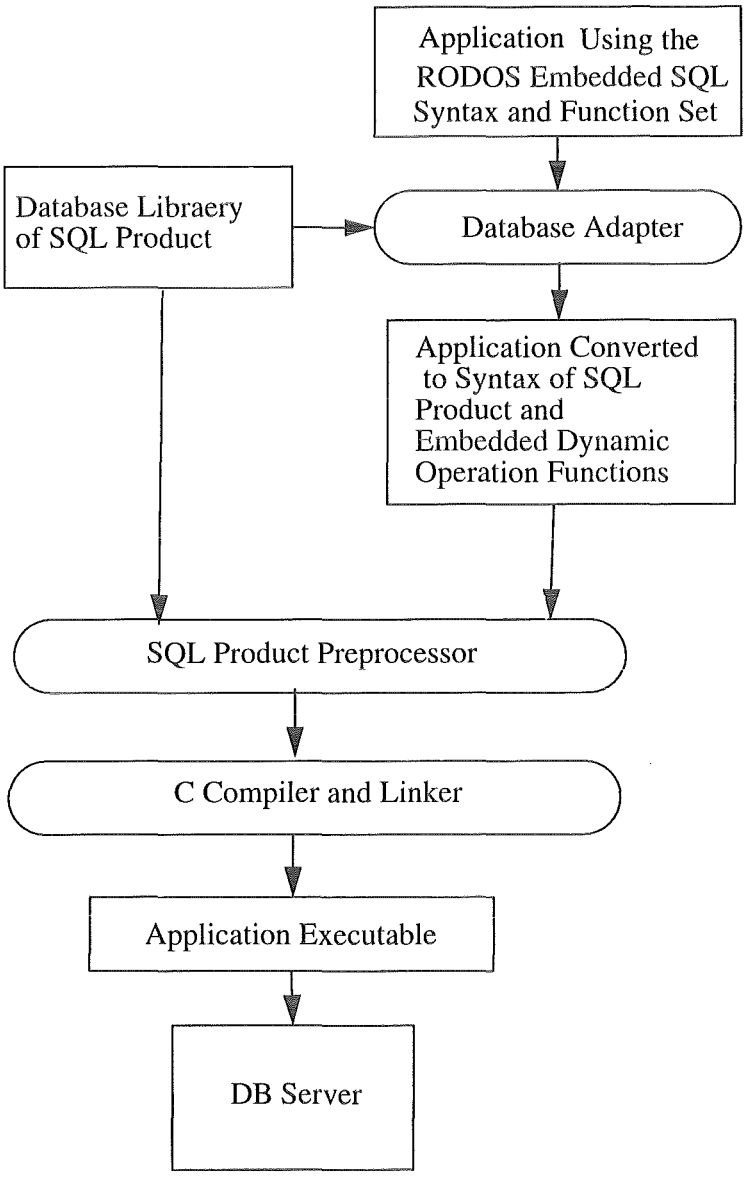


Figure-1 The Description Chart for the concept of RE/SQL

On the other hand, the translator must be able to work dynamically to translate the syntax that the users used in dynamic operations. The Database Adapter would be embedded the dynamic operation into the user's applications during the preprocessing phase.

The most important feature for the concept developed in this method is that the original SQL programming style (embedded SQL) would be maintained effectively, i.e. the users will still use a generic SQL syntax other than some kind of substitute. Particularly, the original SQL users do not have to learn too much about new concept and syntax. And the existing SQL applications could be easily re-written with this RODOS SQL because of the similarity of their syntax. The flexibility of the SQL syntax would be as more as possible remained.

The RE/SQL syntax translator could be developed with very good extendable feature. All kinds of syntaxes could be managed in a extendable file with certain pre-defined rules. In such a way, the translator could easily be extended to produce the syntax coming from any other additional RE/SQL products. On the other hand, the user could define new RE/SQL statements which can be interpreted into C and implementation's SQL statements.

The efforts to standardize the use of dynamic operations would benefit the users. It is expected that the complexity of the dynamic operations would be reduced by supplying a set of macro definitions which will be used to deal with status checking, error handling, and data processing within SQLCA and SQLDA.

The outcome generated from the database adapter will be still in the form of source code, which should be ready to be the input for the preprocessor of SQL products, so that in most of the cases the intermediate results can be checked before compiling, linking, and executing the objects. On the other hand, the syntax checking would be enhanced because of the syntax checking function of the translator.

1.3 About the Software Developed

Based on the concepts described above, a RODOS embedded SQL syntax has been developed, and a related syntax translator (`sesql`) has been programed. `sesql` is a C application which follows C standard on UNIX system. Principally, `sesql` could be run on any UNIX systems, though it has been developed on HP-UX environment.

1.4 About This Document

This Document is divided into four parts. The first and second part decribe the definition and development of the RODOS embedded SQL syntax (RE/SQL). The third part, which serves as a administrator's guide, will include more information about the software itself such as the structure of the program, and how to maintenance the program, etc.The fourth part, which serves as a installation guide, will introduce the packing of the software and provide the procedures to use the software developed.

Chapter 2 The Scope Definition

2.1 Introduction

Because of the differences among ALLBARE/SQL, INGRES/SQL and ANSI/SQL, it is impossible for the RE/SQL to include all of the functions and features which have been implemented in them. Therefore, at the first, it is necessary to define the scope of the functions and features as a goal to be achieved in this task. While doing this work, many analyses must be carried out, such as demand analysis, syntax analysis, feasibility analysis, etc. The objective is to remain as more as possible the functions and features of the current SQL implementations, and to make the product from this task practical. Here, the functions denote the SQL operations which are related to practical SQL statements. The features signify some concepts and conventions beyond the SQL operations. The RE/SQL is consisted of the functions and features. The so-called scope definition means which functions and features should be included in RE/SQL. The work will be done on the basis of the following considerations:

• The user's demands are considered as main aspect to define the scope. The functions and features which will be included in RE/SQL must satisfy the basic requirements of the users to develop their SQL applications.

• Based on the comparison analysis about current SQL implementations, which are ALLBARE/SQL and INGRES/SQL in present time, some critical differences may be identified in the aspects of concept, function and features. In these cases, some functions and features have to be excluded or simplified. These exclusions and simplifications will also be mentioned over through the report.

• For some cases, in which the differences may be eliminated by means of developing a translator and/or a function set, the original functions and features of current implementations should be remained as more as possible.

• Though the syntax analysis has been done, the syntax details will not be discussed while defining the scope. The syntax developing will be included in the next step -- Develop The Syntax For The RODOS Embedded SQL (RE/SQL).

• The contents included in the following will be classified into three main categories, which are the basic SQL element, the SQL statement and the embedded SQL programming features. Each of them is divided into some sub-categories. The basic SQL element includes embedded SQL rules, names, data types, expressions, and search conditions. The SQL statement includes connection management, transaction management, data manipulation, data definition, cursor operations, dynamic operations, and some other statements. The embedded SQL programming features includes host variable declarations, runtime status checking and SQLCA, dynamic operation and SQLDA.

2.2 Embedded SQL Rules (for C)

Statement Prefix

When you are writing an Embedded SQL application, use EXEC SQL (exec sql) as the prefix to each SQL statement. The entire prefix, EXEC SQL, must appear on one line together, though the SQL statement can be continued over more than one line. For example,

```
exec sql select * from table
        into :hostvariable
       where column1 > 500;
```

the following statement is not legal:

```
exec
sql select * from table
   where column1 > 500;
```

Statement Line Continuation

There are no line continuation rules for Embedded SQL statements in C. Statements extend to the terminator. Blank lines can be included.

Statement Terminator

When you are writing an Embedded SQL application, the use of a statement terminator is determined by the rules of the host language. For C, use a semicolon (;) at the end of the SQL command.

Comment Delimiters

To delimit comments in embedded SQL, use the following delimiters:

• The "--" (left side only) delimiter indicates that the rest of the line is a comment. You cannot continue a comment delimited by "--" over to another line. For example,

```
-- this is a comment
```

This kind of comment can be inserted in any line of an SQL statement, except the last line, by prefixing the comment character with at least one space followed by two hyphens followed by one space, for example,

```
exec sql select * from table -- you can write comment here
       where column1 > 500;
```

• Comment delimiters that are specific to the host language. For C, you can use " /* " and " */ " (left and right delimiters, respectively). When you use " /* ... */ " to delimit a comment, the comment can continue over more than one line and/or may appear within or between embedded SQL commands. For example,

```
/* this is a comment */


/* everything from here...


...to here is a comment*/


exec sql select * from table
/* comment on above line */
        into :hostvariable
/* comment on above line */
       where column1 > 500;
```

Label

For C, Embedded SQL statements can have label like C statements. The label must begin with an alphabetic character or an underscore, must be the first word on the line (optionally preceded by white space), and must be terminated with a colon (:). For example,

```
close_cursor: ECEC SQL CLOSE cursor1;
```

The label can appear anywhere a C label can appear. However, although the translator accepted the following, the compiler does not because labels are not allowed before declarations:

```
include_sqlca: EXEC SQL INCLUDE SQLCA;
```

As a general rule, use labels only with executable statements.

String Delimiters

Use single quotes to delimit Embedded SQL string literals. To embed a single quote in a string literal, you must double it. For example:

EXEC SQL INSERT INTO comments

VALUES ('single'' quote followed by double " quote');

Within Embedded SQL statements the double quote and backslash need not be escaped, because they have no special meaning.

To continue a string literal to additional lines, use the backslash (\ ) character. Any leading spaces on the next line are considered part of the string. This follows the C convention.

Use C conventions within the declaration section. You must use double quotes to delimit most C strings. For example:

char *dbname = "personnel";

Notes:

No more new limit on the rules described in this section.

2.3 Names

Basic Names

The following are classified as basic names:

Column names

Constraint names

Correlation names

Cursor names

Group names

Index names

Table names

View names

The rules for basic names:

• A basic name can be up to 20 bytes in length.

• A name can be made up of any combination of letters (A to Z), decimal digits (0 to 9), $, #, or underscore (_). However, the first character can be only an alphabetic character.

• Lowercase letters (a to z) are automatically changed to the corresponding uppercase letters (A to Z) unless enclosed in double quotation marks.

• Table name cannot begin with "ii" or "sq.". These names are reserved for use by INGRES.

Correlation Names

The correlation name specifies a synonym for the immediately preceding table or view. The correlation name can be used instead of the actual table or view name anywhere within the SELECT statement. The correlation name must conform to the syntax rules for a basic name. All correlation names within one SELECT statement must be unique. They cannot be the same as any table name or view name in the FROM clause that does not also have a correlation name associated with it.

If more than one table or view are specified in FROM clause and no correlation names are corresponding to them, the table names or view names are used as correlation names for defaults.

Correlation names are useful when you join a table to itself, example:

```
select a.empname from emp a, emp b
 where a.mgrname = b.empname and a.salary > b.salary;
```

Host Variable Names

Host variables are used to pass information between an application program and RE/SQL. They are ordinary application program variables that happen to be used in SQL commands.

A host variable name must be preceded by a colon (:) when used in an SQL command. When used elsewhere in an application program no colon should be used.

Host variable names must conform to rules for basic names; however, they are allowed to be up to 30 bytes in length. In addition, Host variable names must conform to the rules of the language in which the application program is written (here is C).

Database Names

Database names must always be enclosed in single quotation marks when specified in SQL commands. Generally, this name is used only in CONNECT command.

Notes:

1. '@' will no longer be accepted by RE/SQL in its names, though it can be accepted by both of ALLBARE/SQL and INGRES/SQL.

2. Compound identifiers, which consists of an owner name combined with one or more basic names with periods (.) between them, and which is supported by ALLBARE/SQL, is not meaningful in the RE/SQL because the concept of 'owner' is eliminated.

2.4 Data Types

There are three classed of data types: character, numeric, and date/time. Character strings can be fixed length or variable-length. Numeric strings can be exact or approximate numeric. Date/time string can be date, time, datetime, and interval. Table 1 lists all of these data types.

TABELLE 1 RE/SQL Data Types

character	char(1) - char(2000) varchar(1) - varchar(2000)	character character	A string of 1 to 2000 characters A string of 1 to 2000 characters
numeric	integer smallint float double	4 byte integer 2 byte integer 4 byte floating 8 byte floating	-2147483648 to +2147483647 -32768 to +32767 -1.0e-38 to +1.0e+38 (7 digit precision) -1.0e-38 to +1.0e+38 (16 digit precision)
date/time	date datetime interval	variable for different implementations	variable for different implemen (see following)

Char — strings can contain up to 2000 printing or non-printing characters, but without the null character (`'\0'`). Char strings are stored blank-padded to the declared length. (If the column is nullable, char columns require an additional byte of storage.) For example, if you enter `"ABC"` into a char(5) column, five bytes will be store as `"ABC  "`.

Varchar — strings are variable-length strings, which can also contain up to 2000 printing or non-printing characters, but without the null character (`'\0'`). Varchar strings are stored with a length specifier. The physical storage of specifiers may vary in different SQL implementations. Though varchar columns occupy their declared length, (If the column is nullable, char columns require an additional byte of storage), they can be dealt with by the SQL in a efficient way.

Integer — number occupies 4 bytes. An integer value is whole number in the range -2,147,483,648 to +2,147,483,467, inclusive.

Smallint — number occupies 2 bytes. An integer value is whole number in the range -32768 to +32767, inclusive.

Float	point number occupies 4 bytes, which is single precision value with 7 digit precision.
Double	point number occupies 8 bytes, which is double precision value with 16 digit precision.
Date	is string of the form {YYYY-MM-DD}, where YYYY represents the calendar year, MM is the month, and DD is the day of the month. DATE is in the range from 1582-01-01 to 2382-12-31. Note that the string of DATE are delimited by curly braces.
Datetime	is string of the form {YYYY-MM-DD HH:MI:SS}, where YYYY represents calendar MM is the month, DD is the day of the month, HH is hour, MI is the minute, and SS is the second. The range is from 1582-01-01 00:00:00 to 2382-12-31 23:59:59.
Interval	is string of the form {DDDDDDD HH:MI:SS}, where DDDDDDD is a number of days, HH is a number of hours, MI is a number of minutes, and SS is a number of seconds. The range of INTERVAL is from 0 00:00:00 to 3652436 23:59:59.

Date Operations

The date type is different from one SQL implementation to another. To solve this problem, the new formats of DATE, DATETIME, and INTERVAL data type have been defined, shown above, which are different from any one of the current SQL implementations. The constants of these date types appeared in a embedded SQL statement will be translated by the pre-processor to the format corresponding to a specified SQL product. To enable the pre-processor easily identifying these constants, curly braces instead of single quotations are used as delimiters. On the other hand, to deal with host variables which are assigned with date constants or will be processed within the user application programs, a set of conversion functions, shown in Table 2, will be developed and must be used by the user in their application programs. There are two group of conversion functions, one group is applied to input host variables before they may appear in VALUE clause in INSERT statement, SET clause in UPDATE statement, and search condition in SELECT statement (include subqueries). Another group is used to output host variables after executing of the SQL statements such as INTO clause in SELECT or FETCH statement if some date columns are involved. These conversion functions must be explicitly invoked by the developers of application programs.

TABELLE 2 Date Data Type Conversion Functions

Input Functions:[b]		
to_date(d1,d2)	YYYY-MM-DD	A: YYYY-MM-DD [c] G: dd-mmm-yyyy[d]
to_datetime(d1,d2)	YYYY-MM-DD HH:MI:SS	A: YYYY-MM-DD HH:MI:SS.FFF[e] G: dd-mmm-yyyy HH:MI:SS
to_interval(d1,d2)	DDDDDDD HH:MI:SS	A: DDDDDDD HH:MI:SS.FFF G: DDDDDDD day HH hrs MI mins SS secs
Output Functions:		
form_date(d1,d2)	A: YYYY-MM-DD G: dd-mmm-yyyy	YYYY-MM-DD
from_datetime(d1,d2)	A: YYYY-MM-DD HH:MI:SS.FFF G: dd-mmm-yyyy HH:MI:SS	YYYY-MM-DD HH:MI:SS
from_interval(d1,d2)	A: DDDDDDD HH:MI:SS.FFF G: DDDDDDD day HH hrs MI mins SS secs	DDDDDDD HH:MI:SS

a. A represents ALLBARE/SQL, G represents INGRES/SQL.
b. When d1 is string constant, it must be enclosed in double quotations.
c. YYYY is year, MM is month, DD is day.
d. dd is day, mmm is month in three-letter abbreviations, yyyy is year.
e. HH is hour, MI is minute, SS is second, FFF is thousandths of a second.

The following is an example about usage of input function:

```
/* first, use to_date function to assign
   date constant to host variable*/
to_date("1993-08-16",date_variable);
/* and then, use the host variable as an input variable */
EXEC SQL INSERT INTO test_table (date_column)
```

VALUES (:date_variable);

The following is an example about usage of output function:

```
/* first, use the host variable as an output variable */
EXEC SQL SELECT date_column
INTO :date_variable
FROM test_table
/* same time, use date constant in search condition */
WHERE date_column={1993-08-16};
/* and then, use from_date function to convert internal format
  of date data type to the format defined in RE/SQL */
from_date(date_variable, user_variable);
```

A variety of arithmetic operations could be applied to the columns which has date data type. The. following Table 3 shows the valid operations and the data type of the result.

TABELLE 3 Arithmetic Operations On Date Data Types

DATE	+	INTERVAL	DATE
INTERVAL		DATE	DATE
DATE	,	DATE	INTERVAL
INTERVAL	-	INTERVAL	INTERVAL
DATETIME		INTERVAL	DATETIME
INTERVAL	+	DATETIME	DATETIME
DATETIME	-	DATETIME	INTERVAL
INTERVAL		INTERVAL	INTERVAL
	+		
	+		
	,		
	-		
	+		
	-		
	+		
	,		
	-		

Data Type Conversions

RE/SQL converts the type of a value in the following situations:

- You include values of different types in the same expression.
- You perform a data comparison between values of different type.
- You move data from a host variable to a column or vice versa.

The so-called different type of data includes only the following combinations:

- CHAR and VARCHAR.

• DOUBLE, FLOAT, INTEGER, and SMALLINT.

• DATE

When you use numeric data of different types in an expression or comparison operation, the data type of the lesser type is converted to that of the greater type, and the result is expressed in the greater type. Numeric types have the following precedence:

DOUBLE

FLOAT

INTEGER

SMALLINT

When you compare a CHAR and a VARCHAR string, the shorter string will be padded with ASCII blank to the length of the longer string. The two strings are equal if the characters in the shorter string match those in the longer string and if the excess characters in the longer string are all blank.

Notes:

1. The data type which have been removed from ALLBARE/SQL are DECIMAL, BINARY, VARBINARY, LONGBINARY, LONG BINARY, and TIME. The maximum character length decreased from 3999 to 2000 which is maximum character length of INGRES/SQL.

2. The data type which have been removed from INGRES/SQL are MOMEY, TABLE_KEY, OBJECT_KEY.

3. Date data type is one of the biggest problem area, in which many differences have been identified between ALLBARE/SQL and INGRES/SQL. In RE/SQL, the format of the date type have been re-defined, and a group of date conversion functions would be developed with the rules to use these functions.

4. The differences in data types may affect many important statements of SQL, especially, CREATE TABLE and HOST VARIABLE DECLARATION.

2.4.1 Expressions

An expression specifies a value to be obtained in one of the following ways:

• From a column of a table.

• From a host variable in an application program.

• From a constant.

. By evaluating an aggregate function.

. By a combination of these methods with arithmetic manipulations (*, /, +, -).

Expressions are used for several purposes as follows:

• To identify columns. In the SELECT command, expressions are used in the select list to identify columns to retrieved.

• To identify rows. In the search condition of some commands, expressions help define the set of rows to be operated.

• To define a new column value.

The following rules can be applied to an expression:

• Arithmetic operators can be used between numeric values.

• Elements in an expression are evaluated in the following order:

(1) Aggregate functions and expressions in parentheses are evaluated first.

(2) Unary pluses and minuses are evaluated next.

(3) The * and / operations are performed next.

(4) The + and - operations are then performed.

• You can enclose expressions in parentheses to control the order of their evaluation.

• When two elements have the same data type, the result is of that data type.

• Type conversion, truncation, underflow, or overflow can occur when some expressions are evaluated.

Constants

IntegerValue is a signed or unsigned whole number compatible with INTEGER or SMALLINT data types, for example:

-16746, 155, 5

FloatValue is a signed or unsigned floating point number compatible with DOUBLE or FLOAT data types, for example:

.2E-4

String is a character string compatible with CHAR, VARCHAR data types. String constants are delimited by single quotation marks, for example:

'DON''T JUMP!'

However, two single quotation marks in a row are interpreted as a single quotation mark, not as string delimiters.

DateValue is a character string compatible with DATE, DATETIME or INTERVAL data types. Date constants are delimited by curly braces, for example:

{1993-08-04}

{1993-08-4 15:30:00}

{12 12:21:10}

NullValue is a special value that indicates the absence of a value. Any column in a table, regardless of its data type, can contain null values unless you specify NOT NULL for the column when you create the table. NULL is used as placeholder for a value that is missing or unknown. These properties of null values affect operations on rows containing the following values:

• Null values always sort highest in a sequence of values.

• Two null values are not equal to each other except in a GROUP BY or SELECT DISTINCT operation, or in a unique index.

• An expression containing a null value evaluates to null; for example, five minus null evaluates to null.

Because of these properties, RE/SQL ignores columns or rows containing null values in the situations listed here:

• Evaluating comparisons.

• Joining tables, if the join is on a column containing null values.

• Executing aggregate functions.

In several SQL predicates, described in the "Search Condition" section, you can explicitly test for null values. In an application program, you can use indicator variables to handle input and output null values.

SpecialValue RE/SQL supports several special key word constants. When you use any of these in a statement, RE/SQL assigns the value denoted by the constant to the appropriate variable, or column.

These constants and their meanings are listed in the Table 4. When you use these key words, you must

TABELLE 4 RE/SQL Special Constants

{now | NOW}	The current date and time.
{today | TODAY}	The current date.
{null | NULL}	The null value. See *NullValues* below.
{user | USER}	The current user.

a. These key words are reserved by RE/SQL.

include them in braces.

Aggregate Functions

Aggregate functions specify a value computed using data described in an argument. The argument, enclosed in parentheses, is an expression. The value of the expression is computed using each row that satisfies a SELECT command. Aggregate functions can be specified in the select list and the HAVING clause.

The general syntax of an aggregate function reference takes the form:

function_name([DISTINCT | ALL] columnName)

The key words, DISTINCT and ALL are optional. DISTINCT tells SQL to eliminate duplicate values from the argument before performing the function. The key word ALL indicates the default condition, in which duplicate values are not eliminate. (It makes no sense to use DISTINCT in conjunction with the functions MIN and MAX.)

These aggregate functions are described in the Table 5.

TABELLE 5 Aggregate Function Description

AVG	FLOAT, INTERVAL	computes the arithmetic mean of the values in the argument; null values are ignored.
MAX	The Same As the Argument	finds the largest of the values in the argument; null values are ignored.
MIN	The Same As the Argument	finds the smallest of the values in the argument; null values are ignored.
SUM	INTEGER, FLOAT, INTERVAL	finds the total of all values in the argument; null values are ignored.

TABELLE 5 Aggregate Function Description

COUNT (*)[a]	INTEGER	counts all rows in all columns, including rows containing null values.
COUNT Column-Name	INTEGER	counts all rows in a specific column; rows containing null values are not counted.

a. You cannot specify the COUNT (*) aggregate function with ALL and DISTINCT option.

The GROUP BY clause allows aggregate functions to be performed on subsets of the rows in the table. The subsets are defined by the GROUP BY clause. If the GROUP BY clause is omitted, the entire query result table is treated as one group. If a SELECT list or HAVING clause contains an aggregate function, then any other column references in the SELECT list or HAVING clause which do not appear in the aggregate function must be specified as an operand in a GROUP BY clause associated with the SELECT list or HAVING clause. For example,

select dept, avg(emp_age)

from employee

group by dept;

The following restrictions apply to the use of aggregate functions:

• You cannot nest aggregate functions.

• You can only use aggregate functions, or expressions that include aggregate functions, such as

sum(employee.salary)/25

in the context of a SELECT list or HAVING clause.

• If an aggregate function is computed over an empty, ungrouped table, COUNT returns 1; AVG, SUM, MAX, and MIN return NULL.

• If an aggregate function is computed over an empty group or an empty grouped table, all aggregates return no row at all.

Notes:

1. For the expression, some operations which are supported by INGRES are no longer supported in RE/SQL, such as arithmetic operations ** (exponentiation), string operations.

2. For the constants, some new rules have been developed to solve the problem of date constants, new delimiters { } have been defined for translator which can translate the date format defined here to corresponding format in specified implementation. NOW and TODAY will be replaced with current functions in ALLBARE/SQL.

3. The aggregate functions are basically the same as those of current implementations.

4. All of the other functions in both ALLBARE/SQL and INGRES/SQL are no longer supported by RE/SQL.

2.5 Search Conditions

A search condition specifies criteria for choosing rows to select, update, or delete. Syntactically, a search condition is a single predicate or several predicates connected by the logical operators AND or OR. A predicate is a comparison of expressions that evaluates to a value of TRUE or FALSE. If a predicate evaluates to TRUE for a row, the row qualifies for select, update, or delete operation. If the predicate evaluates to FALSE for row, the row is not operated on. The syntax of search condition is as follows:

[NOT] {Predicate | (searchCondition)}

[{AND | OR} [NOT] {predicate | (SearchCondition)}]

[...]

Parameters

NOT	reverses the value of the predicate that follows it.
AND	evaluates predicates it joins to TRUE if they are both TRUE.
OR	evaluates predicates it joins to TRUE if either or both are TRUE.
Predicate	is one of the predicates described later in this section.
(SearchCondition)	is one of the predicates described later in this section, enclosed in parentheses.

Description

• Predicates in search condition are evaluated as follows:

• Predicates in parentheses are evaluated first.

• NOT is applied to each predicate.

• AND is applied next, left to right.

• OR is applied last, left to right.

• When a predicate contains an expression that is null, the value of the predicate is unknown. Logical operations on such a predicate and the results are listed in the Table 6, where a question mark (?) represents the unknown value:

TABELLE 6 Logical Operations On Predicates

T	T	F	?	T	T	T	T	T	F
F	F	F	F	F	T	F	?	F	T
?	?	F	?	?	T	?	?	?	?

When the search condition for a row evaluates to unknown, the row does not satisfy the search condition and the row is not operated on.

- You can compare only compatible data types.

- A SubQuery expression cannot appear on the left-hand side of a predicate.

BETWEEN Predicate

A BETWEEN predicate determines whether a value is equal to or greater than a second value and equal to or less than a third value. The predicate evaluates to TRUE if a value falls within the specified range. If the NOT option is used, the predicate evaluates to TRUE if a value does not fall within the specified range.

Note that the second value must be less than or equal to the third value. The syntax of the BETWEEN predicate is as follows:

Expression1 [NOT] BETWEEN Expression2 AND Expression3

Parameters

Expression1, 2, 3	specify values used to identify columns, screen rows, or define new column values. The syntax for expression is defined in the "Expressions" section. Both numeric and non-numeric expressions are allowed in BETWEEN predicates.
NOT, AND	are logical operators. NOT reverses the value of the predicate that follows it. AND evaluates predicates it joins to TRUE if they are both TRUE.

Description

- Expression2 and Expression3 constitute a range of possible values for which Expression2 is the lowest possible value and Expression3 is the highest possible value. In the BETWEEN predicate, the low value must come before the high value. Also in the BETWEEN predicate, subqueries are not allowed.

- Comparisons are conducted as described under "Comparison Predicates" later in this section.

Comparison Predicate

A comparison predicate compares two expressions using a comparison operator. The predicate evaluates to TRUE if the first expression is related to the second expression as specified in the comparison operator.

Expression {= | <> | > | >= | < | <=} {Expression | SubQuery}

Parameters

Expression	specify a value used to identify columns, screen rows, or define new column values. The syntax for expression is defined in the "Expressions" section. Both numeric and non-numeric expressions are allowed in comparison predicates.
SubQuery	is a QueryExpression whose result is used in evaluating another query. The syntax of QueryExpression is presented in the description of the SELECT command.
=	is equal to. A comparison predicate using = is also known as an EQUAL predicate.
< >	is not equal to.
>	is greater than.
> =	is greater than or equal to.
<	is less than.
< =	is less than or equal to.

Description

• If either expression is null or if both expressions are null, the predicate evaluates to FALSE.

• Refer to the "Data Types" section for type conversion when you compare values of different types.

• A subquery must return a single value (one column of one row). If the subquery returns more than one value, an error is given. If the subquery returns no rows, the predicate evaluates to unknown.

EXISTS Predicate

An EXISTS predicate tests for the existence of a row satisfying the search condition of a subquery. The predicate evaluates to TRUE if at least one row satisfies the search condition of the subquery.

[NOT] EXISTS SubQuery

Parameters

SubQuery — A subquery is a nested query. The syntax of subqueries is presented in the description of the SELECT command.

