SAP DB

C/C++ Precompiler

© Copyright 2001 SAP AG.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation.
For more information on the GNU Free Documentaton License see http://www.gnu.org/copyleft/fdl.html#SEC4.

Table of Contents

1	**Instructions for the User**	**1 -1**
2	**Embedding SAP DB Calls in an Application Program**	**2 -1**
2.1	Introduction	2 -1
2.2	General Rules	2 -1
2.3	The Declare Section	2 -6
2.4	Host Variables	2 -7
2.4.1	General Host Variable Conventions	2 -7
2.4.2	Predefined Data Types	2 -12
2.4.2.1	DECIMAL	2 -12
2.4.2.2	VARCHAR	2 -13
2.4.2.3	LONGFILE	2 -16
2.4.3	Structures in Host Variables	2 -17
2.5	Indicator Variables	2 -18
2.6	Generating Host Variables	2 -20
2.7	Generating Column Names	2 -22
2.8	Messages Returned via the SQLCA Structure	2 -24
2.9	The Whenever Statement	2 -30
2.10	Dynamic SQL Statements	2 -33
2.10.1	Dynamic SQL Statements without Parameters	2 -33
2.10.2	Dynamic SQL Statements with Parameters	2 -34
2.10.3	Dynamic SQL Statements with Descriptor	2 -36
2.11	SAP DB SQLDA Structure	2 -37
2.12	Using the Descriptor	2 -42
2.13	Overview of the Sequences of SQL Statements	2 -53
2.14	The Macro Mechanism	2 -54
3	**Programming Instructions**	**3 -1**
3.1	Processing Sets and Single Rows	3 -1
3.2	General Instructions	3 -7
3.3	Users	3 -8
3.4	Transactions	3 -9
3.5	Locks	3 -14

3.5.1	Instructions for the Locking	3 -19
3.6	Timeouts	3 -21
3.7	Optimizing SQL Statements	3 -23
3.7.1	Influence of the Search Condition	3 -25
3.7.2	Search Strategies	3 -27
3.7.2.1	Definition of Terms	3 -27
3.7.2.2	Conditions on Key Columns	3 -27
3.7.2.3	Conditions on Single-Column Indexed Columns	3 -28
3.7.2.4	Strategies with Conditions on Multiple-Column Indexed Columns	3 -31
3.7.2.5	Cost-Finding	3 -32
3.7.2.6	Conditions Combined by OR	3 -33
3.7.2.7	Postponing the Search to the FETCH Point in Time	3 -34
3.7.3	UPDATE	3 -35
3.7.4	The EXPLAIN Statement	3 -35
3.7.5	Joins	3 -40
3.7.5.1	EXPLAIN Statements for Joins	3 -41
3.7.6	Ordered Select Statements (single row processing)	3 -44
3.7.7	Instructions to Improve the Performance	3 -45
3.7.8	Additional EXPLAIN Information: Columns O, D, T, M	3 -47
3.7.9	Summary of the Present Chapter	3 -49
4	**Compatibility with Other Database Systems**	**4 -1**
4.1	ANSI	4 -1
4.2	DB2	4 -2
4.3	ORACLE	4 -3
4.4	ANSI ORACLE	4 -4
5	**The SAP DB Precompiler**	**5 -1**
5.1	SAP DB Precompiler Functions	5 -1
5.1.1	Translation of a SAP DB Application Program	5 -3
5.1.2	Precompiler Options	5 -5
5.1.3	User Specifications for a Precompiler Run	5 -12
5.1.4	Precompiler Output Files	5 -13
5.2	Functions of the SAP DB Runtime System	5 -14
5.2.1	Runtime Options	5 -14
5.2.2	User Specifications for the Application	5 -17
5.3	Precompiler Debugging Aids	5 -18
5.3.1	Testing at Precompilation Time	5 -18

5.3.2 The SAP DB Trace Facility ..... 5 -19
5.3.3 The Trace File ..... 5 -20
5.3.4 Profiling ..... 5 -23

6 SAP DB Precompiler Statements ..... 6 -1
6.1 Include Statements ..... 6 -1
6.1.1 exec sql include sqlca ..... 6 -1
6.1.2 exec sql include <filename> ..... 6 -2
6.2 Declare Statements ..... 6 -4
6.2.1 exec sql begin declare section ..... 6 -4
6.2.2 exec sql end declare section ..... 6 -4
6.3 Whenever Statements ..... 6 -5
6.3.1 exec sql whenever sqlwarning ..... 6 -5
6.3.2 exec sql whenever sqlerror ..... 6 -6
6.3.3 exec sql whenever sqlexception ..... 6 -6
6.3.4 exec sql whenever not found ..... 6 -6
6.3.5 exec sql whenever sqlbegin ..... 6 -7
6.3.6 exec sql whenever sqlend ..... 6 -7
6.4 SAP DB Statement ..... 6 -8
6.4.1 exec sql [<n>] <sql statement> ..... 6 -8
6.4.2 exec sql [<n>] <array statement> ..... 6 -8
6.5 Macro Statement ..... 6 -10
6.5.1 exec sql set macro %nnn = <macroline> ..... 6 -10
6.6 Dynamic SQL Statements ..... 6 -11
6.6.1 exec sql [<n>] prepare ..... 6 -12
6.6.2 exec sql [<n>] describe ..... 6 -13
6.6.3 exec sql [<n>] execute ..... 6 -14
6.6.4 exec sql [<n>] execute immediate ..... 6 -15
6.7 SAP DB Cursor Statements ..... 6 -16
6.7.1 exec sql [<n>] declare ..... 6 -17
6.7.2 exec sql [<n>] open ..... 6 -17
6.7.3 exec sql [<n>] fetch ..... 6 -18
6.7.4 exec sql [<n>] close ..... 6 -18
6.8 Trace Statements ..... 6 -19
6.8.1 exec sql set trace on ..... 6 -19
6.8.2 exec sql set trace long ..... 6 -19
6.8.3 exec sql set trace off ..... 6 -20
6.8.4 exec sql set trace line ..... 6 -20

6.9 SAP DB Database Statements ..... 6 -21
6.9.1 exec sql connect ..... 6 -21
6.9.2 exec sql [<n>] set serverdb ..... 6 -21
6.9.3 exec sql [<n>] reconnect ..... 6 -22
6.9.4 exec sql cancel ..... 6 -22
6.10 Other Statements ..... 6 -23
6.10.1 exec command ..... 6 -23
6.10.2 exec sql version ..... 6 -24
6.11 exec sql [<n>] proc ..... 6 -25

7 Appendix 1: Syntax of the Declare Section ..... 7 -1

8 Appendix 2 ..... 8 -1
8.1 Internal Return Codes of the Precompiler ..... 8 -1
8.2 Return Codes of a SAP DB Application ..... 8 -2
8.3 Return Codes of the CPC Call ..... 8 -3

9 Index... ..... 9- 1

1 Instructions for the User

This user manual describes how to embed SAP DB calls in applications written in the programming language C. It is also possible to embed SAP DB calls in C++ programs, provided that the restrictions described in the following sections are taken into consideration for SQL statements, especially for declare sections.

The manual contains an introduction to the concepts, based on examples.

Complete examples are distributed together with the SAP DB installation kit.

Lists of the SQL statements are included for reference purposes. Messages and error codes are described in the Manual Messages: SAP DB, and the operating system dependencies are detailed in the platform-specific SAP DB User manual.

The syntax used for SQL statements is identical to that used in the Reference manual. To summarize, the following notation applies:

```
 ::=     "is in the permissible form"
 <...>   syntax variable which must be replaced
...|...  alternative
 [...]   optional
```

2 Embedding SAP DB Calls in an Application Program

2.1 Introduction

The database language SQL is the interface between application programs and the SAP DB database. Database operations are invoked by SQL statements embedded in application programs. By means of special program variables - the so called host variables - values are interchanged between the program and the database. A language-specific SAP DB precompiler checks the syntax and semantics of the embedded SQL statements and transforms them into calls of procedures of the SAP DB runtime system. The compiler of the programming language used translates the source program generated by the precompiler into machine code.

This manual requires basic knowledge of the elementary SQL statements. All the other possible SQL statements are described in the Reference Manual: SAP DB (7.2), and the meanings of the return codes are described in the Manual Messages: SAP DB.

2.2 General Rules

The keywords "exec sql" precede SQL statements in order to distinguish them from statements of the corresponding programming language. These keywords are not components of the SQL statement.

In addition to SQL statements, it is possible to perform COMMAND system calls and to call DB procedures from programs. The keywords "exec command" precede COMMAND system calls, "exec sql [<n>] proc" precede DB procedures. Otherwise, the rules which apply to SQL statements are valid.

A semicolon terminates each SQL statement.

The keywords "exec" and "sql" must be placed on the same line. They may only be separated by blanks. Each SQL statement, including the "exec sql" prefix, can extend to several lines.

An SQL statement can be interrupted by comments in C format (designated by /* and */). Within an SQL statement, comments can also be placed at the end of a line. They must be preceded by the characters "--".

Identifiers declared in the program must not begin with the characters "sq".

By means of the SQL statement "exec sql include <filename>", any arbitrary source text can be inserted into the program. The source text is stored in the specified file and must not contain an include statement of its own.

Return codes of the database system which, for example, inform about error occurrences, are returned via the global structure SQLCA (SQL Communication Area). The component sqlcode contains encoded messages regarding the execution of an SQL statement. This code should be checked whenever SAP DB has been called.

The variables of the programming language which are used for the transfer of values from and to the database are called host variables. They are declared in a special section called the declare section.

The parts to be analyzed by the precompiler (declare section, include sqlca, and SQL statements) must not be components of preprocessor include statements.

Host variables which are used in SQL statements are preceded by a colon. They have the form ":var" and can therefore be distinguished from SAP DB identifiers. When they are used outside of SQL statements, the colon is omitted.

Example:

```
..... Code .....

exec sql begin declare section;
char  tit [3];
char  firstn [11], lastn [11];
exec sql end declare section;

..... Code .....

/* Creating the table customer */
exec sql create table customer
         (cno fixed (4) key,  title char (2),
          firstname char (10),name char (10));

/* Reading the values */

..... Code .....

/* Inserting into the database */
exec sql insert into customer
         (cno, title, firstname, name)
         values (100, :tit, :firstn, :lastn);

/* Return code output in the case of error */

if (sqlca.sqlcode != 0)
printf ("%d       ",sqlca.sqlcode);

..... Code .....
```

To process null values (undefined values) of the SAP DB database system, special host variables, called indicator variables (indicators), are required. These are specified - also with a preceding colon - immediately after the corresponding host variable. Their value indicates whether null values have been encountered or whether character strings have been truncated when they were passed to host variables.

Within SQL statements, host variables can only be used at those positions where the SQL syntax allows parameters (for more details refer to the Reference manual). In particular, no table names can be specified via host variables. To substitute table and column names, there is a separate macro mechanism available.

When the option check is set, the precompiler tests the SQL statements by making one sequential pass through the program. For a correct check of the SQL statements, it is therefore important to observe the following conventions about the static position of SQL statements:

- Create Table precedes Insert, Update, Delete, Select. These five precede Drop Table.
- Select precedes Fetch. (An unambiguous assignment of select statement and fetch loop must be possible.)
- Declare Cursor precedes Open, Open precedes Fetch, Fetch precedes Close.
- In the case of dynamic statements, Prepare precedes Execute, Declare, Describe, Open, Fetch, and Close.
- In the case of macros, it is necessary to call set macro before using the macro parameter specified in an SQL statement.

These conventions do not affect the dynamic order of the SQL statements at execution time.

Deviations from the above conventions result in confusing warnings when precompiling with the option check, because the context of an SQL statement is not correct. In such a case, the option check makes no sense.

A program can also operate in multi-db mode. Thus operations on up to eight different SAP DB databases can be performed simultaneously; i.e., eight concurrent sessions can exist on different databases. SQL statements of the second database session begin with "exec sql 2". The database session number (sessionno) 2 must be separated with blanks from "exec sql". Similarly, the third database session begins with "exec sql 3" and the fourth with "exec sql 4", etc.

Prior to the connect for the second and any other database session, the servernode and the name of the respective database can be specified with "exec sql n set serverdb <serverdb> [on <servernode>]". If "set serverdb" is not specified, the values are taken from the XUSER file. The keywords and the session number "n" must be placed on the same line and may only be separated by blanks.

Calling xuser creates the XUSER file. It is possible to make eight different entries with "userkey", "username", "servernode", "serverdb", "timeout", "isolation level", "sqlmode", and "locale". (For the generation of the XUSER file, see the platform-specific SAP DB User manual.)

2.3 The Declare Section

The declare section defines an interface for the interchange of values between the database system and the program. All the host variables and indicators to be used in SQL statements are made known to the precompiler in this interface.

The declare section begins with the SQL statement "exec sql begin declare section" and ends with "exec sql end declare section".

The declare section can be repeated within the program.

The basic data types 'struct' and, if defined in a declare section, type names, are permitted. As storage classes, 'typedef', 'static', 'extern', and 'auto' can be specified. Declarations can be made outside functions as well as within functions. Declarators can contain one pointer declarator and up to four array declarators. Arrays of pointers are valid, but not pointers to arrays. If character pointers are used as SQL parameters, the precompiler cannot check the lengths and therefore returns a warning. At runtime, the character pointer must point to a character string delimited by zero. The array boundaries in declarators must also be specified for the storage class 'extern'; in this case, they must lie within the range of the corresponding external definitions.

Besides declarations, a declare section can contain type definitions and #define lines (constant definitions) without parameters for positive integers. The names defined hereby can be specified as array boundaries.

The parts that are to be analyzed by the precompiler (declare section, SQL statements) must not be components of #include files, but must be specified in the source text or in "exec sql include" files.

For efficiency reasons, it is advisable to include only data in the declare section that will actually be used as host variables.

The syntax allowed within the declare section is described in Appendix 1.

2.4 Host Variables

2.4.1 General Host Variable Conventions

- The identifier of a host variable can have a maximum length of 32 characters.
- A host variable must not begin with "sq".
- A host variable can be a structure or an array.
- A structure can contain arrays.
- Arrays of structures are possible.
- Pointer variables of permitted data types and structures are possible. Pointers to pointers or pointers to arrays are not allowed.
- Host variables of type char [ ] can be declared one byte longer than the corresponding SAP DB columns so that they are long enough to receive the delimiting zero byte. When transferring to SAP DB, the contents of the variable (without the zero byte) are passed and the SAP DB column is padded with blanks, if necessary. When transferring to host variables, trailing blanks are deleted and the delimiting zero byte is placed immediately after the last non-blank character or, if there are no blank characters, right on the last character (in this case, the last character will be overwritten!).
- According to the C notation, a host variable name in uppercase letters is not identical to the same host variable name in lowercase letters.

The scalar C host variables can be contrasted with the corresponding SAP DB data types:

Description	Host Variable	ADABAS Data Type
numeric: decimal	float / double long float (4 / 8 / 8 bytes)	fixed (n,m)
integer	int / short int / long int (4 / 2 / 4 bytes)	fixed (10),(5) (4B, 2B)
floating point	float / double / long float (4 / 8 / 8 bytes)	float (6),(15) (4B, 8B)
boolean:	all numeric data types 0 other values	BOOLEAN false true
alphanumeric: character string with binary concluding zero byte	char ... [n+1] n < 4000 n >= 4000	char (n) long
character	char	char (1)
date, time with binary concluding zero byte	char [9]	date, time
timestamp with binary concluding zero byte	char [21]	timestamp

If the data types do not correspond to each other but are of the same category (numeric or character string), then they will be converted. Also numeric values are converted into character strings and vice versa. At precompilation time, the precompiler tests whether the target variable can receive the maximum value of the source variable when values are transferred between database and program. If this is not the case, a warning is issued in the precompiler listing.

SAP DB columns of type "date" hold date information in the form YYYYMMDD (year, month, day); SAP DB columns of type "time" hold time information in the form HHHHMMSS (hour, minute, second), SAP DB columns of the type "timestamp" hold information in the form YYYYMMDDHHMMSSMMMMMM (year, month, day, hour, minute, second, microsecond). The data type "timestamp" is only valid in sqlmode INTERNAL.

Options set during precompilation allow you to vary the representation of "date", "time", and "timestamp".

The following table indicates the conversion possibilities (x) and the errors and warnings that can occur:

Data Type C	Data Type ADABAS					
Host Variable	Fixed	Float	Boolean	Long	Char	Varchar	Date/Time
short int/int/ long int	x 1 2	x 1b 2	x 7	- 4	x 5 6	- 4	- 4
float/double/ long float	x 1a 2	x 2	x 7	- 4	x 5 6	- 4	- 4
char ...[n]	x 5 6	x 5 6	- 4	x 3	x 3	x 3	x 3
char	x 5 6	x 5 6	- 4	- 4	x 3	x 3	x 3

1 An overflow can occur in this case (sqlcode < 0).

1a See 1, if host variable -> SAP DB variable.

1b See 1, if SAP DB variable -> host variable.

2 In such a case, any places after the decimal point, as well as any mantissa places, will be truncated if necessary (indicator value: > 0).

3 In this case, any characters to the right will be truncated, if necessary, whereby sqlwarn1 will be set and the length of the associated SAP DB output column will appear in the indicator (indicator value > 0).

4 Not allowed (sqlcode != 0).

5 An overflow can occur when numeric values are converted into character values (sqlcode < 0).

6 An overflow or an invalid number (sqlcode < 0) can occur when character values are converted into numeric values.

7 The value zero is mapped to "false", all values not equal to zero are mapped to "true".

2.4.2 Predefined Data Types

The following data types are predefined by the precompiler. They can be used in the declare section for the declaration of host variables:

2.4.2.1 DECIMAL

```
<dcmltysp>   ::= <dcmldef> | <dcmlref>

<dcmldef>    ::= 'DECIMAL' <dcmltag> '{' <dcmlscale> '}'

<dcmlref>    ::= 'DECIMAL' <identifier>

<dcmltag>    ::= <identifier> | <empty>

<dcmlscale>  ::= <dcmldigits> | <dcmldigits> ',' <dcmlfract> |
                 <empty>

<dcmldigits> ::= <identifier> | <unsigned_integer>

<dcmlfract>  ::= <identifier> | <unsigned_integer>

<empty>      ::=
```

The data type 'DECIMAL' can be used for the declaration of variables to which fixed point numbers in packed decimal representation are assigned. The number of decimal digits is defined by <dcmldigits> (<= 15), the number of fractional digits is defined by <dcmlfract> (<= <dcmldigits>). The default is 5, 0. These specifications are used during runtime for the conversion of SQL parameters and database columns of the type "fixed". The usage of the data type 'DECIMAL' exactly correlates the value ranges of parameters to those of database columns, so that rounding and overflow errors are not possible. The representation in memory is done in a byte array, where the decimal digits are stored in successive half bytes. The sign is coded in the rightmost (least significant) half byte (+ as 0XC and - as 0XD). The digits occupy the remaining position in right-justified representation.

The precompiler replaces the declaration of 'DECIMAL' variables by a structure declaration:

```
DECIMAL {n, f} v;                  is replaced by
struct {char arr[m];} v,           where
m = (n + 2)/ 2 (integer division)  applies.
```

Examples of valid 'DECIMAL' declarations are:

```
DECIMAL {} a, *b, c [20]; /*  variable declaration
                  with the default values 5, 0 */

DECIMAL {6} d;          /* d is a variable with 6
                  decimal digits */

DECIMAL LONGDEC {15, 2};  /* LONGDEC is a tag for decimal
                  numbers with 15 digits and
                  2 fractional digits */

DECIMAL LONGDEC e, f;     /* LONGDEC is used for the declaration
                  of the variables e and f */
```

2.4.2.2 VARCHAR

```
<vchtysp>     ::= 'VARCHAR' | 'varchar'
```

The data type 'VARCHAR' can be used for the declaration of variables to which character strings of variable length are assigned. A VARCHAR declaration must contain at least either an array declarator or a pointer declarator. The last array declarator defines the maximum length (<= 32767) of the variable. For a

VARCHAR declaration with pointer declarator, the maximum length is undefined; the program has to assign storage space during runtime. The current length of a VARCHAR variable is defined by the length field, zero bytes are ignored in the calculation of the length. The precompiler replaces the declaration of VARCHAR variables by a structure declaration:

```
VARCHAR v {n};                    is replaced by

struct {
        unsigned short len;
        unsigned char arr [n];
        } v;                        where

len                                 is assigned the current length
                                    of the character string and
arr                                 is assigned the characters.

VARCHAR  *v;                        is replaced by

struct {
        unsigned short len;
        unsigned char arr [1];
        } *v;
```

Examples of valid varchar declarations are:

```
VARCHAR a [21], b [100] [133];        /* a is a variable of
                                         length 21 and b an
                                         array of 100 elements
                                         of length 133 */

typedef VARCHAR LONGSTRING [ 65534 ]; /* type definition */

LONGSTRING c, d;                      /* c and d are variables
                                         of type LONGSTRING */

typedef VARCHAR *PVC;                 /* definition of the
                                         pointer type PVC */

PVC p;                                /* declaration of the
                                         VARCHAR pointer p */
```

For example, the following statements can be used to assign storage space to p:

```
n = 100; /* maximum length of the VARCHAR variable */

p = (PVC) malloc (sizeof (p->len) + n * sizeof (p->arr));
```

VARCHAR pointers of a fixed maximum length can be declared in the following way:

```
typedef VARCHAR VC30 [30]; /* VARCHAR type of length 30 */

VC30 *q;                   /* q is a pointer to a VARCHAR of
                              the maximum length 30 */
```

The following statement is used to assign storage space to q:

```
q = (VC30* ) malloc (sizeof (VC30));
```

2.4.2.3 LONGFILE

```
<vlftysp>    ::= 'LONGFILE' | 'longfile'
```

The data type 'LONGFILE' specified in a host variable declaration allows contents of a file to be transferred directly to or from long columns. The host variable must be a character string of any length; its value must be the filename. The SQL statements "insert" and "update" pass the content of the file specified in the host variable to the long column. When selecting a long column, its content is added to the end of the file. If this is not desired, the file must be deleted beforehand (see Section 6.10.1, "exec command"). Errors occurring during file processing have the effect that the SQL statement is aborted and "sqlcode" set.

```
 ..... Code .....

exec sql begin declare section ;

longfile f1[20] = "document.doc";
longfile f2[30] ;

 ..... Code .....

exec sql end declare section ;

 ..... Code .....

    exec sql create table documents(title char (40) key,
                                    document long byte) ;
    exec sql insert into documents values (:id, :f1) ;

 ..... Code .....

    strcpy (f2, "tempdocument.doc") ;
    exec command 'rm tempdocument.doc' result :ret ;
    exec sql select direct document into :f2 from documents
             where key = :id ;

 ..... Code .....
```

2.4.3 Structures in Host Variables

A structure specified as parameter ":var" is expanded into its individual components. This is useful, e.g., in the into clause of the select statement. Data is passed in the order of component declarations. The mapping to table columns described here can only be applied to arrays if the precompiler option compatible is specified. Compare the Section 6.4.2, "exec sql [<n>] <array statement>". For component arrays with several dimensions, the last dimension is run through first. Thereby the indicator variables needed can also be specified as array or structure which must contain at least as many components as the host variable.

