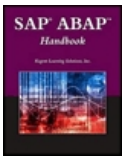


SAP ABAP Handbook

by Kogent Learning Solutions, Inc.

Chapter 8: Accessing Data in the SAP System

Overview

ABAP programs need to access various types of databases to ensure data persistency and implement the business logic of SAP applications. Data persistency means storing data at permanent places so that we can access it as and when required. The SAP system is equipped with a central relational database system (Database Dictionary), which is used to implement the concept of data persistency. A relational database is accessed by using Structured Query Language (SQL), which is a standardized language used to work with databases. A SAP system uses Open SQL statements, a subset of standard SQL, to access and maintain its central relational database. The Open SQL statements consist of a set of ABAP statements used to perform various operations, such as reading, inserting, deleting, and updating data in a database. Using Open SQL statements ensures that the SQL statements in ABAP programs can interoperate between various types of databases. Besides Open SQL statements, which include ABAP code elements, another category of standard SQL statements, Native SQL statements, contain only database manipulation statements. Database tables that are not administered by the ABAP Dictionary can be accessed by using Native SQL.

In this chapter, you learn how database tables are accessed by ABAP programs in an SAP system. You also learn how data is read from database tables by using subqueries and the `SELECT` statement and its various clauses, such as `INTO`, `WHERE`, and `GROUP BY`. Next, you learn how database operations, such as insert, update, and delete, are performed by using the `INSERT`, `UPDATE`, and `DELETE` statements, respectively. You also learn how cursors are used to read data from database tables. Finally, you learn how the `COMMIT WORK` and `ROLLBACK WORK` statements are used to save or revert the final changes made in a database, respectively.

Accessing Database Tables

In a relational database model, data is represented in the form of tables. We know that a table is a two-dimensional matrix consisting of rows or records (also known as lines in SAP) and columns (known as fields in SAP). Programmers generally use standard SQL, which is compatible with all databases, to access database tables. However, standard SQL statements do not ensure interoperability between different types of databases. As a result, SAP uses Open SQL statements, which are interoperable and compatible with all types of databases.

Each work process in the SAP R/3 system has a common database interface. The database interface converts all the database requests into the respective standard SQL statements to interact with the database tables stored in the ABAP Dictionary. Figure 8.1 shows a representation of how a database interface interacts with the components of an SAP R/3 system:

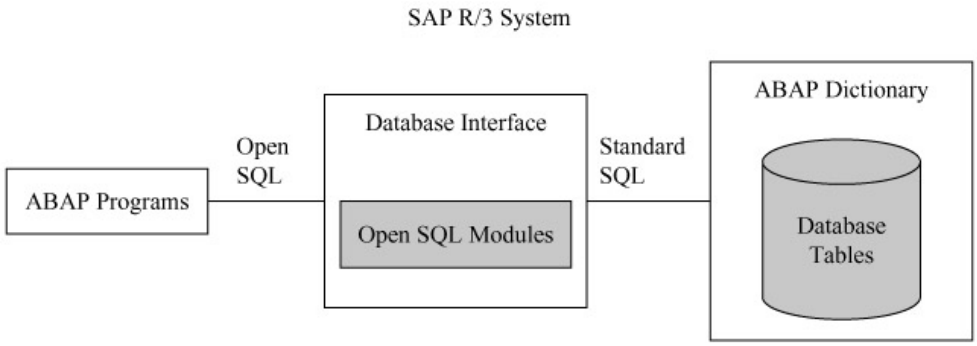


Figure 8.1: Database interface in the SAP R/3 system

As learned previously, in the SAP R/3 system, the ABAP programs communicate with the database by using the database interface. The ABAP Dictionary, which stores the database tables, uses the Open SQL statements to create and change database tables. The ABAP Dictionary allows users to create and administer database tables. It contains metadescriptions of all database tables in the SAP R/3 system.

In an SAP R/3 system, data from a database can be accessed by an ABAP program by using the following types of statements:

- Open SQL

- Native SQL

Now, let's discuss these statements in detail.

Open SQL

Open SQL, a subset of Standard SQL, consists of a set of ABAP statements that perform operations on the databases in a SAP R/3 system. It provides a uniform syntax and semantics for all the database systems supported by SAP. Consequently, ABAP programs that use Open SQL statements can work in any SAP R/3 system, regardless of the database system being used. Open SQL statements can work with database tables that have been created in the ABAP Dictionary. When an ABAP program using Open SQL statements is executed, the SAP Basis component of the SAP R/3 system converts the Open SQL statements into Native SQL statements to access the database.

Note To learn more about the SAP Basis component, refer to Chapter 1.

The Open SQL statements consist of the Data Manipulation Language (DML), which is a part of Standard SQL. In other words, Open SQL statements allow you to read (i.e., to `SELECT`) and change (i.e., to `INSERT`, `UPDATE`, or `DELETE`) data.

In the ABAP Dictionary, you can combine columns of different database tables with a database view. In Open SQL statements, views are manipulated in the same way as database tables. Therefore, any references to database tables in the following sections apply to views as well.

Table 8.1 lists the commonly used Open SQL statements:

Table 8.1: Commonly used open SQL statements

Statement	Description
`SELECT`	Reads data from database tables
`INSERT`	Adds lines to database tables
`UPDATE`	Changes the contents of lines of database tables
`MODIFY`	Inserts new lines into database tables or changes the content of existing lines
`DELETE`	Deletes lines from database tables
`OPEN CURSOR`	Reads lines of database tables using the cursor
`FETCH`	
`CLOSE CURSOR`	

Whenever a user logs in and accesses an SAP database, the user must specify a client. If a database operation is initiated by the user after logon, the SAP system generates a return code specifying whether the operation has been successful.

Now, let's explore automatic client-handling and return code in an SAP system.

Automatic Client-Handling

A single SAP R/3 system can manage the application data of different business processes, such as creating sales and delivery orders in a company. Each of these commercially separate business processes in the SAP R/3 system is called a client, which is assigned with a unique number for identification. When a user logs on to an SAP R/3 system, the user has to specify a client. As a result, every database table in SAP is structured to include the clients of an SAP system. Consequently, when you create a database table in SAP, the first column, `MANDT`, is created automatically to store client-related data.

By default, Open SQL statements use the automatic client-handling feature, which always accesses the current client. The `MANDT` column cannot be manipulated by using the `WHERE` clause of an Open SQL statement if the automatic client-handling feature is turned on. If you try to manipulate the `MANDT` column, the SAP system returns an error either during the syntax check or at runtime. You can, however, manipulate the `MANDT` field after disabling the automatic client-handling feature. The automatic client-handling feature can be disabled by using the `CLIENT SPECIFIED` clause.

Return Code Used in Open SQL

As stated earlier, return code specifies the status of executing a database operation. The status is specified by the following two system variables:

- SY-SUBRC—Specifies whether or not the database operation has been successful. If the SY-SUBRC system variable returns 0, it specifies that the database operation has been successful. Consequently, any value other than 0 signifies an unsuccessful operation.
- SY-DBCNT—Specifies the number of database lines processed after an Open SQL statement has been executed.

Native SQL

Unlike Open SQL, which includes ABAP code elements, Native SQL contains only database manipulation statements. Database tables that are not administered by the ABAP Dictionary can be accessed by using Native SQL. A Native SQL statement is used within the EXEC SQL and ENDEXEC statements, as shown in the following syntax:

```
EXEC SQL [PERFORMING <form>].
<Native SQL statement>
ENDEXEC.
```

In this syntax, note that we have not used a period (.) after the Native SQL statement. Moreover, unlike in ABAP syntax, inverted commas (") and asterisks (*) in Native SQL statements do not convert a line of code into a comment. In addition, you must know whether the table and field names are case-sensitive in the specified database.

The data stored in log columns with types LCHR and LRAW and declared in the ABAP Dictionary can be manipulated by using Open SQL statements. If you use Native SQL statements to access log columns with types LCHR and LRAW, the results might be incorrect. In addition, Native SQL does not support the automatic-client handling feature, unlike Open SQL.

You can use Native SQL statements to perform the following tasks:

- Transfer the values of ABAP fields to database tables
- Read or retrieve the values from database tables and process them in ABAP programs
- Access the tables that are not declared in the ABAP Dictionary

Some disadvantages of Native SQL are as follows:

- The code written between the EXEC and ENDEXEC statements is not checked for syntax errors.
- An ABAP program that contains the database-specific SQL statements is not compatible with different database systems.
- The automatic client-handling feature for client-dependent tables is not supported by Native SQL.

Note An ABAP program contains database-specific SQL statements that do not run under different database systems. If you need to use your ABAP program on more than one database platform, you must use only Open SQL statements.

Reading Data Using the *SELECT* Statement

In Open SQL, the SELECT statement and its various clauses are used to read data from database tables. The syntax to use the SELECT statement to read the data from database tables is:

```
SELECT       <selected_result>
INTO         <target_area>
FROM         <source_database_tab>
[WHERE       <where_condition>]
[GROUP BY    <fields>]
[HAVING      <having_condition>]
[ORDER BY    <fields>].
```

In this syntax, we have used various clauses, such as WHERE, GROUP BY, and ORDER BY, in the SELECT statement.

Table 8.2 describes the various clauses of the SELECT statement:

Table 8.2: Clauses used within the SELECT statement

Clause	Description

SELECT <selected_result>	Defines the structure of data that you want to read from database tables. The structure of the data can be read by specifying the number of lines, column names, and whether duplicate entries are acceptable.
INTO <target_area>	Specifies the target area, where the data has to be sent after being read from the source database table.
FROM <source_database_tab>	Specifies the database table or view from which data is to be read. The FROM clause can also be placed before the INTO clause in a SELECT statement.
WHERE <where_condition>	Specifies the condition based on which data is read from database tables.
GROUP BY <fields>	Groups a selected set of lines into a set of summarized lines, by using the values of one or more columns listed in the <fields> parameter.
HAVING <having_condition>	Sets logical conditions for the lines to be combined by using the GROUP BY clause.
ORDER BY <fields>	Defines a sequence of <fields> parameters for the lines of the result set generated after executing the SELECT statement.

Now, let's discuss all these clauses in detail.

The *SELECT* Clause

In Open SQL, the SELECT clause is used to define the structure of the result set that the user wants to read from a database. The SELECT clause can be divided into two parts, lines and columns, as shown in the following syntax:

```
SELECT <database_tab_lines> <database_tab_columns> ...
```

In this syntax, the <database_tab_lines> expression specifies whether the user wants to read one or more lines and the <database_tab_columns> expression defines the column selection.

You can use the SELECT statement to read any of the following from a database table:

- A single line
- Multiple lines
- A single column
- All columns
- Aggregate data
- Dynamic data

Now, let's explore these in detail, one by one.

Reading a Single Line

The SELECT statement can read a single entry from the database table by using the following syntax:

```
SELECT SINGLE <database_tab_columns> ... WHERE ...
```

In this syntax, the SINGLE clause is used to read a single entry from a database table, and the WHERE clause is used to specify a condition for the selection. For example, you can specify a condition in the WHERE clause by writing the values for the primary key fields of a table. If the WHERE clause does not contain all the key fields of the table, a warning is generated and the SELECT statement reads the first entry according to the key fields that are specified. The result of such a query is either an elementary field or a flat structure, depending on the number of columns specified in the <database_tab_columns> expression. If the system finds a line with the corresponding key, the value of the SY-SUBRC system variable is set to 0; otherwise, it is set to 4.