Description

• Unlike other places in which subqueries occur, the EXISTS predicate allows the subquery to specify more than one column in its select list.

IN Predicate

An IN predicate compares an expression with a list of specified values or a list of values derived from a subquery. The predicate evaluates to TRUE if the expression is equal to one of the values in the list. If the NOT option is used, the predicate evaluates to TRUE if the experssion is not equal to any of the values in the list.

Expression [NOT] IN {SubQuery | (ValueList)}

Parameters

Expression — specify a value used to be obtained. The syntax for expression is defined in the "Expressions" section. Both numeric and non- numeric expressions are allowed in quantified predicates.The expression may not include SubQueries.

NOT — reverses the value of the predicate that follows it.

SubQuery — A subquery is a nested query. The syntax of subqueries is presented in the description of the SELECT command.

ValueList — defines a list of values to be compared against the expression's value. The values in a ValueList can be the constants which are described in "Constants" section and host variables. The syntax for the host variables is as follows:

:HostVariable [:Indicator]

Description

• You can use a list of host variables as the ValueList. If the null indicator for a host variable is less than zero (indicating a NULL value in the corresponding host variable), the value in the host variable is not considered a part of the ValueList.

• Refer to the "Data Types" section for type conversion when you compare values of different types.

LIKE Predicate

A LIKE predicate determines whether an expression contains a given pattern. The predicate evaluates to TRUE if an expression contains the pattern. If the NOT option is used, the predicate evaluates to TRUE if the expression does not contain the pattern.

ColumnName [NOT] LIKE 'Pattern' [ESCAPE 'EscapeChar']

Parameters

ColumnName specify a column.

NOT reverses the value of the predicate that follows it.

Pattern describes what you are searching for in the expression. The pattern can consist of characters only (including digits). Uppercase and lowercase are significant.

You can also use the predicate to test for the existence of a partial match, by using the following symbols in the pattern:

_ represents any single character.

% represents any string of zero or more characters.

The _ and % symbols can be used multiple times and in any combination in a pattern. You cannot use these symbols literally within a pattern unless the ESCAPE clause appears, and the escape character precedes them. Not that they should be ASCII and not your local representations.

EscapeChar describes an optional escape character which can be used to include the symbols _ and % in the pattern.

The escape character must be a single character. When it appears in the pattern, it must be followed by the escaped character, or _ or %. Each such pair represents a single literal occurrence of the second character in the pattern. When the escape character is defined following the word ESCAPE, it must be delimited by single quotes. All other characters are interpreted as described before.

Description

• If an escape character is not specified, then the _ or % in the pattern continues to act as a wildcard. No default escape character is available. If an escape character is specified, then the wildcard or escape character which follows and escape character is treated as a literal. If the character following an escape character is not a wildcard or the escape character, an error results and the command in which it is contained does not process or return any rows.

• You cannot specify Pattern and EscapeChar with host variables (because this feature is not supported by INGRES/SQL).

Quantified Predicate

A quantified predicate compare an expression with a list of values derived from a subquery. the predicate evaluates to TRUE if the expression is related to the values list as specified by the comparison operator and the quantifier.

Expression {= | <> | > | >= | < | <=} {ALL | ANY | SOME} SubQuery

Parameters

Expression	specify a value used to identify columns, screen rows, or define new column values. The syntax for expression is defined in the "Expressions" section. Both numeric and non-numeric expressions are allowed in comparison predicates.
=	is equal to.
< >	is not equal to.
>	is greater than.
> =	is greater than or equal to.
<	is less than.
< =	is less than or equal to.

ALL, ANY, SOME — are quantifiers which indicate how many of the values from the SubQuery must relate to the expression as indicated by the comparison operator in order for the predicate to be true. Each quantifier is explained below:

ALL the predicate is TURE if ALL the values returned by SubQuery relate to the expression as indicated by the comparison operator.

ANY the predicate is TRUE if ANY of the values returned by the SubQuery relate to the expression as indicated by the comparison operator.

SOME a synonym for ANY.

SubQuery — is a QueryExpression whose result is used in evaluating another query. The syntax of QueryExpression is presented in the description of the SELECT command.

Description

- Refer to the "Data Types" section for type conversion when you compare values of different types.
- The ValueList is no longer supported by RE/SQL (because it is not supported by INGRES/SQL)

NULL Predicate

A NULL predicate determines whether a primary has the value NULL. The predicate evaluates to TRUE if the primary is NULL. If the NOT option is used, the predicate evaluates to TRUE if the primary is not NULL.

{ColumnName | Constant | AggregateFun | (Exp.)} IS [NOT] NULL

Parameters

ColumnName	is the name of a column from which a value is to be taken.
Constant	is a specific value; constants are defined in the "Constants" section.
AggregateFun	is a computed value; aggregate functions are defined in the "Aggregate Functions" section.
(Exp.)	is one or more of the above four primaries, enclosed in parentheses.
NOT	reverses the value of the predicate that follows it.

Description

- The primary may be of any data type.

- Because you cannot test for NULL by using the comparison operator "=", you must use this predicate to find out whether an expression is NULL

Notes:

1. The predicates are basically the same as those of current implementations.

2.6 The RE/SQL Statements

Although a brief syntax analysis has been done, the syntax details will not be discussed in this section. The detailed syntax developing will be performed in the next step -- Develop The Syntax For The RODOS Embedded SQL (RE/SQL). In this section, only the functionality will be defined.

2.6.1 Connection Management

Establish A Connection

CONNECT

Terminate A Connection

DISCONNECT

Notes:

1. INGRES/SQL can support multi-session, but ALLBARE/SQL cannot. Therefore, the related session management options and statements will be removed from RE/SQL.

2. When you terminate a connection, the strategy to commit (or rollback) the current transaction may differ in different SQL implementations.

3. Both of the logical and physical structures may be different among the current implementations. Therefore, the options related to these structures may be considered to use a WITH clause.

2.6 Transaction Management

Initiate A Transaction

BEGIN WORK

Terminate A Transaction

COMMIT [WORK]

Set A Rollback Stop

SAVEPOINT

Abort A Transaction

ROLLBACK [WORK]

Notes:

1. Some issues, such as concurrency control, locking, etc., are concerned with transaction management in some SQL implementations. A big difference has been identified in this area. A more detailed analysis is necessary.

2. Though both of ALLBARE/SQL and INGRES/SQL have supported the SAVEPOINT statement, the syntax they used are quite different. In ALLBARE/SQL, the SAVEPOINT statement must use a host variable as the parameter. And the INGRES/SQL does not allow using a host variable as parameter. Despite it is not easy, the pre-processor would solve this problem, see Syntax Development for details.

2.7 Data Definition

Define A Table

CREATE TABLE

Remove A Table

DROP TABLE

Define A Index

CREATE INDEX

Remove A Index

DROP INDEX

Define A View

CREATE VIEW

Remove A View

DROP VIEW

Notes:

1. Both of the logical and physical structures may be different among the current implementations. Therefore, some related options and statements will be removed from RE/SQL. And some others may be considered to use a WITH clause.

2. The differences of data types have been existed between ALLBARE/SQL and INGRES/SQL. Some data types will no longer be supported by RE/SQL. And some others may translated by pre-processor.

3. When user create a table, the storage structure of the table will be determined. This may effect the data retrieval later on. A more detailed analysis is needed for this issue.

2.6.4 Data Manipulation

Input Data

[BULK] INSERT

Remove Data

DELETE

Retrieve Data

[BULK] SELECT

Modify Data

UPDATE

Notes:

1. Data manipulation is the most compatible part in the current implementations. No important differences have been identified.

2. BULK option in INSERT and SELECT statements were original supported only by ALLBARE/SQL. Because of existing similar functions in INGRES/SQL, this function, which is very useful, will be remained by means of pre-processor. See Syntax Development for details.

3. Though cursor operations and dynamic operations are also some kind of data manipulations, they will be discussed in the following separate sections.

2.6.5 Cursor Operations

Issue A Cursor

DECLARE CURSOR

Active A Cursor

OPEN

Retrieve Data With A Cursor

[BULK] FETCH

Modify Data With A Cursor

UPDATE WHERE CURRENT

Remove Data With A Cursor

DELETE WHERE CURRENT

Terminate A Cursor

CLOSE

Notes:

1. Cursor Operation is another most compatible part in the current implementations. No important differences have been identified.

2. BULK option in FETCH statements were original supported only by ALLBARE/SQL. Because of existing similar functions in INGRES, this function, which is very useful, will be remained by means of pre-processor.

2.8 Dynamic Operations

Declare SQLDA Elements (newly defined in RE/SQL)

DECLARE SQLDA

Dynamically Allocate A Format Array Or Data Buffer (newly defined in RE/SQL)

ALLOCATE {SQLDA_SQLFMTARR | SQLDA_DATABUFF}

Free A Dynamically Allocated Format Array Or Data Buffer (newly defined in RE/SQL)

FREE {SQLDA_SQLFMTARR | SQLDA_DATABUFF}

One Time Execute

EXECUTE IMMEDIATE

More Time Execute

PREPARE

and

EXECUTE

Deal With Dynamic Query

DESCRIBE

and

FETCH (Dynamic Version)

Notes:

1. No important differences have been identified, though INGRES/SQL has more optional functions in the dynamic operation statements.

2. To deal with dynamic queries, the SQL Descriptor Area (SQLDA) has to be used. The structures of SQLDA for INGRES/SQL and ALLBARE/SQL have been defined in similar way but different in details. To solve this problem, we will define a set of macros (in C language) to standardize data accessing to SQLDA. See next section for details.

2.9 Other Statements

Include An External File

INCLUDE

Begin A Host Variable Declare Section

BEGIN DECLARE SECTION

Terminate A Host Variable Declare Section

END DECLARE SECTION

Get Message From SQL

INQUIRE_SQL

Error Handling

WHENEVER

Define A New Authorization Group

CREATE GROUP

Add Or Drop User To/From A Group

ALTER GROUP

Grant Authority

GRANT

Revoke Authority

REVOKE

Set Lock Mode

SET LOCKMODE

Notes:

1. Some of the statements in this section, such as statements to control authority, concern with concepts used to control data accessing in current implementations, which are the most difficult area to make these functions available in RE/SQL. A more detailed analysis is needed.

2. Locking managements are also quite different in ALLBARE/SQL and INGRES/SQL. In ALLBARE/SQL, the most of the locking could be controlled when you issue some statements. There is only one special statement for locking, that is LOCK TABLE. In INGRES/SQL, the most of the locking could be controlled by a special statement, SET LOCKMODE. To unify these locking control would be difficult, and needs a more detailed analysis.

3. These two features mentioned above are critical to enable RE/SQL supporting multiple user working environment.

2.10 The RE/SQL Embedded Programming Features

2.10.1 Host Variable Declaration

2.10.1.1 Variable Usage

Host variables are variables used to pass the following information between an application program and RE/SQL:

- data values
- null value indicators
- string truncation indicators
- bulk processing rows to process
- dynamic commands
- messages from the RE/SQL
- database names

It is worth to mention that the host variables can not be used as table name, view name, index name, and column name.

2.10.1.2 Declaring Host Variables

All host variables used in a C program must be declared in declaration parts of the program where variables having the scope of a file, a function, or a block and be declared. Host variables may be declared wherever you can declare variables in C program.

At run time, the scope of a host variable is the same as that of any other C variable declared in the same declaration part. At preprocessing time, however, all host variable declarations are treated as global declarations. Therefore host variables having the same name in different declaration parts must also have the same C type description in each variable declaration.

Host Variable Declaration Section

Host variables must be declared in what is known as a declare section. A declare section consists of the SQL command BEGIN DECLARE SECTION, one or more variable declarations, and the SQL command END DECLARE SECTION, as shown in the following sections.

More than one declare section may appear in a given declaration part. However, a host variable name may appear only once in a given declaration part. Each host variable is declared by using a C type declaration. The declaration contains the same components as any C variable declaration. The data name, which must conform to the rules in "Host Variable Name" in section 2.2, must be the same as the host variable name in the corresponding SQL statement. The data type must satisfy SQL data type and C requirements, which will be described in the following sections.

Indicator Variable

A special type of host variable called an indicator variable, is used in SELECT, FETCH, UPDATE, UPDATE WHERE CURRENT, and INSERT commands to identify null values and in SELECT and FETCH commands to identify truncated output strings. Indicator variable is a two byte integer variable. The declarations of indicator variable are described within the following two sections.

An indictor variable must appear in an SQL command immediately after the host variable whose data it describes. The host variable and its associated indicator variable are not separated by a comma. In SELECT and FETCH commands, an indicator variable is an output host variable containing one of the following indicators, which describe data SQL returned:

0	value is not null
-1	value is null
>0	string value is truncated; number indicates data length before truncation.

In the INSERT, UPDATE, and UPDATE WHERE CURRENT commands, an indicator variable is an input host variable. The value you put in the indicator variable tells SQL when to inset a null value in a column:

>=0	value is not null
<0	value is null

Be sure to use an indicator variable in the SELECT and FETCH commands whenever column accessed may contain null values. A runtime error results if SQL retrieves a null value and the program contains no indicator variable.

Simple Host Variable

The so-called "simple host variables" is C variables which appeared as non-structured variables. Simple host variables can be used for input or for output. The input variables provide data for SQL, and the output

variables receive data from SQL. Be sure to declare the host variables before using them. The declaration section for simple host variables is shown as following:

```
EXEC SQL BEGIN DECLARE SECTION;

short        smallint_variable_name;

|            |

|            |

data type      data_name;

|            |

|            |

short        indicator_variable_name;

EXEC SQL END DECLARE SECTION;
```

The variables, which include indicator variables, can be declared in any order. However, when you use these variables, you must carefully put them in right positions which must exactly correspond to the SQL objects both the order and the data type. The compatible C data type which can be used to declare host variables is described in the following sections.

The following is an example about the usage of simple host variables:

```
EXEC SQL BEGIN DECLARE SECTION;

short        input_varible_name;

char         output_variale_name[30];

short        indicator_variable_name;

char          conditon_variable_name[100];

EXEC SQL END DECLARE SECTION;

 .

 .

/* example of input variable */
```

```
EXEC SQL INSERT INTO test_table (smallint_column_name)

VALUES (:input_variable_name);


/* example of output variable and indicator variable */

EXEC SQL SELECT char_column_name

INTO :output_variable_name :indicator_variable_name

FROM test_table

/* condition_variable_name is an input variable */

WHERE char_column_name=:condition_variable_name;
```

Structure Host Variable

Structure host variables are BULK processing variables which can be used with the BULK option of the SELECT, INSERT, and FETCH commands. BULK table processing is the programming technique you use to SELECT, FETCH, or INSERT multiple rows at a time. Referring to concerned section for the detailed descriptions about BULK processing. In BULK processing, rows are retrieved into or inserted from host variables declared as an array of records. The following is a generic form of the declaration section:

```
EXEC SQL BEGIN DECLARE SECTION;

.

  .

struct StructName {

  data type     ColumnName1;

  data type     ColumnName2;

  sqlind        Col2IndVar;

    .

    .

  data type     ColumnNameN;

  sqlind        ColNIndVar;
```

```
} ArrayName[n];

short StartIndex;

short NumberOfRows;

.

.

EXEC SQL END DECLARE SECTION;
```

When you declare a structure array, note that the order of the select list items for SELECT or FETCH statements, or column name list for INSERT statement, must match the order of the corresponding host variable elements in the structure of the array. Any column that may contain a null value must have an indicator variable immediately following the declaration for the column in array.

When you refer a structure array, note that you must refer the array as a whole. Referring to any single record in array or any single element in the structure is not allowed in the SQL statement. However, you can refer the records or structure elements anywhere in you host program except in any SQL statement.

Two additional host variables may be specified in conjunction with the structure array:

• A *StartIndex* variable: a short integer variable that specifies an array subscript. The subscript identifies where in the array SQL should store the first row in a group of rows retrieved. In the case of an INSERT operation, the subscript identifies where in the array the first row to be inserted is stored. If not specified, the assumed subscript is zero.

• A *NumberOfRows* variable: a short integer variable that indicates to SQL how many rows to transfer into or take from the array, starting at the array record designated by StartIndex. If not specified, the default number of rows is the number of records in the array from the StartIndex to end of the array for an INSERT operation. For a retrieval operation, the default number of rows is the smaller of two values; 1) the number of records in the array from the StartIndex to the end of the array, or 2) the number of rows in the query result. NumberOfRows can be specified only if you specify the StartIndex variable.

The following provides a example of using structure variables:

```
EXEC SQL BEGIN DECLARE SECTION;

struct StructName {

  sqlind     ColumnName1;

  char       ColumnName2[10];

  sqlind     Col2IndVar;
```

```
  double     ColumnName3;
  sqlind     Col3IndVar;
} ArrayName[10];
short StartIndex;
short NumberOfRows;
char  SearchCondition[10];
EXEC SQL END DECLARE SECTION;
.
.
EXEC SQL BULK INSERT INTO example_table
        (ColumnName1,ColumnName2,ColumnName3)
        VALUES (:ArrayName, :StartIndex, :NumberOfRows);
.
.
EXEC SQL BULK SELECT ColumnName1, ColumnName2, ColumnName3
          INTO :ArrayName, :StartIndex, :NumberOfRows
          FROM example_talbe
          WHERE ColumnNmae2 = :SerachCondition;
```

2.10.1.3 Declaring Variable for Data Types

Table 7 summarizes C data declarations for host variables of each RE/SQL data type. Only the type descriptions shown in Table 7 are supported by the RE/SQL translator. Note in particular that the translator does not support user-defined data types.

TABELLE 7 Data Type Declarations

CHAR(n)	char dataname[n+1];
VARCHAR(n)	char dataname[n+1];
SMALLINT	short dataname;
INTERGER	long dataname;
FLAOT	float dataname;
DOUBLE	double dataname;
DATE	char dataname[26];
DATETIME	char dataname[26];
INTERVAL	char dataname[26];

For C strings (character arrays), C has the convention of using an ASCII 0 ('\0'), the null character, to mark the end of the string. Therefore, char host variables declared in C, for CHAR, VARCHAR, DATE, DATETIME, and INTERVAL, must have a size one greater than their column definition, to allow for the null character at the end of the string.

When SQL assigns CHAR data to char array host variable, the total length of the SQL CHAR field is stored in the host variable including any trailing blanks on the right of the data string. An ASCII 0 is then added after the last byte of the string.

For the string of VARCHAR, DATE, DATETIME, and INTERVAL, no trailing blanks are added at the end of string. An ASCII 0 is placed after the last character of the C string based on the specified length of the string.

Because the C string is terminated by ASCII 0, the string could be declared longer than what it needs. Though it is not obviously meaningful to do this, no negative affection would be generated. This feature is helpful to solve the problem which is caused by the difference existed between the current SQL implementations in their length of date data type.

Variable Compatibility

Under the following conditions, SQL performs data type conversion when executing SQL commands containing host variables:

- when the data types of values transferred between your program and database do not match
- when data of one type is moved to a host variable of a different type
- when values of different types appear in the same expression

Data types for which type conversion can be performed are called compatible data types. Table 8 summarizes data type-host variable compatibility. It also points out which data type combinations are incompatible and which data type combinations are equivalent, i.e. require no type conversion. E describes an equivalent situation, C a compatible situation, and I an incompatible situation.

TABELLE 8 C Data Type Equivalency and Compatibility

CHAR	C	I	I	I	I
VARCHAR	C	I	I	I	I
SMALLINT	I	E	C	C	C
INTEGER	I	C	E	C	C
FLOAT	I	C	C	E	C
DOUBLE	I	C	C	C	E

String Data Conversion

When SQL stores the characters in a C string into a CHAR column, the final ASCII 0 is removed and the any remaining position to the right are padded with spaces. When SQL stores the characters in a C string to VARCHAR column, it only stores the string up to but not including the ASCII 0. The length of the string is stored in a header in the front of each VARCHAR data type.

String Data Truncation

If the target host variable used in a SELECT or FETCH operations is too small to hold an entire string, the string is truncated. You can use an indicator variable to determine the actual length of the string in bytes before truncation. SQL puts the actual length of the string n bytes into indicator variable if there is an indicator variable is associated. If a column is too small to hold a string in an INSERT or an UPDATE operation, the string is truncated and stored, but SQL gives no error or warning indication.

Numeric Data Conversion

When you use numeric data of different types in an expression or comparison operation, data of the lesser type is converted into data of the greater type, and the results is expressed in the greater type. SQL numeric types have the following precedence, from highest to lowest:

1. DOUBLE

2. FLOAT

3. INTEGER

4. SMALLINT

Notes:

1. In ALLBARE/SQL, the host variable must be used in a SAVEPOINT statement to refer to a rollback stop. But in INGRES/SQL, it is not allowed to use the host variable in a SAVEPOINT statement. This is a difficult case. To keep SAVEPOINT statement in RE/SQL, an effort must be made to develop some new rules to invoke SAVEPOINT statement, which can be dealt with by the RE/SQL translator. However, the host variables can no longer be directly used in this case by the users. For more details, see SAVEPOINT statement in Develop The Syntax For The RE/SQL.

2. Originally, the BULK processing has been supported only by ALLBARE/SQL. However, there has been some other features in INGRES/SQL which can be used to realize similar function of the BULK processing. It is expected that based on the syntax for BULK statements the equivalent statements could be generated by RE/SQL translator. For more details, refer to BULK statements in Develop The Syntax For The RE/SQL.

3. There is a big difference between the ways you declare structure host variables for BULK processing in ALLBARE/SQL and INGRES/SQL if some indicator variables must be used in the structure. In ALLBARE/SQL, the indicator variables must be declared immediately following the associated structure elements. And in the INGRES/SQL, the indicator variables must be declared as a separated array. This is a very complicated case if we use RE/SQL translator, this might be possible, to automatically make proper changes in the user's application program. Another conservative option is leave this work done be the users themselves, i.e. users write related program segments with both ALLBARE/SQL and INGRES/SQL rules with the compiler conditional statements #if, #else, and #endif.

4. It is more flexible for INGRES/SQL to use structure variables. For example, in INGRES/SQL, the user can refer to single array record or single structure element, and user defined data type is supported. These features are no longer supported by RE/SQL

2.7.2 Runtime Status Checking and the SQLCA

When an SQL command is executed, SQL returns information describing how the command executed. This information signals one or more of the following status condition:

- The command was successfully executed.

- The command could not be executed because an error condition occurred, but the current transaction will continue.

- No rows qualified for a data manipulation operation.

- A specific number of rows were placed into output host variables.

- A specific number of rows qualified for an INSERT, UPDATE, or DELETE operation.

- The command was executed, but a warning condition resulted.

- The command could not be executed because the number of variables in a SELECT or FETCH statement is unequal to the number of columns in the table being operated on. This applies to dynamic precessing only.

- The command could not be executed because an error condition necessitated rolling back the current transaction.

This information is sent to SQL Communications Area (SQLCA) by the SQL after an SQL statement was executed. Based on this information, a program can COMMIT WORK, ROLLBACK WORK, continue, terminate, display a message, or perform error handling. You can use SQLCA in different way:

- You can use the WHENEVER command to perform implicit status checking. This means that SQL checks the variables defined in SQLCA for you, then takes an action based on information you provide in the WHENEVER command.

- You can write C code that explicitly examines one or more of the seven SQLCA elements, the proceeds on the basis of their values. This kind of status checking is called explicit status checking. However, SQLCA has been defined with some differences in the current SQL implementations so that the SQLCA elements can no longer be used directly by the users. The RE/SQL will provide a group of functions to help user accessing to SQLCA. More information about SQLCA will described in the following sections.

- You can use a combination of both implicit and explicit status checking.

In conjunction with status checking of any kind, you can use the INQUIRE_SQL command which retrieve message from the SQL message catalog that describes an error or waring condition.

The SQL Communication Area (SQLCA)

Every SQL C application program must have the SQL Communication Area (SQLCA) declared in the global declaration part. You can use the INCLUDE command to declare the SQLCA:

EXEC SQL INCLUDE SQLCA;

When the translator (from SQL product) parses this command, it insert necessary type definition used in SQLCA into the modified source file. The structures defined for SQLCA may differ from one SQL implementation to another, though these differences are non-essential. The SQLCA structures defined in ALLBARE/SQL and INGRES/SQL are provided in the appendix, a comparison table is also given. Some of the elements in SQLCA C structure are used by SQL itself, here we discuss only elements used by the users. The element names in the structure may be different so that these element names can no longer be referred directly by the users in their application programs. To solve this problem, we use macros instead of element names in the user's application programs. The macros are pre-defined for each current SQL implementation and will be automatically inserted into the program by the RE/SQL translator during the preprocessing phase for a specified SQL implementation. The Table 9 shows the macro definitions for the ALLBARE/ SQL and for the INGRES/SQL. With using these variables, users can perform explicit status checking and error handling, which will be described in the following sections.

TABELLE 9 Macro Definitions For SQLCA[a]

#define SQLCA_SQLCODE	sqlca.sqlcode	sqlca.sqlcode
#define SQLCA_SQLERRD2	sqlca.sqlerrd[2]	sqlca.sqlerrd[2]
#define SQLCA_SQLWARN0	sqlca.sqlwarn[0]	sqlca.sqlwarn0
#define SQLCA_SQLWARN1	sqlca.sqlwarn[1]	sqlca.sqlwarn1
#define SQLCA_SQLWARN2	sqlca.sqlwarn[2]	sqlca.sqlwarn2
#define SQLCA_SQLWARN3	sqlca.sqlwarn[3]	sqlca.sqlwarn3
#define SQLCA_SQLWARN6	sqlca.sqlwarn[6]	sqlca.sqlwarn6

a. Only the interested variables are listed in this table.

SQLCA_SQLCODE is a integer variable indicating the SQL return code. Its value falls into one of three categories:

= 0 The statement executed successfully (though there may have warning messages see SQLCA_SQLWARN0).

< 0 An error occurred. The value of SQLCA_SQLCODE is the negative value of the error number. A negative value sets the SQLERROR condition of the WHENEVER statement.

100 When no rows qualify for one of the following commands, but no error condition exists: SELECT, FETCH, INSERT, UPDATE, DELETE, UPDATE WHILE CURRENT, DELETE WHILE CURRENT. This value sets NOT FOUND condition of the WHENEVER command.

SQLCA_SQLERRD2 can contain one of the following values:

= 0	This value does not make sense because it represents different meaning in different SQL implementations.
> 0	The number of rows processed in the following data manipulation commands: SELECT, FETCH, INSERT, UPDATE, DELETE, UPDATE WHILE CURRENT, UPDATE WHILE CURRENT.
SQLCA_SQLWARN0	If set to "W", at least one other SQLCA_SQLWARN* contains a "W". When "W" is set, the SQLWARNING condition of the WHENEVER statement is set.
SQLCA_SQLWARN1	Set to "W" on truncation of a character string assignment from the database into a host variable. If an indicator variable is associated with the host variable, the indicator variable is set to the original length of the character string.
SQLCA_SQLWARN2	Set to "W" on elimination of nulls from aggregates.
SQLCA_SQLWARN3	Set to "W" when mismatching number of result columns and result host variables in FETCH or SELECT statement (includes dynamic operations).
SQLCA_SQLWARN6	Set to "W" when the error returned in SQLCA_SQLCODE caused the abnormal termination of an open transaction (transaction rolled back).

Explicit Status Checking Techniques

Explicit status checking is useful when you want to test for specific SQLCA values before passing control to one of several locations in your program. Error and warning conditions detected by status checking can be conveyed to the program user in various ways:

• INQUIRE_SQL can be used one or more times after an SQL command is processed to retrieve warning and error messages from the SQL message catalog. (The SQL message catalog contains messages for every negative SQLCA_SQLCODE and for every condition that sets SQLCA_SQLWARN0).

• Your own messages can be displayed when a certain condition occurs.

• You can choose not to display a message; for example, if a condition exists that is irrelevant to the program user or when an error is handled internally by the program.

A typical form of using explicit status checking is shown in the following example:

```
if (SQLCA_SQLCODE == 0)  /* no error, but may have warning */
{
 if (SQLCA_SQLWARN0 == 'W') /* warning occurs */
 {
  /* you may display warning message with using INQUIRE_SQL */
```

```
    EXEC SQL INQUIRE_SQL :SQLMessage;
    printf("%s\n",SQLMessage);
    /* you may deal with warning according to SQLCA_SQLWARN? */
    if (SQLCA_SQLWARN1 == 'W') /* string truncation occurs */
    {
     /* deal with this warning */
     ...
    }
    /* deal with other warning which may occurs */
    ...
  }
}
else if (SQLCA_SQLCODE == 100) /* no row qualified */
{
 printf("No rows qualified for this operation!\n");
}
else if (SQLCA_SQLCODE < 0) /* error condition occurs */
{
 /* you may display error message with using INQUIRE_SQL */
 EXEC SQL INQUIRE_SQL :SQLMessage;
 printf("%s\n",SQLMessage);
 /* you may take other actions to respond error condition */
 ...
}
```

This example just provides an idea to perform status checking. Based on the description about SQLCA elements, you can use them in any way you want.