2.5 Indicator Variables

If null values (undefined values) are to be processed or truncations are to be recognized when assigning column values to host variables, it is necessary to specify an indicator variable in addition to each host variable. These indicator variables must be declared as (long or short) int in the declare section.

An indicator variable is specified in an SQL statement after the pertinent host variable. It begins, like the host variable, with a colon.

Possible Indicator Values:

Indicator Value	Meaning
= 0	The host variable contains a defined value. The transfer was free of error.
= -1	The value of the table column corresponding to the host variable is the null value.
= -2	When computing an expression, an error occurred. The value of the table column corresponding to the host variable is not defined.
> 0	The host variable contains a truncated value. The indicator value indicates the original column length.

When values are transferred from optional columns (i.e., columns where null values can occur) to host variables by a select or fetch statement, an indicator variable must be specified, otherwise a negative sqlcode will be issued and the condition sqlerror will be set when a null value is selected. When precompiling with the option check, the message will appear as a warning at precompilation time.

The indicator variable shows whether a null value has been selected (indicator value is -1). It is also possible to enter a null value into a column by means of the indicator variable. The indicator variable must be set to the value -1 before calling it within an SQL statement. The value in the corresponding host variable will be ignored.

An indicator value > 0 is only set for result host variables. It defines the original column length in the database.

Example:

```
 ..... Code .....

exec sql begin declare section;
char firstn [11], lastn [11];
int  firstnind, lastnind;
exec sql end declare section;

 ..... Code .....

/* Inserting a null value */

firstnind  = -1;
strcpy (lastn, "Tailor");
lastnind = 0;

exec sql insert into customer (firstname, name)
     values (:firstn :firstnind, :lastn :lastnind);

/* Testing for truncation */

exec sql select first name
         into   :lastn :lastnind
         from   customer;

if (lastnind > 0)
printf ("%d      ", lastnind);

 ..... Code .....
```

2.6 Generating Host Variables

A structure can be generated for a table by means of the include statement. This structure can then be used as a host variable. To do so, the option check must be set and the file <filename> must not exist. Then a database session with the predefined user specifications will be opened in order to be able to fetch the corresponding pieces of information from the database and to generate the file <filename>.

The file <filename> contains the structure which was generated from the table. The name of the table (default) or any other name (as clause) can be given to this structure. Another structure can be generated in addition which can be specified as an indicator (ind clause). Component names are derived from a table's column names. The names of the indicator structure are also derived from the table name and column names; they are provided with the prefix "I".

If long columns are contained in the table, a character string of 8240 characters is generated as host variable for each long column. The precompiler returns a warning. The programmer can use an editor to insert the desired length or another host variable.

Example:

A table can be defined by:

```
create table example (
                A fixed (5),
                B fixed (8),
                C fixed (5, 2),
                D float (5),
                E char (80))
```

Then a program can contain the following statements:

```
exec sql begin declare section;
exec sql include "example.h" table EXAMPLE as struct ind;
struct EXAMPLE s, sa [10], *sp;
struct IEXAMPLE indi;
exec sql end declare section;

...
exec sql select * from example;
...
exec sql fetch into :s :indi;
...
exec sql fetch into :sa [4] :indi;
...
sp = &sa [9];
exec sql fetch into :sp :indi;
```

The include statement generates the declarations:

```
struct EXAMPLE {
                short A;
                long B;
                float C, D;
                char E [81];
                };

and

struct IEXAMPLE {
                short IA, IB, IC, ID, IE;
                };
```

2.7 Generating Column Names

The name of a structure or array variable can also be specified as "!<var>", e.g., in the <select list> of the select statement. This has the same effect as the explicit specification of the individual components in their order of definition. Instead of the exclamation mark, the character tilde (~<var>) can also be used. The specifications <[authid.]tablename.> can be placed before "!<var>". They have the effect that they are inserted in front of each column name.

The column names are derived from the variable declarations according to the following rules:

1. The name of the variable "var" itself as well as any specified index references are not taken into account (but compare rule 6).

2. If "cmp1" is a component of the first level, then all the assigned column names begin with "cmp1".

3. If "cmpn" is a component of the n-th level with n > 1, then the assigned column names are continued with "_cmpn".

4. If "cmp" is any component array with m dimensions (m <= 4), then the assigned column names are continued with "i1_i2_..._im", whereby i1 ... im can assume all values from 1 to the number of indexes of the particular dimension and the last dimension is run through first. Leading zeros of the indexes are not taken into account. If the array is single-dimensional, only "i1" (without "_") is generated. If the array components are scalar, every component is assigned to exactly one column name.

5. If "cmp" is any scalar component, then it is assigned to exactly one column name.

6. If "var" is an array with scalar elements, then, deviating from rule 1, the assigned column names begin with "var" and are continued according to rule 4.

Instead of the specification "!<var>", any structured component of "var" can be selected, e.g., "!var.x.y". In this case, a corresponding subset of column names is

generated according to the rules 1 to 4. The column names formed according to the rules 1 to 6 can contain up to 18 characters. Example:

```
..... Code .....

exec sql begin declare section;
typedef char string11 [11];
struct {
       char tit [3];
       struct {
         string11 lastn, firstn [3];
         } name;
       } person;
exec sql end declare section;

..... Code .....

exec sql create table customer
    (cno fixed (4) key, tit char (2),
     name-lastn char(10), name-firstn1 char(10),
     name-firstn2 char(10),name-firstn3 char(10));

/* Reading the values */
..... Code .....

exec sql insert into customer
         (cno, !person )
         values (100, :person);

..... Code .....

/* Has the same effect as the above INSERT */
  exec sql insert into customer
       (cno, tit, name-lastn,
        name-firstn1, name-firstn2, name-firstn3)
       values (100, :person.tit,
       :person.name.lastn, :person.name.firstn[0],
       :person.name.firstn[1],
       :person.name.firstn[2]);

..... Code .....
```

2.8 Messages Returned via the SQLCA Structure

The data structure SQLCA is automatically included in each SAP DB application program. The database stores information about the execution of the last SQL statement in the components of the SQLCA. This section explains the meanings of the SQLCA components that are important for the application programming; the exact structure of SQLCA is described after this section.

The components of the SQLCA have the following meanings:

sqlcaid
is a character string of length 8.
It contains the character string "SQLCA " and serves to find the SQLCA during analyzis of a dump.

sqlcabc
is a 4-byte integer.
It contains the length of the SQLCA in bytes.

sqlcode
is a 4-byte integer.
It contains the return code. The value 0 indicates the successful execution of a statement. Codes greater than 0 indicate normal exceptional situations which can occur during the execution of an SQL statement and which should be handled within the program. Codes smaller than 0, on the other hand, indicate errors which can arise through invalid SQL statements, the violation of restrictions, or database system errors; these errors should result in aborting the program. Precompiler errors lie in the ranges from -700 to -899 and from -9801 to -9820. They are described in the Manual Messages: SAP DB.

Exceptional situations can be:

```
+100   ROW NOT FOUND
+200   DUPLICATE KEY
+250   DUPLICATE SECONDARY KEY
+300   INTEGRITY VIOLATION
+320   VIEW VIOLATION
+350   REFERENTIAL INTEGRITY VIOLATED
+360   FOREIGN KEY INTEGRITY VIOLATION
+400   LOCK COLLISION
+450   LOCK COLLISION CAUSED BY PENDING LOCKS
+500   LOCK REQUEST TIMEOUT
+600   WORK ROLLED BACK
+650   WORK ROLLED BACK
+700   SESSION INACTIVITY TIMEOUT (WORK ROLLED BACK)
+750   TOO MANY SQL STATEMENTS (WORK ROLLED BACK)
```

sqlerrml
is a 2-byte integer.
It contains the length of the error message from "sqlerrmc".

sqlerrmc
is a character string of length 70.
It contains an explanatory text for any sqlcode value not equal to zero. The user can select the language of this text (e.g., English or German) via the SET menu of the tool components.

sqlerrd
is an array of six 4-byte integers.
Sqlerrd indicates in the third element how many rows have been retrieved, inserted, updated or deleted by the SQL statement. If it contains the value -1, the number of rows retrieved is not known. If errors occurred in array statements, it contains the number of the last row which was processed correctly.
The fifth element contains the statement's runtime within the kernel in seconds.
If syntax errors occurred in a SAP DB statement, the sixth element contains the position in the command buffer where an error was detected. The indicated position does not refer to a position within the program text, but to a position within the command buffer at the point in time of sending to the SAP DB kernel. For array statements, this value is undefined. In all the other cases, it is zero.
A value not equal to 0 is also written to the trace file.
The other elements in the array are not used.

sqlwarn0
is a character.
It is set to "W" if at least one of the warnings "sqlwarn1" to "sqlwarnf" has the value "W". Otherwise, "sqlwarn0" has the value " ". The characters "W" for warning set and " " for warning not set apply to all warnings.

sqlwarn1
is a character.
It indicates whether character strings (SAP DB data type "char") have been truncated during the assignment to host variables. When this character has the value "W", an indicator variable may exist which indicates the length of the original character strings.

sqlwarn2
is a character.
It is set if null values occurred and were ignored during the execution of the SQL functions count (not countc(*)), min, max, avg, sum, stddev or variance.

sqlwarn3
is a character.
It is set if the number of result columns of a "select" or "fetch" is not equal to the number of host variables in the into clause.

sqlwarn4
is a character.
It is set if an "update" or "delete" has been executed without a where clause, i.e., on the entire table.

sqlwarn6
is a character.
It is set if, for an operation on date or timestamp in the database, an adaptation to a correct date was made.

sqlwarn8
is a character.
It is set if, for the generation of a result table, it was necessary to search through the entire table(s).

sqlwarna
is a character.
It is not set at the moment. **sqlwarnb**
is a character.
It is set if a time value is > 99 (or > 23 in USA format). The value will be corrected to modulo 100 (or 24).

sqlwarnc
is a character.
It is set if, in the case of a select statement, more rows have been found than are allowed by rowno in the where clause.

sqlwarnd
is a character.
It is set if, in the case of a select statement, the search was performed via a simply-indexed column which may contain null values. Null values are not written to the index list; i.e., the select statement must be formulated differently if you want to obtain the null values.

sqlwarne
is a character.
It is set if the value of the secondary key has changed (transition to the next index list) during the execution of one of the SQL statements "select next", "select prev", "select first" or "select last" by means of a secondary key.

sqlwarnf
is a character.
It is not set at the moment.
sqlresn
is a character string of length 18.
After calling a select statement, it contains the result table name. After other calls "sqlresn" is set to blank.

sqldatetime
is a binary number of 2 bytes length.
It indicates the way in which the data types date, time, and timestamp are interpreted. Values: 1 = Internal, 2 = ISO, 3 = USA, 4 = EUR, 5 = JIS.

The components of SQLCA that are not described in detail are required for internal purposes.

SQLCA Structure

The following data area is automatically included in order to receive the SAP DB error messages:

```
typedef struct     {
                      sqlint4       sqlenv;
                      char          sqlcaid[8];
                      sqlint4       sqlcabc,
                                    sqlcode;
                      sqlint2       sqlerrml;
                      char          sqlerrmc[70];
                      char          sqlerrp[8];
                      sqlint4       sqlerrd[6];
                      char          sqlwarn0,
                                    sqlwarn1,
                                    sqlwarn2,
                                    sqlwarn3,
                                    sqlwarn4,
                                    sqlwarn5,
                                    sqlwarn6,
                                    sqlwarn7,
                                    sqlwarn8,
                                    sqlwarn9,
                                    sqlwarna,
                                    sqlwarnb,
                                    sqlwarnc,
                                    sqlwarnd,
                                    sqlwarne,
                                    sqlwarnf;
                      char          sqlext[12];
                      sqlnname      sqlresn;
                      sqlint2       sqlfill1,
                                    sqlfill2,
                                    sqlfill3,
                                    sqlfill4,
                                    sqlmfetch;
                      sqlint4       sqltermref;
                      sqlint2       sqlfill5,
                                    sqldbmode,
                                    sqldatetime;
                      char          sqlstate [6];
                      sqlargline    sqlargl;
                      sqlgatype     *sqlgap;
                      sqlratype     *sqlrap;
                      sqloatype     *sqloap;
                      sqlmatype     *sqlmap;
                      sqlmftype     *sqlmfp;
                      sqldiaenv     *sqlplp;
```

```
    SQLERROR      *sqlemp,
       sqlcxatype    sqlcxa;
}
sqlcatype;
```

2.9 The Whenever Statement

The function of whenever statements is to perform general error and exception handling routines for all subsequent SQL statements. Different kinds of errors and exceptions can thereby be distinguished. In application programming, whenever statements help to handle error situations. Whenever statements should always be used when "sqlcode" or "sqlwarning" is not checked after every individual SQL statement. Those SQL statements are considered to be subsequent that textually (statically) follow the whenever statement in the program. A whenever error handling routine is valid until it is changed by another whenever statement.

The whenever statement checks four classes of SAP DB return codes (values of sqlwarn0 or sqlcode) and offers four standard actions that can be taken in response. The general format of the whenever statement is:

whenever <condition> <action>

One of the following cases can be specified for <condition>:

sqlwarning	exists when "sqlwarn0" is set to "W". Then, at least one "sqlwarning" is set.
sqlerror	indicates that "sqlcode" has a negative value, which means that an unexpected error occurred. The program should be aborted and analyzed. The possible error codes and adequate user actions in response to them are described in the Manual Messages: SAP DB.

sqlexception	indicates a positive sqlcode value greater than 100. Messages of this kind generally are exceptional cases which should be handled within the program.
not found	is valid when no (further) table row has been found and "sqlcode" has the value +100 (ROW NOT FOUND).

One of the following cases can be specified for <action>:

stop	causes the regular resetting of the current transaction and abortion of the program (COMMIT WORK RELEASE).
continue	does not perform an action when the condition occurs and therefore suspends another whenever condition which has previously been set.
go to <lab>	causes a jump to the indicated label (maximum length of the label: 45 characters).
call <proc>	has the effect that the specified function will be executed. The function call can have a maximum length of 50 characters, parameters inclusive.

If no whenever statements are included, "continue" is the default action for all conditions. User error handling can be implemented by checking "sqlcode". If another whenever statement has previously been specified, it must be suspended by "whenever ... continue".

If SQL statements are issued in the whenever error handling code, then the corresponding whenever action must be set to "continue" in order to avoid an endless loop generated by repeated calls of the whenever error handling routine.

The whenever statements can also be used to call C functions before and after each SQL statement (see Section 6.3, "Whenever Statements").

Example:

```
..... Code .....

    exec sql whenever sqlwarning call warnproc();
    exec sql delete from reservation
             where cno = :cuno;
    exec sql whenever sqlerror stop;

..... Code .....

    exec sql select cno, name from customer ...;

..... Code .....

    exec sql whenever sqlerror call errproc();
```

In this example, the procedure "warnproc" is called for all SQL statements when "sqlwarn0" is set. The standard error handling for a negative sqlcode, on the other hand, varies. Only the select statement lies within the scope of the first whenever sqlerror statement. The second whenever sqlerror statement refers to all subsequent SQL statements. The delete statement is not placed within the scope of a (preceding) whenever sqlerror statement. Therefore, the default action "continue" is operative.

2.10 Dynamic SQL Statements

Dynamic SQL statements serve to support applications which can only decide at runtime which of the SQL statements is to be executed. For example, a user might want to enter an SQL statement from the terminal and to have it executed at once.

Dynamic statements can also be executed in a DB2- or ORACLE-compatible way. In such a case, the option sqlmode must be enabled for precompilation (see Section 4, "Compatibility with Other Database Systems").

2.10.1 Dynamic SQL Statements without Parameters

In the simplest case, an SQL statement is dynamically executed which returns no results except for setting the "sqlcode" in the SQLCA; i.e.: this statement has no host variables. For such a case, there is the dynamic SQL statement "execute immediate".

The SQL statement which is to be executed dynamically can either be specified in a host variable or as a character string (literal).

Example:

```
..... Code .....

exec sql begin declare section;
char statement [40];
exec sql end declare section;

..... Code .....

strcpy (statement, "insert hotel
                    values (27,'Sunshine')");
exec sql execute immediate :statement ;
exec sql execute immediate 'commit work';

..... Code .....
```

2.10.2 Dynamic SQL Statements with Parameters

Dynamic SQL statements can be parameterized with host variables; e.g., the values of the insert statement are to be entered at the terminal.

For dynamic execution of parameterized SQL statements, the SAP DB precompiler offers a procedure which consists of two steps:

- Preparation of the SQL statement to be executed dynamically by means of the prepare statement. In this phase, it will be stipulated how many host variables are to be inserted at which positions within the SQL statement. The positions intended for the host variables are identified by a ?.

- Execution of the prepared SQL statement by means of the execute statement. The SQL statement to be executed is identified by a name which has been given to it during the preparation. In this phase, the statement receives the actual host variable values. Once prepared, an SQL statement can be executed with execute as often as desired, whereby the host variables can vary for every specification.

Example:

```
strcpy (statement, "insert hotel
                    values (27,'Sunshine')");
exec sql prepare stat1 from :statement ;

..... Code .....

exec sql execute stat1;

..... Code .....

strcpy (statement, "select next name,price into?,?
                    from hotel key hno=?");
exec sql prepare stat2 from :statement ;
exec sql execute stat2 using :name,:price,:hno
```

The following example shows how a result table can be processed by means of a cursor. The first step is to prepare the select statement. The second step is to open the result table whereby host variables are assigned to the position indicators. For this reason, the select statement can only be executed at this point in time. The fetch statement is also executed dynamically according to the procedure described above.

Example:

```
       ..... code .....

     strcpy (stm, "select * from hotel where zip = ? and
                   price = ? order by price") ;
     exec sql prepare sel from :stm ;

     printf ("Enter zip code of city. ") ;
     scanf ("%d\n", &plz) ;
     printf ("Enter upper price limit. ") ;
     scanf ("%f\n", &price) ;

     exec sql open csel using :zip, :price ;

     strcpy (stm, "fetch csel into ?, ?, ?, ?, ?") ;
     exec sql prepare fet from :stm ;

/*Processing the result table

     exec sql execute fet using :hno,:name,:zip,:city,:price ;
     while (sqlca.sqlcode == 0)
           {
            /* Processing the data from the current row */

            exec sql execute fet ;
           }

     exec sql close csel ;

     ..... code ...
```

2.10.3 Dynamic SQL Statements with Descriptor

The usage of the descriptor allows different SQL statements and tables to be used for every execution of the SAP DB application without having to change the source code. A descriptor must be used when the structure of the result table created by a select statement is not known and has to be ascertained.

Host variables are not appropriate for the implementation of such a SAP DB application because neither number nor types of the parameters of the SQL statement are known at precompilation time. It is therefore impossible to associate the parameters with SQL variables in the using part of the execute statement. The necessary relations between the parameters of an SQL statement and the program variables are established instead by means of the SQL Descriptor Area (SQLDA), also referred to simply as descriptor.

The SQLDA is generated by the precompiler when a describe statement occurs in the SAP DB application. It can assume any other name and need not be defined in the declare section. The SQLDA contains all the information which is required for associating a program variable, which must be compatible with the SAP DB column, with a parameter of an SQL statement. Such parameters are designated as input parameters when they transfer values to an SQL statement or to the SAP DB database, and they are designated as output parameters when they transfer values from the SAP DB database to the application.