Listing 8.1 shows how to read a single line:

Listing 8.1: Reading a single line from a database table

```
REPORT ZDATA_ACCESS.
```

```
*/Reading the data using SELECT statement

DATA LINE TYPE KNA1.
SELECT SINGLE KUNNR LAND1 NAME1 ORT01 ADRNR
INTO CORRESPONDING FIELDS OF LINE
FROM KNA1
WHERE KUNNR EQ '0000000515'.
IF SY-SUBRC EQ 0.
   WRITE: / LINE-KUNNR, LINE-LAND1, LINE-NAME1, LINE-ORT01,
   LINE-ADRNR.
ENDIF.
```

In Listing 8.1, the `SINGLE` clause is used in the `SELECT` statement, which reads a single entry from the `KNA1` database table, where the value of the `KUNNR` field is equal to 0000000515. The columns specified in the `SELECT` clause are transferred to the corresponding components of the structure `LINE`.

Note All the source code in this chapter uses some standard tables of the SAP R/3 system. Table 8.3 lists these standard tables:

Table 8.3: Some standard tables in the SAP R/3 system

Table Name	Description
MAKT	Material descriptions
MAKV	Material cost distribution
MAKZ	Material cost distribution equivalence numbers
MNA1	General data in customer master
MAKT	Short document: material movement
MARA	General material data

Perform the following steps to create an ABAP program to read a single line from a database (by using the code of Listing 8.1):

1. Open the initial screen of ABAP Editor either by navigating through the SAP menu or by executing the `SE38` transaction code.
2. On the initial screen of ABAP Editor, enter a name for the program, such as, `ZDATA_ACCESS`. Select the `Source code` radio button in the `Subobjects` group box and then click the `Create` button to create a new program, as shown in Figure 8.2:

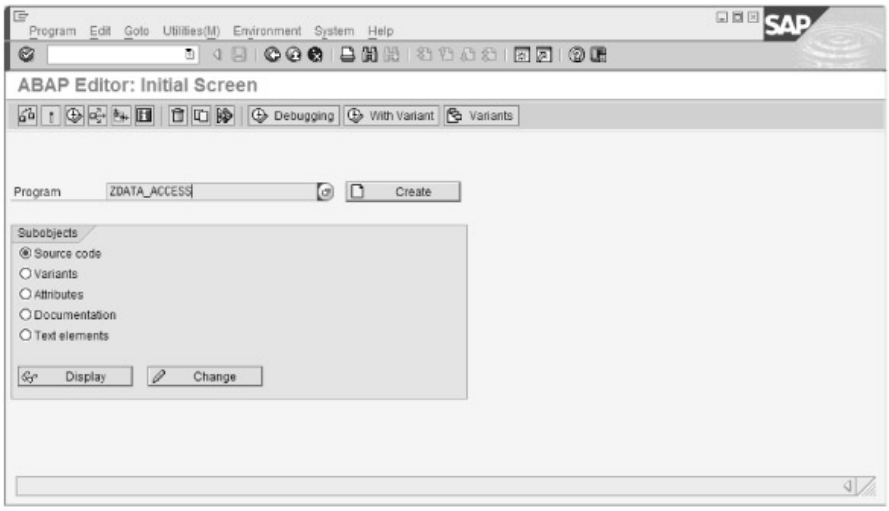


Figure 8.2: Creating a new program

The `ABAP: Program Attributes` dialog box appear.

3. In the ABAP: Program Attributes dialog box, enter a short description for the program in the Title field. Select Executable program as the attributes type, as shown in Figure 8.3:

4. Click the Save button (see Figure 8.3) or press the ENTER key. The Create Object Directory Entry dialog box appears.

5. In the Create Object Directory Entry dialog box, enter the package name, ZKOG_PCKG, beside the Package field, and then click the Save icon.

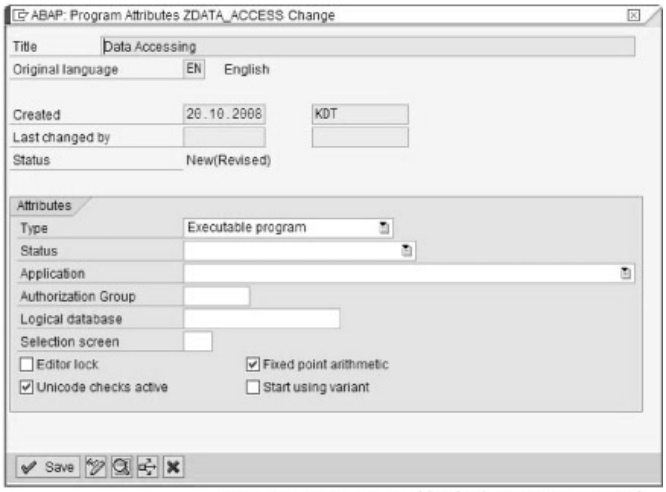


Figure 8.3: The ABAP—program attributes screen

The ABAP Editor: Change Report screen appears, as shown in Figure 8.4:

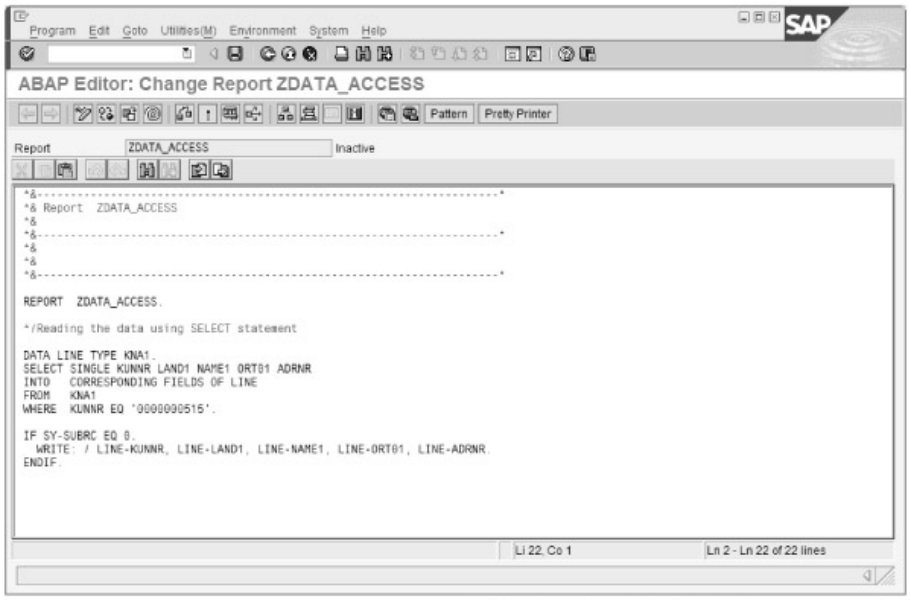


Figure 8.4: Code added to the ABAP editor—change report screen

6. Write the code, as given in Listing 8.1, in the ABAP Editor: Change Report screen (see Figure 8.4).

7. Click the Save icon or press the CTRL+S key combination to save the current changes in the program.

8. Click the Check icon or press the CTRL+F2 key combination to check and remove the syntax error or warning

(if any) in the program. After removing the syntax errors from the program, click the `Activate` icon or press the CTRL+F3 key combination to activate the program.

9. Click the `Direct Processing` icon or press the F8 key to display the output of the program. Figure 8.5 shows the output:

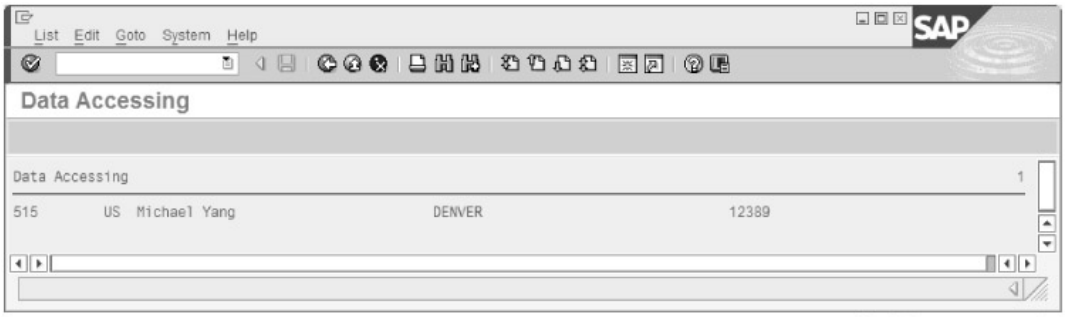


Figure 8.5: Reading a single entry

Note Follow all the steps that have been previously described to generate and view the results of all the programs given in this chapter.

Reading Several Lines

The `SELECT` statement is used to read several entries of a database table by using the following syntax:

```
SELECT [DISTINCT] <database_tab_columns> INTO <target_area>
... WHERE ...
```

In this syntax, the `DISTINCT` clause is used to ensure that no duplicate entries are read. If the `DISTINCT` clause is not used, the SAP system reads all the lines that satisfy the `WHERE` condition. The target area of the `INTO` clause can be an internal table with a line type compatible with the `<database_tab_columns>` expression. If the target area is a flat structure, the `ENDSELECT` statement must be included at the end of the `SELECT` statement. The following syntax shows how to use the `ENDSELECT` statement along with the `SELECT` statement:

```
SELECT [DISTINCT] <database_tab_columns> INTO <target_area>
... WHERE ...
    ...
ENDSELECT.
```

The `SELECT...ENDSELECT` statement, shown in the preceding code snippet, searches the records based on the condition specified in the `WHERE` clause. Each of these records is copied into a structure variable and then processed in the SAP system. The following code snippet shows how to use the `SELECT... ENDSELECT` statement:

```
SELECT * FROM KNA1 INTO cust_nm WHERE NAME1 = 'Roger Zahn'.
 WRITE cust_nm-ORT01.
ENDSELECT.
```

Reading Particular Columns

The following syntax of the `SELECT` statement is used to read single columns from a database table:

```
SELECT <database_tab_lines> <dbtab_col1> [AS <dbtab_col_
alias1>] <dbtab_col2> [AS
<dbtab_col_alias1>] ...
```

In this syntax, `<dbtab_col 1>`, `<dbtab_col 2>...<dbtab_col n>` stands for column names and the `AS` clause is used to assign the alias names `<dbtab_col_alias1>`, `<dbtab_col_alias2>...<dbtab_col_alias n>` corresponding to these columns. An alias name can be used in the following situations:

- When one database table is referred to in the `FROM` clause more than once. In this situation, using the alias name ensures that each column referred to in the `FROM` clause is considered unique.
- When the full names of columns are specified in the `SELECT` statement. The full name of a column is required when a

column specified in the SELECT statement exists in two or more database tables.

The AS clause is used to specify an alias name, <dbtab_col_alias i>, for each column, <dbtab_col i>. The specified alias is used instead of the real name of the column in the INTO and ORDER BY clauses. Listing 8.2 shows how to read the data of the columns specified in the SELECT statement:

Listing 8.2: Reading data of the columns by using the SELECT statement

```
REPORT ZDATA_ACCESS.
*/Reading the data of a particular column of a table, containing
multiple lines
DATA: MyTable TYPE STANDARD TABLE OF KNA1,
   KOG LIKE LINE OF MyTable.
SELECT KUNNR LAND1 NAME1 ORT01 ADRNR
INTO CORRESPONDING FIELDS OF TABLE MyTable
FROM KNA1
WHERE LAND1 NE 'US'.
IF SY-SUBRC EQ 0.
   LOOP AT MyTable INTO KOG.
            WRITE:   / KOG-KUNNR, KOG-LAND1, KOG-NAME1,
            KOG-ORT01, KOG-ADRNR.
   ENDLOOP.
ENDIF.
```

In Listing 8.2, MyTable is an internal table that contains the KOG work area and the same column names as the column names of the KNA1 database table. The SELECT statement reads all the lines of the KNA1 database table that satisfy the condition specified in the WHERE clause. The columns specified in the SELECT statement are then transferred to the corresponding columns of the KOG work area. Figure 8.6 displays the values of the columns stored in the KOG work area of the MyTable table:

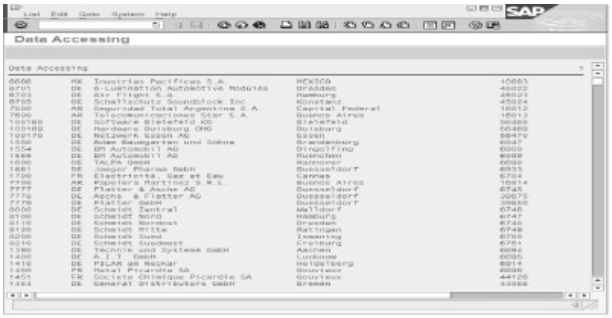


Figure 8.6: Reading particular fields from a table

Reading All Columns

The syntax of the SELECT statement to read all the columns of a database table is:

```
SELECT <database_tab_lines> * ...
```

In this syntax, the asterisk (*) symbol is used to specify all the columns of a database table.