Implicit Error Handling Techniques

Implicit error handling is useful when control to handle warnings and errors can be passed to one predefined point in a WHENEVER command. The WHENEVER command has two components: a condition and an action. The command format is:

EXEC SQL WHENEVER Condition Action;

There are three possible WHENEVER conditions:

• SQLERROR

If WHENEVER SQLERROR is in effect, SQL checks for a negative SQLCA_SQLCODE after

• SQLWARNING

If WHENEVER SQLWARNING is in effect, SQL checks for a "W" in SQLCA_SQLWARN0 after processing an SQL command.

• NOT FOUND

If WHENEVER NOT FOUND is in effect, SQL checks for a 100 in SQLCA_SQLCODE after processing an SELECT or FETCH command.

A WHENEVER command for each of these conditions can be in effect at the same time.

There three possible WHENEVER actions:

• STOP

If WHENEVER Condition STOP is in effect, SQL rolls back the current transaction and terminates the database session and the program when the condition exists.

• CONTINUE

If WHENEVER Condition CONTINUE is in effect, program execution continues when the Condition exists. Any earlier WHENEVER command for the same condition is cancelled.

• GOTO LineLabel

If WHENEVER Condition GOTO LineLabel is in effect, the code routine located at that alpha-numeric line label is executed when the Condition exists. The line label must appear in the function where the GOTO is executes. GOTO and GO TO forms of this action have exactly the same effect.

Any action may be specified for any condition.

Notes:

1. Although the structures defined in the current SQL implementations are different, the structure elements have similar meanings. By means of inserting a set of pre-defined macro definitions for a specified SQL product, which will be used by the users in their application programs, the RE/SQL translator can easily solve this problem.

2. The approaches to runtime status checking in current implementations are also similar. The major differences come from message retrieval command. INQUIRE_SQL is command from INGRES, besides retrieving error or warning message, it has many other options which can be used for runtime status checking. These options will no longer be supported by RE/SQL because there are no equivalent functions in ALLBARE/SQL. The similar command in ALLBARE/SQL is SQLEXPLAIN, which can be used only to retrieve error and/or warning message. However, SQLEXPLAIN could be used to obtain more than one error and/or warning messages at the same time, but the INQUIRE_SQL could obtain only one message at once.

3. The approaches to implicit error handling, which uses WHENEVER command, are the same in the current SQL implementations.

2.7.3 Dynamic Operations and the SQKDA

Dynamic operations support dynamic programming which is the ability to specify a variety of critical program elements, such as queries, SQL statements, and SQL object names, at run time. In applications where table names and column names are not known until run time, or where complete queries must be built based on the application's run-time environment, the "hard-coded" SQL statement is not sufficient. For example, an application might include an expert mode in which the run-time user can type in select queries and browse the results at the terminal. To support applications such as these, SQL provides dynamic operations.

Dynamic operations provide the ability to specify table and column names and build queries at run time. Using dynamic operations, you can:

- execute a statement that is stored in a buffer (EXECUTE IMMEDIATE).

- encode a statement stored in a buffer and execute it multiple times (PREPARE and EXECUTE).

- obtain information about a table at run time (PREPARE and DESCRIBE).

The SQL Descriptor Area (SQLDA) is an integral part of dynamic programming. The SQLDA is a host language structure used by dynamic operation as a storage space for information. When used with the DESCRIBE statement, this information includes the name, data type, and length of the result columns.

All of the current SQL implementations can complete the preprocessing of dynamic commands at run time. Any SQL command except the following, which are designed only for programmatic use, can preprocessed at run time:

ALLOCATE SQLDA_SQLFMTARR

ALLOCATE SQLDA_DATABUFF

BEGIN DECLARE SECTION

BEGIN WORK

BULK operations

CLOSE

COMMIT WORK

CONNECT

DECLARE CURSOR

DECLARE SQLDA

DELETE WHERE CURRENT

DESCRIBE

DISCONNECT

END DECLARE SECTION

EXECUTE

EXECUTE IMMEDIATE

FETCH

FREE SQLDA_SQLFMTARR

FREE SQLDA_DATABUFF

INCLUDE

INQUIRE_SQL

LOCK TABLE

UNLOCK TABLE

OPEN

PREPARE

ROLLBACK WORK

SAVEPOINT

SET READLOCK

UPDATE WHERE CURRENT

WHENEVER

The SQL product translator might create permanently stored sections associated with dynamic commands, which can improve performance of execution of these dynamic operations. For the program developers and the users, this processing can be considered as being transparent, though it may be done in a different way in the different SQL products.

The dynamic operations have four commands that are exclusively for use in a dynamic program: EXECUTE IMMEDIATE, PREPARE, EXECUTE, and DESCRIBE. In addition, all commands that support cursors (DECLARE, OPEN, FETCH, UPDATE, DELETE, CLOSE) have dynamic versions to support dynamically executed queries. These dynamic commands are quite compatible in the current SQL implementations.

Notice that the RE/SQL translator must be able to work dynamically to translate the syntax that are used in dynamic operation, if any, to the syntax of a specified SQL implementation. That means the translator itself, appeared as a C function now, would be embedded into the user's applications during the preprocessing phase if any dynamic operation were used in the user's applications. There are two cases to be considered (this work will be done automatically by the RE/SQL translator, i.e. the users of RE/SQL do not need to consider these problems):

• While executing a dynamic command, a literal string is used as command string. In this case, RE/SQL translator will deal with the literal string according the RE/SQL syntax, and replace the literal string with a translated string in which the syntax will be compatible with a specified SQL implementation.

• While executing a dynamic command, a host variable is used to provide command string. In this case, the RE/SQL translator will embedded a C function version of translator in the user's application program, and meanwhile a C statement to invoke this function before any dynamic statement can be executed, which makes the command syntax in the host variable compatible with a specified SQL implementation. The following example can explain this case:

Before RE/SQL preprocessing, you may write a dynamic operation statement like:

```
EXEC SQL PREPARE ThisCommand FROM :DynamicCommand;
```

and then the translator will replace this statement with the following statements:

```
dynamic_translator(DynamicCommand,DynamicCommandBuffer);

EXEC SQL PREPARE ThisCommand FROM :DynamicCommandBuffer;
```

The dynamic preprocessing function, dynamic_translator(), will be append to your program. The original RE/SQL command will still be stored in DynamicCommand, the modified command string will be copied to a command buffer which is managed by RE/SQL translator.

Dynamic operations in SQL are of two major types, Dynamic Non-Queries which are dynamic operations that do not retrieve rows from the database, and Dynamic Queries which are dynamic operations that do retrieve rows. Note that dynamic queries may have a query result whose format is known to you at programming time, or they may have a query result whose format is unknown.

If the program does not know whether or not the statement is a query, the program can PREPARE and DESCRIBE the statement. The results returned by the DESCRIBE statement will indicate whether or not the statement was a query. The SQLDA_SQLD is set to o if the dynamic command is not a query and to a positive integer if its is a query. The SQLDA data structure is used in any program that may host a dynamic query.

In the following example, if the command is not a query, you branch to function NonQuery() in which you can execute a dynamic non-query. If is a query, you branch to function Query(), where you can execute a dynamic query.

```
EXEC SQL PREPARE ThisCommand FROM :DynamicCommand;

EXEC SQL DESCRIBE ThisCommand INTO SQLDA;

if (SQLDA_SQLD == 0)

 {

  NonQuery();

 }

else if (SQLDA_SQLD > 0)

 {

  Query();

 }
```

It is sometimes necessary to define dynamic data structures that can accommodate either non-queries or queries at run time, which is done through SQLDA and associated data structure and buffer that are used explicitly by the program developers in their application programs. Based on the survey of ALLBARE/SQL and INGRES/SQL, some differences have been identified in this area. An effort has been made to solve this problem, see following for more details.

2.10.1.4 Executing A Dynamic Non-Query

You can use either EXECUTE IMMEDIATE command or the combination of PREPARE and EXECUTE to execute a dynamic non-query statement. EXECUTE IMMEDIATE is most useful if the program executes the statement only once within a transaction. If the program executes the statement many times within a transaction, for example, within a program loop, use the PREPARE and EXECUTE combination: PREPARE the statement once, then EXECUTE as many times as necessary.

Using EXECUTE IMMEDIATE

If you know in advance that a dynamic command will not be a query, you can dynamically preprocess and execute the command in one step using the EXECUTE IMMEDIATE command, its syntax is:

EXEC SQL EXECUTE IMMEDIATE {statement_string | host_variable}

For example, the following command DROP a table:

EXEC SQL EXECUTE IMMEDIATE 'DROP TABLE test_table';

Another example does the same work:

strcpy(Command_Variable,"DROP TABLE test_table");

EXEC SQL EXECUTE IMMEDIATE :Command_Variable;

Using Combination Of PREPARE And EXECUTE

The PREPARE and EXECUTE commands can also be used to execute dynamic non-queries. These two commands, working together, allow your program to create and store a temporary section for a dynamic statement and to execute it as many time as possible. However, a prepared statement is discarded when the transaction in which it was prepared is rolled back or committed. Also, if you prepare a statement with the same name as an existing statement, the new statement supersedes the old statement. Their syntax are:

EXEC SQL PREPARE Command_Name

FROM {statement_string | host_variable}

and

EXEC SQL EXECUTE Command_Name

The following is an example to use these two command:

EXEC SQL PREPARE Command_Name FROM :Dynamic_Command;

for (i=0;i<loop;==i)

EXEC SQL EXECUTE Command_Name;

2.10.1.5 Executing A Dynamic Non-Query

Executing dynamic queries requires setting up a buffer to receive the query result and then extracting the items you want from the buffer. For those operations, you use three special data structures:

• **SQL Descriptor Area (SQLDA)** is a record used to pass information on the location and contents of the other two dynamic data structures, the format array and the data buffer. You set some fields in the SQLDA and pass them to SQL; and SQL passes values back to you in other fields.

• **SQL Format Array** is an array of records with one record for each select list item (column). The attributes of a column in the query result are described in a format array record. When you do not know the format of a query result at programming time, you use format array information to identify where in the data buffer to find each column value and how to interpret it. Such information is sent to format array each time you execute the DESCRIBE command.

• **SQL Data Buffer** is a memory space allocated for holding rows for a query result. SQL puts rows into the data buffer each time you execute the FETCH command.

The details about these data structures will be discussed in the next section. This section only addresses a general procedure to handle dynamic query with cursor operations. Though some specific details differ depending on the query type, in general you handle all types of dynamic query as follows:

• You declare data type for SQLDA, format array and data buffer at C declaration section of the program using following syntax (see next section for more details):

EXEC SQL INCLUDE SQLDA;

EXEC SQL DECLARE SQLDA;

• You specify PREPARE command with following syntax to dynamically preprocessing your query:

EXEC SQL PREPARE MyQuery FROM :DynamicCommand;

• You dynamically allocate space for format array with following RE/SQL command:

EXEC SQL ALLOCATE SQLDA_SQLFMTARR;

• The DESCRIBE command makes format array available to your program information about each column in a query result:

EXEC SQL DESCRIBE MyQuery INTO SQLDA;

• You dynamically allocate space for data buffer with following RE/SQL command:

EXEC SQL ALLOCATE SQLDA_DATABUFF;

• The DECLARE CURSOR command maps the temporary section to a cursor so that the other cursor manipulation commands can be used:

EXEC SQL DECLARE DynamicCursor CURSOR FOR MyQuery;

• The OPEN command attaches SQL buffer space for holding qualifying rows and defines the active set:

EXEC SQL OPEN DynamicCursor;

• The FETCH command evaluates all predicates in the query and transfers rows from the SQL database into SQLDA data buffer:

EXEC SQL FETCH DynamicCursor USING DESCRIPTOR SQLDA;

The USING DESCRIPOR clause indicates to SQL that row should be formatted in accord with a format array identified in the SQLDA and returned to a data buffer identified in the SQLDA. You can then obtain desired data from data buffer according to information stored in format array. The SQLDA, the format array, and the data buffer are discussed later in next section.

The FETCH command retrieve one row with once execution. (Though the dynamic version of FETCH command in ALLBARE/SQL can retrieve multiple rows in one FETCH, we can only retrieve one row in its version in RE/SQL because INGRES/SQL does not support this feature.) You can repeatedly execute the FETCH command until SQL set SQLCA_SQLCODE to 100.

• You free data buffer with following RE/SQL command:

EXEC SQL FREE SQLDA_DATABUFF;

• You free format array with following RE/SQL command:

EXEC SQL FREE SQLDA_SQLFMTARR;

• The CLOSE command closes the cursor:

EXEC SQL CLOSE DynamicCursor;

The COMMIT WORK and ROLLBACK WORK commands also close any open cursors.

Dynamic Query Result Data Structures

Declaring The SQL Descriptor Area (SQLDA)

Every SQL C application program, which uses dynamic operations, must have the SQL Descriptor Area (SQLCA) declared in the global declaration part. You can use the INCLUDE command to declare the SQLDA:

EXEC SQL INCLUDE SQLDA;

When the translator (from SQL product) parses this command, it insert necessary type definition used in SQLDA into the modified source file. The structures defined for SQLCA may differ from one SQL

implementation to another, though these differences are non-essential. The SQLDA structures defined in ALLBARE/SQL and INGRES/SQL are provided in the appendix, a comparison table is also given. Some of the elements in SQLDA C structure are used by SQL itself, here we discuss only elements used by the users. The element names in the structure may be different so that these element names can no longer be referred directly by the users in their application programs. To solve this problem, we use macros instead of element names in the user's application programs. The macros are pre-defined for each current SQL implementation and will be automatically inserted into the program by the RE/SQL translator during the preprocessing phase for a specified SQL implementation. The Table 10 shows the macro definitions for the ALLBARE/SQL and for the INGRES/SQL. With using these variables, users can obtain query result information when you are executing a dynamic query.

TABELLE 10 Macro Definitions For SQLDA[a]

#define SQLDA_SQLN	sqlda.sqln	sqlda->sqln
#define SQLDA_SQLD	sqlda.sqld	sqlda->sqld
#define SQLDA_SQLFMTARR	sqlda.sqlfmtarr	sqlda->sqlvar

a. Only the interested variables are listed in this table.

SQLDA_SQLN is 2-byte integer indicating the number of allocated format array records (one record per select list item). This value must be set by the program before describing a statement. The value must be greater than or equal to zero.

SQLDA_SQLD is 2-byte integer indicating the number of columns in query result. When describing a dynamic SQL statement, if the value in SQLDA_SQLD is zero, then the described statement is not a query.

SQLDA_SQLFMTARR is an SQLDA_SQLN-size array which is declared as record structure to store columns retrieved by FETCH command. The elements in SQLDA_SQLFMTARR will be discussed in the following section.

Setting Up the Format Array SQLDA_SQLFMTARR

SQLDA_SQLFMTARR is an array of record for storing columns retrieved by FETCH command when the users do not know the column format information. Based on information in SQLDA_SQLFMTARR returned from SQL query by DESCRIBE command, the user can analyze column types and find the location of specified data in Data Buffer. SQLDA_SQLFMTARR must be declared in the program declaration section and allocated before you DESCRIBE the SQL statement. The declarations and uses of SQLDA_SQLFMTARR are different in current SQL implementations. Much efforts have been made to standardize the programming techniques for dynamic queries by means of the following methods:

• define a RE/SQL-specific command, which are DECLARE SQLDA. When this command is preprocessed by RE/SQL translator, it will be translated to a group of proper declaration statements in C language, and automatically embedded in the user's source code.

• define two RE/SQL-specific commands, which are ALLOCATE {SQLFMTARR | DATABUFF} and FREE {SQLFMTARR | DATABUFF}, to dynamically allocate and free space of SQLDA_SQLFMTARR and SQLDA_DATABUFF.

• provide a set of C functions, see following section, to manipulate (test and/or obtain) the data store in SQLDA_DATABUFF. On the other hand, a group of C macros are defined to eliminate differences of the structure elements. The elements of structure of SQLDA_SQLFMTARR is no longer transparent to the users, i.e. the users have to use these functions and macros to retrieve data from SQLDA_SQLFMTARR and SQLDA_DATABUFF.

SQLDA_SQLFMTARR must be declared and allocated in size of which it can store all columns (in one row) returned by the FETCH command. The data type for SQLDA_SQLFMTARR is declared at the same time when you declare SQLDA in a INCLUDE command. And then you can allocate space for SQLDA_SQLFMTARR dynamically. The declaration must occur just after you declare SQLDA with a INCLUDE command:

EXEC SQL INCLUDE SQLDA;

EXEC SQL DECLARE SQLDA;

These commands must appear at C global variable declaration section, normally on the top of the file which include SQL statements. No number of record is needed here. These two commands will be translated to the following statements by the RE/SQL translator during the preprocessing phase:

For ALLBARE/SQL:

EXEC SQL INCLUDE SQLDA;

sqlformat_type *sqlfmts;

char *DataBuffer;

For INGRES/SQL:

EXEC SQL INCLUDE SQLDA;

IISQLDA *sqlda;

Then, you allocate space for SQLDA_SQLFMTARR before you use DESCRIBE statement, and free the allocated space after you FETCH data into SQLDA_DATABUFF and obtain data from SQLDA_DATABUFF. The following statements have been developed to support you to do this work:

```
EXEC SQL ALLOCATE SQLDA_SQLFMTARR NumberOfRecord;

. /*you DESCRIBE, FETCH and obtain data */

EXEC SQL FREE SQLDA_SQLFMTARR;
```

These two commands will be translated to the following statements by the RE/SQL translator during the preprocessing phase:

For ALLBARE/SQL:

```
sqlfmts = (sqlformat_type *) calloc(1, NumberOfRecord *
                        sizeof(sqlformat_type));
if (sqlfmts == (sqlformat_type *)0)
{
 /* print error message and exit */
}
sqlda.sqln = NumberOfRecord;
sqlda.sqlfmtarr = sqlfmts;
. /* You DESCRIBE, FETCH and obtain data */
free(sqlfmts);
```

For INGRES/SQL:

```
sqlda = (IISQLDA *)calloc(1, IISQDA_HEAD_SIZE +
                (NumberOfRecord * IISQLDA_VAR_SIZE));
if (sqlda == (IISQLDA *)0)
{
 /* print error message and exit */
}
sqlda->sqln = NumberOfRecord;
. /* You DESCRIBE, FETCH and obtain data */
free(sqlda);
```

One of the main difference of SQLDA declarations between ALLBARE/SQL and INGRES/SQL is that the SQLDA is declared as a variable of structure in ALLBARE/SQL and is declared as a pointer of structure in INGRES/SQL. This may cause difference when you use SQLDA, respectively using "." or "->" to refer to elements of the structure. See the next section for more information.

Setting Up the SQLDA_DATABUFF

Because the SQLDA_SQLFMTARR can contain only the information about each column, you must allocate a space for SQLDA_DATABUFF to actually store the row retrieved by FETCH. This work should be done after you DESCRIBE the query and before you FETCH data, because the SQLDA_DATABUFF must be allocated based on the information of SQLDA_SQLFMTARR. The data type of the DateBuffer has been declared together in DECLARE SQLDA statement. The following statements have been developed to support you to do this work:

```
EXEC SQL ALLOCATE SQLDA_DATABUFF;

. /*you FETCH and obtain data */

EXEC SQL FREE SQLDA_DATABUFF;
```

These two commands will be translated to the following statements by the RE/SQL translator during the preprocessing phase:

For ALLBARE/SQL:

```
/* here DataBuffer is allocated for its maximum value

  to simplify the calculation of buffer size */

DataBuffer = (char *) calloc(1, 2500);

/* set some data field of SQLDA */

sqlda.sqlbuflen = 2500;

sqlda.sqlrowbuf = (int) DataBuffer;

sqlda.sqlnrow = 1;

. /* You FETCH and obtain data */

free(DataBuffer);
```

For INGRES/SQL:

```
for (SQL_ii = 0; SQL_ii < sqlda.sqld; SQL_ii++)

{

 /* allocate space for column data according information
```

```
  in the SQLDA_SQLFMTARR */
switch (abs(sqlda->sqlvar[SQL_ii].sqltype))
{
 case IISQ_INT_TYPE:
  if (sqlda->sqlvar[SQL_ii].sqllen == 4)
   sqlda->sqlvar[SQL_ii].sqldata = (char *)calloc(1, 4);
  else
   sqlda->sqlvar[SQL_ii].sqldata = (char *)calloc(1, 2);
  break;

 case IISQ_FLT_TYPE:
  if (sqlda->sqlvar[SQL_ii].sqllen == 4)
   sqlda->sqlvar[SQL_ii].sqldata = (char *)calloc(1, 4);
  else
   sqlda->sqlvar[SQL_ii].sqldata = (char *)calloc(1, 8);
  break;
 case IISQ_CHA_TYPE:
  sqlda->sqlvar[SQL_ii].sqldata =
        (char *)calloc(1, sqlda->sqlvar[SQL_ii].sqllen+1);
  break;
 case IISQ_VCH_TYPE:
  sqlda->sqlvar[SQL_ii].sqldata =
        (char *)calloc(1, sqlda->sqlvar[SQL_ii].sqllen+2);
  break;
```

```
 case IISQ_DTE_TYPE:
  sqlda->sqlvar[SQL_ii].sqldata =
                    (char *)calloc(1, IISQ_DTE_LEN);
  sqlda->sqlvar[SQL_ii].sqllen = 25;
  break;
 }
/* allocate variable space for null indicator */
 if (slqda->sqlvar[SQL_ii].sqltype > 0)
  sqlda->sqlvar[SQL_ii].sqlind = (short *)0;
 else
  sqlda->sqlvar[SQL_ii].sqlind =
                (short *)calloc(1,sizeof(short));
}
. /* FETCH and obtain data */
for (SQL_ii = 0; SQL_ii < sqlda->sqld; SQL_ii++)
{
 free(sqlda->sqlvar[SQL_ii].sqldata);
 if (sqlda->sqlvar[SQL_ii]. sqltype < 0)
  free(sqlda->sqlvar[SQL_ii].sqlind);
}
```

There are big differences for SQLDA_DATABUFF between ALLBARE/SQL and INGRES/SQL. In ALLBARE/SQL, the SQLDA_DATABUFF is declared and allocated as a whole character variable of array, and the location information are stored in SQLDA_SQLFMTARR. In INGRES/SQL, the SQLDA_DATABUFF is declared and allocated as separated pointers for each record in the structure of SQLDA_SQLFMTARR. This may cause differences when you use SQLDA_DATABUFF. See the next section for more information.

Using the SQLDA_SQLFMTARR and SQLDA_DATABUFF

The SQLDA_SQLFMTARR is defined and used in different ways for different SQL implementations. These differences include:

- the elements defined in the data format structure are not the same for their names.
- the data type is coded in different way.
- the data of columns are stored in different way so that they must be accessed in different way.

To solve these problems, we have defined some group of C macros. In the user's application program, the SQLDA_SQLFMTARR is used as a structure array, which is declared in size of NumberOfRecord that is stored in SQLDA_SQLN. After you FETCH a row from the database to SQLDA_DATABUFF, the number of columns returned is set to SQLDA_SQLD. You may refer the array record as following format:

SQLDA_SQLFMTARR[i].XXXXX

where i has the valid range from 0 to SQLDA_SQLD - 1. XXXXX denotes structure elements in SQLDA_SQLFMTARR. They are C macros defined for the reason mentioned above. Table 11 lists the definitions for all of the valid macros which you can use in your application program.

TABELLE 11 Macro Definitions For SQLDA_SQLFMTARR Elements[a]

#define SQLTYPE	sqltype	sqltype
#define SQLLEN	sqlvallen	sqllen
#define SQLNAME	sqlname	sqlname

a. Only the interested variables are listed in this table.

SQLTYPEspecifies the data type with a coded integer. Because the codes used in different implementations are not the same, we use another group of macros to identify the code in SQLTYPE. Table 12 gives out definitions for these macros. Some codes must be used with the data length SQLLEN. For

TABELLE 12 Macro Definitions For SQLDA Data Type Codes

#define SESQL_INTEGER_TYPE	0	30
#define SESQL_FLOAT_TYPE	4	31
#define SESQL_CHAR_TYPE	2	20

TABELLE 12 Macro Definitions For SQLDA Data Type Codes

#define SESQL_VARCHAR_TYPE	3	21
#define SESQL_DATE_TYPE	10	3
#define SESQL_DATETIME_TYPE	12	3
#define SESQL_INTERVAL_TYPE	13	3

example, when SQLTYPE is equal to SESQL_INTEGER_TYPE, and SQLLEN is equal to 2 or 4, it denotes correspondingly a short integer or a long integer. When SQLTYPE is equal to SESQL_FLOAT_TYPE, and SQLLEN is equal to 4 or 8, it denotes correspondingly a float or a double float. See later of section for details. On the other hand, the codes returned to SQLTYPE by INGRES/SQL may be negative to indicate a nullable column so that it is always necessary to use absolute value of SQLTYPE in your program.

SQLLEN specifies the length of the data stored in data buffer. Generally, it is the practical length of the data. But there are two exceptions for different SQL implementations: 1) while dealing with the DATE, DATETIME, and INTERVAL data type, ALLBARE/SQL returns practical data length to SQLLEN, but INGRES/SQL returns 25 as data length because INGRES/SQL adds ASCII blank to the end of string if it is as long as 25 characters; 2) while dealing with VARCHAR data type, ALLBARE/SQL the length returned to SQLLEN includes 4 byte prefix containing actual length of VARCHAR data, but INGRES/SQL specifies a value which does not include the prefix.

SQLNAME specifies the name of the column with a char string. The string is terminated with '\0'.

Note that the practical data returned by FETCH command in DATABUFF can no longer be referred as elements of structure, which is sqlda.sqlfmtarr[i].sqlvof (short type) that is byte offset of value from beginning of DataBuffer in ALLBARE/SQL and is sqlda->sqlvar[i].sqldata (char pointer type) that is address of column value in INGRES/SQL, because of their incompatible data type. To solve this problem, a group of macros have been defined to return the address of column data and address of null indicator in specified SQLDA_SQLFMTARR record. These macros have been defined for ALLBARE/SQL and INGRES/SQL respectively in Table 13:

TABELLE 13 Macro Definitions For SQLDA_DATABUFF ADDRESSES

#define SQLDA_DAT_AD(i)	&DataBuff[sqlda.sqlfmtarr[i].sqlvof]	sqlda->sqlvar[i
#define SQLDA_IND_AD(i)	&DateBuff[sqlda.sqlfmtarr[i].sqlnof]	sqlda->sqlvar[i
#define SQLDA_IFNULL(i)	sqlda.sqlfmtarr[i].sqlindlen	sqlda->sqlvar[i

SQLDA_IND_AD(i) returns null indicator address as a (short *) type pointer for a nullable column specified with i. Its use depends on the value returned by SQLDA_IFNULL(i) which indicates if the column is nullable. See following for details.

SQLDA_IFNULL(i) returns an integer value. When SQLDA_IFNULL(i) is equal to zero, the column is NOT nullable. In this case, there is NO a short variable in data buffer pointed by SQLDA_IND_AD(i). When SQLDA_IFNULL(i) is NOT equal to zero, the column is nullable. In this case, there is a short variable in data buffer pointed by SQLDA_IND_AD(i). You can obtain this short value by referring the value of this address, *SQLDA_IND_AD(i). Refer to "Declare Host Variable" for description for indicator variable.

SQLDA_DAT_AD(i) returns data buffer address as a (char *) type pointer for the column specified with i. The data in the data buffer must be interpreted with the data type and the data length information which are stored as SQLDA_SQLFMTARR[i].SQLTYPE and SQLDA_SQLFMTARR[i].SQLLEN. Based on the data type, you must perform data conversion for the data type other than (char *) before you assign the data in the data buffer to a proper C variable. Generally, we use following program segment to deal with the data in data buffer:

```
short short_value;

long long_value;

float float_value;

double double_value;

char date_time[26];

char char_str[MAX_CHAR_COLUMN_SIZE];

/* MAX_CHAR_COLUMN_SIZE must be declared in size of that is one

  character long than column size */

struct VARCHAR_TYPE {

  PREFIX_TYPE varchar_length;

  char varchar_string[MAX_VARCHAR_COLUMN_SIZE];

} varchar_str;
```

```
/* PREFIX_TYPE is a macro defined for ALLBARE/SQL
   as long integer, for INGRES/SQL as short integer */.
/* after you FETCH data into data buffer */
/* the following statements are in a loop for variable i */
switch (abs(SQLDA_SQLFMTARR[i].SQLTYPE))
{
 case SESQL_INTEGER_TYPE:
  if (SQLDA_SQLFMTARR[i].SQLLEN == 2)
   short_value = *(short *)SQLDA_DAT_AD(i);
  else
   long_value = *(long *)SQLDA_DAT_AD(i);
  break;
 case SESQL_FLOAT_TYPE:
  if (SQLDA_SQLFMTARR[i].SQLLEN == 4)
   float_value = *(float *)SQLDA_DAT_AD(i);
  else
   double_value = *(double *)SQLDA_DAT_AD(i);
  break;
 case SESQL_CHAR_TYPE:
  memcpy(char_str,SQLDA_DAT_AD(i),SQLDA_SQLFMTARR[i].SQLLEN);
  char_str[SQLDA_SQLFMTARR[i].SQLLEN] = '\0';
  break;
 case SESQL_VARCHAR_TYPE:
  memcpy(varchar_str,SQLDA_DAT_AD(i),
       SQLDA_SQLFMTARR[i].SQLLEN + PREFIX_LENGTH);
```

```
/* PREFIX_LENGTH is a macro defined for ALLBARE/SQL as 0
                        for INGRES/SQL as 2 */
 break;
case SESQL_DATE_TYPE:
case SESQL_DATETIME_TYPE:
case SESQL_INTERVAL_TYPE:
 memcpy(date_time,SQLDA_DAT_AD(i),SQLDA_SQLFMTARR[i].SQLLEN);
 char_str[SQLDA_SQLFMTARR[i].SQLLEN] = '\0';
 break;
}
```

Notes:

1. An extremely effort has been made to standardize the use of dynamic SQL operations so that it could be thought of much easier to use them in a user's application program. The functionality has been remained as more as possible. No important function has been removed.