2.11 SAP DB SQLDA Structure

The values of the parameters are represented by constants which can also be utilized in the program. The meaning of the constants is explained after "::=" (for the values of the constants see Section 6.1.2, "exec sql include <filename>").

sqldaid
Contains the character string "SQLDA" and serves to facilitate the finding of the structure in a dump.

sqlmax
Maximum number of "SQLVAR" entries. For the precompiler-generated SQLDA, a value is automatically assigned to "sqlmax". In all the other cases, the user must set "sqlmax" to a value.

sqln
Number of assigned "SQLVAR" entries (input and output parameters).

sqld
Number of output parameters (the column contents must be placed in a program variable). Columns which can be both input and output parameters are counted here as output parameters.

sqlvar
One "SQLVAR" entry will be created in the SQLDA for each parameter according to the sequence in the SQL statement. The default maximum number for SQLDA entries is 500. The user is allowed to define them in a new variable.

Structure of SQLVAR

SAP DB stores here the following information for each parameter:

colname
For "fetch using descriptor", the name of the column is entered; for all other SQL statements "columnx" is entered, with x as a consecutive number (1 to n) for the columns.

colio
Indicates whether an input or an output parameter is involved.

```
sqlinppar       (0)  ::= input  parameter
sqloutpar       (1)  ::= output parameter
sqlinoutpar     (2)  ::= input/output parameter
```

colmode
Indicates whether null values are allowed. It is a byte whose four low-order bits (lower half-byte) must be interpreted as a bit mask.

```
Bit 0  ::=  mandatory
Bit 1  ::=  optional (null allowed)
Bit 2  ::=  default
Bit 3  ::=  escape character
```

coltype
Supplies the SAP DB type.

```
sqlfixed        (0)  ::= fixed number
sqlfloat        (1)  ::= float number
sqlchar         (2)  ::= character
sqlbyte         (4)  ::= byte
sqloldlongchar  (6)  ::= old long
sqloldlongbyte  (8)  ::= old long byte
sqldate        (10)  ::= date
sqltime        (11)  ::= time
sqlexpr        (12)  ::= float number
sqltimestamp   (13)  ::= timestamp
sqlboolean     (23)  ::= boolean
sqlsmallint    (29)  ::= short integer
sqlinteger     (30)  ::= integer
sqlvarchar     (31)  ::= varchar
```

collength
Number of the total places for numeric columns, otherwise, the number of characters. *Can be set by the user* to the length of the program variables. Such an adaptation is required whenever the number of characters increases by 1 because of the zero byte.

colfrac
Number of places after the decimal point. For coltype = float_number, this array holds -1. *Can be set by the user* to the length of the program variables.

hostindicator
Contains the indicator value. For *input parameters,* it *can be set,* for *output parameters,* it *must be checked,* since in the case of a null value, the program variable will not be overwritten!

hostvartype
Contains the data type designation for the program variable and *must be assigned by the user*. For the describe statement, SAP DB sets this parameter to -1. "collength" and "colfrac" must be changed according to the data type.

```
sqlvint2      (0) ::= integer (2 bytes long)
sqlvint4      (1) ::= integer (4 bytes long)
sqlvreal4     (2) ::= float   (4 bytes long)
sqlvreal8     (3) ::= float   (8 bytes long)
sqlvchar      (7) ::= char-array (terminated with 0 byte)
sqlvstring2  (15) ::= variable string  (2 bytes len, char array)
sqlvuns2     (16) ::= unsigned integer (2 bytes long)
sqlvuns4     (17) ::= unsigned integer (4 bytes long)
sqlvstring1  (20) ::= variable string  (1 byte len, char array)
sqlvint8     (33) ::= integer (8 bytes long)
sqlvstring4  (35) ::= variable string  (4 bytes len, char array)
sqlfile      (40) ::= char-array (terminated with 0 byte)
```

hostcolsize
For ARRAY statements, the size of program variables must be specified here in bytes.

hostvaraddr
Contains the address of the program variable assigned to the parameter. The describe statement initializes it to zero; *the user must assign* the address of the program variable to it. For array statements, this is the address of the first element.

hostindaddr
For array statements, the address of the indicator array *must* be specified here. If indicators are not used, NULL must be specified. For simple SQL statements, the address of a variable *can* be specified here. Then the indicator is written into this variable and "hostindicator" is undefined.

colinfo
Must not be modified, because internal information is stored here that is needed for the conversion of program variables.

SQLDA Declaration

```
typedef struct     {
                      char          sqldaid[8];
                      sqlint4       sqlmax;
                      sqlint2       sqln,
                                    sqld,
                      sqlint4       sqlloop,
                                    sqloffset;
                      sqlint2       sqlkano,
                                    sqlprno,
                                    sqlkamode,
                                    sqlfiller;
                      sqlint4       sqlfiller2;
                      sqlvartype    sqlvar[sqlnmax];
                   }
                   sqldatype;
typedef struct     {
                      sqlnname      colname;
                      sqlint2       colio;
                      char          colmode,
                                    colfill1;
                      sqlint2       coltype;
                      sqlint4       collength;
                      sqlint2       colfrac,
                                    colfiller,
                                    hostvartype,
                                    hostcolsize;
                      sqlint4       hostindicator;
                      void         *hostvaraddr;
                      sqlint4      *hostindaddr;
                      struct SQLCOL col;
                   }
                   sqlvartype;
```

2.12 Using the Descriptor

As a basic rule, dynamic SQL statements with a descriptor must be prepared with the prepare statement. A describe statement issued on the same SQL statement must follow immediately to ensure that during runtime the SQLDA will be provided with the necessary information about the columns to be processed.

The next step depends on the application programming. The SAP DB application must ensure that the SQL statement is provided with appropriate program variables (actually their addresses) as parameters by using the information about the columns stored in the SQLDA. Generally, this will be a distinction of cases by means of which a program variable is to be selected for the SAP DB column and its address is to be entered into the SQLDA.

The SQLDA contains up to 500 SQLVAR entries. Each SQLVAR entry describes the attributes of both a column and the program variables assigned to it. The column description is done by the describe statement, the program variable description is done by the application. "sqln" specifies the number of columns for an SQL statement and is set by the describe statement. The number of SQLVAR entries must not be less than the value of "sqln".

Since parameters can also be included in conditions of SQL statements or serve to provide columns with values, it is necessary to assign values to the corresponding program variables at this point in time.

Finally, the dynamic SQL statement can be executed via the execute statement. The additional specification "using descriptor" must be included in the execute statement only if the SQL statement to be executed contains question marks in place of parameter values.

In the following examples, the four steps

1) execution of the prepare statement,
2) execution of the describe statement,
3) association of the SQL parameters with program variables, and
4) execution (execute) of the dynamic SQL statement

which are necessary for the usage of the descriptor are presented in detail. The first example illustrates the usage of the descriptor only with output parameters and unnamed result tables, the second example also includes input parameters and named result tables.

Example (with output parameters):

A SAP DB application allows interactive queries to a database. For this purpose, the user has to enter a select statement such as follows:

```
select * from reservation

select rno,arrival,departure from reservation where hno = 25

select name from hotel where zip=20005 and price<100.00
```

etc.

This can be formulated within a program in the following manner:

```
        ..... Code .....

        exec sql begin declare section;
        char stmnt [255];
        exec sql end declare section;

        ..... Code .....

/* Processing the select * from reservation */
        printf("SQL statement: \n");
        fgets (stmnt,255,stdin);
        stmnt [strlen(stmnt)-1] = '\0';
        exec sql prepare sel from :stmnt;
        exec sql execute sel;

        ..... Code .....
```

Executing the select statement which was read from the screen generates a result table of an unknown structure. This fact must be taken into consideration when processing the result table.

At execution time of the fetch statement, it must be clear into which program variables the column contents are to be transferred. Therefore the program variables must previously be made known to the fetch statement, which is done by means of the descriptor.

The result table is processed in the following manner:

```
       ..... Code .....

   exec sql prepare fet from
            'fetch using descriptor';
   exec sql describe fet;

/* Providing the SQLDA with user information. */
   set_describe_area ();

/* Processing the result table */
   exec sql execute fet;
   while (sqlca.sqlcode == 0)
   {
/* Processing the data of the current row */
      data_processing();
      exec sql execute fet;
   }

       ..... Code .....
```

Processing the result table:

1) Executing the prepare statement with the parameter "fetch using descriptor". For this purpose, the fetch statement is to be treated like a dynamic SQL statement, i.e., it must be given a <statement name>.

2) Calling the describe statement with the <statement name> of the fetch statement. The descriptor SQLDA is provided with information about the columns to be processed.

3) Using this information, the addresses of appropriate program variables can now be assigned to the SQLDA (set_describe_area ()).

4) Since the program variables are stipulated into which the column contents are to be returned, the fetch statement can now be executed (execute). Subsequently, the corresponding program variables contain the results of the fetch statement.

After processing the first table row (data_processing ()), all the following rows are processed within "fetch-sequence".

The following example illustrates how appropriate program variables can be assigned to the SQLDA:

```
       ..... Code .....

sqlvartype *actvar;
int     lint   [10];
double  lreal  [10];
char    char40 [10][41];
char    char80 [10][81];

       ..... Code .....

set_describe_area ()
{
  int i;
  for (i = 0; i < sqlda.sqln; i++)
  {
   actvar = &sqlda.sqlvar[i];
   switch (( *actvar).coltype)
   {
     case 0: if (( *actvar).colfrac == 0)
             {
                ( *actvar).hostvartype = 1;
                ( *actvar).hostvaraddr = &lint[i];
             }
             else
             {
                ( *actvar).collength = 18;
                ( *actvar).colfrac   = 8;
                ( *actvar).hostvartype = 3;
                ( *actvar).hostvaraddr = &lreal[i];
             };
            break;

     case 2: ( *actvar).hostvartype = 6;
             if (( *actvar).collength < 41)
             {
                 ( *actvar).collength = 40;
                 ( *actvar).hostvaraddr = char40[i];
             }
             else
```

```
                {
                    ( *actvar).collength = 80;
                    ( *actvar).hostvaraddr = char80[i];
                }
                break;
      default: printf ("invalid type! \n");
    }
  }
}

       ..... Code .....
```

The example shows exactly those program variables into which the column contents will be returned. Note that the SAP DB column type must be compatible with the corresponding variable definition. For this reason, variables have been declared in order to receive both character strings of a maximum length of 40 and 80 characters and numbers of different storage and representation. The fact that each type is available in the form of 10 program variables signifies that only SQL statements with up to 10 parameters can be processed. This limitation applies only to the present example. "i" identifies the i-th column in a table and the "SQLVAR" entry in the SQLDA assigned to it.

The information which the describe statement returns to the SQLDA is checked within the function "set_describe_area". The SQLDA is then provided with specifications regarding the program variables. This is illustrated by the following pseudo code section:

```
If the ADABAS column is a fixed type column,
then check, whether it has a fractional part.
  No:  Program variable type is 'integer',
       store the address of 'lint (i)';
       ( The number of digits will not be modified. );
  Yes: The number of digits is set to 18,
       the number of decimal digits is set to 8,
       the type of the program variable
       is 'real',
       store the address of 'lreal (i)'.
If the ADABAS column is a character string,
then ...
```

Executed within a loop, one program variable is assigned to each column of a table which can consist of up to 10 columns. The table contents can subsequently be transferred and processed by rows.

Example (with input/output parameters):

This example illustrates how input and output parameters can be processed by means of the descriptor.

For this purpose, a select statement of the form

```
select * from hotel
         where zip = ? and price <= ? order by price;
```

is executed. Interactive entries made at runtime decide which zip code and which price limit are concerned. The SAP DB application searches the database for hotels of a particular price category in one particular city.

```
       ..... Code .....

set_inp_param ()
{
  int i;
  for (i = 0; i < sqlda.sqln; i++)
  {
   actvar = &sqlda.sqlvar[i];
   if (( *actvar).colio == 0)
   {
      char garbage;
      switch (( *actvar).coltype)
      {
        case 0: if (( *actvar).colfrac == 0)
                {
                   printf("integer:    /bn");
                   scanf ("%ld", &lint[i]);
                   garbage = getchar ();
                }
                else
                {
                   printf("real number: /bn");
                   scanf ("%lf", &real[i]);
                   garbage = getchar ();
                };
                break;
```

```
        case 2: if (( *actvar).collength < 41)
                {
                  printf("text(max. 40 char):\n");
                  fgets (char40[i],40,stdin);
                }
                else
                {
                  printf("text(max. 80 char):\n");
                  fgets (char80[i],80,stdin);
                }
                break;
        default: printf ("invalid type! \n");
      }
    }
  }
}
       ..... Code .....


/* Reading in 'select * from hotel where zip = ?
                    and price = ? order by price' */
     printf("SQL statement: \n");
     fgets (stmnt,255,stdin);
     stmnt [strlen(stmnt)-1] = '\0';
     exec sql prepare sel from :stmnt;
     exec sql describe sel;

 /* The same procedure 'set_describe_area'
    as in the above example is used.                */
      set_describe_area ();
 /* The values for the WHERE clause are entered into
    the program variables of which the addresses have
    been assigned to the SQLVAR entries.            */

    if (sqlda.sqln>sqlda.sqld)
       set_inp_param ();
       exec sql execute sel using descriptor;

 /* Processing the result table.                   */
```

The selection of the program variables used to provide the where clause with values is considerably simplified in the present example. If the search condition has to remain variable, the program variables must be selected and provided with values according to the information about the columns stored in the SQLDA. When calling the execute statement, "descriptor" is specified in the using part instead of a list of host variables. The result table can be processed again with "fetch using descriptor".

Another possibility is to process the result table by means of a cursor:

```
       ..... Code .....

/* Reading in 'select * from hotel where zip = ?
                  and price = ? order by price' */

     printf("SQL statement: \n");
     fgets (stmnt,255,stdin);
     stmnt [strlen(stmnt)-1] = '\0';
     exec sql prepare sel from :stmnt;
     exec sql declare csel cursor for sel;
     exec sql describe sel;

/* The same procedure 'set_describe_area'
   as in the above example is used.                */

      set_describe_area ();

/* The values for the WHERE clause are entered into
   the program variables of which the addresses have
   been assigned to the SQLVAR entries.            */

   if (sqlda.sqln>sqlda.sqld)
      set_inp_param ();

      exec sql open csel using descriptor;

      exec sql prepare fet from
              'fetch csel using descriptor';
      exec sql describe fet;

/* The same procedure 'set_describe_area'
   as in the above example is used.                */

      set_describe_area ();

/* Processing the result table.                    */

       ..... Code .....

      exec sql close csel;

       ..... Code .....
```

Using a cursor for processing a result table has the effect that no execute statement needs to be issued on to the "select", because calling the open statement actually executes the select statement. If question marks are included in a select statement in order to identify parameters, the open statement must be called with "using descriptor" (instead of the host variable list).

2.13 Overview of the Sequences of SQL Statements

General Sequence:

```
exec sql prepare <statement name>  ...;
exec sql describe <statement name>;
providing the SQLDA with information about the program variables;
exec sql execute <statement name> [using descriptor [<variable>]];
```

The specification "using descriptor" is optional.
Default is SQLDA.

When processing the result table by means of a cursor, the following sequence of SQL statements is valid,

for SELECT with parameters:

```
exec sql prepare <select name>  ...;
exec sql declare <cursor name> cursor for <select name>;
exec sql describe <select name>;
providing the SQLDA with information about the program variables;
exec sql open <cursor name> using descriptor;
       ...
exec sql prepare <fetch name>  ...;
exec sql describe <fetch name>;
providing the SQLDA with information about the program variables;
exec sql execute <fetch name>;
       ...
exec sql close <cursor name>;
```

for SELECT without parameters:

```
exec sql prepare <select name>  ...;
exec sql declare <cursor name> cursor for <select name>;
exec sql open <cursor name>;
        ...
exec sql prepare <fetch name>  ...;
exec sql describe <fetch name>;
providing the SQLDA with information about the program variables;
exec sql execute <fetch name>;
        ...
exec sql close <cursor name>;
```

without cursor:

a select statement;

```
exec sql prepare <fetch name> ....;
exec sql describe <fetch name>;
providing the SQLDA with information about the program variables;
exec sql execute <fetch name>;
```

2.14 The Macro Mechanism

The macro mechanism allows a flexible (dynamic) use of table and column names. Without having to modify an application statically, i.e., within the program text, it can work on tables which have the same structure but differ partly in table or column names. Such a name can be equated with a three-digit number between 1 and 128 (syntax: %number) by using the SQL statement "set macro". The number can be inserted into an SQL statement at those positions where tables or columns must be specified. The runtime system of the SAP DB precompiler replaces the number by the name which has previously been specified in the set macro statement. Macro definitions apply to all program units of a SAP DB application (modules, subprograms translated separately, etc.); i.e., a macro value can be defined in one program unit and be used in another one. Host variables can contain table or column names. These names must be declared as strings with a length of up to 30 characters.

The value of a host variable is passed exactly as it was entered. For macros with numbers less than 100, the character string is enclosed in double quotes, "<hostvar>". For macros with numbers greater than 100, the double quotes are omitted.

Example:

```
exec sql begin declare section;
char tabname [31];
exec sql end   declare section;

 ..... Code .....

scanf ("%31s\n",tabname);
exec sql set macro %100 = entry.addresses;
exec sql set macro %101 = :tabname;

 ..... Code .....

exec sql
insert %100  values (:name,:city,:zip,:tel);

exec sql
insert %101  values (:name,:city,:zip,:tel);

 ..... Code .....
```

3 Programming Instructions

3.1 Processing Sets and Single Rows

According to the standard definition of SQL, the statements "select" (declare cursor), "update", and "delete" always apply to sets of table rows. SAP DB offers additional variants of these statements as an extension of the SQL language. These so-called single row statements always address one table row. Therefore single row statements generally make higher concurrency in multi-user mode possible (see the Sections 3.4, "Transactions" and 3.5, "Locks"). The present section describes the main features of set and single row statements. For more details, see the Reference Manual: SAP DB (7.2).

Single Row Selects

SAP DB provides the option of performing a single row select by specifying a key qualification. For a single row select (select ordered statements), the contents of a single table row are transferred to host variables. This is either the first or last table row (select first/select last), a row with a known key value (select direct), or, in relation to a known key position, the preceding or following row (select prev/select next). For single row selects, the column values of the result row must be transferred directly into host variables using the into part of the statement (see the example with Select Direct at the end of this section).

Modifying Single Rows

SAP DB allows the usage of the key qualification also for update and delete statements. As for single row selects, for tables with key columns, the key value is specified by values for all key columns. For tables without key columns, SAP DB provides an internal key column called "syskey (char(8) byte)"; its value can be transferred into the program variables by a select statement and can be used as a key qualification for subsequent single row selects.

Set Requests

The result of a select request without single row qualification is buffered in the database in a so-called result table. The third element of sqlerrd indicates the number of result table rows for some select statements. It has the value -1 if the number of results is unknown at the select point in time.

Result tables can be named or unnamed. In the sqlmodes ANSI and DB2, result tables must be deleted before their names can be used for another result table, whereas in the sqlmodes SAP DB and ORACLE, the result tables are automatically overwritten by the next select statement that creates a result table with the same name.

In sqlmode ANSI, result tables are deleted at the end of a transaction. In sqlmode DB2, result tables are deleted unless they were created "with hold". In the sqlmodes DB2 and SAP DB, result tables are deleted which were generated within a transaction that was concluded by a rollback statement. In sqlmode ORACLE, result tables created via select statements outlast the current transaction and are implicitly deleted at the end of the database session.

Result tables are normally processed sequentially via repeated fetch statements.When doing so, each fetch statement transfers the values of a single result table row into the program variables. As long as there are still result rows available, the return code of the fetch statement has the value 0; after the last row has been processed, the code is +100.

Furthermore, it is possible in sqlmode INTERNAL to fetch the first row of the result table with "fetch first" and the last row of the result table with "fetch last". "Fetch next" and "fetch prev" access the next or previous result table row. If required, it is also possible to position within the result table and to process it repeatedly.

According to the standard definition of SQL, a result table is specified for database set requests in programming languages by the statement "declare <result table name> cursor for <select expression>" and it is read with "open", "fetch", and "close". SAP DB offers this possibility, too.

Examples:

SAP DB SQL Request with an Unnamed Result table:

```
 ..... Code .....

/* Fetching the reservation set to the hotel */

exec sql select arrival, departure
         from reservation
         order by arrival;

exec sql fetch into :arriv, :depart

/* Moving in single steps through the reservation set */

while (sqlca.sqlcode == 0)
 {
 printf("Arrival:%s  Departure:%s\n",arriv,depart);
 exec sql fetch into :arriv, :depart;
 };

 ..... Code .....
```

SAP DB SQL Request with a Named Result table:

```
 ..... Code .....

/* Fetching the reservation set to the hotel */

exec sql select result (arrival, departure)
         from reservation
         order by arrival;

exec sql fetch result into :arriv, :depart;

/* Moving in single steps through the reservation set */

while (sqlca.sqlcode == 0)
 {
 printf("Arrival:%s  Departure:%s\n",arriv,depart);
 exec sql fetch result into :arriv, :depart;
 };

/* Deleting the result table */
exec sql close result;

 ..... Code .....
```

Standard SQL Request:

```
 ..... Code .....

exec sql declare result cursor for
         select arrival, departure
         from reservation
         order by arrival;

 ..... Code .....

/* Fetching the reservation set to the hotel */

exec sql open result;

exec sql fetch result into :arriv, :depart;

/* Moving in single steps through the reservation set */

while (sqlca.sqlcode == 0)
 {
 printf("Arrival:%s  Departure:%s\n",arriv,depart);
 exec sql fetch result into :arriv, :depart;
 };

/* Deleting the result table */

exec sql close result;

 ..... Code .....
```

Example with Select Direct:

```
 ..... Code .....

exec sql
     select direct arrival, departure
     into :arriv, :depart
     from reservation
     key rno = 150;

/* Output after successful search */

if (sqlca.sqlcode == 0)
printf("Arrival:%s  Departure:%s\n",arriv,depart);

 ..... Code .....
```

Example with Update:

```
 ..... Code .....

exec sql update reservation
         set arrival   = :arriv
         key rno = 150;

 ..... Code .....
```

Example with Delete:

```
exec sql delete reservation
         key rno = :reservno;

 ..... Code .....
```

3.2 General Instructions

SAP DB provides a series of system tables which can be used to request data definitions, statistics, and database states. These tables are described in the Reference Manual: SAP DB (7.2). In particular, requests can be issued onto these system tables to determine the columns, especially the key columns, of tables.