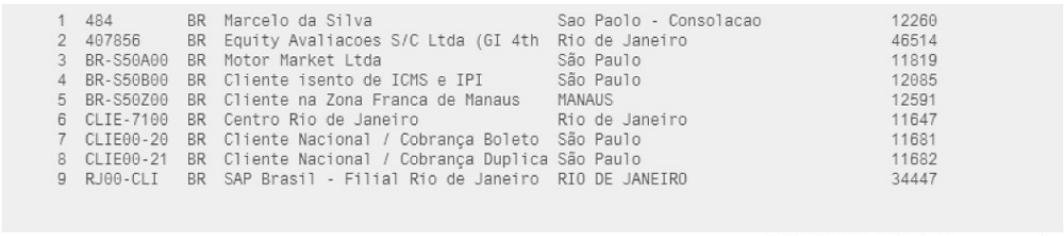
Listing 8.3 shows how to read the data from all the columns of a database table:

Listing 8.3: Reading data from all the columns of a database table

```
REPORT ZDATA_ACCESS.
*/Reading all the columns of a table
DATA KOG TYPE KNA1.
SELECT *
INTO CORRESPONDING FIELDS OF KOG
FROM KNA1
WHERE LAND1 EQ 'BR'.
  WRITE: / SY-DBCNT,
```

```
KOG-KUNNR, KOG-LAND1, KOG-NAME1, KOG-ORT01, KOG-ADRNR.
ENDSELECT.
```

In Listing 8.3, the SELECT statement reads all the lines of the KNA1 database table, where the value of the LAND1 field is BR. All the columns of the KNA1 table are then transferred to the corresponding columns of the KOG work area. Figure 8.7 shows the output of Listing 8.3:



```
1  484        BR  Marcelo da Silva                  Sao Paolo - Consolacao            12260
2  407856     BR  Equity Avaliacoes S/C Ltda (GI 4th  Rio de Janeiro                  46514
3  BR-S50A00  BR  Motor Market Ltda                 São Paulo                         11819
4  BR-S50B00  BR  Cliente isento de ICMS e IPI      São Paulo                         12085
5  BR-S50Z00  BR  Cliente na Zona Franca de Manaus  MANAUS                           12591
6  CLIE-7100  BR  Centro Rio de Janeiro             Rio de Janeiro                    11647
7  CLIE00-20  BR  Cliente Nacional / Cobrança Boleto  São Paulo                      11681
8  CLIE00-21  BR  Cliente Nacional / Cobrança Duplica São Paulo                      11682
9  RJ00-CLI   BR  SAP Brasil - Filial Rio de Janeiro  RIO DE JANEIRO                 34447
```


Figure 8.7: Reading the data from the columns of a table

Note As a best practice, specify only the required columns in the SELECT statement to reduce the time that data takes to process.

Reading Aggregate Data for Columns

The syntax to use the SELECT statement to read aggregate data of a column from a database table is:

```
SELECT <database_tab_lines> <agg>([DISTINCT] <s1>)
[AS <a 1>]
<agg>([DISTINCT] <s2>) [AS <a 2>] ...
```

In this syntax, the <s1> <s2>...<s n > expressions stand for field labels. The <agg> expression represents one of the following aggregate functions:

- **MAX**—Returns the maximum value of the <s i> columns.
- **MIN**—Returns the minimum value of the <s i> columns.
- **AVG**—Returns the average value of the <s i> columns.
- **SUM**—Returns the sum of the values of the <s i> columns.
- **COUNT**—Counts the total lines of the database table.
- **COUNT (**DISTINCT <s i>**)**—Returns the number of different values in the <s i> columns.
- **COUNT(***)—Returns the total number of lines in the result set of the SELECT statement.

The DISTINCT clause ensures that no duplicate values are used to calculate the aggregate of the data. The aggregate functions AVG and SUM are used only with numeric fields. If you use the MAX, MIN, and SUM functions, ensure that the data type of the corresponding columns must be of the ABAP Dictionary type, such as I and F. When the aggregate function is AVG, the ABAP Dictionary type must be FLTP, and when the aggregate function is COUNT, the ABAP Dictionary type must be INT4.

It is to be noted that null values are not used for calculation. If all the lines in a selection contain the null value in the corresponding fields, the result of the calculation also turns out to be null. In ABAP, the null value is interpreted as zero (depending on the data type of the field).

The AS clause is used to define an alternative column name (also called an alias), <a i>, for each aggregate expression. An alias column name is preferred over the actual column name because it helps simplify the query. The field names specified in the SELECT clause for aggregate functions are also included in the GROUP BY clause.

Now, let's consider an example. Assume that the ITEM database table contains two columns and 10 lines, as shown in Table 8.4:

Table 8.4: The ITEM table

COL_1	COL_2
1	3
2	1
3	5
4	7
5	2
6	3
7	1
8	9
9	4
10	3

The following code snippet is used to apply an aggregate function on the data stored in Table 8.4:

```
DATA RESULT TYPE P DECIMALS 2.
SELECT <agg>( [DISTINCT] COL_2 )
         INTO RESULT
         FROM ITEM.
WRITE RESULT.
```

This code snippet generates the result based on different combinations that the aggregate expression `<agg>` forms with the `DISTINCT` clause, as shown in Table 8.5:

Table 8.5: Result of aggregate expressions and the DISTINCT clause

Aggregate Expression	DISTINCT	Result
MAX	No	9.00
MAX	Yes	9.00
MINt	No	1.00
MIN	Yes	1.00
AVG	No	3.80
AVG	Yes	4.43
SUM	No	38.00
SUM	Yes	31.00
COUNT	Yes	7.00
COUNT(*)	---	10.00

Note The `DISTINCT` clause does not work with the `COUNT(*)` aggregate expression.

Specifying Columns Dynamically

You can also use the `SELECT` statement to specify the columns of an internal table dynamically. The internal table can either be empty or contain columns names or aggregate expressions. If the internal table is empty, the SAP system reads all columns. The following syntax is used to specify the columns dynamically:

```
SELECT <database_tab_lines> (<internal_tab>) ...
```

This syntax shows that the name of the internal table, `<internal_tab>`, is written inside the parentheses (). Note that the line type of the `<internal_ tab>` table must be the `C` type (storing character data), with a maximum length of 72 characters.

Listing 8.4 shows how to read columns that are specified dynamically:

Listing 8.4: Specifying and reading columns dynamically

```
REPORT ZDATA_ACCESS.
*/Specify and Reading the columns dynamically
DATA: MyTable TYPE STANDARD TABLE OF KNA1,

   KOG LIKE LINE OF MyTable.
DATA: LINE(200) TYPE C,
      LIST LIKE TABLE OF LINE(200).
LINE = ' Land1 ORT01 '.
APPEND LINE TO LIST.
SELECT DISTINCT (LIST)
       INTO CORRESPONDING FIELDS OF TABLE MyTable
       FROM KNA1.
LOOP AT MyTable INTO KOG.
 WRITE: / KOG-Land1, KOG-ORT01.
ENDLOOP.
```

In Listing 8.4, MyTable is an internal table that retrieves data from the columns of the KNA1 database table. The `DISTINCT` clause in the `SELECT` statement is used to remove the lines with the same content in both the columns. Figure 8.8 shows the values of the `Land1` and `ORT01` fields of the KNA1 table after removing the duplicate entries:

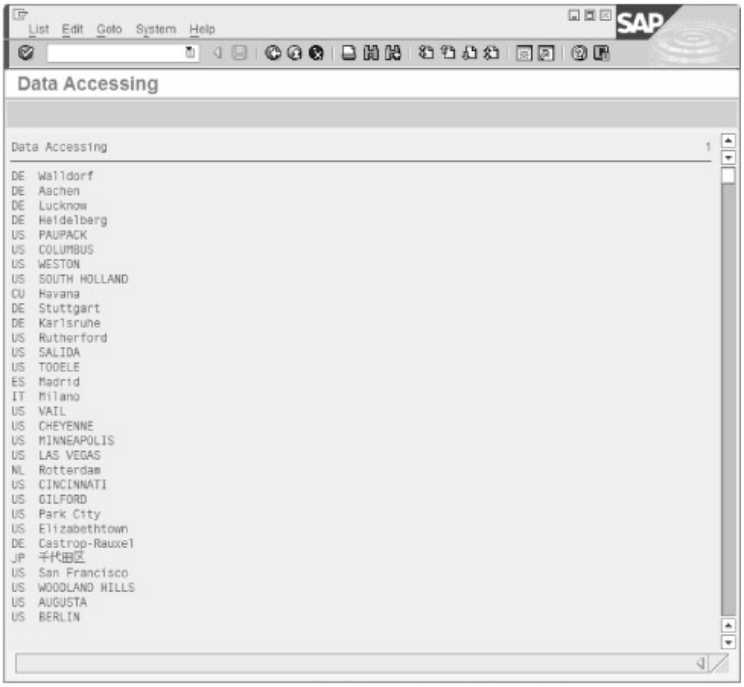


Figure 8.8: Displaying the result after using the DISTINCT clause in the SELECT statement

The *INTO* Clause

The `INTO` clause, in the `SELECT` statement, is used to define the target area. The target area is a variable that contains the field names whose data type is compatible with the field names specified in the `SELECT` statement. The `SELECT` statement uses the following specifications to determine the data type of the target area:

- **The** `<database_tab_lines>` **specification**—Determines the target area, such as a flat or a tabular structure.
- **The** `<database_tab_cols>` **specification**—Determines the structure or the line type of the target area.

The target area is flat when a single line is selected, and it can be either tabular or flat if multiple lines are selected. Note that if the target area is flat, it must be specified in the `SELECT...ENDSELECT` loop construction.

When you specify all the columns of a database table in the `SELECT` statement, the target area is either a structure type or

a type that can be converted into a structure. However, if you specify the names of columns individually in the SELECT statement, the target area can be a component of a structure or a field. The elementary type of fields in the SELECT statement must be compatible with the corresponding elementary components of the target area.

The following syntax of the INTO clause is used to read the data from database table and send it to a work area:

```
SELECT ... INTO [CORRESPONDING FIELDS OF] <work_area> ...
```

In this syntax, <work_area> represents a work area. The use of the CORRESPONDING FIELDS clause is optional after the INTO clause. The CORRESPONDING FIELDS clause is used to limit the amount of data being read from the result set of the SELECT statement and transferred into an ABAP program. Note that the CORRESPONDING FIELDS clause does not limit the amount of data being read from a database table.

When an asterisk (*) symbol is used in the SELECT statement to specify all columns of a database table, the data is transferred into the <work_area> from left to right according to the structure of the database table. However, when individual columns or aggregate expressions are used in the SELECT statement, the columns are transferred into the work area from left to right according to the structure of the work area. Note that when the data is retrieved from a database table into the <work_area>, the values of the components of <work_area> are overwritten. However, the values of the components of <work_area> are not affected by the SELECT statement.