2. To do this work, some newly defined RE/SQL statements have been developed, they are DECLARE SQLDA, ALLOCATE, and FREE. And a procedure to use the dynamic operations has been profiled.

3. On the other hand some group of macros have been defined to access data buffer which is used by dynamic queries. You can find these macros in appendix.

2.11 Conclusions

The most parts of the important SQL functions and features, which have been implemented in ALLBARE/ SQL and INGRES/SQL, have been reviewed on an one by one basis in this report. This report is prepared in the form of a reference manual. As a completed documentation of this work, all necessary information are included except the following sub-titles:

- An One-by-one Command Description
- Transaction Management
- Cursor Operations

• BULK Operations

• Concurrence Controls

The completion of these sub-titles depends on the further work and will be reported during carrying out the following working steps.

Based on the detailed survey for the current SQL implementations, many significant differences have been identified, some of them have been well solved in this report:

• the DATE/TIME data types are defined in different way in current SQL implementations, which include their declarations and data formats. This problem is solved by re-defining the data type and providing a group of data conversion functions. See section "2.3 DATA TYPES" for details.

• the SQL Communication Area (SQLCA) is declared and used in different way in current SQL implementations, which may influence status checking and error handling technical. This problem has been solved by providing a group of pre-defined C macros. See section "4.2 RUNTIME STATUS CHECKING AND THE SQLCA" for details.

• the SQL Descriptor Area (SQLDA) is declared and used in different ways in the current SQL implementations, which may affect dynamic SQL operations. Dynamic SQL operations are thought of one of the most important functions. An extremely effort has been made to standardize the use of dynamic SQL operations so that it could be thought of much easier to use them in a user's application program. The functionality has been remained as more as possible. No important function has been removed. To do this work, some newly defined RE/SQL statements have been developed, they are DECLARE SQLDA, ALLOCATE, and FREE. And a procedure to use the dynamic operations has been profiled. On the other hand some group of macros have been defined to access data buffer which is used by dynamic queries. See section "4.3 DYNAMIC OPERATIONS AND THE SQLDA" for details.

Some small changes made in this report are not mentioned here. Some significant differences identified are expected to solve in the following working steps:

• the differences among the command syntaxes will be eliminated by developing RE/SQL syntax accompanying the RE/SQL translator. The syntax for each command listed in section 3 will be developed on an one-by-one basis in the following step. See Syntax Development for details.

• BULK option in INSERT and SELECT statements were original supported only by ALLBARE/SQL. Because of existing similar functions in INGRES/SQL, this function, which is very useful, will be remained by means of pre-processor. See Syntax Development for details.

• Some statements to control authority, concern with concepts used to control data accessing in current SQL implementations, which are the most difficult area to make these functions available in RE/SQL. Locking managements are also quite different in ALLBARE/SQL and INGRES/SQL. In ALLBARE/SQL, the most of the locking could be controlled when you issue some statements. There is only one special statement for locking, that is LOCK TABLE. In INGRES/SQL, the most of the locking could be controlled by a special statement, SET LOCKMODE. To unify these locking control would be difficult, and needs a more detailed analysis. These two features mentioned above are critical to enable RE/SQL supporting multiple user working environment.

• Though both of ALLBARE/SQL and INGRES/SQL have supported the SAVEPOINT statement, the syntax they used are quite different. In ALLBARE/SQL, the SAVEPOINT statement must use a host variable as the parameter. And the INGRES/SQL does not allow using a host variable as parameter. Despite it is not easy, the pre-processor would solve this problem, see Syntax Development for details.

Some functions and features are thought of very difficult or even impossible to be included in the RE/SQL, which can be considered as the limitations of RE/SQL:

• Because of some data types are not supported by both of ALLBARE/SQL and INGRES/SQL, the data type which have been removed from ALLBARE/SQL are DECIMAL, BINARY, VARBINARY, LONGBINARY, LONG BINARY, and TIME, the data type which have been removed from INGRES/SQL are MOMEY, TABLE_KEY, OBJECT_KEY. The maximum character length decreased from 3999 to 2000 which is maximum character length of INGRES/SQL.

• Almost all of the assistant functions except aggregate functions in both ALLBARE/SQL and INGRES/SQL are no longer supported by RE/SQL.

• NGRES/SQL can support multi-session in one application program, but ALLBARE/SQL cannot. Therefore, the related session management options and statements will be removed from RE/SQL.

It is worth to emphasize that RE/SQL is an off-line tool which preprocesses user's application programs in source code level before they use translator from SQL products. That means that you can not make an application program with RE/SQL to access databases created from different SQL products simultaneously, i.e. you have to make an individual version of application executable for each SQL product before you can access different type databases. Contrastively, INGRES/OpenSQL can on-line connect to multiple sessions through INGRES Gateways, which are databases installed in the network, and which are even created by different SQL products. This feature is high beyond those of RE/SQL, and is high beyond our objectives.

2.12 REFERENCES

[1] ALLBASE/SQL Reference Manual , HEWLET PACKARD, Customer Order Number 36217-90004 January. 1991

[2] ALLBASE/SQL C Application Programming Guide, HEWLETT PACKARD, Customer Order Number 36217-90014 January 1991.

[3] ALLBASE/SQL Database Administration Guide , HEWLETT PACKARD, Customer Order Number 36217-90005 January 1995

[4] INGRES/SQL Reference Manual, Computer Associates, Customer Order Number 64-9(9)-47101. December 1991

[5] INGRES/Embedded SQL Companion Guide for C (Unix) ,Computer Associates, Customer Order Number 64-99-17503. December 1991

[6] INGRES/OpenSQL Reference Manual (Unix) , Computer Associates, Customer Order Number 64-9(9)-47107. December 1991

[7] AMERICAN NATIONAL STANDARD FOR INFORMATION SYSTEMS - DATABASE LANGUAGE - SQL, ANSI X3.135, 1992.

[8] STANDARD SQL RELATIONAL DATA BASE LANGUAGE GUIDE AND REFERENCE MANUAL, Computer Technology Research Corp, by Boris Musteata and Robert Lesser, Patchogue, New York 11772 U.S.A., 1988.

[9] SQL, The Structured Query Language, by Carolyn J. Hursch and Hack L. Hursch, Blue Ridge Summit, 1988.

APPENDIXES

1. The list of the file 'allbase.macros.h' which is the include file for ALLBARE/SQL:

```
/* definitions for SQLCA */

#define SQLCA_SQLCODE       sqlca.sqlcode

#define SQLCA_SQLERRD2      sqlca.sqlerrd[2]

#define SQLCA_SQLWARN0      sqlca.sqlwarn[0]

#define SQLCA_SQLWARN1      sqlca.sqlwarn[1]

#define SQLCA_SQLWARN2      sqlca.sqlwarn[2]

#define SQLCA_SQLWARN3      sqlca.sqlwarn[3]

#define SQLCA_SQLWARN6      sqlca.sqlwarn[6]

/* definitions for SQLDA */

#define SQLDA_SQLN          sqlda.sqln

#define SQLDA_SQLD          sqlda.sqld

#define SQLDA_SQLFMTARR     sqlda.sqlfmtarr

/* definitions for format array */

#define SQLTYPE             sqltype

#define SQLLEN              sqlvallen

#define SQLNAME             sqlname
```

```
/* definitions for data type */
#define SESQL_INTEGER_TYPE      0
#define SESQL_FLOAT_TYPE        4
#define SESQL_CHAR_TYPE         2
#define SESQL_VARCHAR_TYPE      3
#define SESQL_DATE_TYPE         10
#define SESQL_DATETIME_TYPE     12
#define SESQL_INTERVAL_TYPE     13

/* definitions for data buffer address */
#define SQLDA_DAT_AD(i)     &DataBuff[sqlda.sqlfmtarr[i].sqlvof]
#define SQLDA_IND_AD(i)     &DateBuff[sqlda.sqlfmtarr[i].sqlnof]
#define SQLDA_IFNULL(i)     sqlda.sqlfmtarr[i].sqlindlen

/* definitions for other */

#define PREFIX_TYPE         long
#define PREFIX_LENGTH       0
short SQL_ii;
```

2. The list of the file 'ingres.macros.h' which is the include file for INGRES/SQL:

```
/* definitions for SQLCA */
#define SQLCA_SQLCODE       sqlca.sqlcode
```

```
#define SQLCA_SQLERRD2      sqlca.sqlerrd[2]
#define SQLCA_SQLWARN0      sqlca.sqlwarn0
#define SQLCA_SQLWARN1      sqlca.sqlwarn1
#define SQLCA_SQLWARN2      sqlca.sqlwarn2
#define SQLCA_SQLWARN3      sqlca.sqlwarn3
#define SQLCA_SQLWARN6      sqlca.sqlwarn4

/* definitions for SQLDA */
#define SQLDA_SQLN          sqlda.sqln
#define SQLDA_SQLD          sqlda.sqld
#define SQLDA_SQLFMTARR     sqlda.sqlvar

/* definitions for format array */
#define SQLTYPE             sqltype
#define SQLLEN              sqllen
#define SQLNAME             sqlname

/* definitions for data type */
#define SESQL_INTEGER_TYPE      30
#define SESQL_FLOAT_TYPE        31
#define SESQL_CHAR_TYPE         20
#define SESQL_VARCHAR_TYPE      21
#define SESQL_DATE_TYPE         3
#define SESQL_DATETIME_TYPE     3
#define SESQL_INTERVAL_TYPE     3
```

```
/* definitions for data buffer address */
#define SQLDA_DAT_AD(i)      sqlda->sqlvar[i].sqldata
#define SQLDA_IND_AD(i)      sqlda->sqlvar[i].sqlind
#define SQLDA_IFNULL(i)      sqlda->sqltype
/* definitions for other */
#define PREFIX_TYPE          short
#define PREFIX_LENGTH        2
short SQL_ii;
```

Chapter 3 The Syntax for the RODOS Embedded SQL

3.1 Introduction

The syntax for the RE/SQL may differ from the syntax for any one of the current SQL implementations. The statements which will be included have been mentioned in The Scope Definition Of The Functions And Features For The RE/SQL.

The statements will be described on an one-by-one basis with RE/SQL syntax. Because the syntax of the RE/SQL will be preprocessed by RE/SQL translator, which will be coded in the following step, and will be translated to the syntax of a specified current SQL implementation. The equivalent syntaxes of the current SQL implementations, which are ALLBARE/SQL and INGRES/SQL at this moment, will be provided following each RE/SQL statement. Actually, the syntax of the RE/SQL is the input of RE/SQL translator, and the syntaxes for ALLBARE/SQL and INGRES/SQL are the output of the RE/SQL translator, which may be only a part of their original syntaxes.

The following information, if any, will be provided to each of the RE/SQL statement:

- RE/SQL Syntax (S)
- ALLBARE/SQL Syntax (A)
- INGRES/SQL Syntax (I)
- Parameter Explanations for RE/SQL
- Descriptions for RE/SQL Statement
- Notes to Declare Changes and Limits

This report uses the following conventions to describe statement syntax specifications:

`UPPERCASE`	In a syntax statement, commands and keywords are shown in uppercase characters. the characters must be entered in the order shown; however, you can enter the characters in either upper or lowercase.
`italics`	In a syntax statement, a word in italics represents a parameter or argument that you must replace with the actual value.
`punctuation`	In a syntax statement, punctuation characters (other than brackets, braces, vertical bars, and horizontal ellipses) must be entered exactly as shown.

{ }	In a syntax statement, braces enclose required elements. When several elements are stacked within braces, you must select one.
[ ]	In a syntax statement, square brackets enclose option elements.
|	In a syntax statement, vertical bar is used between items in a list to indicate that you should choose one of the items.
. . .	In a syntax statement, horizontal ellipse indicates some items would be used repeatedly.

USING OPTION CLAUSE SUPPORTED BY SESQL/SQL

• Sometimes, you can find that the difference between some SQL statements which supported by current SQL products could not be overcome by syntax translator, because the part of the statement might be complete different thing. For those case, you can use OPTION clause supported by RE/SQL.

• There are two kinds of option clause, which are OPTION and &OPTION. They cause different actions during the syntax translating.

If you use OPTION, the translator will just append the corresponding part of the clause to the main part of the statement which has been translated by the translator. The context of the clause will not be processed, i.e. you have to write those contexts in the syntax of destination SQL product instead of RE/SQL syntax. Example 1 shows how to use OPTION clause.

If you use &OPTION, the translator will append the corresponding part of the clause to the main part of the statement before translating. And then the statement will be translated to the syntax of the destination SQL product as a whole, i.e. you have to write those clause in the RE/SQL syntax. Example 2 shows how to use &OPTION clause.

• Rules to use OPTION clause:

- If necessary, you can use OPTION clause in any one of the RE/SQL statements described in this document.

- OPTION clause, if any, must be the last clause of the statement.

- OPTION clause is constructed by one or more elements, which separated by comma:

[&]OPTION *element-1, element-2, ..., element-n*;

Basically, one element corresponds one SQL product.

- The syntax for element is as following:

'@' = <<*context of clause*>>

here, @ is a single character, so called product identifier, to specify the object product, to which the context of the clause will be assigned. In this moment, the valid product identifier is 'A', for ALLBARE/SQL, and 'I', for INGRES/SQL. Following product identifier, you must place an equal-sign. And then, you can place the context of the clause for that product, which is delimited by double < sign and double > sign. You can not use double < sign and double > sign in the context of the clause.

Example 1: (about using CREATE TABLE command)

• In the following, we will use CREATE TABLE statement as an example. In ALLBARE/SQL, the database object can be stored optionally in different DbeFiles, in this case the user should specify a IN clause in CREATE TABLE statement, e.g.

EXEC SQL CREATE TABLE TestTable (...) IN TestDbeFileName;

In INGRES/SQL, the concept of DbeFiles is not supported. But, similarly, you can use the concept of LocationNames to store the database object in different places by using WITH clause,

EXEC SQL CREATE TABLE TestTable (...)

WITH location=TestLocationName;

It is worth mentioning that the concept of DbeFiles and LocationNames are not the same (physical storage) and the procedures to create them are also completely different. On the other hand, in INGRES/SQL, the WITH clause can support to set many other optional parameters.

• For this case, you can use OPTION clause developed in RE/SQL to overcome this difficulty. OPTION clause can be used to assign a clause of context for a specified SQL product. For above example, the following RE/SQL statement can be correctly translated to object SQL product syntax,

EXEC SQL CREATE TABLE TestTable (...) OPTION

'A' = <<IN TestDbeFileName>>,

'I' = <<WITH location=TestLocationName>>;

In this case, you must follow different procedures to create TestDbeFileName as a DBEFILE and TestLocationName as a LOCATION in advance. The translator can translate this statement to its form in specified SQL product, which just like those statements mentioned above.

Example 2: (about using system catalog)

• In some cases, the application needs to retrieve all table names, which belongs to a specified user, in a database. This information is certainly available in all SQL products, but they are defined and used in different ways. For example, in ALLBARE/SQL, this query should be made as following:

EXEC SQL BULK SELECT name FROM system.table INTO :table_name

WHERE owner='RODOS';

In INGRES/SQL, this query should be made as following:

EXEC SQL BLUK SELECT table_name FROM iitables INTO :table_name

WHERE table_owner='rodos';

• You can find in these two examples that different table names and column name are specified. By using OPTION clause, you can combine the two statements into one, which can be understood by the RE/SQL translator:

EXEC SQL BULK SELECT &OPTION

'A' = <<name FROM system.table INTO :table_name

WHERE owner='RODOS'>>,

'I' = <<table_name FROM iitables INTO :table_name

WHERE table_owner='rodos'>>;

• In order to use other similar information, you need to refer to related manuals of the SQL products.

• Another example to use OPTION clause can be found in the section for CONNECT statement.

• To use OPTION clause, you have to know each of the current SQL syntaxes very well.

3.2 Allocate SQLDA_DATABUFF

The ALLOCATE SQLDA_DATABUFF statement is used to allocate memory space for data buffer in SQLDA, which is used to hold the data retrieved by FETCH command.

Syntaxes

S: EXEC SQL ALLOCATE SQLDA_DATABUFF;

A: DataBuffer = (char *) calloc(1, 2500);

sqlda.sqlbuflen = 2500;

sqlda.sqlrowbuf = (int) DataBuffer;

sqlda.sqlnrow = 1;

I: for (SQL_ii = 0; SQL_ii < sqlda.sqld; SQL_ii++)

```
{
switch (abs(sqlda->sqlvar[SQL_ii].sqltype))
{
 case IISQ_INT_TYPE:
  if (sqlda->sqlvar[SQL_ii].sqllen == 4)
   sqlda->sqlvar[SQL_ii].sqldata = (char *)calloc(1, 4);
  else
   sqlda->sqlvar[SQL_ii].sqldata = (char *)calloc(1, 2);
  break;
 case IISQ_FLT_TYPE:
  if (sqlda->sqlvar[SQL_ii].sqllen == 4)
   sqlda->sqlvar[SQL_ii].sqldata = (char *)calloc(1, 4);
  else
   sqlda->sqlvar[SQL_ii].sqldata = (char *)calloc(1, 8);
  break;
 case IISQ_CHA_TYPE:
  sqlda->sqlvar[SQL_ii].sqldata =
      (char *)calloc(1, sqlda->sqlvar[SQL_ii].sqllen+1);
  break;
 case IISQ_VCH_TYPE:
  sqlda->sqlvar[SQL_ii].sqldata =
      (char *)calloc(1, sqlda->sqlvar[SQL_ii].sqllen+2);
  break;
 case IISQ_DTE_TYPE:
  sqlda->sqlvar[SQL_ii].sqldata =
```

```
                    (char *)calloc(1, IISQ_DTE_LEN);
  sqlda->sqlvar[SQL_ii].sqllen = 25;
  break;
 }
 if (sqlda->sqlvar[SQL_ii].sqltype > 0)
  sqlda->sqlvar[SQL_ii].sqlind = (short *)0;
 else
  sqlda->sqlvar[SQL_ii].sqlind =
                (short *)calloc(1,sizeof(short));
 }
```

Parameters

None.

Descriptions

• If your application program wants to deal with dynamic queries, you must use this statement to allocate necessary space for SQLDA data buffer to hold the data retrieved by FETCH command. To interpret the data holding in data buffer, you must use information stored in format array.

• This statement must be placed at just after you DESCRIBE the query but before you FETCH the data.

• After you use the data buffer, you must free the space allocated by this statement with FREE SQLDA_DATABUFF statement.

Notes

• DECLARE SQLDA statement is a newly defined statement in RE/SQL, which is to standardize the variable declarations and allocations for dealing with dynamic queries.

• The dynamic memory allocation technical is used. For ALLBARE/SQL, the data buffer is allocated in its maximum size to simplify the calculation of buffer size. For INGRES/SQL, the data buffer is allocated according to information in format array.

• See "Dynamic Operations" for more information.

3.3 Allocate SQLDA_SQLFMTARR

The ALLCOCATE SQLDA_SQLFMTARR statement is used to allocate memory space of format array, which is used to accept format information of a dynamic query.

Syntaxes

S: EXEC SQL ALLOCATE SQLDA_SQLFMTARR *NumberOfRecode*;

```
A: {
   sqlfmts = (sqlformat_type *) calloc(1, NumberOfRecode *
                        sizeof(sqlformat_type));
   if (sqlfmts == (sqlformat_type *)0)
   {
    printf("\nruntime error: memory allocation error in\n");
    printf(" ALLOCATE SQLDA_SQLFMTARR statement. \n");
    exit(0);
   }
   sqlda.sqln = NumberOfRecode;
   sqlda.sqlfmtarr = sqlfmts;
  }

I: {
   sqlda = (IISQLDA *)calloc(1, IISQDA_HEAD_SIZE +
                (NumberOfRecode * IISQLDA_VAR_SIZE));
   if (sqlda == (IISQLDA *)0)
   {
    printf("\nruntime error: memory allocation error in\n");
    printf(" ALLOCATE SQLDA_SQLFMTARR statement. \n");
```

```
  exit(0);
 }
 sqlda->sqln = NumberOfRecode;
}
```

Parameters

`NumberOfRecode` specifies the number of the records for format array. When you dynamically execute a query, the column information will be stored in format array in which one column will take a record.

Descriptions

• If your application program wants to deal with dynamic queries, you must use this statement to allocate memory space to hold column information.

• This statement can be embedded in anywhere a C statement can be placed. Mostly, you use this statement just before you execute DESCRIBE command.

• The format array must be allocated in size of which it can store all columns (in one row) returned by the FETCH command.

• After you use the format array, you must free the space allocated by this statement with FREE SQLDA_SQLFMTARR statement.

Notes

• ALLOCATE SQLDA_SQLFMTARR statement is a newly defined statement in RE/SQL, which is to standardize the variable declarations and allocations for dealing with dynamic queries.

• See "Dynamic Operations" for more information.

3.4 Begin Declare Section

The BEGIN DECLARE SECTION statement indicates the beginning of the host variable declaration section in an application program.

Syntaxes

S: EXEC SQL BEGIN DECLARE SECTION;

A: exactly the same as S syntax

I: exactly the same as S syntax

Parameters

None.

Descriptions

• All host variables which are used in an application program must be declared in declaration section(s).

• A single application program can have multiple declaration sections.

• This command is used in conjunction with END DECLARE SECTION command, which marks the end of declaration section.

Notes

None.

3.5 Begin Work

Begins a transaction.

Syntaxes

S: EXEC SQL BEGIN WORK;

A: if (read_lock_mode == NOLOCK)

EXEC SQL BEGIN WORK RU;

else if (read_lock_mode == SHARED)

EXEC SQL BEGIN WORK RR;

I: /* EXEC SQ BEGIN WORK; */

Parameters

None.

Descriptions

• The BEGIN WORK command starts a transaction. Because you can have only one active transaction at a time, you can issue this command only after the following command: CONNECT, COMMIT WORK, ROLLBACK WORK, and SET READLOCK.

• When you issue BEGIN WORK command, a read lock mode is set at the same time, which can be Share Lock (default) or Null Lock. When the first read operation is performed in this transaction, a read lock will be granted by SQL based on the read lock mode. The read lock mode can be set with SET READLOCK command, which can be issued only before any BEGIN WORK command.

• To end a transaction, you must issue a COMMIT WORK, or ROLLBACK WORK, or DISCONNECT commands Otherwise, all locks obtained during the transaction will be hold until you terminate the program execution.

Notes

• For INGRES/SQL, there is no equivalent command of BEGIN WORK, because INGRES/SQL will automatically issue a BEGIN WORK command after CONNECT, COMMIT WORK, and ROLLBACK WORK, which are the places you mostly issue this command. Therefore, RE/SQL translator provides only a C comment statement at where the BEGIN WORK command you issued appears.

• For ALLBARE/SQL, this command can be optionally used to set isolation level (readlock). There are four isolation levels, they are RR, CS, RC, and RU. Only RR (Share lock) and RU (Null lock) can find corresponding readlocks in INGRES/SQL which can be set by SET LOCKMODE command. To eliminate the differences existed, RE/SQL translator automatically declare and use a global C variable, which is `read_lock_mode`, to record the states of the readlocks. This variable can be set by SET READLOCK command which is defined and used only in RE/SQL. And based on this variable, RE/SQL will provide parameters to BEGIN WORK command. See "Concurrence Control though Locks" for details about the topic of locking.

3.6 Close

The CLOSE command is used in an application program to close an open cursor.

Syntaxes

S: EXEC SQL CLOSE *CursorName*;

A: exactly the same as S syntax

I: exactly the same as S syntax

Parameters

CursorName designates the open cursor to be closed.

Descriptions

• CLOSE has no effect if the cursor is not in the open state.

• When you close a cursor, it leaves the open state, its active set becomes undefined, and it can no longer be used in DELETE, FETCH, or UPDATE command. To use the cursor again you must issue an OPEN command to reopen it.

Notes

None.

3.7 Commit Work

Terminates the current transaction.

Syntaxes

S: EXEC SQL COMMIT WORK;

A: exactly the same as S syntax

I: exactly the same as S syntax

Parameters

None.

Descriptions

• The COMMIT WORK command has no effect if you do not have a transaction in progress.

• Once committed, the transaction cannot be aborted, and all changes it made become visible to all users through any statement that manipulates that data.

• The COMMIT WORK command releases all locks held by the transaction, and closes all cursors opened in current transaction.

• Short transactions (frequent COMMIT WORK commands) are recommended to improve concurrence.

Notes

• In both INGRES/SQL and ALLBARE/SQL, the COMMIT WORK command is full compatible. Therefore, the RE/SQL translator will not process this command except syntax checking.

• In ALLBARE/SQL, there is an option parameter in this command, which is RELEASE. This function is included in DISCONNECT command.

3.8 Connect

Establishes a connection between a application and a database.

Syntaxes

S: EXEC SQL CONNECT TO {'*DatabaseName*' | :*HostVariable*};

A: exactly the same as S syntax

I: EXEC SQL CONNECT {'*DatabaseName*' | :*HostVariable*};

Parameters

`DatabaseName` identifies the database (or DBEnvironment for ALLBARE/SQL) to be used. Any path name you specify, unless absolute, is assumed relative to your current working directory.

`HostVariable` identifies a character string host variable containing the name of a database.

Descriptions

• Database is a physical and/or logical data storage on disks, which is created by database administrator. See related reference manuals for information about creating a database. (the term of "database" is equivalent to "DBEvironment" in ALLBARE/SQL).

• The database is supposed that it can be accessed by multiple users. This feature could be affected by the way you create the database, see notes.

• A database session is the period between establishing and terminating a connection to a database by CONNECT command. You must be in a database session to execute any of the SQL commands except the CONNECT command. The CONNECT command must be executed at least and only once during the execution of the application program.

• The database could be connected by more than one SQL application programs at the same time. In this case, the data concurrence control must be considered by the users while developing their applications. See "Concurrence Control through Locks" for details.

Notes

• For the CONNECT statement in RE/SQL, the database name is no longer allowed to contain path, because INGRES/SQL does not support path in the database name. In this case, you have to use OPTION clause to solve this problem. For example, in a RE/SQL application, if the object SQL product is ALLBARE/SQL, the database environment name should be `"/disk2/rodos/rodb/DB1/ResyDBE"`, if the object SQL

product is INGRES/SQL, the database name should be "ResyDBE", in this case you can use following statement:

```
EXEC SQL CONNECT TO OPTION

        'A'=<<'/disk2/rodos/rodb/DB1/ResyDBE'>>,

        'I'=<<'ResyDBE'>>;
```

or:

```
/* Allbase_name and Ingres_name are SQL host variables */

strcpy(Allbase_name, "/disk2/rodos/rodb/DB1/ResyDBE");

strcpy(Ingres_name, "ResyDBE");

EXEC SQL CONNECT TO OPTION 'A'=<<:Allbase_name>>,

        'I'=<<:Ingres_name>>;
```

• INGRES/SQL can support multi-session, that means that more than one sessions can be opened with in one application program. ALLBARE/SQL does not support this feature. Therefore, the related session management options have been removed from RE/SQL.

• For ALLBARE/SQL, you must create you database environment with multi-user configuration, i.e. you must use the following command to create your database environment:

```
START DBE 'DBEnvironmentName' MULTI NEW;
```

See related references manual for details about creation of database.

• You can not use CONNECT command in dynamic operations.

Examples

• The following command connect the program to a database:

```
CONNECT TO 'TestDB';
```

Before you can execute your program, you must make the database available for multi-user mode.

• For ALLBARE/SQL, if you want to enable your database to be used in multiple user mode, you must create you database environment with multiuser configuration by means of ISQL:

```
START DBE 'TestDB' MULTI NEW;
```

• For INGRES/SQL, you can use following INGRES operating command to create database:

```
CREATEDB TestDB;
```

• The following command connect the program to a database, if the connecting is failure, you can get an non-zero SQLCA_SQLCODE:

```
CONNECT TO 'TestDB';

if (SQLCA_SQLCODE != 0)

{

 /* error handling program */

}
```

3.9 Convert after query

Convert date value in a host variable, after a query is executed, to RE/SQL format.