The columns to be selected should always be specified explicitly in a select statement. The statement in the format "select *" should be avoided for two reasons:

1. In this way, modifications of the table definition can enter unnoticed into the select statement, which can lead to errors if sqlwarn3 is not checked.

2. In most cases, the values of all columns in a table are not needed in the application. The set of selected columns should be restricted to the necessary size in order to save time and space.

For set requests without an order specification (order by), the sequence in which the table rows are fetched into the program is not determined. Thus the sequence can differ depending on whether a secondary index is used or not for access support. The sequence can also change from one SAP DB version to the other. The logic of the database application must therefore not rely on the present, but arbitrary output sequence.

If the key value of the row to be accessed is known, a single row statement should be used instead of multiple processing. It should be noted, however, that these single row statements are extensions to the SQL standard.

3.3 Users

Every SAP DB application program operates under a particular SAP DB user identification. Coupling the program to a user identification guarantees data security, because the program can execute only actions valid for this user identification. If a user identification is not specified explicitly (create user ... not exclusive), SAP DB gives every user identification exclusively to one user. The user identification is not bound to a person, but is an authorization profile.

Furthermore, users can be united to form a group. Then every user has the privileges that have been determined for the group.

Before a user can work with a database, he has to open a database session. This means that he has to connect to a database. This is usually done via the connect statement, but can also be done by means of predefined user specifications (XUSER; see the platform-specific User manual).

If a connect statement is specified in the program, the user identification and password can either be included in the text of the program or specified via host variables. The lock mode for the database session has to be specified in addition (for more information, see Section 3.5, "Locks").

"Commit work release" or "rollback work release" must be executed as the dynamically last SQL statement of an application program. "Commit work release" concludes the last transaction, "rollback work release" resets the effects of the last transaction. The additional specification release makes the SAP DB task which was serving the program available again to other database users.

Example:

```
    ..... Code .....

exec sql begin declare section;

char password [19];

exec sql end declare section;

    ..... Code .....

scanf ("%18s",password);
exec sql connect dbuser identified by :password;
exec sql select cno, lastname from customer ....;

   ..... Code .....

exec sql commit work release;
```

3.4 Transactions

An important aspect of SAP DB application programming is the programming of transactions. The SQL statements insert, update, and delete do not modify the addressed table contents irreversibly. All SQL statements are combined to form so-called transactions. Only at the end of a transaction does SAP DB decide whether all the effects of the modifications made during the transaction are committed in the database or cancelled. This guarantees that, e.g., after a system failure, the effects of all SQL statements performed during the last transaction are established completely or not at all in the database. Consequently, it can be ensured in the application program that the database is always in a logically consistent state. In contrast to other relational database systems, data definition SQL statements in SAP DB are also subject to the transaction concept. This means that also data definition SQL statements, any combinations of them, as well as SQL statements such as insert, update, and delete can be rolled back.

The transaction structure also influences the lock behavior of the database and thus possibly the concurrency of user operations.

Finally, there are database states in which the end of all user transactions has to be awaited (e.g., database shutdown; for details see the SAP DB manual Database Manager).

Statements for Transaction Control

All SQL statements placed between the statements "commit work" or "rollback work" form a transaction. "Commit work" records the effects of all the statements performed during the last transaction in the database and opens a new transaction. This statement is normally used to bracket SQL statements to form a transaction.

"Rollback work" resets the effects of all statements of the last transaction and opens a new transaction. This statement can be useful in error cases and exceptional situations.

Connecting to SAP DB implicitly opens the first transaction. The dynamic order of "commit work" or "rollback work" statements at execution time is decisive for the bracketing of SQL statements to form a transaction. Their static order within the program text is not significant.

Subtransactions

In sqlmode INTERNAL, there is the option of defining subtransactions e.g., to atomize the effects of subprograms. The SQL statement "subtrans begin" opens a subtransaction and "subtrans end" closes the subtransaction, removing the position defined by "subtrans begin" from memory. If, instead of "subtrans end", the SQL statement "subtrans rollback" is used, all modifications to the database made within the subtransaction are cancelled.

Subtransactions can be nested. The SQL statements "subtrans end" and "subtrans rollback" always refer to the last open subtransaction.

"Subtrans rollback" or rollback cancel any subtransactions within the last (sub)transaction, even if they were closed with "subtrans end".

SQL statements that conclude subtransactions do not influence set locks, i.e., any existing locks are kept.

In sqlmode ORACLE, savepoints can be used to enable subtransactions. Within a transaction, the SQL statement savepoint can be used to define and name positions in this transaction. All modifications made since the savepoint with the specified name can be cancelled by means of the SQL statement "rollback to". Afterwards, any intermediate positions are no longer known.

Restartable Application Programming

After a system failure and subsequent restart of the database, the effects are restored up to the end of the last completed transaction. An application program, when started again, often cannot identify any more the position at which processing was interrupted. Batch programs, however, should be able to recognize after a breakdown and subsequent restart which transaction was executed last, and to continue processing from this point. This feature is called restartability of application programs. For this purpose, SAP DB offers transaction consistency as a help. A restartable application program must store the internal status of the last transaction (variable values, etc.) in one or several self-defined and self-managed status tables. Subsequent to a breakdown, the status table is also put into the status of the last completed transaction, so that after the program has requested its contents, it can find the re-entry point to continue processing.

The status table is generated before the program starts. At the beginning, a restartable program writes the initiating values of the program status variables into host variables. A subsequent select statement checks whether the last program run was aborted. If this is the case, the program status variables contain the status of the last transaction which was successfully concluded after the last select. If the last program run was regularly terminated, the status table no longer contains information for this program and the select statement delivers an sqlcode not equal to zero. Subsequently, the initiating values will be entered into the status table. During the execution of the program, the program status values must be updated for

each transaction by means of the update statement. At the end of the program, the entry will be deleted from the status table.

Example of Restartability:

```
/*  Defining the table status
    create table status
    (progid char (20) key, transid fixed (2))       */

 ..... Code .....

next_transaction ()
{
   trans_id++
   exec sql update status set transid = :trans_id
                             key progid = :prog_id;
};

insert_transaction ()
{
   exec sql insert into customer  ( ... )
                          values    ( ... );
   next_transaction ();
   exec sql commit work;
};

sel_next_transaction ()
{ ...
};
     .
     .
     .

delete_transaction ()
{ ...
};

 ..... Code .....

strcpy(prog_id, "myprog");

/* Setting the program status variable             */
/* for a normal program start.                     */
trans_id  = 0;
```

```
/* Checking whether the program was aborted.        */
/* If so, the program status variable is            */
/* overwritten with the last recorded status.       */

exec sql whenever not found continue;
exec sql select direct transid
         into  :trans_id
         from  status
         key progid = :prog_id;

if (sqlca.sqlcode != 0)
         exec sql insert into status (progid, transid)
                                values (:prog_id, 0);

 ..... Code .....

/* Within the separate transactions                 */
/* next_transaction is called.                      */

 ..... Code .....

if (trans_id < 1) insert_transaction ();
if (trans_id < 2) sel_next_transaction ();
if (trans_id < 3) update_transaction ();
if (trans_id < 4) sel_fetch_transaction ();
if (trans_id < 5) delete_transaction ();

/* Deleting the entry in the transaction table      */

exec sql delete status
               key progid = :prog_id;

 ..... Code .....
```

3.5 Locks

If a database user wants to update table rows, the rows concerned must be locked for other users for consistency reasons. Only in exceptional cases, the concerned rows need not be locked for reading. SAP DB can set read or write locks. Read-locked table rows (select) can be read but not altered by other users. Write-locked table rows (insert, update, delete), on the other hand, can be neither read nor updated by other users. Locks are normally released at the end of a transaction (commit work, rollback work).

Explicit and Implicit Locks

Write and read locks can be set for tables as well as for table rows. When locks are to be set implicitly, SAP DB places single row locks in most cases. When the number of single row locks held by one user on one table exceeds a given value, SAP DB tries to change these locks into a lock on the entire table. The threshold value from which a table lock is requested is indirectly determined by means of an installation parameter.

The Three SAP DB Lock Modes

SHARE defines a read lock for the specified objects.

EXCLUSIVE defines a write lock for the specified objects.

OPTIMISTIC defines an optimistic lock for the row.

Setting these locks enables controlled and protected execution of evaluations or modifications of data in an application program.

For example, if a data object is only read-locked by a user, it can be read but not modified by any other user.

On the other hand, if a data object is write-locked by a user, no other user can read or modify this data object.

If an optimistic lock is set for a row, any other user can read and even modify this row. But when the owner of the optimistic lock attempts to modify the row thus locked, a check is made. If the row was modified by another user between the setting of the lock and the attempt to modify the row, the modification of the lock holder is rejected with an error code. To change the row, the lock holder has to read it again and to set a lock, if necessary. If no modifications were made by other users in the meantime, the modifications of the optimistic lock holder are performed. Thus it can be ensured that between reading and modifying the row, no modification was made that could be lost by the new modification. If a read- or write-lock is set for a row that was optimistically locked by the same user, then this row is not locked optimistically any more. Then no check is made when the user tries to change the row.

By default, read locks are set implicitly by the system. A non-default lock mode can be determined via the isolation level when opening a database session.

Insert, update, or delete statements write-lock the data objects concerned in any isolation level. Therefore the following description of the different isolation levels only concerns the behavior for select statements.

Isolation Level

A select statement sets a read lock for a row of a table. With the next read operation in this table, this lock is released and replaced by the read lock for the newly read row. This means, only one read lock is held for a table row per table.

This is very helpful in interactive mode, since an interactive user does not need to take care of the setting of locks. In this case, the isolation level 1 is concerned which in sqlmode INTERNAL can also be denoted as isolation level 10. Except for the sqlmode ANSI, this isolation level is also used when no isolation level has been specified in the connect statement.

```
exec sql connect ... isolation level 1;    /* introduction */
 .
 .
exec sql select ...                        /* read loc */
 .
 .
exec sql update ...                        /* write lock */
 .
exec sql commit work;                      /* release of */
 .                                         /* both locks */
```

The statement "commit work" terminates the transaction and releases all the locks implicitly requested.

In OLTP application programs it is better to request locks explicitly. It is therefore recommended to use the isolation level 0 and optimistic locking for OLTP application programs. For this purpose, the isolation level must be set to 0 in the connect statement. The lock statement can subsequently be used to lock any rows (via the key) or tables (via the name). Optimistic locks can only be used for rows.

The isolation level 0 differs from all the other isolation levels by not implicitly setting the read locks when reading; these must be requested explicitly.

```
exec sql connect ... isolation level 0;   /* introduction */
 .
exec sql lock ...                         /* read lock or */
 .
exec sql select ...
 .
exec sql update ...                       /* write lock */
 .                                        /* if not yet */
 .                                        /* locked by */
 .                                        /* preceding lock */
 .                                        /* statement */
 .
exec sql commit work;                     /* release of */
 .                                        /* both locks */
```

If the isolation level is set to 15 in a connect statement, one read lock is set to the read table row per table for a select statement. With the next read operation in the same table, this lock is released and replaced by the lock on the newly read row. This corresponds to isolation level 1. Moreover, the complete table is read-locked and only released at the end of the statement when a statement without a key specification is processed. If the result table is not internally generated for a select statement, the lock is only released for the implicit or explicit close.

Isolation level 15 is only valid for sqlmode INTERNAL.

For isolation level 2, which in sqlmode INTERNAL can also be denoted as isolation level 20, processing a statement without a key specification read-locks the complete table. This lock is only released at the end of the statement. When table rows are read, they are read-locked row by row so that various table rows are read-locked during a select. The locks set to individual table rows remain valid up to the end of the transaction. If the result table is not internally generated for a select statement, the lock is only released for the implicit or explicit close.

Isolation level 3, which in sqlmode INTERNAL can also be denoted as isolation level 30, has the effect that the complete table is read-locked for a statement without a key specification. This lock is kept up to the end of the transaction. This means that repeating the same select statement within the same transaction always yields the same result. This isolation level is used in sqlmode ANSI, if no isolation level has been specified in the connect.

Changing the Isolation Level

In sqlmode INTERNAL, the isolation level chosen for the connect statement can be changed for single select statements. This can be done by means of the specification "with lock isolation level x", where x stands for one of the isolation levels described above. The specified level x can be used for the select statement concerned, regardless of the isolation level selected in the connect statement. The following SQL statements use the isolation level specified for the connect again.

Multi-User Mode

Although selecting the isolation level and using an appropriate transaction size reduce the critical fields where collisions might occur, it can happen that a data object is requested by several users. A user who attempts to lock an object already locked is put in wait state by default, until the object is available again or the maximum waiting time has been exceeded (wait option). For the description of the maximum waiting time, see Section 3.6, "Timeouts".

If the option nowait is specified in the lock statement or in the option "with lock" of a select statement, the release of the locks is not waited for, but a message is returned. If no collision occurred, the desired lock is set.

```
exec sql connect ... isolation level 0;      /* introduction */
 .
 .
exec sql lock ... NOWAIT ...                 /* attempt: */
 .                                           /* read lock */
 .                                           /* write lock */
if (sqlca.sqlcode = 500)
{
    ...
    /* Data object not available */
    /* Please select another function */
}
```

Thus, waiting times are avoided if a user can execute other work. The option nowait can, of course, be used in batch mode if the sequence of work steps is of no importance. The option, however, requires a somewhat more complicated program logic.

3.5.1 Instructions for the Locking

For long transactions, a program can hold many locks over a long period of time, which will probably hinder other users. For this reason, transactions should be as short as possible. On the other hand, locks must be requested after each "commit work" or "rollback work", which can result in increased database activity for very short transactions. The most effective transaction length should therefore be determined taking into consideration recovery duration, lock duration, and locking overhead.

If modifications to one or more tables are to be applied within one transaction, the following transaction programming is recommended:

1) Reading phase with user dialog. All values which are to be entered into the database are collected in host variables. For this purpose, read accesses to the database and interactive entries are necessary. Reading is done without read locks in order not to hinder other users and to avoid deadlocks. No write accesses to the database are used. More details are included in the Reference Manual: SAP DB (7.2) for "select .. with lock optimistic".

2) Locking phase. By means of the lock statement, write locks are requested for all rows or tables to be modified, and also read locks are requested for all rows or tables to be kept statically. This ensures that all rows or tables are accessible before they are write accessed. Rows read without read lock in phase 1 must be read again in order to ensure that these rows have not been modified in the meantime. If modifications have been made, phase 1 has to be repeated after releasing the lock. When optimistic locks (select ... with lock optimistic) are used, it is not necessary to read the rows again, because modifications to the read object made by other users result in a corresponding message for the update or delete statement as long as the corresponding row was not locked explicitly.

3) Writing phase. In this phase the contents of the host variables are entered into the rows or tables previously locked. User dialog is not allowed in the phases 2 and 3.

Deviations from this "textbook" schema, e.g., user dialog with set locks, can lead to decreased concurrency or deadlocks.

A "commit work" is recommendable after each create or drop statement, because these statements set write locks on the SAP DB system information. These locks are released by "commit work".

If it is not essential to have a consistent database state during a long query (e.g., during statistical queries), isolation level 0 can be specified as the lock set mode. In this case, write-locked tables can also be read without read locks, i.e., "readers" will not be blocked by "writers".

When table contents are modified without previously requesting a write lock, this lock will be set implicitly.

3.6 Timeouts

Timeouts enable the database to handle exceptional situations after a certain period of time. The intervals at which timeouts occur can be configured via the SAP DB CONTROL component and depend on the type of application. They are stipulated in such a way that timeouts will not normally occur. Nevertheless, every program must check the timeout sqlcodes and then handle them either via the whenever sqlexception condition or via an explicit test following every SQL statement.

The following timeouts exist:

request_timeout (sqlcode +500)

Cause:
The program has waited for a lock for a longer period of time than is specified in the installation parameter "request_timeout". The lock is not available. Control is returned to the program.

Action:
In principle, the lock can be requested again or the statement which implicitly requested the lock can be repeated. If the program holds its own locks, these should be returned via "rollback work" after a "request_timeout". Then the lock which could not be obtained can be requested again as the first lock. The risk of deadlocks is reduced by this means. If in multiuser mode the "request_timeout" occurs frequently in spite of checking the installation parameter, the transaction should be revised to provide more efficient locking (more single row locks, shorter transactions, another SAP DB lock mode). For long transactions which successively process several tables, all the required tables or table rows should be locked explicitly at the beginning of the transaction. This method reduces the risk of deadlocks in the case of implicit locks (incremental lock request), but other users can be blocked for a longer time.

lock_timeout (sqlcode +600)

Cause:
The program holds a lock for a longer period of time than is specified in the installation parameter "lock_timeout". SAP DB performs a rollback work on the current transaction. (This timeout is only activated when other users request the locked objects.) The message does not appear until the first statement is issued after the period of inactivity.

Action:
The last transaction must be repeated. The cause for the timeout should be checked (e.g., no user interaction when holding locks; the same applies to long computations).

session_timeout (sqlcode +700)

Cause:
The program has not issued an SQL statement for a period longer than is specified in the installation parameter "session timeout". SAP DB performs a "rollback work release" on the current transaction, thus disconnecting the program from the database.

Action:
The program must re-issue the connect statement and repeat the last transaction. If result tables are used, these must be built up again via the select statement.

Waiting for user input can involve a lengthy period of time. User input should therefore be entered immediately after a "commit work". Thus the program holds no locks; therefore it does not hinder other users and there is no risk of a "lock_timeout".
If a "session_timeout" occurs subsequent to a "commit work", it has no effect on the current transaction.

3.7 Optimizing SQL Statements

This section describes the way in which the SAP DB optimizer processes SQL statements. The instructions given in this section and the return messages of the EXPLAIN statement can help to optimize a SAP DB application. It is an important task to find out the optimal SQL statements in an application. This has also decisive influence on the performance of the complete application.

As a tool for optimization, SAP DB provides the EXPLAIN statement. This can be used to easily test the SQL statements within the interactive component QUERY.

The precompilers, moreover, provide the option trace long which, among other things, records the runtime of the SQL statements and provides the possibility of performing an application profile at runtime.

The examples of this section refer to the following declarations:

```
exec sql create table example
         (firstkey       fixed (3)  key,
          secondkey      fixed (4)  key,
          normalcolumn   fixed (5) ,
          invcolumn1     char  (15),
          invcolumn2     char  (9) ,
          multinvcolumn1 char  (22) ,
          multinvcolumn2 char  (5) ,
          multinvcolumn3 fixed (8)      );

exec sql create index example.invcolumn1;

exec sql create index example.invcolumn2;

exec sql create index ind on example (multinvcolumn1,
                              multinvcolumn2,
                              multinvcolumn3 );

exec sql select * from example ...

 ...   where normalcolumn = 3;
 ...   where invcolumn1 like 'M*er';
```

The way in which an SQL statement can be processed in the shortest possible time with a minimum of storage space depends on:

- the manner of physically storing table rows,
- the size of the tables (number of rows and B* tree pages),
- the definition of the table (of the key columns),
- the indexes,
- the kind of SQL statement (select, insert, update, delete),
- the elements of a select statement (<order clause>, <update clause>, <distinct spec>, for reuse),
- the modifications specified in the update statement,
- the search conditions.

These factors influence the processing of an SQL statement especially when large data sets have to be searched through. This will generally happen with the following SQL statements:

- select; namely as <query statement>, <single select statement>, or <select ordered statement> (not as <select direct statement>),
- insert ... select,
- update ... where <search condition>,
- delete ... where <search condition>.

To optimally process these SQL statements, strategies have to be applied which guarantee that the following objects are obtained:

- Only the rows that are actually needed are searched.
- Temporary result tables are kept as small as possible.
- The search is postponed until the fetch time without generating a result table. (The advantages are: no storage usage for results, fast access to the first results, fast comparison of the obtained results with the desired results).

Not all SQL statements allow the search to be postponed until the fetch time. The following overview contains the corresponding SQL statements:

- select on several tables (join)
- select distinct
- select with for reuse specification
- select with order by specification (in most cases).

3.7.1 Influence of the Search Condition

The following conditions can be utilized for the selection of a search strategy:

<column spec> = | < | <= | > | >= <value expr>
<column spec> between <value expr> and <value expr>
<column spec> like <value expr>
<column spec> in <value list>

The in condition can only be utilized when the <column spec> refers to the only key column or to a column for which a single-column index is available (see Section 3.7.2.1, "Definition of Terms").

Conditions like

not (<value expression> <comp op> <value expression>)

are transformed, if possible, into an expression

<value expression> negated<comp op> <value expression> ,

and then processed in this format. If conditions cannot be transformed into one of the above mentioned formats, they cannot be used for the selection of a search strategy.

The order of conditions combined by equivalent Boolean operators has no influence on the selection of a search strategy.

In principle, every search can be performed sequentially through the entire table. This has to be done whenever

- no (<search condition>) is specified,
- no condition is specified either for key columns or for indexed columns.