If the user uses the asterisk (*) symbol to read all the columns of a single database table in the SELECT statement, the INTO clause can be kept empty. The SELECT statement then writes the data by default into the table work area with the same name as the database table. Note that you must declare this table work area by using the TABLES statement.

You can also specify an internal table in the INTO clause in case you have to retrieve a large number of lines from a database table. The following syntax of the INTO clause is used for reading several lines of a database table and placing them into an internal table:

```
SELECT ... INTO|APPENDING [CORRESPONDING FIELDS OF]
TABLE <internal_tab>
                  [PACKAGE SIZE <n>] ...
```

In this syntax, the <internal_tab> internal table is populated with all the lines of the selection. The APPENDING clause is another option that you can use instead of the INTO clause. The main difference between the INTO and the APPENDING clause is related to the existing lines in the <internal_tab> table. When you use the INTO clause, all the existing lines in the <internal_tab> table are deleted. When you use the APPENDING clause, new lines are added to the existing internal table, <internal_tab>. Fields in the internal table that are not affected by the selection are populated with initial values.

The PACKAGE SIZE clause is used to write the selected lines into the <internal_tab> internal table in the form of packets, but not all at once. You can define packets of <n> lines that are written one after the other into the internal table. If you use the INTO clause, each packet replaces the preceding one. If you use the APPENDING clause, the packets are inserted one after the other.

Note The PACKAGE SIZE clause is used only in the SELECT ... ENDSELECT loop, because outside the SELECT statement, the content of the internal table is undetermined.

When the names of the fields of a database table or aggregate functions are specified in the SELECT statement, the INTO clause can also specify the name of fields individually. The following syntax is used to specify the field names in the INTO clause:

```
SELECT ... INTO (<f1>, <f 2>, ...). ...
```

In this syntax, <f1> and <f2> represent the names of fields.

Examples of Using the *INTO* Clause

Listing 8.5 shows a flat structure as a target area in the INTO clause:

Listing 8.5: Showing a flat structure as a target area

```
REPORT ZDATA_ACCESS.
*/An example of INTO clause, where KOG is a flat structure acts
as a target area
```

```
DATA KOG TYPE MARA.
SELECT *
INTO KOG
FROM MARA.
   WRITE: / KOG-MATNR, KOG-ERSDA, KOG-ERNAM, KOG-MTART,
KOG-MBRSH.
ENDSELECT.
```

In Listing 8.5, KOG is a flat structure of the same data type as the target area (a database table named MARA) of the SELECT statement. The individual components of the KOG structure are specified within the WRITE statement. Figure 8.9 shows the values of the MATNR, ERSDA, ERNAM, MTART, and MBRSH fields of the MARA table:

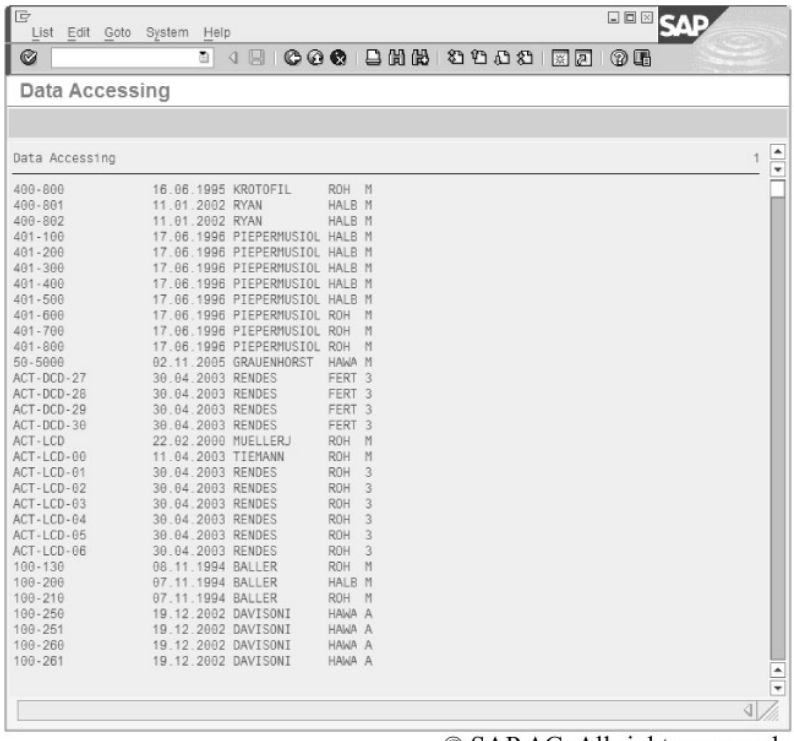


Figure 8.9: Displaying the result of the INTO clause

Listing 8.6 shows how to specify the same name for a data object and a database table:

Listing 8.6: Using the same name for an ABAP data object and a database table

```
REPORT ZDATA_ACCESS.
*/The name of data object and of database are same.
DATA MARA TYPE MARA.
SELECT *
FROM MARA.
   WRITE:    /  MARA-MATNR,      MARA-ERSDA,    MARA-ERNAM,
   MARA-MTART, MARA-MBRSH.
ENDSELECT.
```

The code given in Listing 8.6 displays a warning, when executed, as shown in Figure 8.10:

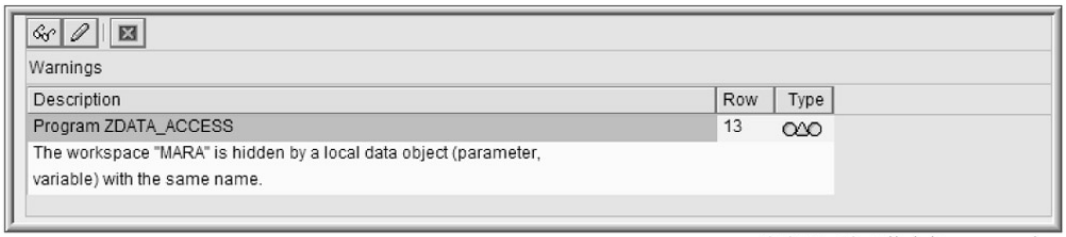


Figure 8.10: The warning generated when the structure and the database table have the same name

In Listing 8.6, a structure with the name similar to the MARA table is created. In other words, the table name and the structure name are the same; that is, MARA. The MARA structure is used implicitly as the target area in the SELECT loop. Because the names of the structure and table are the same, the SAP system overlooks the fact that you are working with an ABAP data object and not a database table.

Listing 8.7 shows an internal table as a target area in the INTO clause:

Listing 8.7: Internal table as a target area

```
REPORT ZDATA_ACCESS.
*/An internal table-MyTable- acts as a target area
DATA: BEGIN OF KOG,
         MATNR TYPE MAKV-MATNR,
WERKS TYPE MAKV-WERKS,

CSPLIT TYPE MAKV-CSPLIT,
KTEXT TYPE MAKV-KTEXT,
END OF KOG,
MyTable LIKE SORTED TABLE OF KOG
             WITH NON-UNIQUE KEY MATNR.
SELECT MATNR WERKS CSPLIT KTEXT
INTO CORRESPONDING FIELDS OF TABLE MyTable
FROM MAKV.
IF SY-SUBRC EQ 0.

LOOP AT MyTable INTO KOG.
   WRITE:  / sy-Tabix, KOG-MATNR, KOG-WERKS, KOG-CSPLIT,
   KOG-KTEXT.
ENDLOOP.
ENDIF.
```

In Listing 8.7, MyTable is a sorted internal table with the KOG line type. MyTable contains the four fields with the same names and data types as the database table MAKV. The CORRESPONDING FIELDS clause is used to place the columns from the SELECT clause into the corresponding fields of MyTable. Figure 8.11 shows the output of Listing 8.7:

```
     1  CH_4200          1100 0010 FeKoAuft HaPr,2 KuPr
     2  CPF10051         3100 0001 Schema ICE
     3  PA-300           1100 0001 Cost Distribution
     4  PQR-100          3100 001  PQR BK 100
     5  T-COP            1000 0001 Pumpe(Kuppelprodukt)
     6  T-FF100          1100 0001 T-FF100 + T-FF300
     7  T-FV100          1100 0001 T-Fv100
     8  T-FV100          3100 0001 T-Fv100
     9  T-MC             1000 0001 Aufteilung
    10  T-MCA            1000 1
    11  T-MCA            1000 2
    12  Y-300            1100 0001 Y-300 and P-300
    13  Y-300            3100 0001 co-product
    14  Y-300B           1100 PI01 PI apportionment
    15  Z-300            1100 0001 Cost distribution
    16  Z-300            3100 0001 Cost distribution
```

Figure 8.11: The CORRESPONDING FIELDS clause is placing fields in an internal table

Listing 8.8 shows how data is read in packets and placed into internal tables by using the `INTO` clause:

Listing 8.8: Reading and placing data in packets into an internal table

```
REPORT ZDATA_ACCESS.
*/Reading the data in packets and keeping them into an
   Internal Table
DATA: KOG TYPE KNA1,
    MyTable TYPE SORTED TABLE OF KNA1
            WITH UNIQUE KEY KUNNR.
SELECT KUNNR NAME1 ORT01 FROM KNA1
INTO CORRESPONDING FIELDS OF TABLE MyTable
       PACKAGE SIZE 10.
   WRITE: / 'Total lines found : ', SY-DBCNT.
LOOP AT MyTable INTO KOG.
   WRITE: / sy-Tabix, KOG-KUNNR, KOG-NAME1, KOG-ORT01.
ENDLOOP.
   SKIP 1.
ENDSELECT.
```

Listing 8.8 shows that the data is read in the packet format. The package size is specified as 10, so that a group of 10 lines are read from the `KNA1` database table, which are then placed in the MyTable internal table of the KOG structure type. The values stored in the corresponding components of the KOG type are displayed as shown in Figure 8.12:

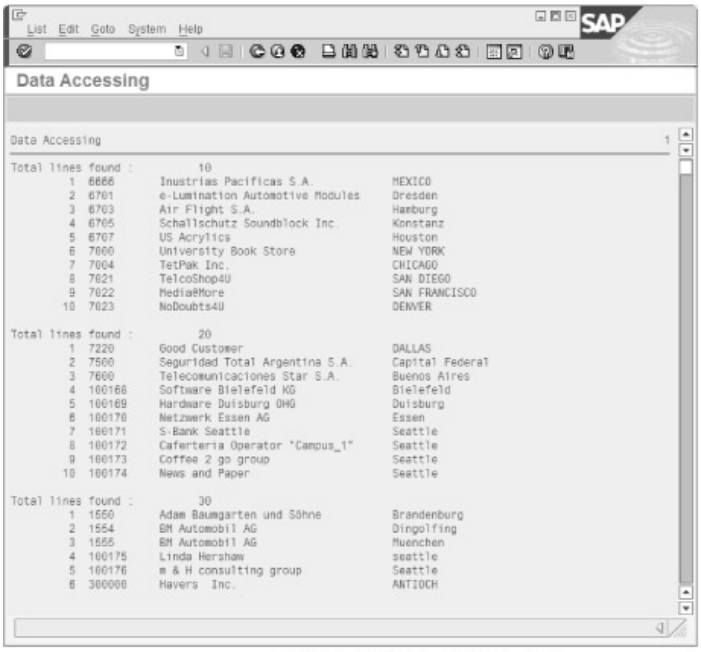


Figure 8.12: Result of the INTO clause

In Listing 8.8, if the `APPENDING` clause is used instead of the `INTO` clause, the output is as it appears in Figure 8.13:

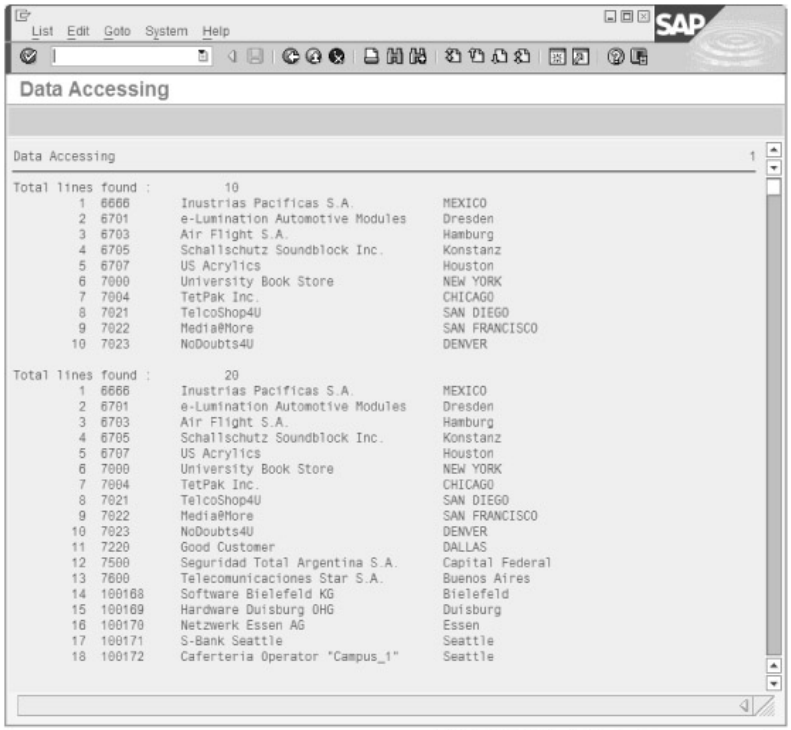


Figure 8.13: Displaying the result of using the APPENDING clause

In Figure 8.13, you can see that the result of the `APPENDING` clause is different from the result of the `INTO` clause (see Figure 8.12). In the case of the `INTO` clause, 10 records are inserted in MyTable and then displayed. However, in the case of the `APPENDING` clause, 10 records are added each time and appended at the end of the previous records in MyTable and then displayed.

Listing 8.9 shows how single fields are specified as a target area by using the `INTO` clause:

Listing 8.9: Specifying single fields as target areas

```
REPORT ZDATA_ACCESS.
*/Specifying individual fields in target area
DATA: AVERAGE TYPE P DECIMALS 2,
      SUM TYPE P DECIMALS 2.
SELECT AVG( MENGE ) SUM( MENGE )
INTO (AVERAGE, SUM)
FROM MARI.
WRITE: / 'Average:', AVERAGE,
       / 'Sum :', SUM.
```

In Listing 8.9, the `SELECT` clause contains two aggregate expressions to calculate the average and sum of the `MENGE` field from the MARI database table. The target fields are called `AVERAGE` and `SUM`. Figure 8.14 shows the average and sum of the values stored in the `MENGE` field:

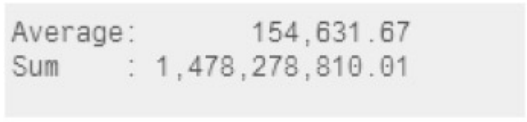


Figure 8.14: Result of the AVERAGE and SUM aggregate functions

Listing 8.10 shows the usage of an alias in the `INTO` clause:

Listing 8.10: Using aliases

```
REPORT ZDATA_ACCESS.
*/Creating and Using aliases-MAXI and MINI
DATA: BEGIN OF LINE,
MAXI TYPE P decimals 4,
       MINI TYPE p decimals 4,
       END OF LINE.
SELECT MAX( MENGE ) AS MAXI MIN( MENGE ) AS MINI
INTO CORRESPONDING FIELDS OF LINE
FROM MARI.
WRITE: / 'Maximum value:', LINE-MAXI,
      / 'Minimum value:', LINE-MINI.
```

In Listing 8.10, the SELECT clause contains two aggregate expressions to calculate the maximum and minimum values of the MENGE field of the MARI database table. A structure LINE is used as the target area and the names of the structure components are used as aliases in the SELECT clause. Figure 8.15 shows the maximum and minimum values of the MENGE field of the MARI database table:

Figure 8.15: The maximum and minimum values of a field

The *FROM* Clause

The FROM clause of the SELECT statement determines the source area, which are database tables from which data needs to be retrieved. You can specify either a single table or multiple tables, linked by using inner or outer joins, in the FROM clause. The syntax to use the FROM clause is:

```
SELECT... FROM <database_tab> <options>...
```

In this syntax, the FROM clause is followed by the name of the <database_ tab> database table and the <options> expression. The <options> expression represents any clause that can be used to control access to the database.

In the SELECT statement, the name of a database table can be specified statically or dynamically. In addition, you can specify and use the alias names (alternative names) of these tables. The following syntax is used to specify the name of a database table statically:

```
SELECT... FROM <database_tab> [AS <database_tab_alias>]
<options> ...
```

In this syntax, <database_tab> represents a database table that must exist in the ABAP Dictionary. The AS clause is used to assign the <database_ tab_alias> alias to the <database_tab> table. Using aliases helps eliminate the ambiguities when you use more than one database table or when a single database table is used more than once in a join expression.

The following syntax is used to specify the name of a database table dynamically:

```
SELECT... FROM (<database_tab>) <options> ...
```

The <database_tab> expression represents the name of a database table in uppercase. When the name of a database table is specified dynamically, you cannot use an empty INTO clause to read all the columns of the database table. It is also not possible to use alternative or alias table names.

The following tasks can be performed by using the FROM clause with the SELECT statement:

- Using JOIN (INNER or OUTER)
- Client-handling
- Disabling data buffering

- Restricting number of lines

Now, let's discuss these tasks in detail, one by one.

Using *JOIN* (*INNER* or *OUTER*)

Sometimes you need to retrieve the data of two or more database tables, such as while creating a report. In such a situation, you can join the tables. To join two or more database tables, you use the JOIN clause in the SELECT statement. In Open SQL, two kinds of joins can be used to join two or more tables, INNER and LEFT OUTER. The following syntax shows how to use the INNER or LEFT OUTER clause to read data from more than one table:

```
... [(] {database_tab_left [AS database_tab_alias_left]}
| join
           {[INNER] JOIN}|{LEFT [OUTER] JOIN}
           {database_tab_right [AS database_tab_alias_right]
           ON <join_cond>}
[)]
  ... .]
```

In this syntax, a join expression contains a left-side table and a right-side table that are joined by using either [INNER] JOIN or LEFT [OUTER] JOIN. As specified in the syntax, a join expression can be an inner join (INNER) or an outer join (LEFT OUTER). Furthermore, a join expression is used in parentheses ().

The inner join is used to join two database tables such that a single result set is obtained. This result set is based on the matching records of the left-side and the right-side tables specified in the join expression. The data in this result set contains all the combinations of rows whose columns meet the <join_cond> condition. The following syntax shows how to implement an inner join:

```
SELECT...
...
FROM <database_tab_left> [AS <database_tab_alias_left>]
[INNER] JOIN <database_tab_right> [AS <database_tab_alias_
right>] ON <join_cond>
...
```

In this syntax, <database_tab_left> and <database_tab_right> are database tables that can be specified statically or dynamically. You can also use alias names of the database tables. The use of the INNER clause is optional; however, the JOIN clause is mandatory if you want specify a join expression.

A join expression links the lines of <database_tab_left> with those of <database_tab_right> that meet the <join_cond> condition. The syntax of the <join_cond> condition is similar to that of the WHERE clause, where one or more conditions are combined by using the AND operator. Each comparison must contain a column from the right-hand table, <database_ tab_right>. Moreover, you can specify columns that have the same name in the <database_tab_left> and <database_tab_right> tables on both sides of a comparison operator.

The LEFT OUTER join is used to display all the records of the left database table and only the matching records of the right database table. The syntax to use the LEFT OUTER join is:

```
SELECT...
...
FROM <database_tab_left> [AS <database_tab_alias_left>] LEFT
[OUTER] JOIN
<database_tab_right> [AS <database_tab_alias_right>] ON
<join_cond>
<options> ...
```

In this syntax, the <database_tab_left> and <database_tab_right> are database tables. The use of the OUTER clause is optional. The tables are linked by using the LEFT JOIN or LEFT OUTER JOIN clause and specifying the <join_cond> condition, similar to the inner join. The final selection includes all the lines of the <database_tab_left> table, but matching lines of the <database_tab_right> table. However, the lines of the <database_tab_ right> table, which do not match the corresponding lines of the <database_ tab_left> table, contain null values.

In the INNER join, all the combinations of matching records of the two database tables are selected. However, in the LEFT OUTER join, all the records of the left-hand database table are selected along with the matching records of the right-hand

database table.

In the LEFT OUTER join, the following restrictions apply to the <join_cond> condition:

- Use either the EQ or = relational operator.
- Specify at least one comparison between the columns of the <database_tab_left> and <database_tab_right> database tables.
- Include the columns of the <database_tab_right> table in the <join_cond> condition when these columns are used in comparisons.

Note Including the name of the columns of the <database_tab_right> table is not necessary in the WHERE clause.

Note that the previous restrictions do not apply while working with an INNER join.

Client-Handling

As already mentioned, you can switch off automatic client-handling in Open SQL statements by using a special clause, CLIENT SPECIFIED. In the SELECT statement, the CLIENT SPECIFIED clause comes after the table specified in the FROM clause, as shown in the following syntax:

```
SELECT ... FROM <database_tab> CLIENT SPECIFIED ...
```

In this syntax, the CLIENT SPECIFIED clause allows you to manipulate the fields of the specified client in the individual clauses of the SELECT statement.

Disabling Data Buffering

One of the important features of the SELECT statement in Open SQL is that it always retrieves or reads the data from the buffer area in the database interface of the current application server. This means that the SELECT statement does not retrieve the data from the database table directly. However, you can disable data buffering of a table by using the BYPASSING BUFFER clause, as shown in the following syntax:

```
SELECT ... FROM <database_tab> BYPASSING BUFFER ...
```

The BYPASSING BUFFER clause ensures that the data being read or retrieved directly from the database tables is the updated one.

Note Buffering of the data is preferred when the data does not change frequently because it improves the performance of the SAP R/3 system.

Restricting the Number of Lines

The SELECT statement can be used to specify a number (fixed or absolute) up to which the rows or lines of a database table can be retrieved or selected, as shown in the following syntax:

```
SELECT ... FROM <database_tab> UP TO <n> ROWS ...
```

In this syntax, the <n> expression represents either a positive number or zero. If <n> is a positive integer, the SAP system can retrieve the <n> lines from a <database_tab> database table. If <n> is zero, the SAP system retrieves all lines from the table that meet the selection criteria.

Now, let's consider some examples of using the FROM clause.