Syntaxes

S: EXEC SQL CONVERT AFTER QUERY FROM :*HostVariable*

TO *C_Variable* WITH {*DATE* | *DATETIME* | *INTERVAL*};

A: if (SESQL_tanslate_date(*HostVariable, C_Variable,*

FROM_{*DATE* | *DATETIME* | *INTERVAL*}, 'A') != 0)

SQLCA_SQLCODE = -2222;

I: if (SESQL_tanslate_date(*HostVariable, C_Variable,*

FROM_{*DATE* | *DATETIME* | *INTERVAL*}, 'I') != 0)

SQLCA_SQLCODE = -2222;

Parameters

HostVariable identifies a character string SQL host variable containing the date value string obtained from a query.

C_Variable identifies a character string C variable to hold the result of the date value string in the format of RE/SQL.

DATE identifies the value in the HostVariable is obtained from a DATE data type column.

DATETIME identifies the value in the HostVariable is obtained from a DATETIME data type column.

INTERVAL identifies the value in the HostVariable is obtained from a INTERVAL data type column.

Descriptions

• To unify the format for DATE data type, you need to use this statement to convert the date string, which is obtained from a query, from corresponding SQL product format to RE/SQL format.

• In this statement, HostVariable is source string, which contains string to be converted. C_Variable is destination string, which holds result of conversion for later processing in your application.

• The conversion preformed by this statement is based on the syntax for date format which is included in **`syntax.file`**. See "The Software Development for the RODOS Embedded SQL".

• When you use this statement, you must pay some attention to error handling. When error occurs in this statement, a special error code -2222 will be generated and assigned to SQLCA_SQLCODE. In this case, instead of using SQLEXPLAIN to retrieve error message, you can find a error message in a C string variable **`DynamicPreprocessMessage`** which is declared in RE/SQL header file.

• See also "The Reference Manual for the RODOS Embedded SQL" about date conversion.

Example

```
/* first, use the host variable as an output variable */

EXEC SQL SELECT date_column

  INTO :date_variable

  FROM TestTable

/* same time, use date constant in search condition */

 WHERE date_column={1993-08-16};

/* and then, use CONVERT AFTER QUERY statement to convert      internal format of date data type to
the format defined in      RE/SQL */

EXEC SQL CONVERT AFTER QUERY

          FROM :date_variable TO user_variable WITH DATE;

if (SQLCA_SQLCODE == -2222)

{

   printf("data conversion error\n");
```

```
    exit(0);
}
```

Notes

• CONVERT AFTER QUERY is a newly defined statement in RE/SQL, which enables the application developers to use a unified DATE data format.

• This statement will be translated to some C statement, from which a convert function will be called. Therefore, when you are using this statement in your applications, you have to link your application to a pre-developed C function library which file name is **dynamic.o**.

3.10 Convert before query

Convert date value in a C string or a C string variable, before a query is executed, to the format of corresponding SQL product.

Syntaxes

S: EXEC SQL CONVERT BEFORE QUERY FROM {*C_Variable* | "*String*"}

TO :*HostVariable* WITH {*DATE* | *DATETIME* | *INTERVAL*};

A: if (SESQL_tanslate_date({*C_Variable* | "*String*"},

HostVariable, TO_{*DATE* | *DATETIME* | *INTERVAL*}, 'A') != 0)

SQLCA_SQLCODE = -2222;

I: if (SESQL_tanslate_date({*C_Variable* | "*String*"},

HostVariable, TO_{*DATE* | *DATETIME* | *INTERVAL*}, 'I') != 0)

SQLCA_SQLCODE = -2222;

Parameters

C_Variable	identifies a character string C variable containing the date value string in the format of RE/SQL.
String	identifies a constant date value string in the format of RE/SQL.
HostVariable	identifies a character string SQL host variable to hold the result of the date value string in the format of corresponding SQL product.

DATE	identifies the value in the HostVariable is obtained from a DATE data type column.
DATETIME	identifies the value in the HostVariable is obtained from a DATETIME data type column.
INTERVAL	identifies the value in the HostVariable is obtained from a INTERVAL data type column.

Descriptions

• To unify the format for DATE data type, you need to use this statement to convert the date string, which will be used in a query, from RE/SQL format to corresponding SQL product format.

• In this statement, C_Variable or String is source string, which contains string to be converted. HostVariable is destination string, which holds result of conversion for use in following query.

• The conversion preformed by this statement is based on the syntax for date format which is included in **syntax.file**. See "The Software Development for the RODOS Embedded SQL".

• When you use this statement, you must pay some attention to error handling. When error occurs in this statement, a special error code -2222 will be generated and assigned to SQLCA_SQLCODE. In this case, instead of using SQLEXPLAIN to retrieve error message, you can find a error message in a C string variable **DynamicPreprocessMessage** which is declared in RE/SQL header file.

• See also "The Reference Manual for the RODOS Embedded SQL" about date conversion.

Example

```
/* first, use CONVERT BEFORE QUERY statement to assign
  date constant to host variable*/
EXEC SQL CONVERT BEFORE QUERY
        FROM "1993-08-16" TO :date_variable WITH DATE;
if (SQLCA_SQLCODE == -2222)
{
  printf("data conversion error\n");
  exit(0);
}
/* and then, use the host variable as an input variable */
EXEC SQL INSERT INTO TestTable (date_column)
```

VALUES (:date_variable);

Notes

• CONVERT BEFORE QUERY is a newly defined statement in RE/SQL, which enables the application developers to use a unified DATE data format.

• This statement will be translated to some C statement, from which a convert function will be called. Therefore, when you are using this statement in your applications, you have to link your application to a pre-developed C function library which file name is `dynamic.o`.

3.11 CREATE INDEX

The CREATE INDEX command creates an index on one or more columns of an existing base table.

Syntaxes

S: EXEC SQL CREATE [UNIQUE] INDEX *IndexName*

ON *TableName* ({*ColumnName* [,...]});

A: exactly the same as S syntax

I: exactly the same as S syntax

Parameters

`UNIQUE`	prohibits duplicates in the index. If UNIQUE is specified, each possible combination of index key column values can occur in only one row of the table. If UNIQUE is omitted, duplicate values are allowed. Because all null values are equivalent, a unique index allows only one row with a null value in an indexed column. When you create a unique index, all existing rows must have unique values in the indexed column(s).
`IndexName`	is the name of the index to be created. A table cannot have two indexes with the same name.
`TableName`	designates the table for which an index is to be created.
`ColumnName`	is the name of a column to be used as an index key. You can specify up to 16 columns in order from major index key to minor index key.

Descriptions

• If the table does not contain any rows, the CREATE INDEX command simply enters the definition of the index in the system catalog. If the table has rows, the CREATE INDEX command enters the definition in the system catalog and builds an index on the existing data.

If the UNIQUE option is specified and the table already contains rows having duplicate values in the index key columns, the CREATE INDEX command is rejected.

If you attempt an INSERT or UPDATE which violates the uniqueness constraint of an index created on the table, then the INSERT or UPDATE will fail.

• The index created is maintained automatically by ALLBARE/SQL until the index is deleted by a DROP INDEX command or until the table it is associated with is dropped.

• Indexes cannot be created for views.

• Index entries are sorted in ascending order. Null compare higher than other values for sorting.

Notes

• In different SQL implementations, the basic elements of this command are compatible, but many options do not.

• A WITH clause is not included in RE/SQL, which is supported by INGRES/SQL.

• A CLUSTERING and ASC | DESC options are not included in RE/SQL, which are supported by ALLBARE/SQL.

3.12 Create Table

The CREATE TABLE command defines a table.

Syntaxes

S: EXEC SQL CREATE TABLE *TableName*

({*ColumnDefinition*} [,...]);

A: EXEC SQL CREATE PUBLIC TABLE *TableName*

({*ColumnDefinition*} [,...]);

I: exactly the same as S syntax

Parameters

TableName is the name of the table to be created.

ColumnDefinition defines an individual column in a table. Each table must have at least one column. The syntax for a CREATE TABLE column definition is presented below:

ColumnName ColumnDataType [**WITH NULL** | NOT NULL]

ColumnName is the name to be assigned to one of the columns in the new table. No two columns in the table can be given the same name.

*ColumnDataType*indicates what type of data the column can contain. Some data types require that you include a length. The following table shows what data type you can specify here:,

TABELLE 14 RE/SQL Data Type Translation[a]

character	CHAR(1 - 2000) VARCHAR(1 - 2000)	CHAR(1 - 2000) VARCHAR(1 - 2000)	CHAR(1 - 2000) VARCHAR(1 - 2000)
numeric	INTEGER SMALLINT REAL DOUBLE	INTEGER SMALLINT REAL DOUBLE PRECISION	INTEGER SMALLINT REAL FLOAT8
date/time	DATE DATETIME INTERVAL	DATE DATETIME INTERVAL	DATE DATE DATE

a. See also Table 1 in "Data Types" section for more information.

The table also lists the corresponding data type for ALLBARE/SQL and INGRES/SQL, according to which the RE/SQL translator will translate the data type you specify here to the corresponding data type for the specified SQL product.

WITH NULL means the column accepts null values. If no value is supplied by the user, a null value will be inserted. If neither WITH NULL nor NOT NULL is specified, WITH NULL is used as default. This syntax is compatible with INGRES/SQL. For ALLBARE/SQL, this key word will be translated to DEFAULT NULL by RE/SQL translator.

NOT NULL means the column cannot contain null values. If NOT NULL is specified, any command that attempts to place a null value in the column will be rejected.

Descriptions

• A table can have maximum of 255 columns; a table row can be a maximum of 2000 bytes wide.

• A VARCHAR column requires two bytes in addition to its declared length, to store the length of the string. Nullalbe columns require one additional byte to store the null indicator.

Notes

• This command is one of the most incompatible command, many features in this command have to be removed.

• For INGRES/SQL, a WITH clause is removed.

• For ALLBARE/SQL, Locking control, unique constraint, default value specification, and storage structure control are removed.

• For ALLBARE/SQL, PUBLIC locking strategy is used to make the table accessible by multi users, which is the default case for INGRES/SQL.

• For most of the cases, if necessary, the parameters removed will be provided with default value. Please reference to related manuals for more information.

3.13 Create View

The CREATE VIEW command creates an view of a table, another view, or a combination of tables and views. CREATE VIEW expands an asterisk (*) in a select list into a list of column names. an entry is created in the system catalog view containing the view definition

Syntaxes

S: EXEC SQL CREATE VIEW *ViewName* ({*ColumnName*} [,...])

AS *SubSelect*;

A: exactly the same as S synta

I: exactly the same as S syntax

Parameters

ViewName	is the name of the view to be created.
ColumnName	specifies the names to be assigned to the columns of the new view. The names are specified in an order corresponding to the columns of the query result produced by the subquery. You must specify the column names if any column of the query result is defined by a computed expression, aggregate function, or constant in the select list of the query result was obtained by joining to tables that have a column of the same name or if the expansion of an * in the select list results in duplicate column names. If you do not specify column names, the columns of the view are assigned the same names as the columns from which they are derived. The * is expanded into the appropriate list of column names provided no duplicates are found.

SubSelect is a SELECT statement from which the view is derived. The query may refer to tables or views or a combination of tables and views. The SELECT statement used in this command may not include any UNION or UNION ALL operations.

Descriptions

• You use a view in any SQL manipulation statement as the same as you use a table. However, if you want use DELETE, INSERT, or UPDATE command on a view, the view must be updatable. A view is updatable only if the subselect from which it is derived matches the following updatablility criteria:

- No DISTINCT, GROUP BY, or HAVING clause is specified in the subselect, and no aggregate function appears in the select list.

- The FROM clause specifies exactly one table, which must be an updatable table.

- To use INSERT and UPDATE commands through views, the select list in the view definition must not contain any arithmetic expressions. It must contain only column names.

- For DELETE WHERE CURRENT and UPDATE WHERE CURRENT commands operating on cursors defined with views, the views definition must not contain subqueries.

- For non-cursor UPDATE, DELETE, and INSERT commands, the view definition must not contain any subqueries which contain in either FROM clause a table reference to the same table as the outermost FROM clause.

• You cannot define a index on a view.

• You cannot use host variables in the CREATE VIEW command.

• You cannot use an ORDER BY clause when defining a view.

• The index created is maintained automatically by ALLBARE/SQL until the index is deleted by a DROP INDEX command or until the table it is associated with is dropped.

• Indexes cannot be created for views.

• Index entries are sorted in ascending order. Null compare higher than other values for sorting.

Notes

• In different SQL implementations, the basic elements of this command are compatible, but many options do not.

• In INGRES/SQL, the subquery may include UNION operation, but may not in ALLBARE/SQL.

• A WITH CHECK OPTION is not included in RE/SQL, which are supported by INGRES/SQL.

• In INGRES/SQL, you can use host variables in some places. In ALLBARE/SQL, you cannot use host variables in any places.

3.14 Declare Cursor

The DECLARE CURSOR command associates a cursor with a specified SELECT command. A cursor is a pointer you use in an application program to indicate the row in the set of rows retrieved with the SELECT command on which you want to operate.

Syntaxe

S: EXEC SQL DECLARE *CursorName* CURSOR FOR

{*SelectCommand* | *CommandName*}

[FOR UPDATE OF {*ColumnName*} [,...]];

A: exactly the same as S syntax

I: exactly the same as S syntax

Parameters

`CursorName`	is the name to be assigned to the newly declared cursor. Two cursors in an application program cannot have the same name. The cursor name must conform to the SQL syntax rules for a basic name.
`SelectCommand`	is a SELECT command which determines the rows and columns to be processed by means of the cursor. The rows defined by the query expression when you open the cursor are called the active set of the cursor.
`CommandName`	is specified when declaring a cursor for use with a SELECT command dynamically preprocessed earlier with a PREPARE command.
`FOR UPDATE OF`	specifies the column or columns which may be updated using this cursor. The order of the column names is not important. The column(s) to be updated need not appear in the select list of the SELECT command. Do not use the DISTINCT option or an ORDER BY or GROUP BY clause in the SELECT command if you use a FOR UPDATE clause.

Descriptions

• A cursor must be declared before you refer to it in the other cursor command.

• The active set is defined when you issue the OPEN command. You can operate on the rows in the active set with the FETCH, UPDATE WHERE CURRENT, and DELETE WHERE CURRENT commands.

• Use the FETCH command to position the cursor on the row you want to update or delete.

• Use the DELETE WHERE CURRENT command to delete a row in the active set.

• Use the UPDATE command with the CURRENT OF option to update columns; you can update the columns identified in the FOR UPDATE OF clause of the DECLARE CURSOR command. The restrictions that govern updating via a cursor are described below.

• Use the CLOSE command when you are finished operating on the active set.

• A cursor is said to be updatable when you can use it in DELETE WHERE CURRENT OF CURSOR or UPDATE WHERE CURRENT OF CURSOR commands to modify the base table. A cursor is updatable only if the query from which it is derived matches the following updatablility criteria:

- No ORDER BY, UNION, or UNION ALL operation is specified.

- No DISTINCT, GROUP BY, or HAVING clause is specified in the outermost SELECT clause, and no aggregate appears in its select list.

- The FROM clause specifies exactly one table, whether directly or through a view. If it specifies a table, the table must be an updatable table. If it specifies a view, the view definition must satisfy the cursor updatablility rules stated here.

- For the UPDATE WHERE CURRENT command, the select list in the cursor definition must not contain any arithmetic expressions. It must contain only column names.

- For DELETE WHERE CURRENT and UPDATE WHERE CURRENT commands, the SelectCommand parameter must not contain any subqueries or reference any view whose view definition contains a subquery.

Notes

None.

3.15 Declare SQLDA

The DECLARE SQLDA statement is used to declare some variables for SQLDA in an application program. It can be thought of a supplement of the INCLUDE SQLDA statement.

Syntaxes

S: EXEC SQL DECLARE SQLDA;

A: sqlformat_type *sqlfmts;

char *DataBuffer;

I: IISQLDA *sqlda;

Parameters

None.

Descriptions

• If your application program wants to deal with dynamic queries, you must use this statement to declare some necessary variables.

• This statement must be placed at just following statement of INCLUDE SQLDA.

Notes

• DECLARE SQLDA statement is a newly defined statement in RE/SQL, which is to standardize the variable declarations for dealing with dynamic queries.

• See "Dynamic Operations" for more

3.16 Delete

The DELETE command deletes a row or rows from table.

Syntaxes

S: EXEC SQL DELETE FROM *TableName* [WHERE *SearchCondition*];

A: exactly the same as S syntax

I: exactly the same as S syntax

Parameters

TableName designates a table from which any rows satisfying the search condition are to be deleted.

WHERE *SearchCondition*p specifies which rows are to be deleted. If not rows satisfy the search condition, the table is not changed. If the WHERE clause is omitted, all rows are deleted.

Descriptions

• If all rows of a table are deleted, the table is empty but continues to exist unless you issue a DROP TABLE command.

• If an error is detected during a DELETE command, the command will have no effect and no rows will have been deleted.

• The search condition is effectively executed for each row of the table or view before any row is deleted. If the search condition contains a subquery, each subquery in the search condition is effectively executed for each row of the table and the results used in the application of the search condition to the given row. If any executed subquery contains an outer reference to a column of the table, the reference is to the value of that column in the given row.

Notes

• This command is basically compatible in ALLBARE/SQL and INGRES/SQL, but some differences still exist. ALLBARE/SQL can delete rows through a view, which function is not supported by INGRES/SQL. The correlation name can be used in INGRES/SQL but not in ALLBARE/SQL. These functions are removed from RE/SQL.

3.17 Delete where current

The DELETE WHERE CURRENT command deletes the current row of an active set. The current row is the row pointed to by a cursor after the FETCH command is issue.

Syntaxes

S: EXEC SQL DELETE FROM *TableName* WHERE CURRENT OF *CursorName*;

A: exactly the same as S syntax

I: exactly the same as S syntax

Parameters

`TableName`	designates a table from which any rows satisfying the search condition are to be deleted.
`CursorName`	specifies the name of a cursor. The cursor must be open and positioned on a row of the table. The DELETE WHERE CURRENT command deletes this row, leaving the cursor with no current row. (The cursor is said to be positioned between the preceding and following rows of the active set). You cannot use the cursor for further updates or deletions until you reposition it using a FETCH command, or until you close and reopen the cursor.

Descriptions

• Although the SELECT command associated with the cursor may specify only some of the columns in a table, the DELETE WHERE CURRENT command deletes an entire row.

• The DELETER WHERE CURRENT command can be used on an active set associated with a cursor defined using the FOR UPDATE clause.

• If the cursor is not currently pointing at a row when the DELETE is executed, then SQL generates an error indicating the need to issue a FETCH statement to position the cursor on a row.

Notes

• This command is basically compatible in ALLBARE/SQL and INGRES/SQL, but some differences still exist. ALLBARE/SQL can delete rows through a view, which function is not supported by INGRES/SQL. In INGRES/SQL, some options in DECLARE CURSOR statement may affect the execution of this command, which are not supported by ALLBARE/SQL. Reference to DECLARE CURSOR command.

3.18 Describe

The DESCRIBE command is used in an application to get information about the result of a dynamic command such as a command preprocessed with the PREPARE command.

Syntaxes

S: EXEC SQL DESCRIBE *CommandName* INTO SQLDA;

A: exactly the same as S syntax

I: exactly the same as S syntax

Parameters

CommandName	identifies a dynamically preprocessed command to be executed in an application program. The command name corresponds to one specified in a previous PREPARE command.

Descriptions

• If the command name refers to a SELECT command, the DESCRIBE command returns the number of columns in the query result, along with each column's name, length, and data type. On the basis of this information, the program can parse column values in the query result from a data buffer declared in the program. The program reads the query result by associating the CommandName with a cursor and using the cursor manipulation commands (OPEN, FETCH, DELETE WHERE CURRENT, and CLOSE).

• If the command name does not refer to a SELECT command, DESCRIBE sets the SQLD field of the SQLDA data structure to zero.

Notes

• See "Dynamic Operations" for more information.

3.19 Disconnect

Terminates access to a database.

Syntaxes

S: EXEC SQL DISCONNECT;

A: EXEC SQL COMMIT WORK RELEASE;

I: exactly the same as S syntax

Parameters

None.

Descriptions

- Before terminating data access to the database, the current transaction is committed.
- Any locks still hold are released. Any cursors still open are closed.

Notes

- The related session management options have been removed from RE/SQL, because it does not support multiple session feature.
- When you terminate a connection, the strategy to commit (or rollback) the current transaction may differ in different SQL implementations. Therefore an option parameter is added to RE/SQL, which asks users explicitly to issue the strategy of ending the current transaction.
- You can not use DISCONNECT command in dynamic operations.

3.20 Drop Index

The DROP INDEX command deletes the specified index.

Syntaxes

S: EXEC SQL DROP INDEX *IndexName*;

A: exactly the same as S syntax

I: exactly the same as S syntax

Parameters

IndexName is the name of the index to be deleted.

Descriptions

None.

Notes

• A FROM clause in ALLBARE/SQL is not included in RE/SQL syntax.

3.21 Drop Table

The DROP TABLE command deletes the specified table.

Syntaxes

S: EXEC SQL DROP TABLE *TableName*;

A: exactly the same as S syntax

I: exactly the same as S syntax

Parameters

TableName is the name of the table to be deleted.

Descriptions

• Any indexes, views, permits, and integrities defined on that table are automatically dropped also.

Notes

None.

3.22 Drop View

The DROP VIEW command deletes the specified view.

Syntaxes

S: EXEC SQL DROP INDEX *ViewName*;

A: exactly the same as S syntax

I: exactly the same as S syntax

Parameters

ViewName is the name of the view to be deleted.

Descriptions

• This command does not affect the base tables on witch the views were defined.

• You cannot use this command on system views.

Notes

None.

3.23 End Declare Section

The END DECLARE SECTION statement indicates the end of the host variable declaration section in an application program.

Syntaxes

S: EXEC SQL END DECLARE SECTION;

A: exactly the same as S syntax

I: exactly the same as S syntax

Parameters

None.

Descriptions

• This command is used in conjunction with BEGIN DECLARE SECTION command, which marks the end of declaration section.

Notes

None.

3.24 Executer

The EXECUTE command executes a command that has been prepared for execution by the PREPARE command.

Syntaxes

S: EXEC SQL EXECUTE *CommandName*;

A: exactly the same as S syntax

I: exactly the same as S syntax

Parameters

CommandName	identifies a dynamically preprocessed command to be executed in an application program. The command name corresponds to one specified in a previous PREPARE command.

Descriptions

• It is recommended that you use this command when you need to execute a dynamic statement more than once in your program.

Notes

• See "Dynamic Operations" for more information.

3.25 Execute Immediate

The EXECUTE IMMEDIATE command dynamically prepares and executes a non-query SQL command.

Syntaxes

S: EXEC SQL EXECUTE IMMEDIATE {:*HostVariable* | *'String'*};

A: {

```
  if (dynamic_translator(DynamicCommandBuffer,
                         {HostVariable | "String"})
   EXEC SQL EXECUTE IMMEDIATE :DynamicCommandBuffer;
  else
   SQLCA_SQLCODE = -1111;
}
```

```
I: {
   if (dynamic_translator(DynamicCommandBuffer,
                    {HostVariable | "String"})
    EXEC SQL EXECUTE IMMEDIATE :DynamicCommandBuffer;
   else
    SQLCA_SQLCODE = -1111;
}
```

Parameters

HostVariable	identifies a host variable containing the SQL command to be executed.
String	identifies a SQL command string.

Descriptions

• If you know in advance that the command to be dynamically preprocessed is not a SELECT command, you can prepare it and execute it in one step with this command.

• It is recommended that you use this command only when you need to execute a dynamic statement just once in your program. To execute a dynamic statement more than once, it is better to use PREPRE and EXECUTE commands.

• When you use this statement, you must pay some attention to error handling. When error occurs in this statement, a special error code -1111 will be generated and assigned to SQLCA_SQLCODE. In this case, instead of using SQLEXPLAIN to retrieve error message, you can find a error message in a C string variable **DynamicPreprocessMessage** which is declared in RE/SQL header file.

• You can not use the EXECUTE IMMEDIATE command for any of the following commands:

ALLOCATE SQLDA_SQLFMTARR

ALLOCATE SQLDA_DATABUFF

BEGIN DECLARE SECTION

BEGIN WORK

BULK operations

CLOSE

COMMIT WORK

CONNECT

DECLARE CURSOR

DECLARE SQLDA

DELETE WHERE CURRENT

DESCRIBE

DISCONNECT

END DECLARE SECTION

EXECUTE

EXECUTE IMMEDIATE

FETCH

FREE SQLDA_SQLFMTARR

FREE SQLDA_DATABUFF

INCLUDE

SQLEXPLAIN

LOCK TABLE

UNLOCK TABLE

OPEN

PREPARE

ROLLBACK WORK

SAVEPOINT

SELECT

SET READLOCK

UPDATE WHERE CURRENT

WHENEVER

Notes

• A dynamic translator has been developed, which is a C function and its source code will be automatically embedded in application program by RE/SQL translator. In this way, the statements which are written in RE/SQL syntax can be dynamically translated to specified SQL syntax before they can be executed.

• See "Dynamic Operations" for more information.

3.26 Fetch

The FETCH command advances the position of an opened cursor to the next row of the active set and copies selected columns into the specified host variables. The row to which the cursor points is called the current row. The following two forms of the FTECH command are described individually:

• The form used to fetch one or multiple rows into host variables.

• The form used to fetch rows while using dynamic operations.

1. FETCH row(s) into host variables

Syntaxes

S: EXEC SQL [BULK] FETCH *CursorName* INTO *VariableList*;

A: exactly the same as S syntax

I: exactly the same as S syntax for without BULK option,

with BULK option:

```
{
  /* transfer data between Buffer and (Buffer_s, Buffer_i) */
  for (SQL_ii = 0; SQL_ii < NumberOfRows; SQL_ii++)
  {
    EXEC SQL FETCH CursorName
          INTO :Buffer_s[StartIndex+SQL_ii]
             :Buffer_i[StartIndex+SQL_ii];
    if (SQLCA_SQLCODE == 100)
```

```
    {
      NumberOfRows = SQL_ii;
    }
  }
  /* transfer data between (Buffer_s, Buffer_i) and Buffer */
}
```

Parameters

BULK	is specified in an application program to retrieve multiple rows with a single execution of the FETCH command. After a BULK FETCH command, the current row is the last row fetched.
CursorName	identifies a cursor. The cursor's active set, determined when the cursor was opened, and the cursor's current position in the active set determine the data to be returned by each successive FETCH command.
INTO	The INTO clause defines where to place rows fetched.
VariableList	identifies one or more host variables for holding columns of the row(s) returned in application program. The syntaxes for host variables may vary depending on if you use BULK option. The syntax for missing BULK option is as follows:

{:*HostVariable*[:*Indicator*]}[,...]

HostVariable	identifies the host variable corresponding to one column in the row. The number of the host variables must be the same as the number of the items in SelectList. If the SelectList is a "*", the number of the host variables must be the same as the number of the columns in specified table. And the order of the host variables must match the order of their corresponding items in the SelectList.
Indicator	names an indicator variable, an output host variable whose value (see following) depends on whether the host variable contains a null value: • 0 meaning the column's value is not null. • -1 meaning the column's value is null. • >0 meaning the column's value is truncated. The syntax for having BULK option is as follows:

{*:Buffer, :StartIndex, :NumberOfRows*}

`Buffer` is a host array of structure containing rows that are the result from the SELECT command. This array contains elements for each column to be returned and indicator variables for columns that can contain null values. Whenever a column can contain nulls, an indicator variable must be included in the array definition immediately after the definition of that column. This indicator variable is an integer that can have the following values:

>= 0 the variable's value is not null.

< 0 the variable's value is null. (generally, you use -1).

`StartIndex` is a host variable whose value specifies the array subscript denoting where the first row to be inserted is stored. You must specify this variable.

`NumberOfRows` is a host variable whose value specifies the number of rows to retrieve. You must specify this variable, the value of this variable is the smaller of two values; 1) the number of records in the array from the StartIndex to the end of the array, or 2) the number of rows in the query result.

Descriptions

• There must be a one-to-one correspondence between variables specified in the INTO clause of FETCH and expressions in the SELECT clause of the DECLARE CURSOR statement. IF the number of variables or structure elements for Bulk operation does not match the number of expressions, the warning will be generated.

• The variables listed in the INTO clause, or the structure elements if using BULK variable, must be type-compatible with the values being retrieved. If a result expression is Nullalbe, then the host variable that will receive that value must have an associated null indicator.

• If the statement does not fetch a row -- a condition that occurs after all rows in the set have been processed -- the SQLCA_SQLCODE is set to 100 (condition NOT FOUND) and no values are assigned to the variable.

Notes

• The BULK SELECT is not supported by INGRES/SQL, but can be simulated with a group of C and SQL statements. Reference to "Bulk Operations" section for more information about using the BULK operations.