If the possible non-sequential search strategies are more costly than the sequential search, the table is processed sequentially.

When conditions for key columns exist, the search is limited to the corresponding part of the table.

3.7.2 Search Strategies

3.7.2.1 Definition of Terms

A single-column index is a named or unnamed index containing one column only. A single-column indexed column is a column on which a single-column index was created.

A multiple-column index is a named index containing various columns. A multiple-column indexed column is a column which is an element of one multiple-column index at least.

An index list is a list of keys belonging to an index. The following is true for every index column: all table rows with keys which are specified in the list contain the same value in this column.

3.7.2.2 Conditions on Key Columns

If search conditions are specified for key columns, then two cases can be distinguished:

- If an EQUAL or IN condition refers to the only key column, then the corresponding row(s) is (are) directly accessed.

- For all the other conditions specified on key columns, an upper and a lower limit are determined for the valid range of keys. All strategies, even those implemented with indexes, utilize this knowledge of the permitted range of keys.

3.7.2.3 Conditions on Single-Column Indexed Columns

If conditions are applied to single-column indexed columns, four cases have to be distinguished:

- EQUAL/IN condition:
  If an EQUAL condition is specified for a single-column indexed column, only the rows with keys contained in the pertinent index list are accessed.
  If an IN condition is specified, the rows with keys contained in the index lists are accessed.

- Several EQUAL conditions combined by AND:
  If several EQUAL conditions are specified for different single-column indexed columns, an intersection of the corresponding index lists is formed. Only the rows with keys which are contained in all the indicated index lists are accessed.

- Conditions of ranges of values:
  The specification of one condition (<, <=, >, >=) on one of the two limits of the value range (lower or upper limit) will suffice for the selection of a search strategy.
  If both limits are to be specified, it makes no difference to the search strategy whether this specification is made by one BETWEEN condition or by two conditions (<= or >=) combined by AND defined for the same column.

  The rows accessed are always the rows with keys that are contained in the index lists and that are designated by the range of values.

- Conditions of ranges of values combined by AND which are defined on ten columns:
  If there is at least one value range restriction for one single-column indexed column and if there is one EQUAL condition or at least one value range restriction for each of up to 9 other single-column indexed columns, then the following action is taken:
  One logical index list, which need not necessarily be physically available, is created for each of the available columns. An intersection is formed from all these index lists. Only the rows with keys contained in all index lists are accessed.

Examples:

```
... where firstkey >= 123
```

Starting with the row with the key "123", the table is sequentially searched.

```
... where invcolumn1 = 'Miller' and firstkey >= 123
```

Starting with the key "123", the complete index list with the value 'Miller' is processed up to the end of the list.

```
... where invcolumn1 = 'Miller' and invcolumnD2 < 'C'
```

A logical index list is created which contains all index lists of INVCOLUMN2 and which begins with a value less than 'C' (' ', 'A', 'B').

The intersection of the logical index list and of the list containing the value 'Miller' is formed and completely processed.

```
... where invcolumn1 = 'Miller' and invcolumn2 = 'Don'
```

The intersection of both index lists is formed and completely processed.

```
... where invcolumn1 in ('Miller', 'Smith', 'Hawk')
```

The intersection of both index lists is formed and completely processed.

```
... where invcolumn2 > 8965 and firstkey = 34 and
        secondkey between 12 AND 18
```

All index lists of invcolumn2 with values greater than 8965 are processed. The lists are, however, only considered between the key boundaries "34, 12" and "34, 18".

For the EQUAL/IN condition and the conditions of ranges of values, there are some enquiries for which accessing the rows is not necessary, because all the required values are contained in the index list(s).

3.7.2.4 Strategies with Conditions on Multiple-Column Indexed Columns

If conditions are specified for multiple-column indexed columns, two cases can be distinguished.

- Equality conditions:
  Several equality conditions are specified for several multiple-column indexed columns which form a complete named index. The rows with keys that are contained in the corresponding index list of the named index are accessed.

- Equality conditions or conditions of ranges of values:
  Several equality conditions or conditions of ranges of values combined by AND are specified for several multiple-column indexed columns.
  Let a named index be formed by several columns (number n). If there is an equality or range condition for the first k (k<=n) columns of the index (a condition containing < or > can only be specified for the k-th column), only the rows with keys contained in the index lists of the specified range of values are accessed.

Examples:

```
... where multinvcolumn1 = 'Düsseldorf'  and
          multinvcolumn2 = '40223'       and
          multinvcolumn3 = 10000
```

The complete index list of the named index "ind" with the values 'Düsseldorf', '40223', and 10000 is processed.

```
... where multinvcolumn1 = 'Düsseldorf'  and
          multinvcolumn2 between '40221' and '40238'
```

The index lists including the values 'Düsseldorf', '40221', (binary zeros) and 'Düsseldorf', '40238', (binary ones) are processed.

For both strategies, there are enquiries for which accessing the rows is not necessary, because all the required values are contained in the index list(s).

3.7.2.5 Cost-Finding

The costs are calculated for every possible strategy.
This is necessary because

- there can be several strategies, of which the best must be determined
- there are cases in which the search performed by means of an index is more costly than the sequential search.

The costs refer to the estimated number of I/O processes which will have to be performed as a consequence of the selected strategy.
These costs are also output as a result of the EXPLAIN statement (see Section 3.7.4, "The EXPLAIN Statement").

3.7.2.6 Conditions Combined by OR

The search for the best strategy must not yet be concluded when, for example, the following conditions apply at the same time:

- The sequential search is the best strategy.
- There are search conditions combined by OR.

The conditions combined by OR which have not yet been considered are analyzed in order to find strategies better than the sequential search.

The procedure is as follows:

- The search condition is transformed into the disjunctive normal form.

  Example:

```
 b1 and b2 and (b3 or b4 and b5)
               =
(b1 and b2 and  b3)
               or
(b1 and b2 and        b4 and b5)
```

- Analysis of the new expression:
  Every subexpression is analyzed separately. If the result of this analysis is the sequential search through the complete table, then this is the only applicable strategy. If the analysis gives a better search strategy for every subexpression, there are generally as many strategies as subexpressions that have been analyzed.

- Cost-finding:
  The costs of the different strategies are added up. If the total is less than the cost of the sequential search, the different strategies are applied.

3.7.2.7 Postponing the Search to the FETCH Point in Time

One object of optimization is to save storage space, i.e., to avoid the generation of result tables. Apart from the cases in which the element FOR REUSE or an existing join enforces the creation of result tables, building a result table is avoided as far as possible.

With an ORDER BY specification, it is only possible to do without a result table under the following conditions:

- There is no strategy better than the sequential search,

  and

- neither <distinct spec> nor FOR REUSE is specified,

  and

- there is a single-column indexed column according to which a sort is performed,

  or

- there are several columns according to which a sort has to be performed which, in a specified sequence and order (ascending or descending), form a named index.

The EXPLAIN output indicates whether the result table is generated (RESULT IS COPIED) or not (RESULT IS NOT COPIED).

3.7.3 UPDATE

The type of statement has an effect especially on the update. If the new value of a column is calculated in an arithmetic expression, an index of this column cannot always be used for the search. Thus the SQL statement

```
update <table name>
set    columnx = columnx + 3
where  columnx in (100, 103, 106, 109, 112)
```

could lead to incorrect results if the index lists with the values 100, 103, 106, 109, 112 were processed little by litte. This must also be taken into consideration when FOR UPDATE is used in the SELECT statement.

3.7.4 The EXPLAIN Statement

Using this statement, the user can inform himself of the strategy applied for the execution of the specified select statement.

The following overview shows the strategies which are distinguished:

- TABLE SCAN
  Sequential search through the complete table.

- SINGLE INDEX COLUMN USED (INDEX SCAN)
  Sequential search through the complete specified single-column index.

- MULTIPLE INDEX COLUMN USED (INDEX SCAN)
  Sequential search through the complete named multiple-column index.

- RANGE CONDITION FOR KEY COLUMN
  Sequential search through a part of the table.

- RANGE CONDITION FOR KEY COLUMN (SUBQ)
  Sequential search through a part of the table using each matching row of the subquery.

- EQUAL CONDITION FOR KEY COLUMN
  The table has only one key column to which an EQUAL condition is applied. The corresponding table rows are directly accessed.

- EQUAL CONDITION FOR KEY COLUMN (SUBQ)
  The table has only one key column to which an EQUAL condition is applied. The corresponding table rows are directly accessed using each matching row of the subquery.

- IN CONDITION FOR KEY COLUMN
  The table has only one key column to which an IN condition is applied. The corresponding table rows are directly accessed.

- EQUAL CONDITION FOR INDEXED COLUMN
  An EQUAL condition is applied to a single-column indexed column. The related index list is used to directly access the corresponding table rows.

- EQUAL CONDITION FOR INDEXED COLUMN (SUBQ)
  An EQUAL condition is applied to a single-column indexed column. The related index list is used with each matching row of the subquery to directly access the corresponding table rows.

- IN CONDITION FOR INDEXED COLUMN
  An IN condition is applied to a single-column indexed column. The related index lists are used to directly access the corresponding table rows.

- RANGE CONDITION FOR INDEXED COLUMN
  A range condition is applied to a single-column indexed column. The index lists within the specified range are used to directly access the corresponding table rows.

- 'ORDER BY' VIA INDEXED COLUMN
  No strategy better than the sequential search has been found. The index of the column specified after 'ORDER BY' is utilized.

- EQUAL CONDITION FOR MULTIPLE INDEX
  An equality condition is applied to every column of the named multiple-column index. The related index list is used to directly access the corresponding table rows.

- RANGE CONDITION FOR MULTIPLE INDEX
  There are equality or range conditions which are applied to the first k columns of a named multiple-column index. The index lists within the specified range are used to directly access the corresponding table rows.

- RANGE CONDITION FOR MULTIPLE INDEX (SUBQ)
  There are equality or range conditions which are applied to the first k columns of a named multiple-column index. The index lists within the specified range are used with each matching row of the subquery to directly access the corresponding table rows.

- IN CONDITION FOR MULTIPLE INDEX
  An IN condition was specified for the first column of a named multiple-column index; or equality conditions were specified for the first k-1 columns of such an index and an IN condition was specified for the kth column of that index. The related index lists are used to directly access the corresponding table rows.

- 'ORDER BY' VIA MULTIPLE INDEX
  No strategy better than the sequential search has been found. The multiple-column index is used; the columns of this index were specified after 'ORDER BY' in correct sequence and order (ASC/DESC).

- INTERSECTION OF COLUMN INDEXES
  There are several equality or range conditions which are applied to several single-column indexed columns. The intersection of the respective index lists is formed. The intersection index list is used to directly access the corresponding table rows.

- DIFFERENT STRATEGIES FOR OR TERMS
  The analysis of the conditions combined by AND has not produced a strategy better than the sequential search. Conditions combined by OR have been transformed and analyzed, giving the result that various strategies have been found for the different elements of the search condition. Each strategy is displayed according to the above mentioned rules.

- CATALOG SCAN
  A sequential search is performed on the catalog.

- CATALOG SCAN USING USER EQUAL CONDITION
  A sequential search is performed through the catalog entries describing the objects of the identified user.

- CATALOG KEY ACCESS
  The qualification contains equal conditions. This allows the query to be processed by accessing the key columns in the catalog (e.g., equal conditions for OWNER and TABLENAME in queries performed on DOMAIN.TABLES).

- NO STRATEGY NOW (ONLY AT EXECUTION TIME)
  The corresponding column values in a correlated subquery are only known at the subquery's execution time. The strategy for the most effective access to the corresponding table of the subquery will not be determined until these values are available.

Search conditions that use the IN operator to compare a column with a subquery are converted in the following way before the strategic search:

```
columnx     IN (SELECT ...) ==> columnx  = ANY (SELECT
columnx NOT IN (SELECT ...) ==> columnx <> ALL (SELECT ...)
```

For an EXPLAIN issued for a request containing such a search condition, an EQUAL or RANGE condition is displayed, although the query does not contain such a condition.

To determine the costs of strategies which compare the value of a column with the matching rows of a subquery, it is necessary to know the number of matching rows. Therefore, the costs can only be determined after processing the subquery.

For an EXPLAIN issued for a SELECT with subqueries, the possible strategies are determined, but the subqueries are not executed; therefore 'NO STRATEGY NOW (ONLY AT EXECUTION TIME)' is usually entered in the STRATEGY field for an outer query.

Certain selects are so complicated that they are divided into several internal select steps. This is indicated in the EXPLAIN output by several output lines and by the indication 'INTERNAL TEMPORARY RESULT' displayed as the table name. Only a sequential search is possible on such internal temporary results.

The EXPLAIN statement is described in the Reference Manual: SAP DB (7.2).

3.7.5 Joins

A join can be performed across 16 tables at the most. Thereby the tables are joined step by step, i.e., a join is formed from two tables. Each additional table is then combined with the join result to form another join result.

The procedure according to which a result table is generated can be described in pseudo code such as follows:

```
PROCEDURE:
- search through the first table and store the result
  in a temporary result table ordered according to
  the JOIN columns
- loop: As long as another table has to be searched
        through,
      - combine the existing temporary result table
        with the new table
      - store the result in a new temporary result
        table ordered according to the JOIN columns
      - delete the old temporary result table
  end of loop
- the last temporary result table is the result
  table desired by the user.
```

Time or space can only be saved when the temporary result tables are as small as possible and when the rows of the tables to be combined to form a new join can be accessed directly.

The SAP DB optimizer therefore tries to put small tables with restrictive conditions at the beginning of the series of tables to be processed in order to obtain small temporary result tables.

The sequence of the tables' specification in the <from clause> of the select statement has no influence on the sequence of processing. The sequence of processing is determined by the SAP DB optimizer.

3.7.5.1 EXPLAIN Statements for Joins

The EXPLAIN statement can also be applied to joins. It shows:

- the sequence in which the tables will be processed when the select statement is performed,
- whether the rows of a new table can be accessed directly or via an index, starting from the join column values of the old temporary result table,
- the strategy according to which the corresponding new table is searched through when the rows of this table cannot be accessed directly or via an index.

The following strategies can be applied to access rows of the new table starting from the join column values of the old temporary result table:

- JOIN VIA KEY COLUMN
  The join column is the only key column. Table rows of the new table are accessed directly.
- JOIN VIA KEY RANGE
  The join column whose name is displayed is the first key column. Within the range of keys, table rows of the new table are accessed sequentially.
- JOIN VIA INDEXED COLUMN
  The join column is a single-column indexed column. Access is made via the index of the column whose name is displayed.
- JOIN VIA MULTIPLE KEY COLUMNS
  The specified join columns can be combined to form the key of the new table. This key consists of several columns. The table rows of the new table are accessed directly.

- JOIN VIA RANGE OF MULTIPLE KEY COLUMNS
  The specified join columns can be combined to form the initial part of the key of the new table. This key consists of several columns. Within the range of keys, the table rows of the new table are accessed sequentially.

- JOIN VIA MULTIPLE INDEXED COLUMNS
  The specified join columns can be combined to form a complete multiple-column index. Access is made via this index.

- JOIN VIA RANGE OF MULTIPLE INDEXED COL.
  The specified join columns can be combined to form the initial part of the index consisting of several columns. Within the index range, the table rows are accessed.

Comment on the multiple-column key strategies or index strategies:

If the column lengths of the two columns to be compared within a join step are not equal, these strategies cannot be utilized. To bypass this restriction, it is recommended to use the same domain for the definition of the colums to be joined.

Examples:

```
exec sql create table one ( keyf fixed(6) key,
                f    fixed(3),... );        /* 1000 rows*/

exec sql create index one.f;

exec sql create table ten1 ( keyft1 fixed(6) key,
                ft1    fixed (3),... );    /*10000 rows*/

exec sql create table ten2 ( keyft2 fixed(6) key,... );
                                           /*10000 rows*/

exec sql EXPLAIN select one.key, ten1.keyft1, ten2.keyft2
         from one, ten1, ten2
         where ten1.keyft1 < 100
          and ten1.ft1    = one.keyf
          and one.f       = ten2.keyft2
          and ten2.keyft2 < 100;
```

The EXPLAIN statement causes the following output to be made:

```
TABLE    COLUMN_     STRATEGY                              PAGE
NAME     OR_INDEX                                          COUNT

TEN1                 RANGE CONDITION FOR KEY COLUMN       1250
ONE      KEYF        JOIN VIA KEY COLUMN                   125
TEN2     KEYFT2      JOIN VIA KEY COLUMN                  1463
                     RESULT IS COPIED, COSTVALUE IS         97
```

3.7.6 Ordered Select Statements (single row processing)

Since these statements do not use any storage space, this section only explains the way in which runtime can be shortened.

Every single row select can be performed by means of the sequential search starting from the first/last row or from the row identified by a key up to the first row which fulfills the condition. It is obvious that all available information about key ranges should be specified in order to reduce the number of rows to be searched through.

The syntax of the single row select permits the specification of the lower limit (select next) of a key range as well as the specification of an upper limit (search condition). For select first/last, it is possible to specify both limits in the search condition. For select next/prev, only the limit which is not specified in the <key spec list> is found from the search condition.

To find the rows which meet a specified condition, it is better to apply an index than to perform a sequential search.

If the index is directly specified in the program with INDEX or INDEXNAME, it must always be used for the search. If no index is specified, any index can be used for which the SAP DB optimizer finds equality conditions so that exactly one index list is designated.

Examples:

```
exec sql select next * from example
         key  firstkey = 3, secondkey = 0
         where firstkey <= 70 and secondkey = 20;
```

In this example, the range between 3,0 and 70,20 is searched through.

```
exec sql select first * from example
         index invcolumn1 = 'Miller'
         where normalcolumn = 4711;
```

The index of invcolumn1 is used and every index list, starting from 'Miller', is processed, if necessary.

```
exec sql select first * from example
         index invcolumn1 = 'Miller'
         where normalcolumn = 4711
           and invcolumn1   = 'Miller
```

The index of invcolumn1 is used. Only the index list with the value 'Miller' is processed, not all the index lists with values greater than 'Miller'.

3.7.7 Instructions to Improve the Performance

This section contains instructions which help to improve the runtime of applications.

- If a database is built, the definition of the tables should be derived from the structures previously investigated. When defining the key columns, it should be ensured that the most selective columns to which conditions are most frequently applied are placed at the beginning of the key. This guarantees that only a very small part of the table has to be considered during the processing of the select.

- Only columns of high selectivity should be indexed. No single-column index should be created on columns such as sex or personal status because of the small number of distinct values. These columns could be used very seldom for a non-sequential strategy, because this would normally be more costly than the sequential search.

- For relatively static data sets, many columns can be indexed. If it makes sense, multiple indexes should be created. As for the key column definition, it should be ensured that the most selective columns which are frequently used in EQUAL conditions are specified at the beginning of the multiple index.

- Not all the columns which are used in conditions should be indexed. The space required for the indexes and the overhead for their maintenance would be considerable.

- If many updates have been made to a table, UPDATE STATISTICS should be performed.

- SAP DB implicitly performs UPDATE STATISTICS when it determines that a table has been modified to a certain extent. SAP DB, however, is not able to recognize any change relevant to the correct determination of the currently best strategy and to execute an implicit UPDATE STATISTICS.

- Only conditions which are not met by all rows should be formulated. Frequently, applications are built in which the user defines the values of a condition. If the user does not specify any values, default values are entered into the condition so that it always yields "true". The database system must then evaluate such an inefficient condition for every row to be checked. It is better to issue various select statements which depend on user input.

- The most selective conditions should be placed at the beginning of the search condition.

- The specification

```
columnx between 1 and 5
```

is better than

```
columnx in (1,2,3,4,5)
```

- The specification

```
columnx in (1,13,24,...)
```

is better than

```
columnx=1 or columnx=13 or columnx=24 or ...
```

3.7.8 Additional EXPLAIN Information: Columns O, D, T, M

1. Column O (Only Index)

'*' - The strategy only uses the specified single-column or multiple-column index for command processing. The data of the base table is not accessed. It is required that only columns contained in the index structure are accessed in the sequence of < select column>s or <where clause>, if any.

2. Column D (Distinct Optimization)

- Column D only has an entry when column O contains an '*'.

'C' - (Complete Secondary Key)
In the sequence of <select column>s, all columns of a single-column or multiple-column index (and only those !) have been specified in any order after the keyword DISTINCT. Each time, the values of the corresponding index columns are accessed only once. A result table is not built (e.g., SELECT DISTINCT <all columns of the index> FROM ...).

'P' - (Partial Secondary Key)
In the sequence of <select column>s, the first k (k < total number of columns in the index) columns of a multiple-column index have been specified in any order after the keyword DISTINCT. Each time, the values of the corresponding index columns are accessed only once. A result table is not built (e.g., SELECT DISTINCT <first k columns of the multiple-column index> FROM ...).

'K' - (Primary Key)
In the sequence of <select column>s, all columns of a single-column or multiple-column index and the first k (k <= total number of columns in the key) columns have been specified in any order after the keyword DISTINCT. Each time, the values of the corresponding index and key columns are accessed only once. A result table is not built (e.g., SELECT DISTINCT <all columns of the index + the first k columns of the key> FROM ...).

3. Column T (Temporary Index)

'*' - An internal temporary index is built. In this index, the keys of the hit rows are sorted in ascending order. The hit rows have been determined by the corresponding key columns. The base table is accessed via this temporary index.