Examples of Using the *FROM* Clause

Listing 8.11 shows how to specify a database table statically:

Listing 8.11: Specifying a database table statically

```
REPORT ZDATA_ACCESS.
*/Specifying a database table statically
DATA KOG TYPE MAKV.
SELECT *
INTO KOG
```

```
FROM MAKV UP TO 6 ROWS.
WRITE: / KOG-MATNR, KOG-WERKS, KOG-CSPLIT, KOG-KTEXT.
ENDSELECT.
```

The code given in Listing 8.11 reads six lines from the MAKV database table and places them into the KOG structure. The MATNR, WERKS, CSPLIT, and KTEXT components of the KOG structure are displayed, as shown in Figure 8.16:

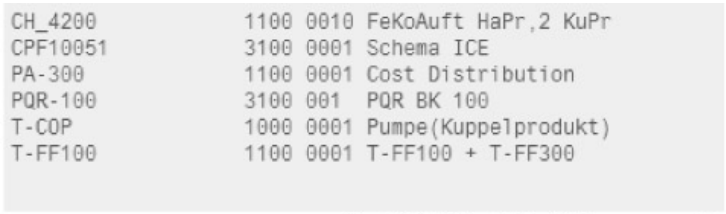


Figure 8.16: Reading lines from a table

Listing 8.12 shows how to switch off automatic client-handling by using the `CLIENT SPECIFIED` clause:

Listing 8.12: Using the CLIENT SPECIFIED clause

```
REPORT ZDATA_ACCESS.
*/Using the CLIENT SPECIFIED clause
DATA KOG TYPE MARI.
DATA name(10) TYPE c VALUE 'MARI'.
SELECT *
   INTO KOG
   FROM (name) CLIENT SPECIFIED
   WHERE MANDT = '800' and USNAM = 'BUROW'.
      WRITE: / KOG-MBLNR, KOG-MATNR, KOG-BWART, KOG-USNAM.
ENDSELECT.
```

In Listing 8.12, automatic client-handling is switched off by using the `CLIENT SPECIFIED` clause. The KOG structure contains the data of the MARI database table, where the value of the `MANDT` field is 800 and that of the `USNAM` field is BUROW. The identical components of the KOG structure are displayed. Figure 8.17 shows the output of Listing 8.12:

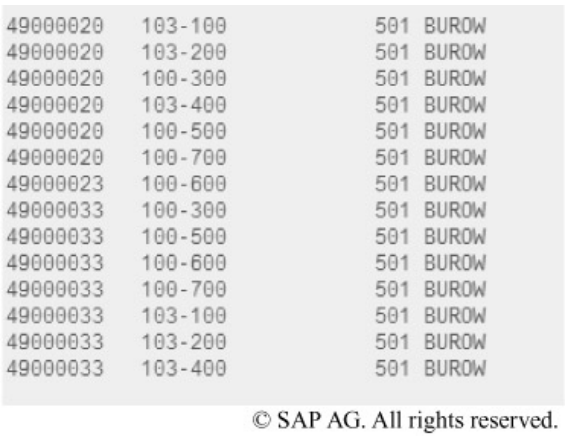


Figure 8.17: Displaying the result when a client is specified manually

Listing 8.13 shows how two database tables are joined by using the `INNER` join:

Listing 8.13: Joining two database tables by using the INNER join

```
REPORT ZDATA_ACCESS.
*/Using INNER join in between two tables
DATA: BEGIN OF KOG,
          MATNR TYPE MARA-MATNR,
```

```
          ERNAM TYPE MARA-ERNAM,
MAKTX TYPE MAKT-MAKTX,
END OF KOG,
MYTABLE LIKE SORTED TABLE OF KOG
             WITH UNIQUE KEY MATNR ERNAM MAKTX.
SELECT MR~MATNR MR~ERNAM MK~MAKTX
   INTO CORRESPONDING FIELDS OF TABLE MYTABLE
   FROM (MARA AS MR
   INNER JOIN MAKT AS MK ON MK~MATNR = MR~MATNR)
WHERE MR~ERNAM = 'RUDISILL'.
WRITE: 'Material Number', ' ', 'Created by', ' ', 'Material
Description'.
skip.
LOOP AT MYTABLE INTO KOG.
   AT NEW MATNR.
          WRITE: / KOG-MATNR.
ENDAT.
   WRITE: KOG-ERNAM, KOG-MAKTX.
ENDLOOP.
```

In Listing 8.13, the ERNAM field of the MARA table and the MAKTX field of the MAKT table are joined by the INNER JOIN based on the MATNR field of both the tables. An alias name is assigned to each table; for instance, MR is an alias name of the MARA table and MK is an alias name of the MAKT table. Listing 8.13 creates a list of material numbers and their descriptions, where the value of the ERNAM field is RUDISILL (the ERNAM field contains the name of the person who has created the material). Figure 8.18 shows the list of material numbers created by RUDISILL:

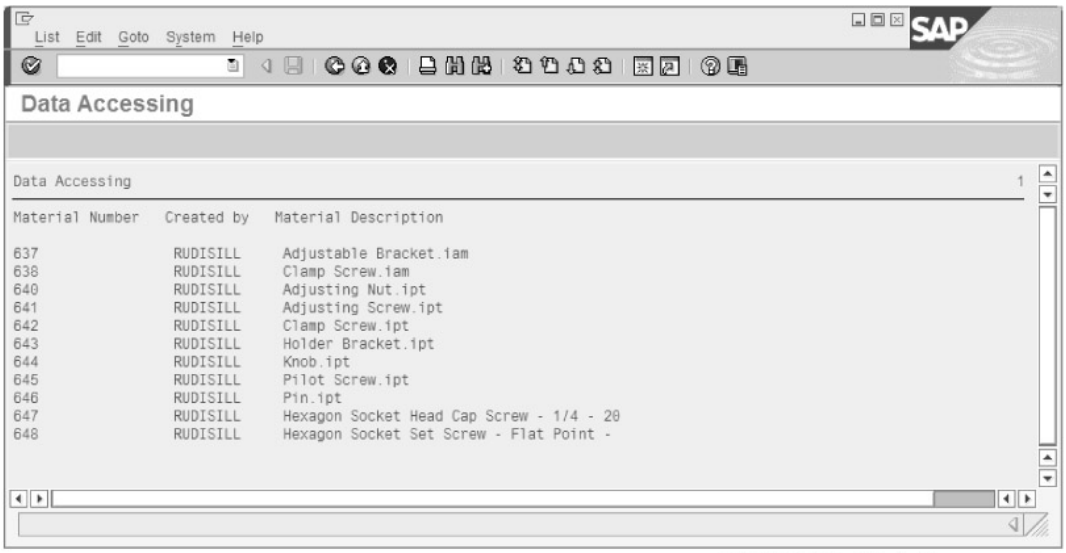


Figure 8.18: Displaying the result of using the INNER join

Listing 8.14 shows how two database tables are joined by using the LEFT OUTER join:

Listing 8.14: Joining two database tables by using the LEFT OUTER join

```
REPORT ZDATA_ACCESS.
*/Using LEFT OUTER join in between two tables
DATA: BEGIN OF KOG,
        MATNR TYPE MARA-MATNR,
        ERNAM TYPE MARA-ERNAM,
         SPRAS TYPE MAKT-SPRAS,
END OF KOG,
MyTable LIKE SORTED TABLE OF KOG
WITH NON-UNIQUE KEY MATNR.
SELECT MR~MATNR MR~ERNAM MK~SPRAS
   INTO CORRESPONDING FIELDS OF TABLE MyTable
   FROM MARA AS MR
```

```
LEFT OUTER JOIN MAKT AS MK ON MR~MATNR = MK~MATNR AND MK~SPRAS =
'J'.
LOOP AT MyTable INTO KOG.
   WRITE: / KOG-MATNR, KOG-ERNAM, KOG-SPRAS.
ENDLOOP.
```

In Listing 8.14, the ERNAM field of the MARA table and the SPRAS field of the MAKT table are joined by the `LEFT OUTER JOIN` based on the MATNR field of both the tables, where the value in the SPRAS field of the MAKT table is J. An alias name is assigned to each table; MR for the MARA table and MK for the MAKT table. Figure 8.19 shows the result of the `LEFT OUTER` join:

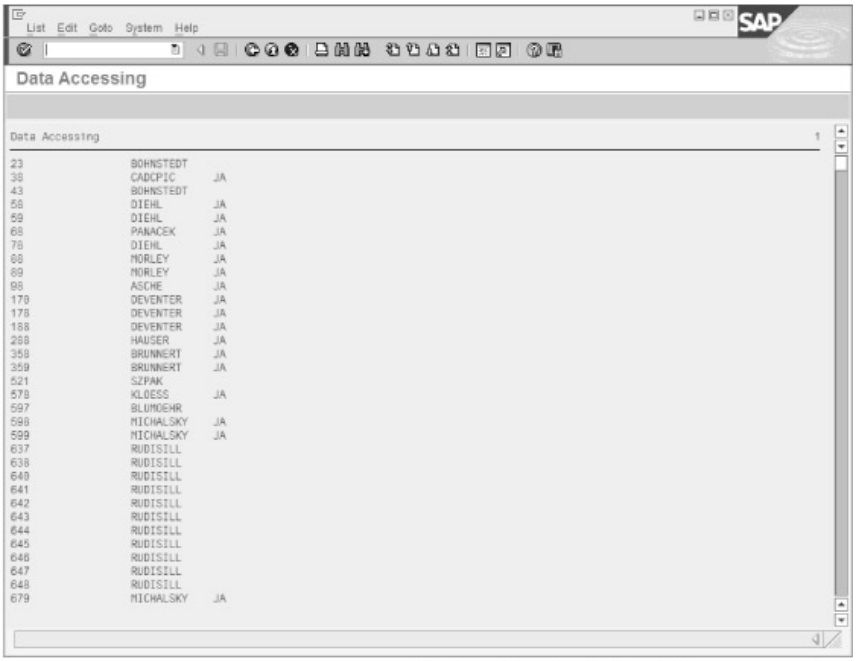


Figure 8.19: Displaying the result of using the LEFT OUTER join

Now, replace the `LEFT OUTER` join with the `INNER` join in Listing 8.14. The output is shown in Figure 8.20:

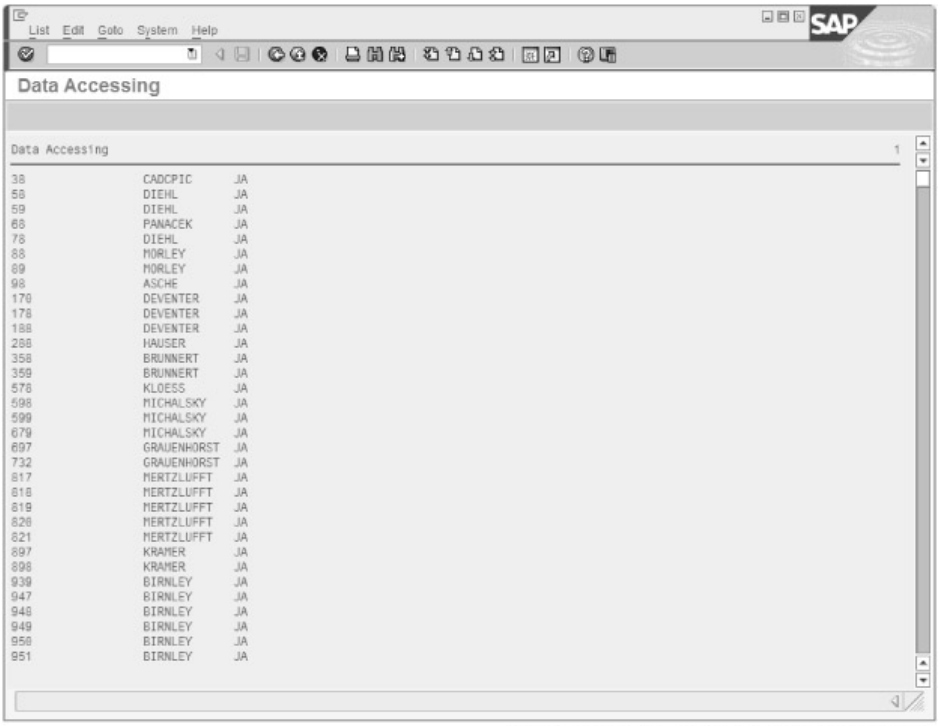


Figure 8.20: Displaying the result when the LEFT OUTER join is changed to the INNER join

You can see that the result displayed in Figure 8.20 differs from the result displayed in Figure 8.19. The `INNER` join shows all the combinations of the matching records of the MARA and MAKT database tables, where their `MATNR` fields have equal values, and the SPRAS field of the MAKT table contains the value J. However, in the case of the `LEFT OUTER` join, all the records of the MARA table are shown along with the matching records of the MAKT table, where the `MATNR` fields of both the tables have the same value and the SPRAS field of the MAKT table contains the value J.