2. FETCH rows for dynamic operation

Syntaxes

S: EXEC SQL FETCH *CrusorName* USING DESCRIPTOR SQLDA;

A: exactly the same as S syntax

I: exactly the same as S syntax

Parameters

CursorName	identifies a cursor. The cursor's active set, determined when the cursor was opened, and the cursor's current position in the active set determine the data to be returned by each successive FETCH command.
USING DESCRIPTOR	The USING DESCRIPTOR clause defines where to place rows selected in accord with a dynamically preprocessed SELECT command and described by a DESCRIBE command. You cannot use the BULK option if you employ the USING DESCRIPTOR clause.

Descriptions

• If the statement does not fetch a row -- a condition that occurs after all rows in the set have been processed -- the SQLCA_SQLCODE is set to 100 (condition NOT FOUND) and no values are assigned to the variable.

• The cursor identified by cursor name must be an open cursor.

FREE SQLDA_DATABUFF

The FREE SQLDA_DATABUFF statement is used to release memory space for data buffer in SQLDA, which is allocated by ALLOCATE SQLDA_DATABUFF.

Syntaxes

S: EXEC SQL FREE SQLDA_DATABUFF;

A: free(DataBuffer);

```
I: for (SQL_ii = 0; SQL_ii < sqlda->sqld; SQL_ii++)
  {
   free(sqlda->sqlvar[SQL_ii].sqldata);
   if (sqlda->sqlvar[SQL_ii]. sqltype < 0)
    free(sqlda->sqlvar[SQL_ii].sqlind);
  }
```

Parameters

None.

Descriptions

• After you use the data buffer, you must free the space by this statement.

Notes

• FREE SQLDA_DATABUFF statement is a newly defined statement in RE/SQL, which is to standardize the variable declarations and allocations for dealing with dynamic queries.

• See "Dynamic Operations" for more information.

3.27 Free SQLDA_SQLFMTARR

The FREE SQLDA_SQLFMTARR statement is used to release memory space for format array, which is allocated by ALLOCATE SQLDA_SQLFMTARR.

Syntaxes

S: EXEC SQL FREE SQLDA_SQLFMTARR;

A: free(sqlfmts);

I: free(sqlda);

Parameters

None.

Descriptions

• After you use the format array, you must free the space by this statement.

Notes

• FREE SQLDA_SQLFMTARR statement is a newly defined statement in RE/SQL, which is to standardize the variable declarations and allocations for dealing with dynamic queries.

• See "Dynamic Operations" for more information.

3.28 Include

The INCLUDE statement is used to declare SQLCA and/or SQLDA in an application program.

Syntaxes

S: EXEC SQL INCLUDE {SQLCA | SQLDA};

A: exactly the same as S syntax

I: exactly the same as S syntax

Parameters

SQLCA	is an area for SQL output messages concerning the status of each of SQL command.
SQLDA	is an area for use in conjunction with dynamic preprocessing of SELECT commands.

Descriptions

• You must always include the SQLCA in your SQL application program by using INCLUDE statement.

• If your application program wants to deal with dynamic queries, you must include the SQLDA by using INCLUDE statement.

Notes

None.

INCLUDE SQL_HEADER

The INCLUDE SQL_HEADER statement is used to include necessary macro definitions and variable declarations in an application program, which is used by RE/SQL translator.

Syntaxes

S: EXEC SQL INCLUDE SQL_HEADER;

A: #include "albase.macros.h"

```
EXEC SQL BEGIN DECLARE SECTION;
short save_point_0;
short save_point_1;
short save_point_2;
short save_point_3;
short save_point_4;
short save_point_5;
short save_point_6;
short save_point_7;
```

```
  short save_point_8;

  short save_point_9;

  char DynamicCommandBuffer[2500];

  EXEC SQL END DECLARE SECTION;
```

I: #include "ingres.macros.h"

```
  EXEC SQL BEGIN DECLARE SECTION;

  char DynamicCommandBuffer[2500];

  EXEC SQL END DECLARE SECTION;
```

Parameters

None.

Descriptions

• You must always include the SQL_HEADER in your SQL application program. This statement must be placed at where is only following the statements of including your system include files in your application program.

Notes

• INCLUDE SQL_HEADER is a newly defined statement in RE/SQL.

• "allbase.macros.h" is an including file for ALLBARE/SQL. "ingres.macros.h" is an including file for INGRES/SQL.

3.29 Insert

The INSERT command adds a row or rows to a table. The following three forms of the INSERT command are described individually:

• The form used to add a single row by specifying a value for each concerned column.

• The form used to add multiple rows by using BULK facility.

• The form used to add rows defined by a SELECT command. This form copies rows from one or more tables into a table.

1. INSERT single row

Syntaxes

S: EXEC SQL INSERT INTO *TableName* [(*ColumnName* [,...])]

VALUES (*ColumnValue* [,...]);

A: exactly the same as S syntax

I: exactly the same as S syntax

Parameters

TableName	designates a table to which a row is to be added.
ColumnName	specifies a column for which values are supplied.
VALUES	The VALUES clause specifies the values corresponding to the columns in the column name list. If no column name list exists, the clause specifies the values corresponding to the columns in CREATE TABLE command.
ColumnValue	provides column values when you insert a row. Column value must be a constant which is defined in "Expression" section.

Descriptions

• If you omit the column name list, there must be a one-by-one correspondence between the constants in the VALUES clause and the columns in the table which are specified when you create the table. That is, the VALUES list must have a value of appropriate data type for each column and the values must be listed in an order corresponding to the order of the columns in the table.

• If you omit any of the table's columns from the column name list, the INSERT command places the default value of the respective column definitions in the omitted columns. For columns with no default value, the null value is placed in the omitted columns. If the table definition specifies NOT NULL for any of the omitted columns, the INSERT command fails.

• If an error is detected during a INSERT command, the command will have no effect and no row will have been inserted.

Notes

• In INGRES/SQL, a expression can be used to specify a column value. That means that you can combine constants with +, -, *, /, operations. In ALLBARE/SQL, the expression can not be used to specify a column value. So, this feature is removed from RE/SQL.

• When you specify a column value, some constants defined by RE/SQL, which are included in { }, will be preprocessed by RE/SQL translator. These constants include DATE, DATETIME, INTERVAL, and some special constants, such as {now}, {today}, {user}, and {null}. If you use host variables to represent date constants, you must explicitly invoke a C conversion function before you invoke this command. See "Data Type" for details.

2. INSERT multiple rows with BULK facility

Syntaxes

S: EXEC SQL BULK INSERT INTO *TableName*

[(*ColumnName* [,...])]

VALUES (*:Buffer, :StartIndex, :NumberOfRows*);

A: exactly the same as S syntax

I:

```
{
  /* transfer data between Buffer and (Buffer_s, buffer_i) */
  for (SQL_ii = 0; SQL_ii < NumberOfRows; SQL_ii++)
  {
    EXEC SQL REPEATED INSERT INTO TableName
        [(ColumnName [,...])]
        VALUES(:Buffer_s[StartIndex+SQL_ii]
            :Buffer_i[StartIndex+SQL_ii]);
  }
}
```

Parameters

TableName	designates a table to which a row is to be added.
ColumnName	specifies a column for which values are supplied.
VALUES	The VALUES clause specifies the values corresponding to the columns in the column name list. If no column name list exists, the clause specifies the values corresponding to the columns in CREATE TABLE command.
Buffer	is a host array of structure containing rows that are the input for the INSERT command. This array contains elements for each column to be inserted and indicator variables for columns that can contain null values. Whenever a column can contain nulls, an indicator variable must be included in the array definition

immediately after the definition of that column. This indicator variable is an integer that can have the following values:

>= 0 the variable's value is not null.

< 0 the variable's value is null. (generally, you use -1).

StartIndex is a host variable whose value specifies the array subscript denoting where the first row to be inserted is stored. No default value is provided, you must specify this variable.

NumberOfRows is a host variable whose value specifies the number of rows to insert. No default value is provided, you must specify this variable.

Descriptions

• If you omit the column name list, there must be a one-by-one correspondence between the constants in the data structure and the columns in the table which are specified when you create the table. That is, the data structure must have a value of appropriate data type for each column and the values must be listed in an order corresponding to the order of the columns in the table.

• If you omit any of the table's columns from the column name list, the INSERT command places the default value of the respective column definitions in the omitted columns. For columns with no default value, the null value is placed in the omitted columns. If the table definition specifies NOT NULL for any of the omitted columns, the INSERT command fails.

• If an error is detected during a INSERT command, the command will have no effect and no row will have been inserted.

• For CHAR and VARCHAR data, if a string literal is shorter than the target column, it is padded with blanks; if it is longer than the target column, the string is truncated.

Notes

• The host variable structure can contain only constants. Some constants, such as DATE, DATETIME, and INTERVAL must be explicitly converted before it is assigned to host variable by using date function provided by RE/SQL See "Data Type" for details.

• The BULK INSERT is not supported by INGRES/SQL, but can be simulated with a group of C and SQL statements. Reference to "Bulk Operations" section for more information about using the BULK operations.

3. INSERT rows defined by a SELECT command

Syntaxes

S: EXEC SQL INSERT INTO *TableName* [(*ColumnName* [,...])]

SubQuery;

A: exactly the same as S syntax

I: exactly the same as S syntax

Parameters

TableName	designates a table to which a row is to be added.
ColumnName	specifies a column for which values are supplied.
SubQuery	defines the rows to be inserted based on one or more tables in the database. The name of the target table cannot appear within the FROM clause or in a FROM clause of any subquery. The query cannot contain an INTO clause or a UNION operation. The data types of each column in the select list must be compatible with the data types of corresponding columns in the target table. The first elect list item defines the first column in the target table, the second select list item defines the second column in the target table, and for forth. The number of select list items must equal the number of columns in the target table. Any column in the target table can contain null values only if it was not defined with the NOT NULL attribute. Therefore ensure either that select list items are not null for any NOT NULL target column, or that the NOT NULL target columns have default values defined for them.

Descriptions

• If you omit any of the table's columns from the column name list, the INSERT command places the default value of the respective column definitions in the omitted columns. For columns with no default value, the null value is placed in the omitted columns. If the table definition specifies NOT NULL for any of the omitted columns, the INSERT command fails.

• If an error is detected during a INSERT command, the command will have no effect and no row will have been inserted.

Notes

• Only those constants with { } which may appear in subquery is needed to be preprocessed by translator.

3.30 Lock Table

The LOCK TABLE command provides a means of explicitly acquiring a lock on a table to override the automatic locking provided by SQL.

Syntaxes

S: EXEC SQL LOCK TABLE *TableName* IN {SHARED | EXCLUSIVE} MODE;

A: exactly the same as S syntax

I: EXEC SQL SET LOCKMODE ON *TableName* WHERE LEVEL = TABLE,

READLOCK = {SHARED | EXCLUSIVE};

Parameters

TableName	specifies the table to be locked.
SHARED	allows other transactions to read but not change the table during the time you hold the lock.
EXCLUSIVE	prevents other transactions from reading or changing the table during the time you hold the lock.

Descriptions

• Of the two lock types described here, the highest level is exclusive (X), and the lowest level is shared (S). When you request a lock on an object which is already locked with a higher severity lock, the request is ignored.

• This command can be used to avoid the overhead of acquiring many small locks when scanning a table. For example, if you know that you are accessing all the rows of a table, you can lock the entire table at once instead of letting SQL automatically lock each individual page or row as it is needed.

• LOCK TABLE can be useful in avoiding deadlocks by locking tables in a predetermined order.

• You must issue a UNLOCK TABLE command explicitly to release the locks held after you terminate the transaction. Otherwise, the locks can be released automatically but the lock mode will still in the mode you specified in LOCK TABLE command.

• This command can be used only in one place, where is immediately after CONNECT TO, COMMIT WORK, ROLLBACK WORK, and BEGIN WORK (if you use this command to begin a transaction) command. You can not use this command in any on going transaction.

• This command must be used in pairs with UNLOCK TABLE command. If you issued a LOCK TABLE command before the transaction has begun, you must issue a UNLOCK TABLE command, immediately after the transaction ends (committed or rolled back).

Notes

• A SHARE UPDATE lock option is removed from ALLBARE/SQL. A NOLOCK lock option is removed from INGRES/SQL.

• The real time of granting the lock for specified table is different. For ALLBARE/SQL, the lock is granted at the same time while you issue LOCK TABLE command. For INGRES/SQL, the lock is not granted while you issue LOCK TABLE command. The lock will be granted only when you issue the first command which

has reading and/or writing operations. That is to say, this command dose not grant locks but only sets a lock mode. So, we recommend that you should issue a statement with reading or writing operation immediately after you issue LOCK TABLE command, because it is possible that you would fail to grant the lock on this table if any other user grants a lock on this table in advance

• Reference to UNLOCK TABLE command and "Concurrence Control through Locks".

3.31 Open

The OPEN command is used in an application program to open an open cursor, that is, make the cursor and its associated active set available to manipulate.

Syntaxes

S: EXEC SQL OPEN *CursorName*;

A: exactly the same as S syntax

I: exactly the same as S syntax

Parameters

`CursorName`	designates the open cursor to be opened. The cursor name must first be defined with a DECLARE CURSOR command.

Descriptions

• OPEN command executes the SELECT command specified when the cursor was declared, which determines the cursor's active set, and positions the cursor immediately before the first row returned. No rows are actually available to your application program until a FETCH command is executed.

• Updates are not permitted on cursors that involve joining two or more tables or cursors that involve sort operations.

• More than one cursor can be opened at one time during a session.

• All open cursors are automatically closed when a transaction terminates.

Notes

• A READONLY option is removed from INGRES/SQL syntax.

• A KEEP CURSOR option is removed from ALLBARE/SQL syntax.

3.32 Prepare

The PREPARE command dynamically preprocesses an SQL command for later execution.

Syntaxes

S: EXEC SQL PREPARE *CommandName* FROM {:*HostVariable* | *'String'*};

A: {

if (dynamic_translator(DynamicCommandBuffer,

{*HostVariable* | *"String"*})

EXEC SQL PREPARE *CommandName* FROM :DynamicCommandBuffer;

else

SQLCA_SQLCODE = -1111;

}

I: {

if (dynamic_translator(DynamicCommandBuffer,

{*HostVariable* | *"String"*})

EXEC SQL PREPARE *CommandName* FROM :DynamicCommandBuffer;

else

SQLCA_SQLCODE = -1111;

}

Parameters

CommandName	identifies a dynamically preprocessed command to be executed in an application program. The command name corresponds to one specified in a previous PREPARE command.
HostVariable	identifies a host variable containing the SQL command to be executed.
String	identifies a SQL command string.

Descriptions

• A command to be dynamically preprocessed in an application program must be terminated with a semicolon.

• You cannot prepare a command which contains host variables.

• In an application program, a dynamically preprocessed command is automatically deleted from the system at the end of the transaction in which it was prepared. It cannot be executed in any other transaction.

• When you use this statement, you must pay some attention to error handling. When error occurs in this statement, a special error code -1111 will be generated and assigned to SQLCA_SQLCODE. In this case, instead of using SQLEXPLAIN to retrieve error message, you can find a error message in a C string variable `DynamicPreprocessMessage` which is declared in RE/SQL header file.

• You can not use the PREPARE command for any of the following commands:

ALLOCATE SQLDA_SQLFMTARR

ALLOCATE SQLDA_DATABUFF

BEGIN DECLARE SECTION

BEGIN WORK

BULK operations

CLOSE

COMMIT WORK

CONNECT

DECLARE CURSOR

DECLARE SQLDA

DELETE WHERE CURRENT

DESCRIBE

DISCONNECT

END DECLARE SECTION

EXECUTE

EXECUTE IMMEDIATE

FETCH

FREE SQLDA_SQLFMTARR

FREE SQLDA_DATABUFF

INCLUDE

SQLEXPLAIN

LOCK TABLE

UNLOCK TABLE

OPEN

PREPARE

ROLLBACK WORK

SAVEPOINT

SET READLOCK

3.33 Update where current

WHENEVER

• You use the DESCRIBE command to determine whether the prepared command is a SELECT command. If the command is SELECT command, other information provided by the DESCRIBE command helps you determine how much storage to dynamically allocate for the query result; then you reference the command name in a DECLARE CURSOR command and use the cursor to execute the dynamically preprocessed command. If the command being preprocessed is not a SELECT command, you reference the command name in an EXECUTE command later in the current transaction to execute the dynamically preprocessed command.

Notes

• A dynamic translator has been developed, which is a C function and its source code will be automatically embedded in application program by RE/SQL translator. In this way, the statements which are written in RE/SQL syntax can be dynamically translated to specified SQL syntax before they can be prepared.

• See "Dynamic Operations" for more information.

ROLLBACK WORK

Aborts part or all of the current transaction.

Syntaxes

S: EXEC SQL ROLLBACK WORK [TO *:SavePointMarker*];

A: EXEC SQL ROLLBACK WORK [TO *:SavePointMarker* (host variable)];

I: EXEC SQL ROLLBACK WORK [To *SavePointMacro* (literal string)];

Parameters

TO	The TO clause is used to roll back to a savepoint without ending the current transaction. If the TO clause is omitted, ROLLBACK WORK ends the current transaction and undoes any changes that have been made in the transaction.
SavePointMarker	is a save point marker, which is defined by RE/SQL, to specify a savepoint that you issued in a SAVEPOINT command. See SAVEPOINT command for details about the save point marker.

Descriptions

• When you omit the TO clause, all changes you have made to the database since the most recent BEGIN WORK command are undone. In an application program, all open cursors are automatically closed, and all locks held by the transaction are released. Any savepoint defined in the transaction are lost and become invalid. The transaction is ended.

• The TO clause may not be used if any cursor were opened in this transaction. Issuing a ROLLBACK WORK to a savepoint in this case results in an error, and no rollback is done.

• When you specify the TO clause, all changes you have made to the database since the designated savepoint are undone. And the save point marker used must be appear in a SAVEPOINT statement before. In an application program, all open cursors are automatically closed, all locks obtained since the savepoint was set are released (for ALLBARE/SQL), or no locks are released (for INGRES/SQL). Any savepoint defined more recently than the designated savepoint are lost and become invalid. The designated savepoint is still valid and can be specified in a future ROLLBACK WORK command. The transaction is not ended.

Notes

• In both INGRES/SQL and ALLBARE/SQL, the ROLLBACK WORK command is full compatible, if you do not specify the TO clause. The savepoint is implemented in different ways in ALLBARE/SQL and INGRES/SQL. The problem concerned could be solved by RE/SQL translator, which provides a group pre-defined macro to the users. See SAVEPOINT command for details.

• In ALLBARE/SQL, there is an option parameter in this command, which is RELEASE. This function is included in DISCONNECT command.

SAVEPOINT

Declares a savepoint marker within the current transaction.

Syntaxes

S: EXEC SQL SAVEPOINT *:SavePointMarker*;

A: EXEC SQL SAVEPOINT *:SavePointMarker* (host variable of short);

I: EXEC SQL SAVEPOINT *SavePointMarker* (macro of literal string);

Parameters

SavePointMarker is a save point marker, which is defined by RE/SQL in different forms, to specify a savepoint that you issued in a SAVEPOINT command.

There is a big conflict between ALLBARE/SQL and INGRES/SQL. In ALLBARE/SQL, a savepoint must be issued with a integer host variable. In INGRES/SQL, a savepoint must be specified by a literal string instead of a host variable. To solve this problem in RE/SQL, some pre-defined save point markers, shown in Table 2, have to be used.

TABELLE 15 Pre-defined Savepoint Markers

SAVE_POINT_0	:SAVE_POINT_0	SAVE_POINT_0
SAVE_POINT_1	:SAVE_POINT_1	SAVE_POINT-1
...	...	...
SAVE_POINT_9	:SAVE_POINT_9	SAVE_POINT_9

a. corresponding host variables are declared in INCLUDE SQL_HEADER statement.

b. the markers are considered as literal strings.

You must use one or some of these macros to specify savepoint(s) instead of any other kind identifier, such as host variable, number, literal string. There is no special order requirement when use these macros.

The number of the markers pre-defined is 10, which is default number used by RE/SQL. If this number of savepoints are not enough for you application program, you have to add some more marker definitions by yourself in the file of 'syntax.file' about INCLUDE SQL_HEADER statement.

Descriptions

• Specify the savepoint marker in the TO clause of a ROLLBACK WORK command to roll back to a specified savepoint.

• Do not assign any value to the save point markers, it makes no sense.

• You must use a semi-colon prefixing the markers, though they are not considered as host variables here. The RE/SQL translator will make a proper processing.

Notes

None.

Example

```
BEGIN WORK;

command-1;

SAVEPOINT SAVE_POINT_1;

command-2;

command-3;

ROLLBACK WORK TO SAVE_POINT_1;

/* work of command-2 and command-3 is undone. */

command-4;

COMMIT WORK;

/* work of command-1 and command-2 is committed; */

/* transaction ends. */
```

3.34 Select

The SELECT command retrieves data from one or more tables. The retrieved data is presented in the form of a table, called the result table or query result. A SELECT command is a syntactically complete SQL statement containing one or more SELECT statements bur having a single query result, it can be used either as an individual command or as a subquery in another SELECT statement or other statements:

• When you use SELECT command as an individual command, the following cases are possible:

• A simple use without BULK option, which can retrieve only one row as query result.

• A full use with BULK option, which can obtain multiple rows as query result.

• A full use within dynamic operations.

• When you use SELECT command with other statement, the following cases are possible:

• A SELECT command can be used as part of other statements, such as INSERT and DECLARE CURSOR.

• A SELECT command can be used as a subquery within a search conditions in some statements, which have a WHERE or HAVING clause, such as DELETE, UPDATE and SELECT itself.

Syntaxes

S: EXEC SQL [BULK] SELECT [ALL | DISTINCT] *SelectList*

INTO *VariableList*

FROM *TableList*

[WHERE *SearchCondition1*]

[GROUP BY *ColumnName* [,...]]

[HAVING *SearchCondition2*]

[UNION [ALL] *FullSelect*]

[ORDER BY *ColumnID* [ASC | DESC] [,...]];

A: exactly the same as S syntax

I: exactly the same as S syntax for without BULK option,

with BULK option:

{

/* transfer data between Buffer and (Buffer_s, Buffer_i) */

SQL_ii = *StartIndex*;

EXEC SQL REPEATED SELECT [ALL | DISTINCT] *SelectList*

INTO *:Buffer_s*[SQL_ii]*:Buffer_i*[SQL_ii]

FROM *TableList*

[WHERE *SearchCondition1*]

[GROUP BY *ColumnName* [,...]]

[HAVING *SearchCondition2*]

[UNION [ALL] *FullSelect*]

[ORDER BY *ColumnID* [ASC | DESC] [,...]];

EXEC SQL BEGIN;

SQL_ii++;

```
  if ((SQL_ii - StartIndex) == NumberOfRows)
  {
    EXEC SQL ENDSELECT;
  }
  EXEC SQL END;
  /* transfer data between (Buffer_s, Buffer_i) and Buffer */
  NumberOfRows = SQL_ii - StartIndex;
}
```

Parameters

`BULK` is specified in an application program to retrieve multiple rows with a single execution of the SELECT statement.

Do not use this option with any other statements. Do not use this option in dynamic query. When you use this option, the host variable in INTO clause must be compatible, see syntax for INTO clause later in this section. See "Bulk Operations" for details.

`ALL` prevents elimination of duplicate rows from the result. The ALL option is assumed as default option.

`DISTINCT` ensures that each row in the query result is unique. All null values are considered equal. You cannot specify this option if the select list contains an aggregate function with DISTINCT in the argument.

`SelectList` tells how the columns of the result table are to be derived. The syntax of SelectList is presented below:

{ * | *Table.** | *CorrelationName.** | *Expression*

| [*Talbe.*]*ColumnName* | *CorrelationName.ColumnName*}[,...]

`*` includes, as columns of the result table, all columns of all tables specified in the FROM clause.

`Table.*` includes all columns of the specified table in the result.

`CorrelationName.``*`includes all columns of the specified table in the result. The correlation name is a synonym for the table as defined in the FROM clause.

Expression produces a single column in the result table; the result column values are computed by evaluating the specified expression for each row of the result table.

The expression can be of any complexity. For example, it can simply designate a single column of one of the tables specified in the FROM clause, or it can involve aggregate functions, multiple columns, and so on. When you specify one or more aggregate functions in a select list, the only other entity you can specify is the name(s) of the column(s) you group by.

*Table.ColumnName*includes a particular columns from the named table.

Correlation. includes a specified columns from the table whose correlation

ColumnName name is defined in the FROM clause.

INTO The INTO clause defines host variables for holding row(s) returned in application program. Do not use this clause for SELECT statements associated with a cursor or dynamically preprocessed SELECT statements, subqueries, or any but first SELECT block in a SELECT statement.

VariableList identifies one or more host variables for holding columns of the row(s) returned in application program. The syntaxes for host variables may vary depending on if you use BULK option.

The syntax for missing BULK option is as follows:

{:*HostVariable*[:*Indicator*] }[,...]

HostVariable identifies the host variable corresponding to one column in the row. The number of the host variables must be the same as the number of the items in SelectList. If the SelectList is a "*", the number of the host variables must be the same as the number of the columns in specified table. And the order of the host variables must match the order of their corresponding items in the SelectList.

Indicator names an indicator variable, an output host variable whose value (see following) depends on whether the host variable contains a null value:

- 0 meaning the column's value is not null.
- -1 meaning the column's value is null.
- >0 meaning the column's value is truncated.

The syntax for having BULK option is as follows:

{*:Buffer, :StartIndex, :NumberOfRows*}

Buffer is a host array of structure containing rows that are the result from the SELECT command. This array contains elements for each column to be returned and indicator variables for columns that can contain null values. Whenever a column

can contain nulls, an indicator variable must be included in the array definition immediately after the definition of that column. This indicator variable is an integer that can have the following values:

>= 0 the variable's value is not null.

< 0 the variable's value is null. (generally, you use -1).

StartIndex is a host variable whose value specifies the array subscript denoting where the first row to be inserted is stored. You must specify this variable.

NumberOfRows is a host variable whose value specifies the number of rows to retrieve. You must specify this variable, the value of this variable is the smaller of two values; 1) the number of records in the array from the StartIndex to the end of the array, or 2) the number of rows in the query result.

FROM The FROM clause identifies the tables referenced anywhere in the SELECT statement.

TableList provides the table names and optionally the correlation names. The syntax for TableList is as following:

{[*OwnerName.*]*TableName* [*CorrelationName*]

[,...]}

TableName identifies a table to be referenced. The tables can be listed in any order. *CorrelationName* specifies a synonym for the immediately preceding table. The correlation name can be used instead of the actual table name anywhere within the SELECT statement. The correlation name must conform to the syntax rules for basic name. All correlation names within one SELECT statement must be unique. They can not be the same as any table name in the FROM clause that does not also have a correlation name associated with it.

OwnerName must be used when you are not the owner of the table. In this case, *CorrelationName* must also be used to refer to this table in other part of the statement, because *OwnerName* is not allowed to use in other part of the statement.

Correlation names are useful when you join a table to itself. You name the table twice in the FROM clause, and assign it two different correlation names.

WHERE *SearchCondition1* The WHERE clause determines the set of rows to be retrieved. Rows for which *SearchCondition1* is false or unknown are excluded from processing. IF the WHERE clause is omitted, no rows are excluded. Aggregate functions cannot be used in the WHERE clause.

Rows that do not satisfy *SearchCondition1* are eliminated before groups are formed and aggregate functions are evaluated.

When you are joining tables, the WHERE clause also specifies the condition(s) under which rows should be joined. If you omit a join condition, SQL joins each row in each table in the FROM clause with each row in all other tables in the FROM clause.

SearchCondition1 may contain subqueries. Each subquery is effectively executed for each row of the outer query and the results used in the application of SearchCondition1 to the given row. If any executed subquery contains an outer reference to a column of a table in the FROM clause, then the reference is to the value of that column in the given row.

Refer to the "Search Conditions" section for additional information on search conditions.

GROUP BY *ColumnName*The GROUP BY clause identifies the columns to be used for grouping when aggregate functions are specified in the select list and you want to apply the function to groups of rows.

When you use the GROUP BY clause, the select list can contain only aggregate functions and columns referenced in the GROUP BY clause. If the select list contains an * or a TableName.* construct, then the GROUP BY clause must contain all columns that the * includes. Specify the grouping column names in order from major to minor.

Null values are considered equivalent in grouping columns. If all other columns are equal, all nulls in a column are placed in a single group.

If the GROUP BY clause is omitted, the entire query result table is treated as one group.

HAVING *SearchCondition2* The HAVING clause specifies a test to be applied to each group. Any group for which the result of the test is false or unknown is excluded from the query result. This test, referred to as *SearchCondition2*, can be a predicate containing either and aggregate function or a column named n the GROUP BY clause. If the query contains no GROUP BY clause, any HAVING predicate that does not contain an aggregate function must contain a column named in the select list.

Each subquery in *SearchCondition2* is effectively checked for each group created by the GROUP BY clause, and the result is used in the application of *SearchCondition2* to the given group. If any executed subquery contains an outer reference to a column, then the reference is to the values of that column in the given group. Only grouping columns can be used as outer references in a subquery in *SearchCondition2*.