4. Column M (More Qualifications)

'*' - On index or key columns, there are conditions which cannot be used to directly restrict the range for an access via index (e.g., in case of an EQUAL/IN condition on the first and third column of a multiple-colun index, only the first condition in the strategy will be used for access). Those conditions affect the corresponding index strategy. They are only used to restrict access to the base table.

3.7.9 Summary of the Present Chapter

The attempt was made in this chapter to show the importance of optimal SQL statements and the ways in which these can be created. The descriptions given in this chapter should enable the user to find useful formulations for the enquiries. Before including these statements into a program, they can be checked for correctness within the SAP DB component QUERY and for the performance obtained by them by means of EXPLAIN statements.

If the user consequently follows the instructions given in this chapter from the very beginning, later tuning overhead, if any, should be minimal.

SQL is a powerful database query language, which can save an application quite a lot of work. But as shown by the last example in Section 3.7.7, "Instructions to Improve the Performance", there are limits beyond which the performance of an application can be improved, not by expanding an SQL command, but by adding some additional lines of code to the application.

4 Compatibility with Other Database Systems

4.1 ANSI

The SAP DB database system is compatible with ANSI SQL-92 (entry level). Application programs must be precompiled using the option "sqlmode ANSI". The sqlmode is automatically passed to the object program. Some parts of the statements which are not operative for SAP DB generate a warning (sqlwarnf).

The sqlcode is accepted by SAP DB, except for a few cases (see the Manual Messages: SAP DB).

The following conversions are done:

```
INTERNAL +250       to   ANSI        -250
INTERNAL +300       to   ANSI        -300
INTERNAL +320       to   ANSI        -320
INTERNAL Warning1   to   ANSISTATE  01004
INTERNAL Warning2   to   ANSISTATE  01003
```

Isolation level 3 or 30 (internal) is set as default. Valid are the isolation levels 0, 1, 2, 3, 4.

Dynamic statements with descriptors operate on the SAP DB SQLDA. The SAP DB SQLDA structure must be included for dynamic SQL in ANSI mode. This is done automatically while precompiling.

If the user wants to issue SAP DB specific statements in the application program, these have to be preceded by "exec sapdb" instead of "exec sql".

For a detailed description, see the Reference Manual: SAP DB (7.2) for the ANSI mode.

4.2 DB2

The SAP DB database system is compatible with DB2. Application programs must be precompiled using the option "sqlmode DB2". The sqlmode is automatically passed to the object program. Some parts of the statements which are not operative for SAP DB generate a warning (sqlwarnf).

The sqlcode is accepted by SAP DB (see the Manual Messages: SAP DB).

Isolation level 1 or 10 is set as default. Valid are the isolation levels 0, 1, 2, 3, 4.

The corresponding SQLDA structures must be included for dynamic SQL in DB2 mode. These structures are compatible with the DB2 SQLDA structures, but they contain internal SAP DB information in addition. Therefore, the corresponding include files should be used which are automatically included during precompilation.

If the user wants to issue SAP DB specific statements in the application program, the statements must be preceded by "exec sapdb" instead of "exec sql".

For a detailed description, see the Reference Manual: SAP DB (7.2) for the DB2 mode.

4.3 ORACLE

The SAP DB database system is compatible with ORACLE. Application programs hmust be precompiled using the option "sqlmode ORACLE". The sqlmode is automatically passed to the object program. Some parts of the statements which are not operative for SAP DB generate a warning (sqlwarnf).

The sqlcode is accepted by SAP DB, except for a few cases (see the Manual Messages: SAP DB).

The following conversions are done:

```
INTERNAL Warning3    to    ORACLE               -1008
INTERNAL     -804    to    ORACLE               -1007
INTERNAL     -813    to    ORACLE               -1034
INTERNAL    -4000    to    ORACLE                   0
INTERNAL     -743    to    ORACLE            Warning1
INTERNAL      100    to    ORACLE  1403 ROW NOT FOUND
```

Isolation level 1 or 10 is set as default. The valid isolation levels are 0, 1, 2, 3, 4.

The corresponding SQLDA structures must be included for dynamic SQL in ORACLE mode. These structures are fully compatible with the ORACLE SQLDA structures, but they contain internal SAP DB information in addition. Therefore, the corresponding include files should be used which are automatically included during precompilation.

If the user wants to issue SAP DB specific statements in the application program, these have to be preceded by "exec sapdb" instead of "exec sql".

For a detailed description, see the Reference Manual: SAP DB (7.2) for the ORACLE mode.

4.4 ANSI ORACLE

For the description, see Section 4.3, "ORACLE". Character host variables are converted as in sqlmode ANSI.

```
INTERNAL 100    to    ANSI-ORA  100  ROW NOT FOUND
```

5 The SAP DB Precompiler

The SAP DB precompiler is a program which is executed before the programming language compiler. The basic function of the precompiler is to translate all the SQL statements into statements of the corresponding programming language. In this process, the precompiler only translates those pieces of information contained in a program which are needed for compiling the SQL statements. The language compiler will then decide whether the program is correct according to the programming language description.

The precompiler has additional functions which are described in the following section.

5.1 SAP DB Precompiler Functions

- The precompiler opens the database session under a user name which can be specified as a parameter for the call. This user name does not need to be identical to the SAP DB user name used during the execution of the program. If the username is not specified as a parameter, then the connect that is statically placed in the module as the first statement is taken. If the first statement is not a connect, a database session is opened with predefined user specifications.

- The precompiler checks the compatibility of the corresponding host variables and SAP DB column types. In doing so, the rules specified for the conversion possibilities apply.

- The precompiler actually executes administrative statements (create table, etc.) and resets their effects at the end of the precompiler run.

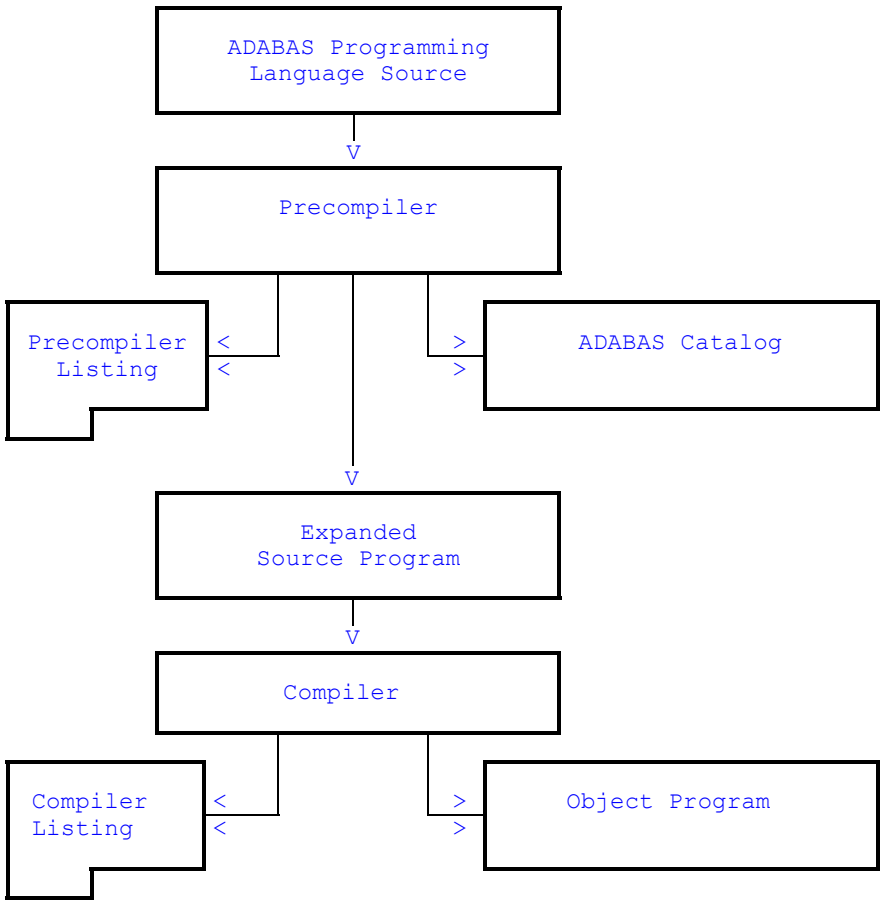
- The precompiler analyzes the syntax of table statements (insert, select, etc.) and checks them with regard to the privileges. When accessing existing tables, it is important that they are already available for the SAP DB user whose name is used by the precompiler.

- The precompiler enters the names of the tables the program processes into system tables. Afterwards, the relations of the programs and used tables can be shown via select statements performed on the tables DOMAIN.MOD_USES_TAB or DOMAIN.MOD_USES_COL. These tables are described in the Reference Manual: SAP DB (7.2).

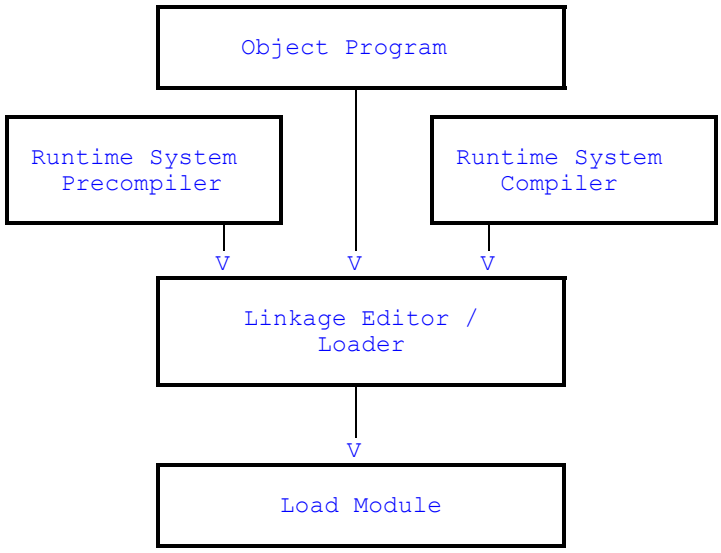
- The precompiler generates the compiler input, thereby transforming the embedded SQL statements into calls of procedures of the SAP DB precompiler runtime system.

5.1.1 Translation of a SAP DB Application Program

(1) Start of the SAP DB precompiler and of the compiler of the chosen programming language.



(2) Generation of a load module



(3) Subsequently, the SAP DB application program can be executed.

5.1.2 Precompiler Options

This section contains a survey of all functions of the SAP DB precompiler which can be controlled by options. The syntax of these options is described separately in the platform-specific SAP DB User manual.

ansi c
Code is generated which is compatible with ANSI C.

c++
Code is generated which is compatible with C++.

check/nocheck
The precompiler checks the syntax of all SQL statements, it checks whether the table and column names are available and whether their types are compatible with the host variables in use. The precompiler opens a database session either under the user name specified in the option or in the connect statement or under the predefined user name, and executes all SQL statements according to the order of their occurrence in the program. Possible error messages of the database system are stored as warnings in the precompiler listing. SQL statements which cannot be executed at the time of precompilation (e.g., dynamic statements) are only checked with regard to syntax errors; they generate warnings. The system tables DOMAIN.MOD_USES_TAB or DOMAIN.MOD_USES_COL, which contain the program-data relationships, are maintained. When starting the precompiler run, all entries relating to the connect user name and program name or module name are deleted from the corresponding tables. At the regular end of the precompiler run (no errors), new entries are made in the corresponding tables. The option nocheck does not establish a connection to the database. It is therefore possible to precompile without a started database.

check/syntax
The syntax of all SQL statements is checked and possible SAP DB error messages are written to the precompiler listing.

comment
All SQL statements are written as comments into the source file saved for the compiler.

compatible
With Version 6.1, new language elements are integrated in the embedded SQL (see Section 6.4.2, "exec sql [<n>] <array statement>"). These elements can change the interpretation of programs written for former releases. The option compatible enforces the old interpretation and thus guarantees upward compatibility. This option need only be specified if an error message occurs during re-precompilation of an existing program. In any case, new programs should be written in such a way that they can be precompiled without this option.

date-time/eur
This option can be used to set the date and time representation to "europe". It is passed to the application program.

date-time/iso
This option can be used to set the date and time representation to "iso". It is passed to the application program.

date-time/jis
This option can be used to set the date and time representation to "jis". It is passed to the application program.

date-time/usa
This option can be used to set the date and time representation to "usa". It is passed to the application program.

Representation:

Date-Time	Date	Time	Timestamp
ISO	yyyy-mm-dd	hh.mm.ss	yyyy-mm-dd-hh.mm.ss.mmmmmm
USA	mm/dd/yyyy	hh:mm AM	yyyy-mm-dd-hh.mm.ss.mmmmmm
EUR	dd.mm.yyyy	hh.mm.ss	yyyy-mm-dd-hh.mm.ss.mmmmmm
JIS	yyyy-mm-dd	hh:mm:ss	yyyy-mm-dd-hh.mm.ss.mmmmmm
INTERNAL (Default)	yyyymmdd	hhhhmmss	yyyymmddhhmmssmmmmmm

extern
A module precompiled with the option extern can be linked to modules of other precompilers. The module itself must not contain a main function. Each function of the module containing SQL statements must be defined according to the following schema:

```
void xfunc_ (sqlca,...)
sqlcatype *sqlca; ...
{
...
}
```

The identifier SQLCA for the corresponding formal parameter is prescribed.
A COBPC program could call this function in the following way:

```
IDENTIFICATION DIVISION.
...
WORKING-STORAGE SECTION.
EXEC SQL INCLUDE SQLCA END-EXEC.
...
EXEC SQL CONNECT DEMO IDENTIFIED BY DEMO END-EXEC
...
         CALL 'xfunc' USING SQLCA ...
...
```

The first SQL statement must be executed in the calling module which is precompiled without the extern option.

help
This option displays all the precompiler and precompiler runtime options (application programs) on the terminal.

isolation level(<level number>)
If the option check or syntax is set, the level number 10 (default value) is always specified for the connect. Under this number, the locks must be set to a database during precompilation. This option, if specified, is only passed to the application program and has only an effect on the session with the number 1. It overrides the

level specification for all connect statements with the session number 1 made in an application program. It can be overridden by the same runtime option when starting the application program.

list
A precompiler listing is generated. If this option is not specified, the precompiler listing only consists of warnings and error messages.

margins (<leftmargin>, <rightmargin>)
This option can be used to determine the range of lines in which the precompiler has to work.

max packetsize (<maxpacketsize)
This option can be used during precompilation to determine the size of a static statement in bytes. The default is set to 16000 bytes. This size is retrieved from the database at runtime.

nowarn
The warnings -733, -735, -853, -884, and -885 are not output.

precom
The compiler will not be started after precompilation. The source file is saved for the compiler.

profile
The precompiler generates code for profiling the SQL statements. Profiling is started with a runtime option.

program (<program name>)
When using the option check, a program name can be specified here related to which the names of the tables and columns used in the program are stored in system tables. These entries can be looked up via a select performed on the tables DOMAIN.MOD_USES_TAB or DOMAIN.MOD_USES_COL. The default value of <program name> is the filename of the source program.

serverdb (<serverdb>)
If the option check or syntax is set, a serverdb name can be specified here under which the database will be accessed at precompilation time. This option has an effect on all sessions with the number 1. When precompiling, this option overrides either the specifications made in the connect statement or the predefined user specifications. It is not passed to the application program.

servernode (<servernode>)
If the option check or syntax is set, a node name can be specified here under which the database will be accessed at precompilation time. This option has an effect on all sessions with the number 1. When precompiling, this option overrides either the specifications made in the connect statement or the predefined user specifications. It is not passed to the application program.

silent
No output is made to the screen.

sqlmode/SAP DB
This is the "native mode" of a SAP DB application and the default value of the sqlmode option.

sqlmode/ANSI
This mode ensures ANSI compatibility of all SQL commands. Within the program, variables and statements must be defined according to the ANSI standard. When this option is enabled, some positive error situations are transformed into negative error messages (see Section 4.1, "ANSI Compatibility"). They are passed to the application program. The option must be specified with the main module which contains the first connect statement.

sqlmode/ANSIORA
This mode ensures ORACLE compatibility of all SQL statements. The host variables are converted as in sqlmode ANSI; i.e., they are padded with blanks. The error message "Row Not Found" is returned with the code +100.

sqlmode/DB2
This mode ensures DB2 compatibility of all SQL statements. The precompiler generates the "DB2 struct SQLDA" definition. Within the program, variables and statements must be defined according to the DB2 standard.

sqlmode/ORACLE
This mode ensures ORACLE compatibility of all SQL statements. Within the program, variables and statements must be defined according to the ORACLE standard. When this option is enabled, some positive error situations are transformed into negative error messages (see Section 4.3, "ORACLE Compatibility"). They are passed to the application program. The error message "Row Not Found" is returned with the code 1403.

timeout (<timeout>)
This option can be used to specify the session_timeout for session 1. When precompiling, this option overrides either the specifications made in the connect statement or the predefined user specifications. It is not passed to the application program.

trace file (<trace filename>)
This option can be used to specify a name other than the default name for the trace file. If no other trace option was specified, the option "trace short" is simultaneously enabled by default. The filename is standardized in the platform-specific SAP DB User manual according to the operating system conventions. This option is only passed to the application program. It can be overridden in the application program by the same runtime option.

trace file (.pid)
The string "pid" or "PID" has the effect that the trace file name is built with the process id: "pid<process id>.pct". The default "Trace Off" is enabled if no other trace option was specified.

trace long
Each SQL statement is written into a file, together with the values of all host variables, the sqlcode (if not equal to zero), the warnings (if warnings exist), and the "sqlerrd (index_3)" value. This option is only passed to the application program. It can be overridden in the application program by the same runtime option.

trace short
Each executed SQL statement is written into a file, together with the sqlcode (if not equal to zero), the warnings (if warnings exist), and the "sqlerrd (index_3)" value. This option is only passed to the application program. It can be overridden in the application program by the same runtime option.

user (<userid> <password>)
If the option check or syntax is set, a user name can be specified here under which the database will be accessed. This option only has an effect on the session with the number 1. When precompiling, this option overrides either the specifications made in the connect statement or the predefined user specifications. It is not passed to the application program.

userkey (<userkey>)
If the option check or syntax is set, a userkey can be specified here. The pertinent user, the database, timeout, cachelimit, and sqlmode are fetched from the XUSER file. The corresponding database can then be accessed by means of this user id. This option only has an effect on the session with the session number 1.

version
This option can be used to see information about the version used.

5.1.3 User Specifications for a Precompiler Run

User specifications for the precompiler call are only required when the option check (default) or syntax is set. If the application runs in multi-db mode, user specifications are needed for each of the up to eight database sessions. If the precompiler option nocheck is set, a database session is not opened, so that no user specifications are necessary. The concept of how a user can connect to a database, the possible user specifications, and the syntax according to which the user specifications have to be made are described in the platform-specific SAP DB User manual.

The following precedence rules are applied to the user specifications (user, password, servernode, serverdb, timeout, sqlmode, isolation level, and locale):

1. If the statically first SQL statement of a database session is not a connect statement, then all the user specifications are fetched from the XUSER file.

2. If the statically first SQL statement of a database session is a connect statement, then all the specifications made in this statement are taken and the missing specifications are fetched from the XUSER file.

3. For the database session with the number 1, the user specifications can be overridden by means of the corresponding precompiler options.

The precompiler option isolation level is passed to the application program. It overrides the specifications made in the application program. Isolation level can be overridden by means of runtime options.

5.1.4 Precompiler Output Files

Source and error listing
Errors are written to the precompiler listing. If errors occur before this file can be opened (e.g., in the option specifications), they are written into a special error file (sqlerror) which is described in more detail in the platform-specific SAP DB User manual.

Object module
Together with other object modules and the runtime system, the file is linked to an executable module.

Trace file
This file contains the executed SQL statements.

5.2 Functions of the SAP DB Runtime System

- Values are assigned from host variables to database columns and vice versa and values are converted.
- Null values (undefined column values) are dneoted by indicator variables.
- If database errors occur, the default error handling routines defined in the program are performed.