The *WHERE* Clause

The `WHERE` clause is used to specify one or more conditions to read or retrieve data from a database table. This clause is also used in the `OPEN CURSOR`, `UPDATE`, and `DELETE` statements (discussed later in this chapter). The syntax of the `WHERE` clause is:

```
SELECT ... WHERE <where_condition> ...
```

In this syntax, the `<where_condition>` expression represents one or more conditions. The conditions may include the use of one or more comparison operators, such as >, <, and ==, or some special expressions, such as `GT`, `LT`, and `EQ`. You can also combine multiple conditions into a single condition using the logical operators, such as `AND`, `OR`, and `NOT`. Moreover, the conditions may also be specified dynamically.

The column names included in the conditions, specified by using the `WHERE` clause, can also appear in the `SELECT` statement. The result of a condition specified in the `WHERE` clause can be true, false, or unknown. The result of a condition is unknown when the column used in the condition has a null value. Using the `WHERE` clause, a line is retrieved from a database table only if the condition specified on the respective columns is true for the line. The following tasks can be performed by using the `WHERE` clause within the `SELECT` statement:

- Comparing column values using relational operators
- Specifying range operators
- Matching a pattern
- Specifying a list
- Checking subqueries

- Checking selection tables
- Checking null values
- Specifying negating conditions
- Specifying linking conditions
- Specifying dynamic conditions
- Specifying tabular conditions

Now, let's discuss these tasks in detail, one by one.

Comparing Column Values by Using Relational Operators

The syntax to compare the value of a column of any data type with another value is:

```
SELECT ... WHERE <s> <operator> <f> ...
```

In this syntax, the <s> expression represents a column of one of the database tables named in the FROM clause. The <f> expression represents a column in a database table specified in the FROM clause, a data object, or a scalar subquery.

Table 8.6 describes a list of relational operators and their meanings:

Table 8.6: Relational operators and their meanings

Operator	Meaning
EQ	Equal to
=	Equal to
NE	Not equal to
<>	Not equal to
><	Not equal to
LT	Less than
<	Less than
LE	Less than or equal to
<=	Less than or equal to
GT	Greater than
>	Greater than
GE	Greater than or equal to
>=	Greater than or equal to

The values of the operands can be converted to other data types, if necessary. The conversion may be dependent on the platform and code page.

Specifying Range Operators

The following syntax is used to determine whether the value of a column lies within a specified range of values:

```
SELECT ... WHERE <s> [NOT ] BETWEEN <f 1> AND <f 2> ...
```

The condition is evaluated to true if the value of the <s> column lies between the data objects <f1> and <f2> in the case of the BETWEEN range operator and does not lie between the values of the data objects <f1> and <f2> in the case of the NOT BETWEEN range operator. You cannot use the BETWEEN clause in the ON condition of the FROM clause.

Matching a Pattern

The syntax to determine whether the value of a column matches a pattern is:

```
SELECT ... WHERE <s> [NOT ] LIKE <f> [ESCAPE <h>] ...
```

In this syntax, the condition is true if the value of the <s> column matches or does not match the pattern specified in the data object <f>. The data type of the column must be alphanumeric. The <f> data object must be of the C type.

You can use the following wildcard characters in the <f> data object:

- % for a sequence of characters (including spaces)
- _ for a single character

For example, ABC_EFG% matches the strings ABCxEFGxyz and ABCxEFG but does not match ABCEFGxyz. If you want to use two wildcard characters explicitly in the comparison, use the ESCAPE clause to specify an escape symbol in the <h> expression. If the ESCAPE clause is preceded by the <h> expression, the wildcards and the escape symbol itself lose their usual function within the pattern <f>. The use of _ and % corresponds to Standard SQL usage; however, the logical expressions can use the other wildcard characters (+ and *) in an ABAP program.

Note You cannot use the LIKE clause in the ON condition of the FROM clause.

Specifying a List

The syntax to find out whether the value of a column is stored in a list of values is:

```
SELECT ... WHERE <s> [NOT ] IN (<f 1>, ......, <f n>) ...
```

The condition is true if the value of the <s> column exists in the list <f1> ... <fn> in the case of the IN clause, or does not exist in the list <f1> ... <fn> in the case of the NOT IN clause.

Checking Subqueries

The following syntax is used to find whether the value of a column is stored in a scalar subquery:

```
SELECT ... WHERE <s> [NOT ] IN <subquery> ...
```

The condition is true if the value of <s> is stored in the result set of the <subquery> scalar subquery in the case of the IN clause, or not stored in the result set of the <subquery> scalar subquery in the case of the NOT IN clause.

The following syntax is used to find whether the selection of a subquery contains any lines:

```
SELECT ... WHERE [ NOT ] EXISTS <subquery> ...
```

This condition is evaluated to true if the result set of the <subquery> subquery contains at least one record in the case of the EXISTS clause, and no record in the case of the NOT EXISTS clause. The subquery does not have to be scalar.

Checking Selection Tables

The following syntax is used to find whether or not the value of a column satisfies the conditions in a selection table:

```
SELECT ... WHERE <s> [NOT ] IN <seltab> ...
```

This syntax uses two operators, IN and NOT IN. The IN operator validates that the value specified in the <s> column matches with the values of the <seltab> expression. The NOT IN operator validates that the value specified in the <s> column does not match with the values stored in the <seltab> expression. The <seltab> expression can be either a real selection table or a RANGES table.

Checking Null Values

The following syntax is used to ascertain whether the value of a column is null:

```
SELECT ... WHERE <s> IS [NOT ] NULL ...
```

This syntax uses two operators, IS NULL and IS NOT NULL. The IS NULL operator validates that the value specified in the <s> column is null. Alternatively, the IS NOT NULL operator validates that the value specified in the <s> column is not null.

Specifying Negating Conditions

The syntax to negate the result of a WHERE clause is:

```
SELECT ... WHERE NOT <cond> ...
```

The condition is true if the <cond> condition specified by the WHERE clause is evaluated to false, and vice versa. The result of an unknown condition remains unknown when negated.

Specifying Linking Conditions

At times, you might need to specify more than one condition in a WHERE clause. In such cases, you can use the AND and OR operators in the WHERE clause, as shown in the following syntax:

```
SELECT ... WHERE <cond 1> AND <cond 2> ...
SELECT ... WHERE <cond 1> OR <cond 2> ...
```

This syntax demonstrates two WHERE clauses; the first WHERE clause uses the AND operator and the second WHERE clause uses the OR operator. The AND operator returns true if the <cond1> and <cond2> conditions are true, whereas the OR operator returns true if one or both the conditions are true.

When NOT, AND and OR operators are used simultaneously, the priority level of the NOT operator is higher than that of the AND operator and AND takes priority over OR. However, parentheses () can be used to control the processing sequence.

Specifying Dynamic Conditions

The syntax used to specify a condition dynamically is:

```
SELECT ... WHERE (<internal_tab>) ...
```

In this syntax, the <internal_tab> expression represents an internal table with line type C and a maximum length of 72 characters. In this code, you may use only literals for an internal table, not the names of data objects. The internal table can also be left empty.

Use the following syntax if you want to specify only a part of the condition dynamically:

```
SELECT ... WHERE <cond> AND (<internal_tab>) ...
```

You cannot link a static and a dynamic condition by using the OR operator.

Note Dynamic conditions are specified in the WHERE clause of the SELECT statement.

Specifying Tabular Conditions

The WHERE clause of the SELECT statement has a special variant that allows you to derive conditions from the lines and columns of an internal table, as shown in the following syntax:

```
SELECT ... FOR ALL ENTRIES IN <internal_tab> WHERE
<cond> ...
```

In this syntax, the <internal_tab> expression represents an internal table and the <cond> expression represents one or more conditions of the WHERE clause. The <cond> conditions accept the fields of internal tables as operands. You can use the FOR ALL ENTRIES clause to compare the records or lines of a database table against all lines of the internal table. Consequently, the SAP system selects the lines from the database table that satisfy the condition for each line of the internal table. The result set of the preceding SELECT statement consists of the matching records of both the internal table and the database table. Duplicate lines are eliminated automatically from the result set. If the <internal_tab> internal table is empty, the FOR ALL ENTRIES clause is discarded and all the entries are retrieved or read from the database table.

The internal table <internal_tab> must have a structured line type, and each field that occurs in the condition <cond> must be compatible with the column of the database with which it is compared. The fields of an internal table cannot be compared by using the LIKE, BETWEEN, and IN operators.

You can use the FOR ALL ENTRIES clause to replace nested select loops by operations on internal tables. Using the FOR ALL ENTRIES clause can improve the performance significantly for large sets of selected data.

Examples of Using the *WHERE* Clause

In this section, we consider some examples of using the WHERE clause to specify various conditions on the ZSTUDENT_DATA table, which is a user-defined table. The ZSTUDENT_DATA table contains the STUDENTID, STUDENTNAME, SUBJECTCODE, MARKS, ADDRESS, CITY, CLASS, MONTHLYFEES, and YEARLYFEES fields. The records of the ZSTUDENT_DATA table are shown in Figure 8.21:

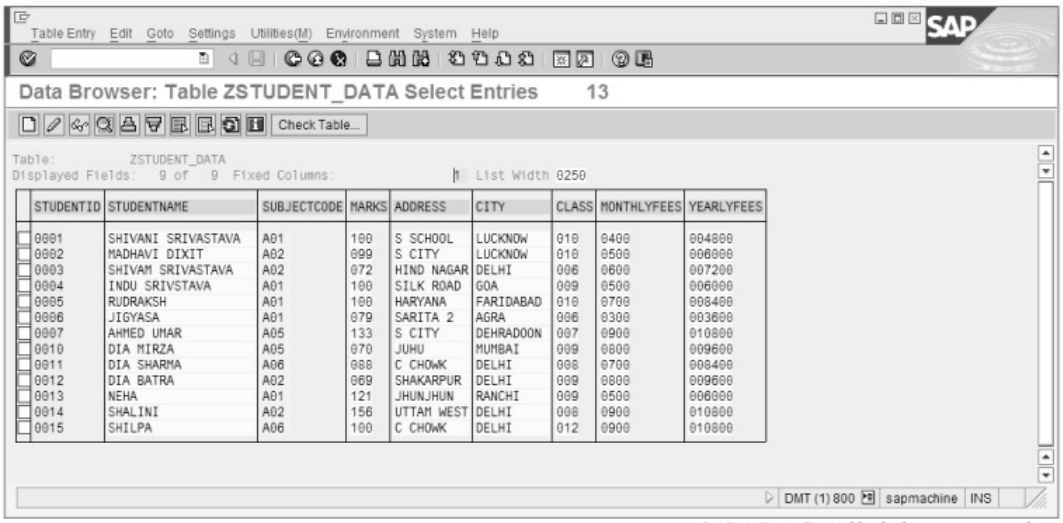


Figure 8.21: Displaying the data of the ZSTUDENT_DATA table

Table 8.7 lists various conditions that can be specified with the WHERE clause on the ZSTUDENT_DATA table:

Table 8.7: Examples of the WHERE clause

Condition	Description

`TABLES: ZSTUDENT_DATA.`
`SELECT * FROM ZSTUDENT_DATA WHERE CITY = 'DELHI'.`
`ENDSELECT.`
	Demonstrates the use of a simple `WHERE` clause. The condition specified in the `WHERE` clause is evaluated to true if the `CITY` field has the value DELHI.