UNION [ALL] *FullSelect* unites two SELECT results into a combined SELECT result. The *FullSelect* is referenced as the second SELECT, in which it can include all clauses from SELECT to HAVING except that it can not include INTO clause.

The union of two sets is the set of all elements that belong to either or both of the original sets. Because a table is a set of rows, the union of two tables is possible. The resulting table consists of all rows appearing in either or both of the original tables.

ALL indicates that duplicates are not removed from the result table when UNION is specified. If UNION is specified with ALL, duplicates are removed.

`ORDER BY` — The ORDER BY clause sorts the result table rows in order by specified columns Specify the sort key columns in order from major sort key to minor sort key.

`ColumnID` — The *`ColumnID`* must correspond to a column in the select list. You can identify a column to be sorted by giving its name or by giving its ordinal number, with the first column in the select list being column number 1. You must use a column number when referring to columns in the query result that are derived from column expressions. You must also use a column number to refer to columns if the expression contains more than one subqueries.

`ASC | DESC` — For each column you can specify whether the sort order is to be ascending or descending. If neither ASC nor DESC is specified, ascending order is used.

Descriptions

• When you do not specify BULK option, the SELECT command is a simplest version of SELECT command to retrieve for single row. In this case, it can retrieve only one row from the database. The search condition in WHERE and GROUP BY and HAVING clause must be specified in such a way that only one can be qualified as the result of SELECT command. If the SELECT statement does try to retrieve more than one row, an error occurs and the result variables in INTO clause hold information from the first row. The UNION and ORDER BY clause cannot be included in this simple case. In the INTO clause, you must use compatible syntax for holding single row.

• When you specify the BULK option, the SELECT command is full select statement. Multiple rows can retrieved as a single execution of the SELECT statement. In the INTO clause, you must use compatible syntax for BULK operation.

• The BULK option and INTO clause cannot be used in dynamic queries and cursor operations.

• The clauses must be specified in the order given in the syntax diagram.

• A result column in select list can be derived in any of the following ways:

- A result column can be computed, using an arithmetic expression, form values in a specified column of a table listed in the FROM clause.

- Values in result columns of a single table can be combined in an arithmetic expression to produce the result column values.

- Values in several columns of a single table or view can be combined in an arithmetic expression to produce the result column values.

- Values in result columns of various different tables can be combined in an arithmetic expression to produce the result column values.

- Aggregate function (AVG, MAX, MIN, SUM, and COUNT) can be used to compute result column values over groups of rows. Aggregate functions can be used alone or in an expression. If you specify more than one aggregate function containing the DISTINCT option, all these aggregate functions must operate on the same column. If the GROUP BY clause is not specified, the function is applied over all rows that satisfy the query. If the GROUP BY clause is specified, the function is applied once for each group defined by GROUP BY clause. When you use GROUP BY clause, the select list can contain only aggregate functions and columns referenced in the GROUP BY clause. The select list cannot contain a construct containing an asterisk (*). Do not use the DISTINCT option in the argument of a function if that option precedes the select list.

- A result column containing a fixed value can be created by specifying a constant or an expression involving only constants.

• In addition to specifying how the result columns are derived, the select list also controls their relative position from left to right in the result row. The first result column specified by the select list becomes the leftmost column in the result row.

• When you use SELECT command with DECLARE CURSOR command, the following clauses cannot be included: INTO, GROUP BY, HAVING, ORDER BY, and UNION. If you specify a FOR UPDATE clause in DECLARE CURSOR command, only one table can be referred in FROM clause.

• Result columns in the select list are numbered from left to right. The leftmost column is number 1. Result columns can be referred to by column number in the ORDER BY clause; that is especially useful if you want to refer to a column defined by an arithmetic expression.

• To join tables, list the tables in the FROM clause, and specify a join condition in the WHERE clause. To join a table with itself, define correlation names for the table in the FROM clause; use the correlation names in the select list and the WHERE clause to qualify columns named more than once in the select list and the WHERE clause.

• When you use the SELECT command as a subquery, it may not contain any UNION or UNION ALL operations, and the following rules are provided:

- Subqueries are used to retrieve data that is then used in evaluating a search condition.

- A subquery may be used only in the following types of predicates:

  - EXISTS predicate.

  - Quantified predicate.

  - IN predicate.

  - Comparison predicate.

- A subquery may be used in the WHERE or HAVING clause of SELECT statements and in WHERE clause of UPDATE, INSERT, and DELETE statements.

- A subquery may also be nested in the WHERE or HAVING clause of another subquery.

• When you use the GROUP BY clause, one answer is returned per group, in accord with select list:

- The WHERE clause eliminates rows before groups are forme.

- The GROUP BY clause groups the resulting rows.

- The HAVING clause eliminates groups.

- The select list aggregate functions are computed for each group.

• When you include a UNION operator in a SELECT command, some rules are provided. For the following, assume that T1 is the result of the subquery on the left of the UNION operator, and T2 is the result of the subquery on the right of the UNION operator:

- T1 and T2 must have the same number of columns. (They may be derived from tables with varying numbers of columns.)

- The union is derived by first inserting each row of T1 and each row of T2 into a result table and then eliminating any redundant rows unless ALL is specified.

- The ORDER BY clause can specify the ordinal number or the column name of a column in the leftmost query in a UNION.

- You can use INTO clause only in the SELECT command on left side of UNION.

• If an error is detected during a SELECT command, the command will have no effect and no row will have been returned.

Notes

• SELECT command is one of the most compatible command.

• When you specify a value in search condition, some constants defined by RE/SQL, which are included in { }, will be preprocessed by RE/SQL translator. These constants include DATE, DATETIME, INTERVAL, and some special constants, such as {now}, {today}, {user}, and {null}. If you use host variables to represent date constants, you must explicitly invoke a C conversion function before you invoke this command. See "Data Type" for details.

• The BULK SELECT is not supported by INGRES/SQL, but can be simulated with a group of C and SQL statements. Reference to "Bulk Operations" section for more information about using the BULK operations.

3.35 SET READLOCK

Sets default lock mode for reading data from database.

Syntaxes

S: EXEC SQL SET READLOCK {NOLOCK | SHARED};

A: read_lock_mode = {NOLOCK | SHARED};

I: EXEC SQL SET LOCKMODE WHERE READLOCK = {NOLOCK | SHARED};

Parameters

NOLOCK — means that the transaction reads data without obtaining any lock. I provides the greatest degree of concurrence. It is also called Read Uncommitted which does not wait for locks on user data to be released.

Because no locks are obtained, data read with NOLOCK may be modified by other transactions. Use this readlock in applications in which the reading of uncommitted data is not of concern.

SHARED — means that the transaction must grant a Share Lock to before reading data. It is also called Repeatable Read which guarantees that data pages selected or updated by the transaction are changed by other transactions until the current transaction ends with a COMMIT WORK or ROLLBACK WORK command.

Use this readlock when you need to read the same data more than once in the current transaction with the assurance that the data has not changed.

Descriptions

• No default readlock parameter is provided. If you omit the readlock parameter, an error message will be generated by RE/SQL translator.

• When you issue a statement that reads data from the database, such as a select statement, a readlock must be acquired before the data can be read. SQL grants readlock only when you issue a query statement instead of when you issue SET READLOCK command itself.

• This command can be used only in one place, where is after CONNECT TO, COMMIT WORK, and ROLLBACK WORK commands, and before BEGIN WORK command.

• You can not use this command in any on going transaction. Once you set the readlock in a certain mode, this mode will keep effective until you next time set to another mode.

Notes

• For ALLBARE/SQL, the readlock can be controlled by setting isolation level. There are four isolation levels, they are RR (Repeatable Read, i.e. Share Lock), CS (Cursor Stability), RC (Read Committed), and RU (Read Uncommitted, i.e. Null Lock). These isolation levels can be set when you issue a BEGIN WORK command. Only RR and RU can find corresponding readlocks in INGRES/SQL which can be set by SET LOCKMODE command.

• To eliminate the differences existed, RE/SQL translator automatically declare and use a global C variable, which is `read_lock_mode`, to record the states of the readlocks. Based on this variable, we developed this RE/SQL statement, which can be used to set `read_lock_mode` variable. And based on this variable, RE/SQL will provide a proper parameter (RR or RU) when you issue BEGIN WORK command in ALLBARE/SQL application program, or will issue a SET LOCKMODE command in INGRES/SQL application command.

• The Cursor Stability (CS) and the Read Committed (RC) readlock are removed from ALLBARE/SQL. A Exclusive readlock is removed from INGRES/SQL.

• See "Concurrence Control through Locks" for details.

3.36 Sqlexplain

The SQLEXPLAIN command places a message describing the meaning of a return code into a host variable. The text of messages comes from the SQL message catalog.

Syntaxes

S: EXEC SQL SQLEXPLAIN :*HostVariable*;

A: EXEC SQL SQLEXPLAIN :*HostVariable*;

I: {

EXEC SQL SQLEXPLAIN (:*HostVariable* = ERRORTEXT);

SQLCA_SQLCODE = 0;

}

Parameters

HostVariable identifies a host variable used to hold an SQL error message.

Descriptions

• The host variable is a character string which should be sufficient to store the error message retrieved, otherwise the message will be truncated. If no message retrieved, a blank message is returned.

• If more than one error occurs, SQLEXPLAIN can be used to obtain more than one message. You execute SQLEXPLAIN repeatedly until the SQLCA_SQLCODE becomes zero. (this is true only for ALLBARE/SQL.)

• The message describes the meaning of a return code. SQL puts a return code into the SQLCA after each SQL command in a program is executed. The SQLCA is an area for information on errors, warnings, truncation, null values, and other conditions related to the execution of an SQL command.

Notes

• SQLEXPLAIN command can be used to retrieve many other messages in INGRES/SQL, these functions are not included in RE/SQL.

Examples

```
EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;

  char msg[256];

EXEC SQL END DECLARE SECTION;

/* execution of a SQL command */

while (SQLCA_SQLCODE != 0)

{

  EXEC SQL SQLEXPALIN :msg;

  printf("message: %s\n", msg);

}
```

3.37 Unlock Table

The UNLOCK TABLE command provides a means of explicitly releasing a lock granted by LOCK TABLE command.

Syntaxes

S: EXEC SQL UNLOCK TABLE *TableName*;

A: /* EXEC SQL UNOCK TABLE *TableName*; */

```
I: {

    EXEC SQL SET LOCKMODE WHERE LEVEL = PAGE;
```

```
    if (read_lock_mode == NOLOCK)
      EXEC SQL SET LOCKMODE WHERE READLOCK = NOLOCK;
    else if (read_lock_mode == SHARED)
      EXEC SQL SET LOCKMODE WHERE READLOCK = SHARED;
  }
```

Parameters

TableName specifies the table to be unlocked.

Descriptions

• You must issue a UNLOCK TABLE command explicitly to release the locks held after you terminate the transaction. Otherwise, the locks can be released automatically but the lock mode will still in the mode you specified in LOCK TABLE command.

• This command must be used in pairs with UNLOCK TABLE command. If you issued a LOCK TABLE command before the transaction has begun, you must issue a UNLOCK TABLE command, immediately after the transaction ends (committed or rolled back).

Notes

• The way to release the lock for specified table is different. For ALLBARE/SQL, the locks granted will be automatically released when the transaction ends. No UNLCOK TABLE is necessary. For INGRES/SQL, a UNLOCK TABLE command is necessary, because the lock mode will not change, though the locks can be released, when the transaction ends. Once the lock mode is set, it will keep in that mode until you explicitly issue a UNLOCK TABLE command.• Reference to LOCK TABLE command and "Concurrence Control through Locks".

3.38 Update

The UPDATE command updates the values of one or more columns in all rows of a table or in rows that satisfy a search condition.

Syntaxes

S: EXEC SQL UPDATE *TableName*

SET {*ColumnName* = *Expression*} [,...]

[WHERE *SearchCondition*];

A: exactly the same as S syntax

I: exactly the same as S syntax

Parameters

TableName	designates a table in which any rows satisfying the search condition are to be updated.
SET	is a clause in which you specify the columns and values to be updated.
ColumnName	designates a column to be updated. You can update several columns of the same table with a single UPDATE command.
Expression	is any expression that does not contain an aggregate function. The expression is evaluated for each row qualifying for the update operation. The data type of the expression must be compatible with the column's data type. Host variables or constants can also be specified as an expression. See "Expression" section for details.
WHERE *SearchCondition*	specifies which rows are to be updated. If not rows satisfy the search condition, the table is not changed. All rows for which the search condition is true are updated as specified in the SET clause. The search condition cannot contain an aggregate function. If the WHERE clause is omitted, all rows are updated.

Descriptions

• If the WHERE clause is present, then the search condition is evaluated for each row of the table before updating any row. Each subquery in the search condition is effectively executed for each row of the table, and the results used in the application of the search condition to the given row. If any executed subquery contains an outer reference to a column of the table, the reference is to the value of that column in the given row.

• If an error is detected during a multi-row UPDATE operation, the command will have no effect and no rows will have been updated.

Notes

• This command is basically compatible in ALLBARE/SQL and INGRES/SQL, but some differences still exist. ALLBARE/SQL can update rows through a view, which function is not supported by INGRES/SQL. The correlation name can be used in INGRES/SQL but not in ALLBARE/SQL. These functions are removed from RE/SQL.

• A FROM clause in INGRES/SQL is not included in RE/SQL syntax.

UPDATE WHERE CURRENT

The UPDATE WHERE CURRENT command updates the values of one or more columns in current row associated with a cursor. The current row is the row pointed to by a cursor after the FETCH command is issue.

Syntaxes

S: EXEC SQL UPDATE *TableName*

SET {*ColumnName* = *Expression*} [,...]

WHERE CURRENT OF *CursorName*;

A: exactly the same as S syntax

I: exactly the same as S syntax

Parameters

TableName	designates a table in which any rows satisfying the search condition are to be updated.
SET	is a clause in which you specify the columns and values to be updated.
ColumnName	designates a column to be updated. You can update several columns of the same table with a single UPDATE command.
Expression	is any expression that does not contain an aggregate function. The expression is evaluated for each row qualifying for the update operation. The data type of the expression must be compatible with the column's data type. Host variables or constants can also be specified as an expression. See "Expression" section for details.
CursorName	Designates an opened cursor. The current row of the cursor is updated as specified by the SET clause. The column(s) named in the SET clause must also be named in the FOR UPDATE clause of the DELCARE CURSOR command defining the cursor. After the update, the row updated remains the current row.

Descriptions

• If an error is detected during a multi-row UPDATE operation, the command will have no effect and no rows will have been updated.

Notes

• This command is basically compatible in ALLBARE/SQL and INGRES/SQL, but some differences still exist. ALLBARE/SQL can update rows through a view, which function is not supported by INGRES/SQL.

• When you specify a expression, some constants defined by RE/SQL, which are included in { }, will be preprocessed by RE/SQL translator. These constants include DATE, DATETIME, INTERVAL, and some special constants, such as {now}, {today}, {user}, and {null}. If you use host variables to represent date

constants, you must explicitly invoke a C conversion function before you invoke this command. See "Date Type" for details.

WHENEVER

The WHENEVER is a command used in an application program to specify an action to be taken depending on the outcome of subsequent SQL commands.

Syntaxes

S: EXEC SQL WHENEVER *Condition Action*;

A: exactly the same as S syntax

I: exactly the same as S syntax

Parameters

`Condition`	can be one of the following three key words:
• SQLERROR	refers to a test for the condition SQLCA_SQLCODE < 0.
• SQLWARNING	refers to a test for the condition SQLCA_SQLWARN0 = 'W'.
• NOT FOUND	refers to a test for the condition SQLCA_SQLCODE = 100.
`Action`	can be one of the following three key words:
• STOP	causes a ROLLBACK WORK command and terminates the application program, whenever a SQL command produces the specified condition.
• CONTINUE	means no special action is taken automatically when a SQL command produces the specified condition. Sequential execution will continue.
• GO TO `Label`	causes a jump to the specified `Label` whenever the condition is found to be true after executing a SQL command. The `Label` must conform to the SQL syntax rules for a basic name as well as the requirements of the host language. GOTO is the same as the GO TO.

Descriptions

• SQLCA_SQLCODE and SQLCA_SQLWARN0 are fields in the SQLCA, a data structure SQL uses to return status information about SQL commands.

• A WHENEVER statement affects all SQL command that come after it in the source program listing, up to the next WHENEVER statement for the same condition.

• You can write code of your own to check the SQLCA for error or warning conditions, whether or not you use the WHENEVER statement.

Notes

• In SQLEXPLAIN, some additional conditions and action is supported, such as SQLMESSAGE and DBEVENT conditions and CALL action, these functions are not included in RE/SQL.

Examples

```
EXEC SQL INCLUDE SQLCA;.

EXEC SQL WHENEVER SQLERROR STOP;

EXEC SQL WHENEVER NOT FOUND GO TO 9000;

EXEC SQL CONNECT TO 'database';

/* Execution of the program terminates if the CONNECT TO

  command cannot be executed successfully */

SELECT * FROM TestTable;

/* If no rows are qualified, control is passed to the

  statement labeled 9000 */

9000: printf("message: no rows are qualified\n");
```

Chapter 4 The Administrator's guide

4.1 Overview

The readers of this chapter will be the users who responsible to maintenance this software. The RE/SQL syntax translator has been developed with a very good extendable feature. All kinds of syntaxes could be managed in a extendable file, namely **syntax.file**, with certain pre-defined rules. In such a way, the translator could easily be extended to produce the syntax coming from any other additional RE/SQL products. On the other hand, the user could define new RE/SQL statements which can be interpreted into C and implementation's SQL statements.

This chapter will explain the working mechanism of the translator by providing a logical flow chart of the program. And then, the format of the syntax file will be defined. An currently used syntax file, which includes syntaxes for ALLBARE/SQL and INGRES/SQL, is given in Appendix A.

4.2 The Working Mechanism of the Translator

Figure-2 is the logical flow chart of the program which reflects the working mechanism of the translator. The following will explain this figure based on the numbers on it.

¿ After the program is started, at the first, the parameters which are specified by the user in command line will be processed. Three parameters must be obtained in this step, which are source (input) file name, destination product identifier, and destination (output) file name.

¡ All syntaxes have been defined and included in a syntax file that name is **syntax.file**, which includes all statements for all currently available SQL products. Based on the destination product identifier obtained in ¿, only the syntax for the specified product will be loaded into a pre-defined structure array which will be dynamically allocated in the memory. See next section for detailed information about syntax file.

¬ Based on the names obtained in ¿, the source code file and the output file are opened in this step.

Beginning from this step, the program enters a loop to deal with the statements in source code file one by one. After a statement is read into a statement buffer, a conditional branch is applied based on if it is a SQL statement, i.e. if it is prefixed by EXEC SQL. If it is not a SQL statement, branching to *f*, in which this statement will be echoed directly to output file. If it is a SQL statement, a preprocessing work is done on the statement before branching to for later convenience, e.g. the EXEC SQL prefix including any possible preceding label is eliminated, all embedded comments, if any, are removed, and all additional blanks are also removed. A "clean" SQL statement is provided in this step for later use.

☞*In above four steps, errors may occur. Mostly, those errors are fatal errors which will cause immediately terminating of the program. The fatal error handling is not included in the Figure-2. The possible errors could be command line error in ①, syntax file not found error and syntax file internal error in ②, source code file not found error in ③, delimiters for comments or strings unpaired error in ④.*

⑤ In this step, the text in statement buffer is written to output file without any processing.

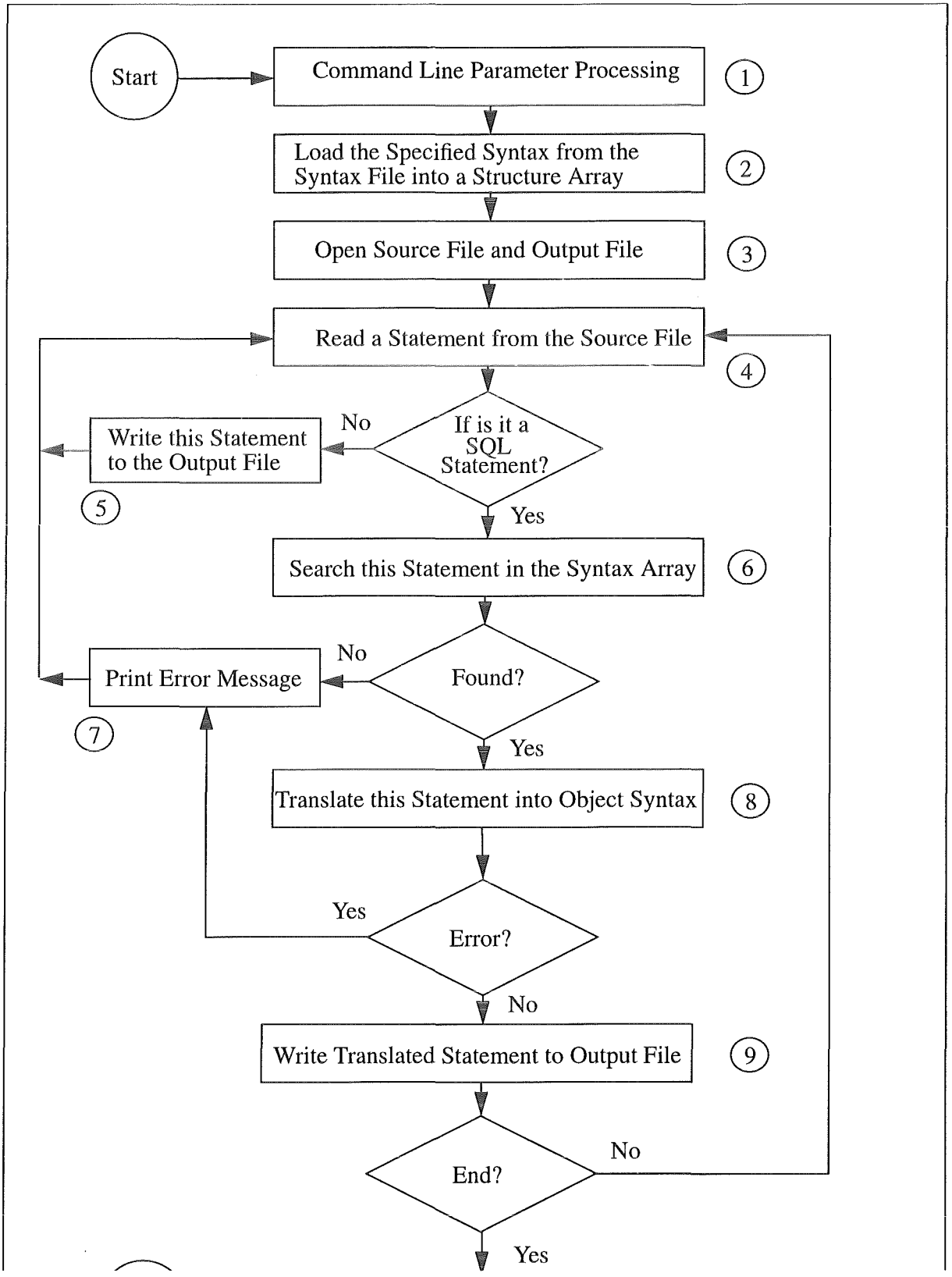
⑥ The program extracts a certain number of characters from the beginning of the statement as the key word with which it could locate the syntax for this statement in the syntax array. If the extracted key word could not match any statements in syntax array, a error message will be generated in ⑦. Otherwise, the translator function will be invoked in ⑧.

⑦ An error message is generated and printed on the screen. All errors printed are counted to provide an error summary before terminating the program.

⑧ An translator function is invoked. Based on the corresponding syntax found in syntax array, the statement in the buffer is translated into its destination product syntax. If any error is detected during translating, which is mostly RE/SQL syntax error, the program will branch to ⑦. Otherwise, the translator will generate statement in specified syntax and write it to output file in ⑨. This is the most important step. For more information about syntax translation, see next section.

⑨ In this step, the statement generated by the translator function is written to output file. And then, the program tests if this statement is the last statement in the input file. If no end of file detected, the program will loop to ④. Otherwise, the program goes to terminate in ⑩.

⑩ Before the program terminates, all files opened are closed, all memories allocated are released, a summary of the execution of the program is reported.


Start
Command Line Parameter Processing
1
Load the Specified Syntax from the Syntax File into a Structure Array
2
Open Source File and Output File
3
Read a Statement from the Source File
4
If is it a SQL Statement?
No
Write this Statement to the Output File
5
Yes
Search this Statement in the Syntax Array
6
Found?
No
Print Error Message
7
Yes
Translate this Statement into Object Syntax
8
Error?
Yes
No
Write Translated Statement to Output File
9
End?
No
Yes

4.3 The Format for the Syntax File and the Syntax Array

The RE/SQL syntax translator has been developed with a very good extendable feature. All kinds of syntaxes could be managed in a extendable file, namely **`syntax.file`**, with certain pre-defined rules. In such a way, the translator could easily be extended to produce the syntax coming for any other additional RE/SQL products. On the other hand, the user could define new RE/SQL statements which can be interpreted into C and implementation's SQL statements.

It is worth to highlight the overall structure of the syntax file and its storage in memory, because the translator program has been developed based on this file. On the other hand, the administrator of the program must know all aspects of the syntax file and related rules to maintenance this program and/or extend the translator to support any additional SQL products.

After surveying the currently available SQL implementations, some syntax features have been noted as following:

• The syntaxes for some statements are exactly the same. In this case, no translating is needed.

• Some of the statements have similar structures, but some insignificant differences could be identified, e.g. some statements have same function, same number of parameters, but different statement key words have been used. In this case, a one word by one word syntax translating is needed.

• Some statements existed in a SQL implementation is absent in another SQL implementation. But the functions for those absent statements in a SQL implementation could be simulated by a combination of some SQL statements and/or C statements in another SQL implementation. In this case, a more strong syntax translating is needed, in which the parameters must be extracted from the source statements and then the new syntax could be re-built based on the syntax provided in syntax file.

• The formats for date data type used in different SQL products are not the same. The format conversion for date data type should be considered not only during the syntax translation phase but also while inserting and retrieving data into/from the database.

• The key words used to define the data types in different SQL products are not the same, which appear mostly in the CREATE TABLE command. A special processing should be applied to this command.

• Bulk processing, which allow user to retrieve multiple rows in one SELECT command, are implemented with significant differences. A special processing for bulk operation commands and related host variable declarations should be considered.

• If any dynamic operations are used in the user's applications, the syntax in those dynamic operation command must be dynamically translated to destination SQL product syntax.

All aspects listed above reflect the major considerations while developing the translator program and related formats and rules of syntax file.

The Overall Structure of the Syntax File

The syntax file, namely **syntax.file,** is an UNIX text file, which can be referred in Appendix A. The syntaxes defined in this file are some text lines terminated with <return> keys. The whole file is divided into five parts which include syntax translating information for different purposes, in which each part terminates by identifying a line preceding with '@@'. Figure-3 shows the file structure.

```
①

@@ /* end of syntaxes for all statements */

②:

@@ /* end of syntaxes for data type definitions */

③

@@ /* end of syntaxes for null indications */

④

@@ /* end of syntaxes for constant specifications */

⑤:

@@ /* end of syntaxes for date conversions */
```

Figure-3 The Overall Structure of the Syntax File

The following descriptions address to Figure-3:

¿ This part contains the syntax definitions for all RE/SQL statements. Please reference to "The development of the Syntax for the RODOS Embedded SQL (RE/SQL)".

¡ This part contains the key word translating syntaxes for the RE/SQL DATA TYPE definition in CREATE TABLE command.

¬ This part contains the key word translating syntaxes for the RE/SQL NULL INDICATION in CREATE TABLE command.

④ This part contains the key word translating syntaxes for the RE/SQL CONSTANT which are mostly used in DATA MANUPULATION commands.

⑤ This part contains the date conversion formats which are used by DATE CONVERSION and DATA MANUPULATION commands.

The following general rules can be applied to the syntax file:

• Each part terminates when the identifier '@@' is encountered. The identifier must be the first two characters of the line. The comments following '@@' identifier will not be processed by the program. They are only the comments.

• Each part of the syntax consists of a number of 'syntax units'. The so-called syntax unit is constructed by a couple of lines to define a RE/SQL statement for its implementations for all destination SQL products. The format to define a syntax unit will be described later in this section.

• Each syntax unit begins with a '@' character, and it is considered, at the same time, as the end of last unit. Any blank line will be omitted while reading these syntax units into memory. Generally, we use a blank line between every two units to make the file easier to read by the users.

• The maximum number of syntax units, in whole six parts, is now set to 80 in the program by a macro named MAX_SYNTAX_NUMBER. In current syntax file, about 59 syntax units have been used.

The Format of the Syntax Unit

Every part of the syntax file is composed of a number of syntax units which are basic units of the syntax file. All syntax units in the syntax file use a unique general format and follow similar rules to build the syntax contexts. The date conversion format is an exception, which will be described separately. This section mainly defines the format of the syntax unit. The rules to build the syntax contexts will be described in later section. Figure-4 shows the format of the syntax unit.

@CommandKeyWord@ [*ParameterDefinitions*] [*Operators*]

ProductIdentifier 1: *SyntaxContext 1*

ProductIdentifier 2: *SyntaxContext 2*

.