5.2.1 Runtime Options

This section contains a survey of all runtime options. They can be specified when a SAP DB application is started.

isolation level(<level number>)
A level number can be specified here under which the locks will be set to a database. This option only has an effect on the session with the number 1. It overrides the level number in the specified connect statement and the level number passed during the precompiler run.

mfetch (<number>)
This option can be used to optimize access for fetch statements. Number indicates the number of 8k byte buffers into which rows of different result tables can be stored simultaneously, thus enabling faster access to the rows. <number> is preset to 1. If a trace option is active, the mfetch option has no effect.

no select direct fast
This option can be used to disable the optimization of the select direct statements. This is necessary especially if the error -9806 occurs. The optimization only has an effect when repeating the select direct statement.

profile
This option can be used to enable a profile which calculates the complete time of processing. If the system table SYSPROFILE is available for the LOCALSYSDBA, the values will be written to this table (see Section 5.3.4, "Profiling").

serverdb (<serverdb>)
This option has an effect on every session with the number 1. The option overrides the specifications of the connect statement made in the application program or the predefined user specifications.

servernode (<servernode>)
This option has an effect on every session with the number 1. The option overrides the specifications of the connect statement made in the application program or the predefined user specifications.

timeout (<timeout>)
This option can be used to specify the session timeout for the session with the number 1. This option overrides the timeout specifications made in the indicated connect statement.

trace alt (<statement count>)
Trace output is made alternately to two files. When doing so, as many SQL statements are recorded in each file as are specified in <statement count>. If there are more SQL statements than indicated by <statement count>, the files will be overwritten cyclically. The trace files are named "fn.pct" and "fn.prot". The SQL statements are recorded together with the contents of the host variables (trace long). Date and time specifications cannot be suppressed.

trace file (<trace filename>)
This option can be used to specify a name other than the default name for the trace file. If no other trace option was specified, the option trace short is simultaneously enabled by default. The filename is standardized in the platform-specific SAP DB User manual according to the operating system conventions. It can also be specified (optionally) as a character string constant. The option overrides the option transferred during the precompiler run. Only trace files with default trace filenames are not buffered on output.

trace file (.pid)
The string "pid" or "PID" has the effect that the trace file name is built with the process id: "pid<process id>.pct". The default "Trace Off" is enabled if no other trace option was specified.

trace long
Each SQL statement is written into a file, together with the values of all host variables, the sqlcode (if not equal to zero), the warnings (if warnings exist) and the "sqlerrd (index_3)" value. This option overrides the option transferred during the precompiler run and the statements specified in the application program.

trace no date/time
This option can be used to suppress the output of the date and time specifications for the start and end of the execution of an SQL statement made into the trace file.

trace short
Each executed SQL statement is written into a file, together with the sqlcode (if not equal to zero), the warnings (if warnings exist) and the "sqlerrd (index_3)" value. This option overrides the option transferred during the precompiler run and the statements specified in the application program.

trace time (<seconds>)
SQL statements are only recorded if their execution time is greater than or equal to <seconds>. The SQL statements are recorded together with the contents of the host variables (trace long). Date and time specifications cannot be suppressed.

user (<userid> <password>)
If the application program will be performed under a user other than specified in the connect statement or in the predefined user specifications, this user can be specified here. This option has only an effect on the session with the number 1.

userkey(<userkey>)
A userkey can be specified here. The pertinent user, the database, timeout, cachelimit, and isolation level are fetched from the XUSER file. The corresponding database can then be accessed by means of this user id. This option only has an effect on the session with the number 1. It overrides either the user specified in the connect statement or the predefined user specifications.

5.2.2 User Specifications for the Application

There can be 1 to 8 concurrent database sessions (multi-db mode), so that user specifications must be available for each database session.

The following precedence rules apply to the user specifications:

1. If no connect statement for the database session is specified in the application program, the user specifications are taken from the XUSER file.

2. If a database session is opened via a connect statement, the user specifications made in the connect statement are taken. Missing specifications are fetched from the corresponding XUSER parameter combination.

3. The user specifications can be overridden for the database session with the number 1.

If the isolation level was set by means of the corresponding precompiler option, it can only be overridden by a runtime option.

5.3 Precompiler Debugging Aids

Test functions are available at precompilation time as well as at the application's runtime.

5.3.1 Testing at Precompilation Time

Depending on option settings, the SAP DB database system can either check only the syntax of an SQL statement (syntax) or also the availability of the specified tables and columns and the compatibility of their types with host variables (check). For this purpose, a database session is opened and all SQL statements are executed at precompilation time. After precompilation, the effects of the SQL statements are cancelled. The option user specified in the program determines under which identification this database session is to be performed. Any connect statement in a program will be rejected with the error message "-3005 INVALID COMMAND". The SQL statements are executed in the sequence of their static placement within the text, not in the dynamic order of the program statements. For various statements, e.g., dynamic SQL statements, only a syntax check can be performed.

Since the control flow of the application is not taken into consideration when testing with the precompiler, the interrelations of the SQL statements cannot be traced by means of this mechanism.

The results of the precompiler tests are stored in the precompiler listing.

5.3.2 The SAP DB Trace Facility

The trace of a SAP DB application can be activated at three levels; whereby all results are recorded in a special trace file.

The lowest level is that of the SQL statement. The precompiler directives "trace" or "trace long" and "trace off" enclose those SQL statements whose execution is to be recorded in the trace file. The directives are written into the program like SQL statements, e.g.:

```
exec sql set trace long ;
```

The trace file contains all SQL statements enclosed by the directives; including those SQL statements which are executed within called subroutines. The name of the trace file is standardized according to the operating system conventions.

The next level comprises program units - such as modules, subroutines which can be translated separately etc. The SQL statements contained there are recorded in the trace file according to their logical sequence within the program. Any trace statements present are rendered inoperative. The trace file is enabled with precompiler options. The first <trace filename> addressed is the name of the trace file. This name is either formed from the <trace filename> or from the program name and a suffix.

The highest level is the trace of the entire application. If a trace option is specified when calling a SAP DB application, a trace file is opened for all executed SQL statements.

A comment line can be inserted into the trace file via the SQL statement "exec sql set trace line".

5.3.3 The Trace File

The trace file contains information sent to or received from the interface to the SAP DB kernel. The SQL statements are only recorded for parse requests to the kernel. The parse identification (parseid) can be used to find out which SQL statement is currently executed.

The information contained in the trace file depends on the trace statement or trace option.

For a simple trace, only the sequence of SQL statements sent to the database system is recorded: this aids in checking dynamic SQL statements and macros.

A declare cursor statement is only executed with the opening of the result table (open statement). In the trace file, it can therefore be found exactly at this place. The open statement is only denoted by a comment.

"sqlerrd(index_3)" contains the number of processed rows.

Example:

```
select next hno, name, zip, city, price into :chno, :cname,
:czip, :ccity, :cprice from hotel key hno = :chno
SQL STATEMENT   :  FROM MODULE : CHBL    AT LINE : 328
PARSEID: OUTPUT:  000014DBD00000013D000000
PARSEID: INPUT :  000014DBD00000013D000000
SQLERRD(INDEX_3) : 1
START   :  DATE :  1996-02-17    TIME :  0013:39:44
END     :  DATE :  1996-02-17    TIME :  0013:39:44

SQL STATEMENT   :  FROM MODULE : CHBL    AT LINE : 315
PARSEID: INPUT :  000014DBCD0000013C002C00
WARNING: W-------8-------
SQLERRD(INDEX_3) : 4
START   :  DATE :  1996-02-17    TIME :  0013:39:44
END     :  DATE :  1996-02-17    TIME :  0013:39:44
```

If the detailed form of the trace file is required, the input and output values of the host variables involved are also included.

Example:

```
select next hno, name, zip, city, price into :chno, :cname,
:czip, :ccity, :cprice from hotel key hno = :chno
SQL STATEMENT   :  FROM MODULE : CHBL    AT LINE : 328
PARSEID: OUTPUT:  000014D1D00000013D000000
PARSEID: INPUT :  000014D1D00000013D000000
INPUT   :    6: chno                          :    10
OUTPUT :     1: chno                          :    11
OUTPUT :     2: cname                         :  CONGRESS
OUTPUT :     3: czip                          :   48226
OUTPUT :     4: ccity                         :  DETROIT
OUTPUT :     5: cprice                        :      87.50
SQLERRD(INDEX_3) : 1
START   :  DATE :  1996-02-17    TIME :  0013:12:26
END     :  DATE :  1996-02-17    TIME :  0013:12:26

SQL STATEMENT   :  FROM MODULE : CHBL    AT LINE : 315
PARSEID: INPUT :  000014D1CD0000013C002C00
INPUT   :    1: chnr                          :    81
WARNING: W-------8-------
SQLERRD(INDEX_3) : 4
START   :  DATE :  1996-02-17    TIME :  0013:12:26
END     :  DATE :  1996-02-17    TIME :  0013:12:26

SQL STATEMENT   :  FROM MODULE : CHBL    AT LINE : 319
PARSEID: INPUT :  000014D1CE0000011C002A00
OUTPUT :     1: cadat                         : 11.11.95
OUTPUT :     2: cedat                         : 30.11.95
SQLERRD(INDEX_3) : 1
START   :  DATE :  1996-02-17    TIME :  0013:12:26
END     :  DATE :  1996-02-17    TIME :  0013:12:26
```

If the detailed form of the trace file is only required for the program run, the names of the host variables are not available. The input and output values of the host variables concerned only get the numbers of the parameters.

Example:

```
select next hno, name, zip, city, price into :chno, :cname,
:czip, :ccity, :cprice from hotel key hno = :chno
SQL STATEMENT  :  FROM MODULE : CHBL    AT LINE : 328
PARSEID: OUTPUT:  000014D9D00000013D000000
PARSEID: INPUT :  000014D9D00000013D000000
INPUT  :    6: PARAMETER                        :    10
OUTPUT :    1: PARAMETER                        :    11
OUTPUT :    2: PARAMETER                        : CONGRESS
OUTPUT :    3: PARAMETER                        :  48226
OUTPUT :    4: PARAMETER                        : DETROIT
OUTPUT :    5: PARAMETER                        :      87.50
SQLERRD(INDEX_3) : 1
START  :  DATE :  1996-02-17    TIME :  0013:38:45
END    :  DATE :  1996-02-17    TIME :  0013:38:45

SQL STATEMENT  :  FROM MODULE : CHBL    AT LINE : 315
PARSEID: INPUT :  000014D9CD0000013C002C00
INPUT  :    1: PARAMETER                        :    81
WARNING: W-------8-------
SQLERRD(INDEX_3) : 4
START  :  DATE :  1996-02-17    TIME :  0013:38:45
END    :  DATE :  1996-02-17    TIME :  0013:38:45

SQL STATEMENT  :  FROM MODULE : CHBL    AT LINE : 319
PARSEID: INPUT :  000014D9CE0000011C002A00
OUTPUT :    1: PARAMETER                        : 11.11.95
OUTPUT :    2: PARAMETER                        : 30.11.95
SQLERRD(INDEX_3) : 1
START  :  DATE :  1996-02-17    TIME :  0013:38:45
END    :  DATE :  1996-02-17    TIME :  0013:38:45
```

5.3.4 Profiling

When the option profile is enabled during an application program's run, statistical data on the processed SQL statements can be obtained. To obtain results, however, the table SYSPROFILE (system table) of the LOCALSYSDBA must have been created when initializing the database. If this has not been done, the table has to be created first.

Definition of the table:

```
create table sysprofile (
          username  char  (18) key,
          progname  char  (18) key,
          modname   char  (18) key,
          language  char  (12) key,
          lineno    fixed ( 7) key,
          parseid   char  (12) byte key,
          rundate   date,
          runcount  fixed (10),
          seconds   fixed (12,3)
          stmbegin  char  (3900),  ) ;
```

For every SQL statement, the beginning of statement, date of runtime, number of calls and accumulated realtime is entered into the table. The realtime consists of the time taken by the processing of a statement within an application program including all data conversions and ime needed by the SAP DB kernel. The time needed to enter this information into the table SYSPROFILE, however, is not included. The key of a row consists of the following specifications: user name, program name, module name, language of the application program, and line number of the statement within the application program related to the source and the internal parseid. The parseid is integrated into the key in order to be able to distinguish dynamic statements. With the enabled trace option, the time required for writing the trace to a file affects the profiling. Therefore, it is not convenient to activate the profile and trace options at the same time.

The entries to the SYSPROFILE table are made within the transactions of the application program. Therefore, they are only stored in the table when the application program issues a COMMIT WORK.

When the option is enabled, old entries made for username, program name, and language are always deleted at the beginning of the program. After the run, the table LOCALSYSDBA.SYSPROFILE can be looked up by means of QUERY and the stored data can be evaluated. The entries remain in the table until they are deleted explicitly or the program is restarted with the PROFILE option.

6 SAP DB Precompiler Statements

6.1 Include Statements

Include statements are generally used to insert text into a source program at precompilation time. This text will be included in place of the include statement.

Within the source texts of the applications, include statements are allowed at the following positions:

- SQLCAs can be at any place where data definitions are permitted;
- "include <filename>" can be at any place where the program text to be inserted is permitted. The source text is located in the file to be specified. It must not contain any "exec sql include <filename>" statement.

6.1.1 exec sql include sqlca

This statement identifies the place within the program text where the SQLCA has to be declared. If any SQL statement is included in the program text, the SAP DB C precompiler generates the required declarations independently. This statement can therefore be omitted; it only serves to guarantee the compatibility with other systems.

6.1.2 exec sql include <filename>

```
exec sql include <fname> [<declare clause>]

  <declare clause> ::= <table clause> | <dbproc clause>
  <table clause>   ::= table <tname> [<as clause>] [<ind clause>]
  <dbproc clause>  ::= dbproc <dbprocname>
                       [<as clause>] [<ind clause>]
  <as clause>      ::= as var    [<variable declarator>] |
                       as type   [<type declarator>] |
                       as struct [<structure tag>]
  <ind clause>     ::= ind [<declarator>] [<structure tag>]
  <fname>          ::= "<character seq>" | '<character seq>'
  <tname>          ::= <identifier>
```

If <fname> is available as a file, the text contained in it will be inserted into the calling program text. A <declare clause>, if any, will be ignored in this case.

If there is no file <fname>, either a <table clause> or a <dbproc clause> must be specified. The precompiler run must be performed with the option check. The specifications required for the database session are taken from the XUSER file or from the options.

<table clause>
A table <tname> must exist in the database. Declarations are derived from the table definition. These declarations help to assign variables to the columns of the table. The declarations are entered into the file <fname> and simultaneously into the calling program text. They can then be recalled from the generated file if they are needed for further precompilation.

<dbproc clause>
A DB procedure <dbprocname> must exist in the database. Declarations are derived from the DB procedure. These declarations help to assign variables to the parameters of the DB procedure. The declarations are entered into the file <fname> and simultaneously into the calling program text. They can then be recalled from the generated file if they are needed for further precompilation.

<as clause>
A structure declaration for SQL parameters is generated. The component names correspond to the column names of the table ("case significance"), the component types are compatible with the column types. A variable declaration is generated by "as var", a "typedef" declaration by "as type" and a "structure tag" declaration by "as struct". The default is "as var". Several array declarators and at most one pointer declarator can be specified. Default for declarators or "structure tag" is the table name.

<ind clause>
Only when <ind clause> is specified, a structure declaration for indicators is generated in addition. The component names are derived from the column names by prefixing an "I" to them, and all component types are "short". The kind of declaration results from the "as clause". The default for declarator or "structure tag" is the derivation from the table name by prefixing it with an "I".

Also see Section 2.6, "Generating Host Variables".

6.2 Declare Statements

The declare sections contain data definitions which can occur in SQL statements. The data definitions can be distributed over several declare sections.

6.2.1 exec sql begin declare section

This statement introduces a section where host variables are declared. It can only occur at places within the program where variable declarations are permitted.

6.2.2 exec sql end declare section

This statement closes a declare section.

6.3 Whenever Statements

Error and exception handling routines for SQL statements can be programmed by means of whenever statements.

The whenever statements must be placed before the SQL statements that they affect. Whenever actions are valid until they are changed by further whenever statements, or up to the end of the program. The static position in the source program and not the control flow determines the scope of a whenever statement.

```
<whenever action> ::= <call> |
                      <continue> |
                      <go to> |
                      <stop>

<call>            ::= call <subprogram>
<continue>        ::= continue
<go to>           ::= goto <label>
<stop>            ::= stop
```

<subprogram> is a function call.

<label> is a C skip label. 50 characters are available for <subprogram call>, 45 characters for <label>.

6.3.1 exec sql whenever sqlwarning

```
exec sql whenever sqlwarning <whenever action>
```

The <whenever action> is executed whenever the SAP DB system issues a warning. A warning exists when sqlwarn0 is set to "W". Consequently, other sqlwarnings exist.

6.3.2 exec sql whenever sqlerror

```
exec sql whenever sqlerror  <whenever action>
```

The <whenever action> is executed whenever the SAP DB system reports an error. An error message exists when sqlcode has a negative value.

6.3.3 exec sql whenever sqlexception

```
exec sql whenever sqlexception <whenever action>
```

The <whenever action> is executed each time the SAP DB system reports an exceptional case. An exceptional case exists when sqlcode has a value greater than 0.

6.3.4 exec sql whenever not found

```
exec sql whenever not found   <whenever action>
```

The <whenever action> is executed each time SAP DB issues the message "not found". "Not found" corresponds to the sqlcode=100.

6.3.5 exec sql whenever sqlbegin

```
exec sql whenever sqlbegin   <call>
```

The specified C function is executed before each SQL statement that is placed after this whenever statement in the source text.

6.3.6 exec sql whenever sqlend

```
exec sql whenever sqlend   <call>
```

The specified C function is executed after each SQL statement that is placed after this whenever statement in the source text.

6.4 SAP DB Statement

6.4.1 exec sql [<n>] <sql statement>

This statement precedes all SQL statements described in the Reference Manual: SAP DB (7.2).

```
exec sql [<n>] <sql statement>

  <n>           ::= number of the database session
                            1 .. 8  , 1 = default
```

6.4.2 exec sql [<n>] <array statement>

The array statement has the general format:

```
exec sql [<n>] <array statement>

  <array statement> ::= [<for clause>] <sql statement>
  <for clause>      ::= for <loop parameter>
  <sql statement>   ::= <verb> <...> <parameter>,
                        ...<...> <parameter>,... <...>
  <loop parameter>  ::= <unsigned integer constant>
                      | :<integer variable>
  <verb>            ::= select
                      | select into (* not implemented *)
                      | fetch
                      | insert
                      | update
                      | delete

  <parameter>       ::= :<array variable>
  <n>               ::= number of the database session
                              1 .. 8  , 1 = default
```

An array statement is formed from an SQL statement by replacing all scalar and structured parameters with arrays of the corresponding type of element. All dimensions must be identical, otherwise a warning is issued and the statement with the smallest dimension is executed. It is possible to specify an indicator array with the same dimension for each parameter array. The elements of the indicator array must be arrays of the type integer.

The result of an array statement is the same as that of the underlying SQL statement repeated n times; each array element of the parameters is applied to one data row (except for the where clause in the select). The repetition factor n results from the minimum of the array dimension and <loop parameter>, if specified. The number of the rows k successfully processed is returned in sqlca.sqlerrd [2]. k < n is true if sqlca.sqlcode != 0, i.e., if an error occurs during execution. The <loop parameter>, on the other hand, is always an input parameter, so no new value will be assigned to it. The values of the (i -1)th elements of the parameter arrays and indicator arrays are valid for the i-th processed row.

The SQL statement "select into" is only valid in "oracle" mode. Each output parameter must have an indicator. If no data row can be found for a where condition, the select statement produces the null value (indicator = -1). Sqlerrd[2] contains the number of data rows found.

Arrays as parameters were already allowed in former precompiler versions; but instead of being interpreted in the context described here, they were mapped columnwise (like structures). For existing applications using this kind of array as parameters, the option comp has been installed. Array statements to be inserted into a program which has to be precompiled with comp must contain the <for clause>.

6.5 Macro Statement

6.5.1 exec sql set macro %nnn = <macroline>

```
<macroline>     ::= <table name>.<column name> |
                    <table name> |
                    <column name>|
                    <variable name>

<variable name> ::= :<prog identifier>
<table name>    ::= [<auth id>.]<identifier>
<auth id>       ::= <identifier>
<column name>   ::= <identifier>
```

<macroline> can have up to 30 characters.

<prog identifier> corresponds to an identifier and is of the type character string.

"%nnn" is the macro identification, whereby "nnn" is an integer with 1 <= nnn <= 128 .

Macro definitions have global validity for all modules of the database application. With regard to the control flow, the macro must be defined before its call within the application.

6.6 Dynamic SQL Statements

SAP DB statements cannot only be written statically into an application program, but also be dynamically generated or read in and executed during a program run. The SQL statements to be performed dynamically can contain host variables as input and output parameters.

Dynamic statements can also be executed in a DB2- or ORACLE-compatible way. To do so, the corresponding sqlmode must be specified for precompilation. The syntax of the statements has therefore been extended. The statements described in the Reference Manual: SAP DB (7.2) for the SQLMODE DB2 and ORACLE must be used for these sqlmodes.

Dynamic statements are processed in two steps or, when a descriptor is used, in three steps.

- The SQL statement specified as a parameter is prepared and named for its execution by means of the prepare statement. Host variables which possibly exist within the SQL statement to be prepared must be designated by question marks or are described via the descriptor.

- If the prepare statement stipulates that the host variables are to be described via a descriptor, the describe statement must be called afterwards. This statement generates an SQLDA (SQL-Descriptor-Area) and provides it with information that is needed for the parameter identification. Using this information, the user can associate appropriate program variable values with parameters.

- Once the prepare or describe statement is successfully executed, the prepared SQL statement can be performed. This is done by the SQL statement execute which requires

    - the name qualified in the prepare statement for the SQL statement to be executed,
    - the parameters to be included in place of the question marks. The usage of the descriptor must be indicated ("...using descriptor").

"execute immediate" offers another possibility of dynamically executing SQL statements. The statement to be performed does not need to be prepared with prepare, but it must not contain host variables.

6.6.1 exec sql [<n>] prepare

```
exec sql [<n>] prepare <statement name>
         [  into <descriptor name>
         [  using <using clause> ] ]
            from <statement source>

  <statement source>   ::= <variable name>
                         | <string constant>
  <using clause>       ::= names | labels | any
  <variable name>      ::= :<prog identifier>
  <string constant>    ::= '<character seq>'
  <statement name>     ::= <identifier>
                         | < variable name >
  <n>                  ::= number of the database session
                                    1 .. 8  , 1 = default
```

This statement prepares an SQL statement for dynamic execution. Thereby all host variables needed at execution time are designated by a question mark (?). The prepared SQL statement is subsequently identified via <statement name>. <prog identifier> corresponds to an identifier and is of the type character string. The <descriptor name> and the using clause have no effect on the processing. They only serve to ensure the DB2 and ORACLE compatibility of the statement.

6.6.2 exec sql [<n>] describe

```
exec sql [<n>] describe <statement name> [ into <descriptor name>
         [  using  <using clause> ] ]

  <n>             ::= number of the database session
                              1 .. 8  , 1 = default
```

The describe statement ensures that the information required for associating a program variable with a parameter is stored in the descriptor <descriptor name> for an SQL statement that is to be executed dynamically. The <descriptor name> can be any variable of the type "sqldatype" or "struct SQLDA" (if sqlmode DB2 or ORACLE is set). If "into" is not specified, the variable SQLDA is taken by default. Within a program, the descriptor must always be of the same type, i.e., it must always be either a SAP DB structure, or ORACLE structure, or DB2 structure. An "SQLVAR" entry is generated in the SQLDA for each parameter specified in the SQL statement. These entries are created in the order of occurrence of the parameters.

A prerequisite is that a prepare was issued for the SQL statement to be executed dynamically. After a describe statement, the program variables must always be provided as parameters by means of the column information now available in the SQLDA (see Section 2.12, "Using the Descriptor").

6.6.3 exec sql [<n>] execute

```
exec sql [<n>] [<for clause>]
         execute <statement name> [<using clause>]

  <for clause>     ::= for <loop parameter>
  <loop parameter> ::= <unsigned integer constant>
                     | :<integer variable>
  <using clause>   ::= using <using expr>
  <using expr>     ::= <parameter list>
                     | descriptor <descriptor name>
  <parameter list> ::= <parameter, ...>
  <parameter>      ::= :<host variable>
                       [ :<indicator variable> ]
                        | <array variable>
                       [ :<indicator array variable> ]
  <n>              ::= number of the database session
                                 1 .. 8  , 1 = default
```

SQL statements with <array variable> are only allowed for array statements.

The SQL statement identified by <statement name> is executed. During execution, either all question marks are replaced one to one by the corresponding <parameter list> values or a descriptor is used. A prerequisite is that a prepare statement or, in the case of a descriptor, a describe statement is executed with the same <statement name> or with the same <descriptor name>. For the <descriptor name>, the same definitions apply as described for the describe statement.

6.6.4 exec sql [<n>] execute immediate

```
exec sql [<n>] execute immediate <statement source>

  <n>             ::= number of the database session
                              1 .. 8  , 1 = default
```

The SQL statement specified in a host variable or as a character string is executed. It must neither contain host variables nor be prepared by a prepare statement. If the SQL statement is specified as a character string, the macro mechanism can be applied.

6.7 SAP DB Cursor Statements

The cursor concept is an alternative to the named and unnamed result tables which can be generated with the select statement. It is supported for compatibility reasons; but its use is more complicated than that of the result tables.

A result table is specified by the statement declare cursor, and created and opened for processing by the open statement. A cursor determines a position within the result table from which a table row can be accessed by means of a fetch statement.

The result table exists until a corresponding close statement is executed or until the program is terminated.

The declare cursor statement is not written into the trace file, because it is not sent to the SAP DB kernel. This is only done for the open statement.

```
<cursor name>           ::= <identifier> |
                            <macro identification>
<macro identification>  ::= %nnn   with 1 <= nnn <= 128
<statement>             ::= <select statement>
                          | <statement name>
<parameter list>        ::= <parameter , ...>
<parameter>             ::= :<host variable> [
                            :<indicator variable> ]
```

6.7.1 exec sql [<n>] declare

```
exec sql [<n>] declare <cursor name> cursor for <statement>

  <n>            ::= number of the database session
                           1 .. 8  , 1 = default
```

This statement specifies a result table. The select statement is either qualified as a character string or prepared with the prepare statement and identified by the <statement name>. Parameters are subsequently assigned via an open statement in the using part.

6.7.2 exec sql [<n>] open

```
exec sql [<n>] open <cursor name>  [
                      |  into    <parameter list>
                      | using    <parameter list>
                      | using    descriptor  <descriptor name> ]

  <n>            ::= number of the database session
                           1 .. 8  , 1 = default
```

This statement generates the result table specified by the corresponding declare cursor statement and sets the cursor on the first row. If host variables are required, they must be specified in the using part.

The select statement identified by <cursor name> is executed. During execution, all question marks are either replaced one to one by the <parameter list> values or are associated with program variables via the descriptor. When the descriptor is used, the information needed about the program variables must have been assigned to the SQLDA (see Section 2.12, "Using the Descriptor").

6.7.3 exec sql [<n>] fetch

```
exec sql [<n>] fetch  [ <fetchspec> ] [ <cursor name>]
                        |  into   <parameter list>
                        |  into    descriptor  <descriptor name>
                        |  using   descriptor  <descriptor name>

  <n>            ::= number of the database session
                           1 .. 8  , 1 = default
  <fetchspec>    ::= first | last | next | prev | same
                     pos (<unsigned integer> | pos (:<host-var>)
```

This statement assigns the values of a result table row to the corresponding host variables and sets the cursor to the next row.

If a descriptor is specified, the values are assigned to the corresponding program variables which are described in the SQLDA. The indicator values are also assigned to the SQLDA (see Section 2.12, "Using the Descriptor").

6.7.4 exec sql [<n>] close

```
exec sql [<n>] close <cursor name>

  <n>            ::= number of the database session
                           1 .. 8  , 1 = default
```

This statement closes the specified result table.

6.8 Trace Statements

The following SQL statements provide a mechanism which supports partial testing of database applications.

Those parts of the application which are to be tested must begin with the SQL statements "set trace on" or "set trace long" and end with "set trace off". According to the control flow of the program, the SQL statements included in the part to be tested are stored in a file which can then be evaluated. All SQL statements in called translation units (external subroutines, modules, etc.) are also tested. Set trace statements can be specified at any places where, in accordance with the syntax rules, statements of the programming language are permitted.

Trace options which can be specified will suspend the trace statements for a translation unit. Under UNIX, the name of the file containing the test results receives the suffix ".pct". As an alternative, the filename can be specified after the trace option (<trace filename>).

6.8.1 exec sql set trace on

After this statement, all SQL statements completely generated are written into a trace file, together with the sqlcode (if not equal to zero) and any warnings returned.

6.8.2 exec sql set trace long

After this statement, all SQL statements completely generated are written into a trace file, together with the host variable values, the sqlcode (if not equal to zero) and any warnings returned.

6.8.3 exec sql set trace off

After this statement, writing of the SQL statements into a trace file is suppressed.

6.8.4 exec sql set trace line

The <trace line> is written into the trace file as a comment.

```
exec sql set trace line <trace line>

  <trace line>          ::= <parameter>
                          | <string constant>
  <parameter>           ::= :<prog identifier>
  <string constant>     ::= '<character seq>'
```

6.9 SAP DB Database Statements

These SQL statements help to establish a connection to a database.

6.9.1 exec sql connect

This statement can be used to establish a connection to a database. The user identification and the password can be specified either directly in the statement (see the Reference Manual: SAP DB (7.2)) or using the mechanisms described in the platform-specific User manual (XUSER, options). The database is defined using the statement "set serverdb" or XUSER or options.

6.9.2 exec sql [<n>] set serverdb

```
exec sql [<n>]  set serverdb <serverdb> [ on <servernode> ]

  <serverdb>           ::= <string constant> (maximum of 18 bytes)
                         | <variable name>
  <servernode>         ::= <string constant> (maximum of 64 bytes)
                         | <variable name>
  <string constant>    ::= '<character seq>'
  <variable name>      ::= :<prog identifier>
  <n>                  ::= number of the database session
                                    1 .. 8  , 1 = default
```

This statement can be used to define the name of a new database. It must be issued before a connect statement. The specified name has global validity for all modules of the DB application. Servernode can have up to 64 characters. Serverdb can have up to 18 characters. <prog identifier> is a character string of length 18.

If the specifications of serverdb name and servernode name are missing, they are taken from the nth entry of the XUSER file.

6.9.3 exec sql [<n>] reconnect

```
exec sql [<n>] reconnect

  <n>            ::= number of the database session
                             1 .. 8  , 1 = default
```

After a timeout or session end, this statement can be used to have a connect performed for this database session with the user specifications of the last connected user.

Before this SQL statement is executed, a connect with the above mentioned user specifications is performed.

6.9.4 exec sql cancel

```
exec sql cancel

  exec sql cancel <cancel session>;

  <cancel session>   ::= <n>
                       | <parameter>
                       | CURRENT
  <n>                ::= number of the database session
  <parameter>        ::= :<identifier>  must be an int variable
```

This statement serves the DBMS (kernel) to cancel the processing of an SQL statement. The SQL statement then has no effect.

6.10 Other Statements

6.10.1 exec command

This statement can be used to specify operating system commands from programs and to have these commands executed.

```
exec command  sync    <command> result <resultparameter>

exec command  async   <command>

  <command>          ::= <string const> | <parameter>
  <parameter>        ::=  :<prog identifier>
  <resultparameter>  ::=  :<prog identifier>
                          must store a binary number of 2 bytes
                          length.
```

This command statement executes an operating system command. The command syntax depends on the operating system.

In SYNC mode, the program waits until <command> execution has been terminated. In ASYNC mode, <command> is executed in background.

6.10.2 exec sql version

This statement can be used to request the versions of the general runtime environment and of the precompiler dynamic link library.

```
exec sql version <parameter>, <parameter>

  <parameter>        ::=  array variable   (up to 40 bytes long)
```

The first parameter returns the version of the general runtime environment.
The second parameter returns the version of the precompiler runtime.

6.11 exec sql [<n>] proc

```
exec sql [<n>] proc <db procedure>

  <db procedure>       ::= <db procedure name>
                           [ ( <parameterlist> ) ]
  <parameterlist>      ::= <parameter> , ...
  <parameter>          ::= :<host variable>
                           [ :<indicator variable> ]
  <n>                  ::= number of the database session
                                    1 .. 8  , 1 = default
```

This SQL statement can be used to call DB procedures which have been stored in the database by means of SQL-PL.

7 Appendix 1: Syntax of the Declare Section

```
statement   ::= prepcomline | declaration

prepcomline ::= '#define' identifier constant

constant    ::= identifier | unsigned_integer

declaration ::= decspec declist ';'

decspec     ::= storclspec | typespec | decspec storclspec |
                decspec typespec

storclspec  ::= 'external' | 'static' | 'auto' | 'typedef' | empty

typespec    ::= inttysp | flptysp | chartysp | vlftysp |
                dcmltysp | strutysp | identifier | 'const' |
                'volatile'

inttysp     ::= 'short' | 'long' | 'int' | 'unsigned'

chartysp    ::= 'char' | 'VARCHAR' | 'varchar'

vlftysp     ::= 'LONGFILE' | 'longfile'

flptysp     ::= 'float' | 'long' | 'double'

dcmltysp    ::= dcmldef | dcmlref

strutysp    ::= strudef | struref

strudef     ::= 'struct' structag '{' fieldlist '}'

struref     ::= 'struct' identifier

dcmldef     ::= 'DECIMAL' dcmltag '{' dcmlscale '}'

dcmlref     ::= 'DECIMAL' identifier

structag    ::= identifier | empty

fieldlist   ::= typespec declist ';' |
                fieldlist typespec declist ';'

dcmltag     ::= identifier | empty

dcmlscale   ::= dcmldigits | dcmldigits ',' dcmlfract | empty
```

```
declist     ::= possinitdec |  declist ',' possinitdec

dcmldigits  ::= identifier | unsigned_integer

dcmlfract   ::= identifier | unsigned_integer

possinitdec ::= declarator | declarator '=' initializer

declarator  ::= identifier | '*' identifier | declarator
                '[' arrdim ']'

arrdim      ::= identifier | unsigned_integer

initializer ::= expression | '{' initlist '}'

initlist    ::= initializer |  initlist ',' initializer

expression  ::= basexpr | basexpr ( expression ) basexpr

empty       ::=
```

8 Appendix 2

8.1 Internal Return Codes of the Precompiler

0 : Precompilation and compilation free of errors

1 : Precompiler error

2 : Precompilation free of errors, but compiler error

3-120 : Number of errors returned by the precompiler

126 : Errors in precompiler file management

127 : The precompiler was called, not the compiler.

8.2 Return Codes of a SAP DB Application

These return codes are generated by an application which was incorrect or which was terminated with "whenever stop".

0 : No error

1 : XUSER file read error or character set load error.

2 : Database has not been started.

3 : Too many users on a database.

4 : Unknown user-password-combination.

5 : Whenever stop.

6 : Option error.

7 : Tracefile error.

8 : Incorrect values assigned to SQLXA (system error).

8.3 Return Codes of the CPC Call

0 : Precompiler and compiler run free of errors.

and 0 : Precompiler ok; when option set, only precompiling.

1 : Precompiler errors.

2 : Precompiler ok, compiler errors.

9 Index

- A -

SAP DB optimizer **3-23**; 3-41; 3-45
ANSI **4-1**
ansi C option **5-5**
ANSI ORACLE **4-4**
array statement 2-25; **6-8**; 6-14
as clause 2-20; **6-3**

- B -

begin declare section **6-4**

- C -

C++ option **5-5**
call **2-31**; 6-5
cancel **6-22**
check option 2-4; 2-18; **5-5**; 5-18
close 3-2; **6-18**
colfrac **2-40**; 2-42
colinfo **2-41**
colio **2-39**; 2-42
collength **2-40**; 2-42
colmode **2-39**; 2-42
colname **2-39**; 2-42
coltype **2-39**; 2-42
column name generation **2-22**
comment option **5-5**
compatible option 2-16; **5-6**; 6-9
connect **6-21**
continue **2-31**; 2-32; 6-5

costs of a search strategy **3-33**; 3-34
cursor 2-36

- D -

data types, predefined 2-12
database session **2-4**
database session number **2-4**; 6-8; 6-12; 6-13; 6-14; 6-15; 6-17; 6-18; 6-21; 6-22; 6-25
date **2-9**
date-time option 2-9; **5-6**
DB procedures 2-1; **6-25**
DB2 ..... 2-33; **4-2**; 6-13
dbproc clause **6-2**
DECIMAL 2-12
declare 3-1; 3-2; **6-17**
declare section 2-2; **2-6**; 6-4; 7-1
declare section, syntax **7-1**
describe 2-37; 2-43; 2-44; 2-47; 6-11; **6-13**
descriptor **2-37**; 2-43; 2-47; 2-51; 6-11; 6-14; 6-17; 6-18
dynamic SQL statements 2-33; 2-35; 2-37; 2-43; 5-5; 6-11

- E -

end declare section **6-4**
exec sapdb 4-1; 4-2; 4-3
exec command 2-1; **6-23**
exec sql n 2-4
exec sql proc **6-25**
exec sql version **6-24**
execute 2-35; 2-37; 2-43; 2-44; 2-52; **6-14**
execute immediate 2-33; **6-15**
EXPLAIN statement 3-23; 3-35; **3-36**; 3-44
extern option **5-7**

- F -

fetch 2-39; 2-45; 2-47; 3-2; **6-18**

- G -

go to **2-31**
goto 6-5

- H -

help option **5-7**
host variable 2-1; **2-2**; 2-6; **2-7**; 2-16; 2-20; 2-35; 5-14; 5-16; 6-4
host variable generation 2-20; 6-2
hostcolsize **2-40**; 2-42
hostinaddr **2-41**; 2-42
hostindicator **2-40**; 2-42
hostvaraddr **2-41**; 2-42
hostvartype **2-40**; 2-42

- I -

identifier **2-7**
include <filename> 2-2; 6-2
include sqlca 6-1
include statement 2-20; **6-1**
ind clause 2-20; **6-3**
index 3-25; **3-29**; 3-45
index list **3-28**
indicator 2-3; 2-6; 2-16; **2-17**; 5-14; 6-18
indicator array 2-16; 2-41; 6-9
indicator value **2-17**; 2-40
indicator variable **2-17**
instructions for the locking 3-19
isolation level **3-15**; 3-17; 3-20

isolation level option **5-7**; **5-14**

- J -

join 3-26; 3-35; **3-41**; 3-42

- L -

list option **5-8**
lock_timeout **3-22**
long column 2-15
LONGFILE 2-15

- M -

macro mechanism 2-3; **2-56**; 6-15; 6-16
macro statement **6-10**; 6-16
margins option **5-8**
max packetsize, option **5-8**
mfetch option **5-14**
multi-db mode **2-4**; 5-12

- N -

no select direct fast option **5-14**
not found 2-31; **6-6**
nowarn option **5-8**

- O -

open 2-55; 3-2; **6-17**
ORACLE 2-33; **4-3**; 6-13
orderd select statements **3-45**

- P -

parameter.....**2-3**; 2-16; 2-37; 2-38; 2-41; 2-43; 2-44; 2-55; 6-11; 6-13
precom option **5-8**
precompiler debugging aids **5-18**
precompiler listing **5-13**
precompiler options.....**5-5**
precompiler output files 5-13
prepare.....2-35; 2-43; 2-44; 2-47; 6-11; **6-12**
proc 2-1
profile option.....**5-8**; **5-15**; 5-23
program option.....**5-8**
program variable 2-37; 2-40; 2-43
programming instructions **3-1**

- R -

reconnect **6-22**
request_timeout.....**3-21**
return code 2-24; 4-1; 4-2; 4-3
return codes/SAP DB application 8-2
return codes/CPC call.....8-3
return codes/precompiler.....8-1
runtime optimization **3-46**
runtime options **5-14**
runtime trace **5-19**

- S -

search at FETCH point..... 3-25; **3-35**
search condition 2-52; 3-25; **3-26**; 3-27; 3-28; 3-34; 3-36; 3-45; 3-47
search strategy..... 3-25; 3-26; **3-28**; 3-34
select ordered statements.....3-1
serverdb option..... **5-9**; **5-15**
servernode option..... **5-9**; **5-15**
session_timeout.....**3-22**

sessionno **2-4**; 6-8; 6-12; 6-13; 6-14; 6-15; 6-17; 6-18; 6-21; 6-22; 6-25
set macro 2-56; **6-10**
set serverdb 2-4; **6-21**
set servernode 2-4
set trace line **6-20**
set trace long **6-19**
set trace off **6-20**
set trace on **6-19**
silent option **5-9**
single row statements 3-1
SQL statement **6-8**
SQLCA (SQL Communication Area) 2-2; **2-24**; 6-1
sqlcabc **2-24**; 2-29
sqlcaid **2-24**; 2-29
sqlcode 2-2; 2-17; **2-24**; 2-29; 2-30; 4-1; 4-2; 4-3; 6-6; 6-9; 6-19
sqld **2-38**; 2-42
SQLDA (SAP DB) **2-38**
SQLDA (ANSI) 4-1
SQLDA (DB2) 4-2
SQLDA (ORACLE) 4-3
SQLDA (SQL Descriptor Area) 2-37; 6-11; 6-13
sqldaid **2-38**; 2-42
sqldatetime **2-28**; 2-29
sqlerrd **2-25**; 2-29; 3-2; 5-11; 5-16; 5-20; 6-9
sqlerrmc **2-25**; 2-29
sqlerrml **2-25**; 2-29
sqlerror 2-17; 2-30; 2-32; 5-13
sqlexception 2-25; 2-31; 3-21; **6-6**
sqlmax **2-38**; 2-42
sqlmfetch 2-29
sqlmode option 2-33; 4-1; 4-2; 4-3; 4-4; 6-11
sqlmode option (INTERNAL) **5-9**
sqlmode option (ANSI) **5-9**
sqlmode option (ANSIORA) **5-9**
sqlmode option (DB2) **5-10**
sqlmode option (ORACLE) **5-10**
sqln **2-38**; 2-42

sqlresultn **2-28**; 2-29
sqlvar **2-38**; 2-42; 6-13
sqlwarn0 ..... **2-26**; 2-29; 2-30; 2-32; 6-5
sqlwarn1 ..... 2-11; **2-26**; 2-29
sqlwarn2 ..... **2-26**; 2-29
sqlwarn3 ..... **2-26**; 2-29; 3-7
sqlwarn4 ..... **2-26**; 2-29
sqlwarn6 ..... **2-26**; 2-29
sqlwarn8 ..... **2-27**; 2-29
sqlwarna **2-27**; 2-29
sqlwarnb **2-27**; 2-29
sqlwarnc **2-27**; 2-29
sqlwarnd **2-27**; 2-29
sqlwarne **2-27**; 2-29
sqlwarnf 2-26; **2-28**; 2-29; 4-1; 4-2; 4-3
sqlwarning 2-30
stop **2-31**; 6-5; 8-2
syntax option **5-5**; 5-18
system tables **5-2**; 5-5; 5-8; 5-15; 5-23

- T -

table clause **6-2**
time **2-9**
timeout option **5-10**; **5-15**
timeouts 3-21
timestamp 2-9
tool components 2-25
trace 5-19
trace alt option **5-15**
trace file **5-20**
trace file option **5-10**; **5-15**
trace file, option **5-10**; **5-16**
trace long option **5-11**; 6-19
trace no date/time option **5-16**
trace option **6-19**

trace short option **5-11**; **5-16**
trace time option **5-16**

- U -

user option **5-11**; **5-16**; 5-18
user specifications 3-8; **5-12**; 5-17
userkey option **5-11**; **5-16**

- V -

VARCHAR 2-13
version option **5-11**

- W -

whenever action 2-31; **6-5**
whenever condition 2-30; 3-21
whenever sqlbegin **6-7**
whenever sqlend **6-7**
whenever sqlerror **6-6**
whenever sqlwarning **6-5**
whenever statement 2-30; 2-31; **6-5**

- X -

XUSER 6-21
XUSER file **2-5**; 5-11; 5-16; 6-21