.

ProductIdentifier N: *SyntaxContext N*

Figure-4 The Format of the Syntax Unit

A syntax unit consists of a couple of syntax lines, which depends on the number of SQL products that can be processed by the translator. If the user includes N products in the syntax file, each syntax unit consists of N+1 syntax lines. The current syntax file includes two SQL products, which are ALLBARE/SQL and INGRES/SQL.

The so-called syntax line is a logical line which will be treated as a single character string by the translator. However, the syntax line could take more than one text lines in the syntax file, because the <return> key is not used as terminator of the syntax context. A syntax context terminates when another product identifier or another command key word is encountered.

The first line of the syntax unit includes the information about source RE/SQL statement. As an exception, this line can not include any <return> key, i.e. it must be finished in one text line. The following N lines contains the informations for each of the destination SQL products. Those lines are syntax lines which can take as many text lines as you need.

The following descriptions address to Figure-4:

@CommandKeyWord@ is used by the translator program to locate the syntax unit for the statement to be processed. The command key word, which must be in lower case and is delimited between two '@' characters, should specify an unique statement in RE/SQL command set. If the command key words for two or more syntax units are partially the same, i.e. one complete key word is the first few characters of another key word, the syntax unit with longer command key word should be defined before the one with shorter command key word in the syntax file. For example, command EXECUTE is partially the same as command EXECUTE IMMEDIATE. In this case, the syntax unit for EXECUTE IMMEDIATE command should be defined before defining EXECUTE command.

ParameterDefinitions are optionally used by the translator program to extract the parameters in the source statement, which are certain parts of the source statement. The detailed format will be defined in the later section.

Operators are optionally used to indicate that the translator program performs some special process during translating the statement. See later section for detailed information.

`ProductIdentifier`:is a single letter which identifies the destination SQL product. For each translatable SQL product, which is ALLBARE/SQL or INGRES/SQL in this moment, a single letter must be selected as an identifier. Generally, you can use the first letter of the product name. Here we use A as the identifier of ALLBARE/SQL, I as the identifier of INGRES/SQL. The letter used must be in upper case and immediately followed with a colon character. The product identifier and the followed colon should take the first two character of the line. The product identifier has the same meaning as it was defined in the section 2.2.1.

SyntaxContext contains the syntax of the destination SQL product. At least one blank should be inserted between the product identifier and the syntax context. The appearance of the statement generated by translator is possibly as it is formatted in syntax context. You can include <return> key and the preceding blanks in the continuous lines. The rules to build the syntax context will be discussed in the later section.

In the following, a very simple example is provided to give you a brief view of the syntax unit:

@disconnect@

A: EXEC SQL COMMIT WORK RELEASE;

I: EXEC SQL %s;

The Format of the Syntax Array

In order to enable the translator working in an efficient way, it is necessary to reside all syntax units of the syntax file in the memory. Obviously, during an execution of the translator, only the syntax context for one of those SQL products is needed to load into the memory, which is specified when the user invokes the translator program. Therefore, a C structure array is defined for this purpose as following:

```
struct SYNTAX
{
  char *se_stx;
  short id_len;
  char *ob_stx;
} syntax[MAX_SYNTAX_NUMBER];
```

Here, one member of the array is used to store one syntax unit. The structure element `*se_stx` and `*ob_stx` are two character pointer, which point two character buffers, to store the syntax context for RE/SQL syntax (source syntax) and the syntax context for one of the SQL products syntax (object syntax). Because the length of the syntax context may vary from some characters to some hundred characters, the character buffers used here will be dynamically allocated by the translator based on the real length of the syntax contexts. `id_len` is a integer variable to specify the length of the command key word in RE/SQL syntax, which is used by the translator while searching the key word in the array (fixed length comparison of the string is used).

All syntax units in the syntax file are stored in the syntax array continuously in the same order as they are in the syntax file. These syntax units are still divided into six parts, which can be addressed by a couple of integer variables:

short n_stx, n_type, n_null, n_const, n_date;

and the following is the map of the syntax array:

syntax[0] /* here is the first syntax unit of 1st part */

```
. /* here is the last syntax unit of 1st part */

syntax[n_stx] /* here is the first syntax unit of 2nd part */

. /* here is the last syntax unit of 3rd part */

syntax[n_type] /* here is the first syntax unit of 3rd part */

.

. /* here is the last syntax unit of 4th part */

syntax[n_null] /* here is the first syntax unit of 4th part */

. /* here is the last syntax unit of 5th part */

syntax[n_const] /* here is the first syntax unit of 5th part */

. /* here is the last syntax unit of 6th part */

syntax[n_date]
```

4.4 The Rules to Build the Syntax Context

A couple of rules have been developed to build the syntax context. In section 3.1, the overall working mechanism of the translator has been explained. To illustrate the detailed translating process, s simplified logical chart is provided in Figure-5:

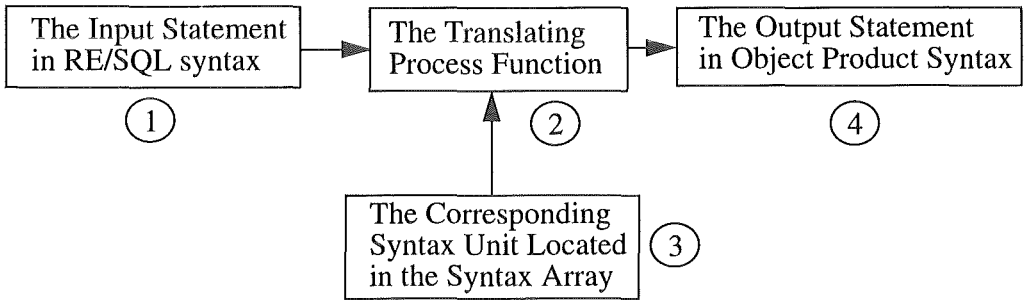


Figure-5 The Simplified Logical Chart for Translating Process

¿ All of the statements which are embedded in the source code file in RE/SQL syntax will be translated one by one. This chart shows only the process to translate a single statement. When each of the embedded statement is read to a character buffer by the program, it is made up to become a "clean string" of the statement, in which the EXEC SQL prefix including any possible label is eliminated, all embedded comments and additional blanks are picked out.

¡ The translating process function is a C function which does really translating work. Before it is invoked, a corresponding syntax unit in the syntax array is located based on the key word matching.

¬ A syntax unit includes two parts. One part contains information about source statement, e.g. the command key work which is used to locate that syntax unit, and optionally, some parameter definitions which can be used to extract a certain part of the source statement that will be embedded in the object statement, and some operators which indicates some special processing may be applied. Another part contains a format to build object statement which is composed of strings, and parameters obtained from source statement.

A statement in a specified object syntax will be generated by the translating process function based on both the source statement and the corresponding syntax unit.

In the following, all of the rules used to build syntax context will be described based on some real cases:

Case 1: **Using %s operator**. This case is the simplest case which includes two sub-cases. One is that the object syntax is exactly the same as the source syntax. Another one is that the object syntax is completely different with the source syntax. In both cases, none parameter is involved. For the first case, an operator "%s" is defined to completely copy the input statement to the place where it appears in the object syntax context. For the second case, you can write any string as you want. The following example just includes the both cases:

@disconnect@

A: EXEC SQL COMMIT WORK RELEASE;

I: EXEC SQL %s;

In this example, the RE/SQL statement DISCONNECT is defined without any clause and parameter. For ALLBARE/SQL, we find an equivalent statement COMMIT WORK RELEASE which is completely different. For INGRES/SQL, this statement is exactly defined as DISCONNECT.

This operator can not be applied to data type conversion syntax, constant conversion syntax, and date data type syntax (③④⑤⑥ in Figure-3);

Case 2: **Using %n operator to deal with parameter**. In this case, some parts of the source statement should appear in the object statement, which are possibly clauses or elements of the statement. We can define those parts of the source statement as "parameters" in the `ParameterDefinitions` field of the syntax unit. And then, these parameters can be used in the product syntax context at a proper place. A parameter can be defined as the following format:

{KeyWord %0 KeyWord}, {KeyWord %1 KeyWord}, ...

You can define more than one parameters in one syntax unit. The parameter definitions should be finished in the first line of the syntax unit (no continuous lines are allowed). The separator of every two parameter definitions could be a comma or a blank. In each parameter, a operator "%n" is used to represent the parameter, in which n should start from number of zero. The `KeyWords` are used to extract a certain part of the source statement, in which those key words are elements that must appear in the source statement. In the case of that a key word used in a parameter definition could not find a matching word in the source statement, an error will generated by RE/SQL translator. Besides the statement elements could be used as key words, two special key words are defined, which are FF and EE. FF denotes that the parameter starts from the place just following the command key word in the source statement which is defined in `@CommandKeyWord@` field. EE denotes that the parameter would end when the statement terminates with a ";". On the other hand, the parameters defined in the parameter definition field do not have to be used in every product syntax context. The following example includes these cases:

@connect to@ {FF %0 EE};

A: EXEC SQL %s;

I: EXEC SQL CONNECT %0;

In this example, the RE/SQL statement CONNECT is defined with a parameter which indicates the database to be connected. For ALLBARE/SQL, this statement is exactly defined so that we need only to copy the source statement to object statement. For INGRES/SQL, we find that the syntax is basically the same but without TO in the command key word. In this case, we need to define a parameter, which indicates the database identifier. Based on the syntax defined in this syntax unit, a RE/SQL statement:

CONNECT TO 'database1';

can be translated to its ALLBARE/SQL syntax:

CONNECT TO 'database1';

or its INGRES/SQL syntax:

CONNECT 'database1';

The parameter definitions may be different in some special cases, which are parameter in BULK operation statement and DATE format conversion. See following for details.

Case 3: **Using n operator**. For some special cases, some processing should be applied to the source statement or parameters. n operator, which could be optionally specified in the `Operators` field of the syntax unit, could inform the translator program to do those special processing. Four n (n=1, 2, 3, 4) operators are currently defined. Each of them has special meaning and can be used only in certain places. In the `Operators` field, you can use only one of these operators.

1 - Expression Processing Operator. For most of the data manipulation statements, the syntax used by different SQL products are basically the same. So, in most cases, you would use %s operator to copy the source statement to object statement. On the other hand, most of these statements could use expressions in their clauses to specify some conditions. Some constants defined in RE/SQL, such as date (data type) value and special value, may be different from any one of the current SQL products. (see Reference Manual for RODOS Embedded SQL - EXPRESSION section). In this case, before the source statement is copied to the object statement, the constants in the statement must be converted to its object syntax. 1 operator is used to inform the translator program to do this thing. It should be used when the source statement includes any possible expression, which are known as SELECT, DELETE, UPDATE, INSERT, and statements including sub-SELECT, such as CREATE VIEW, DECLARE CURSOR, and all BULK operation statements. The syntax used to convert the constants can be found in syntax file (⑤⑥ in Figure-3). The following is an example:

@select@ 1;

A: EXEC SQL %s;

I: EXEC SQL %s;

Based on the syntax defined in this syntax unit and constant converting syntax in syntax file, a RE/SQL statement:

```
SELECT * FROM TestTable
    WHERE date={today} or date={1994-01-01};
```

can be translated to its ALLBARE/SQL syntax:

```
SELECT * FROM TestTable
    WHERE date=CURRENT_DATE or date='1994-01-01';
```

or its INGRES/SQL syntax:

```
SELECT * FROM TestTable
    WHERE date='today' or date='01-jan-1994';
```

2 - Data Type Definition Processing Operator. This operator is used only for CREATE TABLE statement, because the key words, which are used to define date type in this statement, are different in current SQL implementations. The syntax used to convert the constants can be found in syntax file (③④ in Figure-3). The following is an example:

@create table@ {FF %0 EE} 2;

A: EXEC SQL CREATE PUBLIC TABLE %0;

I: EXEC SQL %s;

Based on the syntax defined in this syntax unit and constant converting syntax in syntax file, a RE/SQL statement:

```
exec sql create table TestTable
  (pchar char(1000),
   pvarchar varchar(100), pdouble double,
   preal real with null,
                psmallint smallint, pinteger integer,
                pdate date, pdatetime datetime, pinterval interval
  );
```

can be translated to its ALLBARE/SQL syntax:

```
EXEC SQL CREATE PUBLIC TABLE TestTable
        (pchar CHAR(1000),
         pvarchar VARCHAR(100),
         pdouble DOUBLE PRECISION,
         preal REAL DEFAULT NULL,
         psmallint SMALLINT,
         pinteger INTEGER,
         pdate DATE,
         pdatetime DATETIME,
```

pinterval INTERVAL);

or its INGRES/SQL syntax:

```
EXEC SQL CREATE TABLE TestTable
        (pchar CHAR(1000),
         pvarchar VARCHAR(100),
         pdouble FLOAT8,
         preal REAL WITH NULL,
         psmallint SMALLINT,
         pinteger INTEGER,
         pdate DATE,
         pdatetime DATE,
         pinterval DATE);
```

3n - Parameter Processing Operator. Sometimes, before a parameter, which is defined in *ParameterDefinitions* field, is used to build object statement, some processing should be applied to it. The number n specify which parameter should be processed. The following three cases will be automatically identified and processed based on some conditions:.

(1) **The first character of the parameter is a colon**. Sometime, the parameter may be a host variable preceded with a colon ":", and when this host variable appear in the object syntax context the colon ":" should be eliminated. The following is an example:

```
@savepoint@ {FF %0 EE} $3$0;

A: EXEC SQL %s;

I: EXEC SQL SAVEPOINT %0;
```

In this example, the parameter is preceded with ":", which is identical to ALLBARE/SQL syntax. But, for INGRES/SQL, the host variable is not allowed to be used here, it should be a integer variable (see SAVEPOINT statement in "The Development of the Syntax for the RODOS Embedded SQL").

(2) **The first character of the parameter is a single quotation**. Sometime, the parameter may be a string within two single quotation " ' ", and when this string appear in the object syntax context the single quotation should be change to a double quotation " " ". The following is an example:

```
@execute immediate@ {FF %0 EE} $3$0;

A: {
  if (SESQL_dynamic_preprocessor(DynamicCommandBuffer, %0, 'A') == 0)
    EXEC SQL EXECUTE IMMEDIATE :DynamicCommandBuffer;
  else
    sqlca.sqlcode = -1111;
}
I: {
  if (SESQL_dynamic_preprocessor(DynamicCommandBuffer, %0, 'I') == 0)
    EXEC SQL EXECUTE IMMEDIATE :DynamicCommandBuffer;
  else
    sqlca.sqlcode = -1111;
}
```

In this example, it is actually that a SQL string is converted to a C string.

(3) **The parameter looks like TO :hostvariable**. This is a special case for ROLLBACK statement. Because the TO clause of this statement is optional, the TO with the followed host variable as a whole is defined as a parameter:

```
@rollback work@ {FF %0 EE} $3$0;

A: EXEC SQL %s;

I: EXEC SQL ROLLBACK %0;
```

In this example, the parameter, which is optional, is preceded with "TO :", which is identical to ALLBARE/SQL syntax. But, for INGRES/SQL, the host variable is not allowed to be used here, it should be a integer variable (see ROLLBACK statement in "The Development of the Syntax for the RODOS Embedded SQL").

4n - Converting to Uppercase Operator. This operator is used in a similar way as the way you use 3n operator, it indicates that the parameter should be converted to its uppercase. For example:

```
@set readlock@ {FF %0 EE} $4$0;

A: read_lock_mode = %0;
```

I: EXEC SQL SET LOCKMODE ON SESSION WHERE READLOCK = %0;

In this example, the parameter %0 appear in ALLBARE/SQL syntax as a C macro which is defined in uppercase (which are NOLOCK and SHARE. See SET READLOCK statement in "The Development of the Syntax for the RODOS Embedded SQL").

$*ProductIdentifier*$ - BULK Variable Declaration Processing Operator. This operator is used specially by BEGIN DECLARE SECTION statement. Because the BULK variable declarations are different in the current SQL implementations, some processing should be applied to the BULK variable declarations. See BULK OPERATIONS section in "The Reference Manual for the RODOS Embedded SQL" for the detailed description. Here, *ProductIdentifier* has the same meaning as it was defined earlier in this guide, which indicates that if the object syntax is specified as *ProductIdentifier*, then the structure variable (BULK variable) declaration following this BEGIN DECLARE SECTION statement will be processed based on the rules developed in BULK OPERATIONS section in "The Reference Manual for the RODOS Embedded SQL". For example, the syntax unit of BEGIN DECLARE SECTION is defined as follows:

@begin declare section@ I;

A: EXEC SQL %s;

I: EXEC SQL %s;

that means, if you specify INGRES/SQL (with -i option in command line) as object syntax, all BULK variable declarations following the BEGIN DECLARE SECTION will be processed.

4.5 The Run-time RE/SQL related C function Library

For some SQL statement, the statement is not completed in the application source code until the application is executed. In this case, the translating works should be done during the run-time of the user's applications. For RE/SQL, two cases are related to this problem. One is dynamic operations. Another is DATE format conversions. For these cases, some C functions should be developed in advance to build a RE/SQL related run-time C function library. The file containing these run-time C functions, which name is `dynamic.c` and `dynamic.o`, is provided in the packing of this software. When you embed any dynamic operation statement or date format conversion statement in your application, the file `dynamic.o` should be linked to the executable of your program.

4.6 The Run-time RE/SQL Related C Functions

Currently, only two functions (including some subroutines called by them) are included in the library. They are described as follows:

(1) **Dynamic Translating Function**. The function is defined as follows:

short SESQL_dynamic_preprocessor(Object,

Source, Productidentifier)

char *Object, *Source, ProductIdentifier;

where, the parameter `Source` could be a command string delimited by two double quotations or a host variable containing a command. Based on the syntaxes provided in the first part of the syntax file, in which includes all statements that could be dynamically translated, the statement contained in `Source` will be translated to the syntax specified by `Productidentifier` and stored into `Object`. At this moment, the possible values of `Productidentifier` are `'A'` and `'I'`. This function could be invoked in the syntax context of a syntax unit. Currently, it is invoked by the syntax units for EXECUTE IMMEDIATE and PREPARE statements (see Appendix A for this two statements).

When the function finishes the translating work, a integer value will be returned. The value 0 indicates success and value 1 denotes that an error occurs. The error processing will be described later in this section.

(2) **Dynamic Date Format Converting Function**.

The function is defined as follows:

short SESQL_translate_date(Source, Object, Operator)

char *Object, *Source, Operator;

where, the parameter `Source` could be a date (include datetime and interval) string delimited by two double quotations or a host variable containing a date string. Based on the syntaxes provided in the last part of the syntax file, in which includes all date formats, the date data type contained in `Source` will be converted based on the specified `Operator` and stored into `Object`.

When the function finishes the translating work, a integer value will be returned. The value 0 indicates success and value 1 denotes that an error occurs. The error processing will be described later in this section.

4.7 The RE/SQL Run-time Error Processing

As mentioned above, errors may occur during run-time translating. In this case, a error message will be generated and stored into a C character variable, namely `DynamicPreprocessMessage` which is declared in RE/SQL include file (ingres.macros.h and allbase.macros.h).

When you invoke these run-time function in you syntax file, a condition branch, which is based on the return values of the function, should be used to process possible errors. In order to keep agreement of general SQL error processing, a special negative number should be set to RE/SQL communication area variable `sqlca.sqlcode`. See syntax units for EXECUTE IMMEDIATE, PREPARE, and CONVERT DATE statements for examples.

Chapter 5 Installation Guide

5.1 Copying RE/SQL Files

This section will list the files included in this software. These files can be divided into two parts, which will be listed separately in the following.

The first part includes the files needed to execute the RE/SQL Syntax Translator (called as translator in the following):

sesql	the executable of the translator, see section 4.2 for using it
syntax.file	the syntax file includes user defined syntaxes by means of which the syntax will be translated, see section 3.3 & Appendix A
dynamic.o	the object file which includes assistant function set, see section 3.5 for description
allbase.macros.h	the including file defined for ALLBARE/SQL product, see Appendix B for listing
ingres.macros.h	the including file defined for INGRES/SQL product, see Appendix C for listing
example.input	the example(s) for each RE/SQL statement, see Appendix D

The second part includes the files needed to build the RE/SQL Syntax Translator:

sesql.c	the source code file, which includes main program
syntax.c	the source code file, which includes syntax translating functions
dynamic.c	the source code file, which includes assistant function set
bulk.c	the source code file, which includes functions related to bulk operations
sesql.m	the make file to build the executable of the translator

To install this software on an HP workstation, you need to copy only the files in the first part into your working directory. To install this software on other kind of workstation, you need to copy the files, except **sesql** and **dynamic.o**, in the both parts into your working directory, and then type the following UNIX command to rebuild the executable:

make -f sesql.m

After the installation, you can test if the program works by typing the following command:

sesql example.input -i -o ingres.output

or

sesql example.input -a -o allbase.output

If the program works, an output file (ingres.output in Appendix E or allbase.output in Appendix F) will be produced, and some messages will be displayed on the screen.

5.1.1 Managing RE/SQL Files

When you are using RE/SQL, you need always to take care a couple of files which are supplied by RE/SQL.

- **`SESQL`** -- ***Translator Executable File***. It is necessary for you to know where the translator executable is located after installing RODOS/SQL software. You need to add a search path in operating system environment or you always use file name with full path to invoke the translator.

- **`syntax.file`** -- ***Syntax File***. It is a data file which will be used (read) either by translator during processing the source code or by your application during its run-time. This file is supposed to stay at current directory from which you invoked the translator or you execute your application. See also section 4.1, 3.3, and Appendix A.

- **`dynamic.o (.c)`** -- ***Run-time C Function Library***. This is a C function library file which is necessary to be linked to your application. See also section 3.5.

- **`allbase.macros.h`** -- ***Header File for ALLBARE/SQL***. This is a header file defined for use when object SQL product is ALLBARE/SQL. This file will be automatically included when you use RE/SQL statement INCLUDE SQL_HEADER. When you compile your application code (after preprocessing it with ALLBARE/SQL preprocessor), this file should be found by C compiler in suitable place, which is current directory as default.

- **`ingres.macros.h`** -- ***Header File for INGRES/SQL***. This is a header file defined for use when object SQL product is INGRES/SQL. This file will be automatically included when you use RE/SQL statement INCLUDE SQL_HEADER. When you compile your application code (after preprocessing it with INGRES/SQL preprocessor), this file should be found by C compiler in suitable place, which is current directory as default.

When you include any header files in a SQL DECLARE SECTION, those files will be processed by translator. You need consider this case ***only if the object SQL products is INGRES/SQL***.

The original header file will be remained and a substitute header file will be generated by translator. In this case, the translator will locate and read original header files based on file names supplied in include statement, for example:

EXEC SQL BEGIN DECLARE SECTION;

#include "../include/header_file_name.h";

EXEC SQL END DECLARE SECTION;

First, the translator will process file `"../include/hfile.h"` and generate a substitute file `"../include/hfile.h.s"` in same directory.

And then, the file name in include statement will be modified to:

#include "../include/hfile.h.s";

Finally, the C compiler will use generated header files instead of original header files.

The translator has capability to deal with chain including files. For example, if you have a include statement in `"../include/hfile.h"`,

```
#include "hfile1.h"
```

then a substitute file `"hfile1.h.s"` is generated in the same directory, and the corresponding include statement is modified:

```
#include "hfile1.h.s"
```

and so on.

5.2 Using the Translator

5.2.1 Invoking the Translator

The translator, whose executable is **`sesql`**, can be invoked by the users as an UNIX command line. If **`sesql`** is located in current directory or you have add a search path for it, you can invoke it just using its name. Otherwise, you have to use a full name with path to invoke it.

The syntax of the command is as following:

sesql *InputFileName -Productidentifier* -o *OutputFileName*

Parameters

`InputFileName` identifies the name of input file containing the source code to be translated. You must specify a full name of input file, no any default is assigned for this parameter.

*-ProductIdentifier*is single letter which identifies the destination SQL product. For each translatable SQL product, which is ALLBARE/SQL or INGRES/SQL at this moment, a single letter must be selected as an identifier. Generally, you can use the first letter of the product name. Here we use A as the identifier of ALLBARE/SQL, I as the identifier of INGRES/SQL. The letter used is not case sensitive. You must specify this parameter, no default is assigned for this parameter.

*-o OutputFileName*specifies the name of output file containing the source code for destination SQL product. If you omit this parameter, a default name *InputFileName.sql* will be provided.

If the translator executes successfully, the following messages will be printed on the screen:

RE/SQL translator input file: XXX.XXX

RE/SQL granulator output file: XXX.XXX

XXX RE/SQL statements have been translated.

RE/SQL translating has successfully finished.

5.2.2 Errors Detected by Translator

The translator has error detecting capability. Three kinds of errors could be detected by the translator which are described as following:

Command Line Error

If you include any errors in **sesql** command, the command would not be executed, and a message will be displayed as following:

an error message...

Usage: sesql {input_file_name}

{A | I | ...}

[-O output_file_name]

where: A - to ALLBARE/SQL

I - to INGRES/SQL

default output file name is input_file_name.sql

Fatal Embedded SQL Statement Error

The so-called fatal error is the error which is so serious that the translator could not continuously process the source code. Generally, the fatal errors are related to various delimiters used by embedded SQL statements, which must be appear in pare such as string delimiter, comment delimiter, command terminator, etc. Once such an error occurs, the translator will terminate and an error message will be printed as following:

Fatal error: an error message... (at line XXX).

Non-fatal Embedded SQL Statement Error

The so-called non-fatal error is the error which is not so serious that the translator could continuously process the source code after an error message is printed. Though the error checking capacity of the translator is not complete, the most of the obvious errors could be detected. Once such an error occurs, the translator will count the number of the errors and an error message will be printed as following:

error: an error message... (at line XXX).

If the translator finds any error, which including fatal and non-fatal errors, the following messages will be printed on the screen:

RE/SQL translator input file: XXX.XXX

RE/SQL granulator output file: XXX.XXX.

. error message....

XXX RE/SQL statements have been translated.

RE/SQL translating has finished with XXX errors.

5.3 The Procedure to Use This Software

Based on the concept described in section 1.2, the users could use this software in the way of their convenience. However, this section will provide a proposed procedure for the general use of this software. The procedure will be described step by step in the following:

Step 1: You embed SQL statements with RE/SQL syntax in your application program which is your original source code file, e.g. its name is `sql-ex.s.c`. Multiple source code files are allowed. Here, we assume that all C functions including main function and functions with SQL statements are stayed in `sql-ex.s.c` file.

Step 2: You invoke **`sesql`**, which is the translator, to translate the RE/SQL syntax in original source code file into the syntax of the specified SQL product. The output file coming from this step is the source code for the specified SQL product, and is ready to be preprocessed by the preprocessor supplied with the product. For ALLBARE/SQL product, you can invoke **`sesql`** as following:

```
sesql sql-ex.s.c -a -o sql-ex.a.sc
```

For INGRES/SQL product, you can invoke **`sesql`** as following:

```
sesql sql-ex.s.c -i -o sql-ex.i.sc
```

Step 3: In this step, you begin to follow the procedure to use a specified SQL product. For ALLBARE/SQL product, you can use the preprocessor as following:

```
psqls EXAMPLEDBE -i sql-ex.a.sc -p sql-ex.a.c -d -r
```

where `sql-ex.a.sc` is the input file for the preprocessor, `sql-ex.a.c` is the output file from the preprocessor, `EXAMPLEDBE` is an example database environment.

For INGRES/SQL product, you can use the preprocessor as following:

```
esqlc -fsql-ex.i.c sql-ex.i.sc
```

where `sql-ex.i.sc` is the input file, `sql-ex.i.c` is the output file.

Step 4: In this step, you begin to follow the procedure to use compile and link your application. For ALLBARE/SQL product, you can do it as following:

```
cc -c sql-ex.a.c

cc sql-ex.a.o [dynamic.o] -lsql -lcl -lportnls -o sql-ex.r
```

where `sql-ex.a.r` is the executable for use with ALLBARE/SQL database environment.

For INGRES/SQL product, you can do it as following:

```
cc -c sql-ex.i.c

cc sql-ex.i.o [dynamic.o] $II_SYSTEM/ingres/lib/libingres.a \
  -lm -lc -o sql-ex.i.r
```

where `sql-ex.i.r` is the executable for use with INGRES/SQL database.

Note: It is worth mentioning that the users must maintenance only the original source code file, which is `sql-ex.s.c` in this procedure. It is possible that some errors could occur in step 2, 3 and 4.

The translator has an error checking capability by which the most obvious errors could be detected. However, error checking of the translator might not be sufficient so that some errors might be remained even if the step 2 had been passed successfully. By the way, all errors in the embedded SQL statements would be found in the step 3 by the preprocessor of the SQL product. It is also possible to find some C syntax errors, if any, in the step 4, because the step 2 and 3 check only for those embedded SQL statements in your application program. Wherever the errors occur, you must find and correct the errors in original source code file, and re-start the procedure from the step 2. The output file from the step 1 is a readable source code file too. Sometimes, you need to locate the errors in this file if any errors occur in the step 2.