`TABLES: ZSTUDENT_DATA.`
`SELECT * FROM ZSTUDENT_DATA WHERE YEARLYFEES GE 8000.`
`ENDSELECT.`
	Shows the use of the `GE` operator with the `WHERE` clause. The condition specified in the `WHERE` clause is evaluated to true if the `YEARLYFEES` field contains numbers greater than or equal to 8000.

`TABLES: ZSTUDENT_DATA.`
`SELECT * FROM ZSTUDENT_DATA WHERE ADDRESS NE 'C CHOWK'.`
`ENDSELECT.`
	Illustrates the use of the `NE` operator with the `WHERE` clause. The condition specified in the `WHERE` clause is evaluated to true if the `ADDRESS` field does not contain the string C CHOWK.

`TABLES: ZSTUDENT_DATA.`
`SELECT * FROM ZSTUDENT_DATA WHERE MONTHLYFEES BETWEEN 500 AND 800.`
`ENDSELECT.`
	Exhibits the use of the `BETWEEN` range operator with the `WHERE` clause. The condition specified in the `WHERE` clause is evaluated to true if the field `MONTHLYFEES` contains numbers between 500 and 800.

`TABLES: ZSTUDENT_DATA.`
`SELECT * FROM ZSTUDENT_DATA WHERE YEARLYFEES NOT BETWEEN 6000 AND 10000.`
`ENDSELECT.`
	Demonstrates the use of the `NOT BETWEEN` range operator with the `WHERE` clause. The condition specified in the `WHERE` clause is evaluated to true if the field `YEARLYFEES` contains numbers not between 6000 and 10000.

`TABLES: ZSTUDENT_DATA.SELECT * FROM ZSTUDENT_DATA WHERE STUDENTID NOT BETWEEN '0003' AND '0010'.` `ENDSELECT.`	Shows the use of the `NOT BETWEEN` range operator with the `WHERE` clause. The condition specified in the `WHERE` clause is evaluated to true if the field `STUDENTID` is four characters long and its values do not lie between 0003 and 0010.

`TABLES: ZSTUDENT_DATA.`
`SELECT * FROM ZSTUDENT_DATA WHERE STUDENTNAME LIKE '%SRIVASTAVA'.`
`ENDSELECT.`
	Illustrates the use of the `LIKE` operator with the `WHERE` clause. The condition specified in the WHERE clause is evaluated to true if the field `STUDENTNAME` contains a string containing the pattern SRIVASTAVA at the end of a student's name.

`TABLES: ZSTUDENT_DATA.`
`SELECT * FROM ZSTUDENT_DATA WHERE CITY NOT LIKE '_A%'.`
`ENDSELECT.`
	Demonstrates the use of the `NOT LIKE` operator with the `WHERE` clause. The condition specified in the `WHERE` clause is evaluated to true if the field `City` contains a value whose second character is not A.

`TABLES: ZSTUDENT_DATA.`
`SELECT * FROM ZSTUDENT_DATA WHERE STUDENTNAME LIKE 'DIA#%'`
`ESCAPE '#'.`
`ENDSELECT.`
	Demonstrates the use of the `LIKE` operator with the `WHERE` clause to search according to the specified characters. The condition specified in the `WHERE` clause is evaluated to true if the values contained in the field `STUDENTNAME` begin with DIA.

`TABLES: ZSTUDENT_DATA.`
`SELECT * FROM ZSTUDENT_DATA WHERE CITY IN ('AGRA', 'MUMBAI', 'LUCKNOW').`
`ENDSELECT.`
	Shows the use of the IN operator with the `WHERE` clause. The condition specified in the `WHERE` clause is evaluated to true if the column `CITY` contains one of the values AGRA, MUMBAI, or LUCKNOW.

`TABLES: ZSTUDENT_DATA.`
`SELECT * FROM ZSTUDENT_DATA WHERE CITY NOT IN ('DELHI', 'GOA').`
`ENDSELECT.`
	Demonstrates the use of the `NOT IN` operator with the `WHERE` clause. The condition specified in the `WHERE` clause is evaluated to true if the column `CITY` does not contain the values DELHI or GOA.

`TABLES: ZSTUDENT_DATA.`
`SELECT * FROM ZSTUDENT_DATA WHERE (SUBJECTCODE = 'A01' OR SUBJECTCODE = 'A02') AND NOT (CITY = 'DELHI' OR CITY = 'LUCKNOW').`
`ENDSELECT.`
	Exemplifies the use of the `OR`, `AND`, and `NOT` operators with the `WHERE` clause. The condition specified in the `WHERE` clause is evaluated to true if the field `SUBJECTCODE` contains the value A01 or A02 and the field `CITY` contains neither DELHI nor LUCKNOW.

We have explored various examples of using different operators with the `WHERE` clause. Now, let's consider an example of using dynamic conditions, as shown in Listing 8.15:

Listing 8.15: An example of dynamic conditions

```
REPORT ZDATA_ACCESS.

*/Specifying a condition dynamically
DATA: CITYDATA(72) TYPE C,
      CITYDETAIL LIKE TABLE OF CITYDATA.
PARAMETERS: CITY_A(10) TYPE C, CITY_B(10) TYPE C.
DATA KOG TYPE KNA1-ORT01.
CONCATENATE 'ORT01 = ''' CITY_A '''' INTO CITYDATA.
APPEND CITYDATA TO CITYDETAIL.
CONCATENATE 'OR ORT01 = ''' CITY_B '''' INTO CITYDATA.
APPEND CITYDATA TO CITYDETAIL.
CONCATENATE 'OR ORT01 = ''' 'GERA' '''' INTO CITYDATA.
APPEND CITYDATA TO CITYDETAIL.
LOOP AT CITYDETAIL INTO CITYDATA.
      WRITE CITYDATA.
ENDLOOP.
SKIP.
SELECT ORT01 INTO KOG FROM KNA1 WHERE (CITYDETAIL).
   WRITE / KOG.
ENDSELECT.
```

In Listing 8.15, `CITYDATA` is a variable of `C` type and `CITYDETAIL` is a variable of type `CITYDATA`. The value of `CITYDATA` is appended to `CITYDETAIL`.

Figure 8.22 shows the output of Listing 8.15, displaying two text fields, `CITY_A` and `CITY_B`:

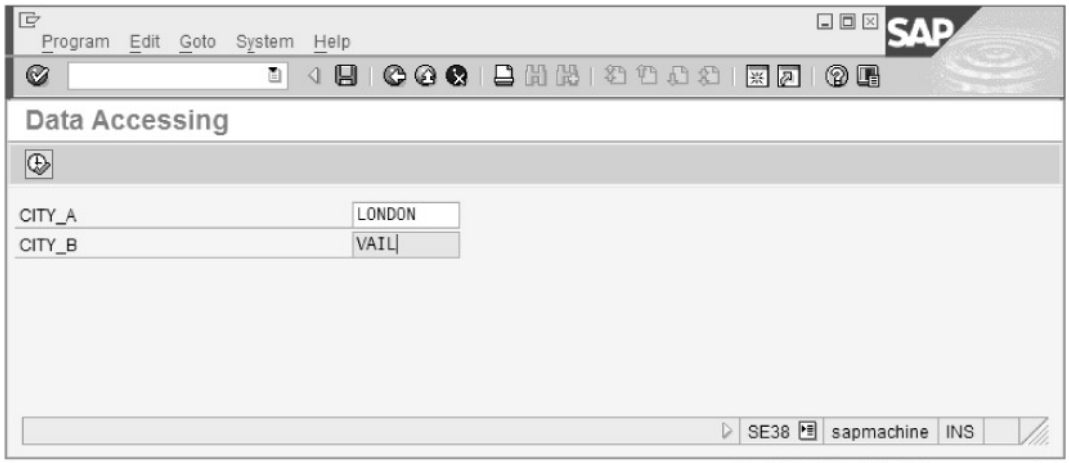


Figure 8.22: Displaying the CITY_A and CITY_B text fields

You can enter the data dynamically by entering the values in the `CITY_A` and `CITY_B` text fields. For instance, enter LONDON and VAIL in the `CITY_A` and `CITY_B` fields, respectively, and then click the Execute icon, as shown in Figure 8.22. The resultant output is shown in Figure 8.23:

Now, let's consider some examples of how to specify a tabular condition in a `WHERE` clause.

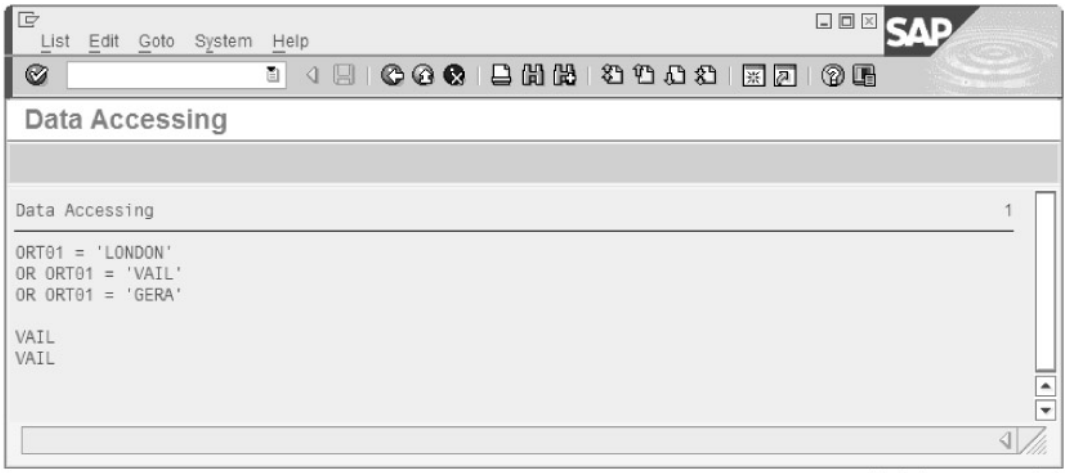


Figure 8.23: Result of dynamic conditions

Listing 8.16 shows the first example of specifying a tabular condition:

Listing 8.16: Example of specifying a tabular condition

```
REPORT ZDATA_ACCESS.

*/Specifying a tabular condition
DATA: BEGIN OF RECORD,
     KUNNR TYPE KNA1-KUNNR,
     LAND1 TYPE KNA1-LAND1,
     NAME1 TYPE KNA1-NAME1,
     ORT01 TYPE KNA1-ORT01,
   END OF RECORD,
   MYTABLE LIKE TABLE OF RECORD.
RECORD-LAND1 = 'FR'.
RECORD-ORT01 = 'Paris'.
APPEND RECORD TO MYTABLE.
RECORD-LAND1 = 'GB'.
RECORD-ORT01 = 'London'.
APPEND RECORD TO MYTABLE.
```

```
SELECT KUNNR LAND1 NAME1 ORT01
INTO CORRESPONDING FIELDS OF RECORD
FROM KNA1
FOR ALL ENTRIES IN MYTABLE
WHERE LAND1 = MYTABLE-LAND1 AND ORT01 = MYTABLE-ORT01.
  WRITE: / RECORD-KUNNR, RECORD-LAND1, RECORD-NAME1, RECORD-
  ORT01.
ENDSELECT.
```

In Listing 8.16, the following tabular conditions are specified:

- The `LAND1` column contains FR and the `ORT01` column contains Paris.
- The `LAND1` column contains GB and the `ORT01` column contains London.

Figure 8.24 shows the result after specifying the tabular conditions:

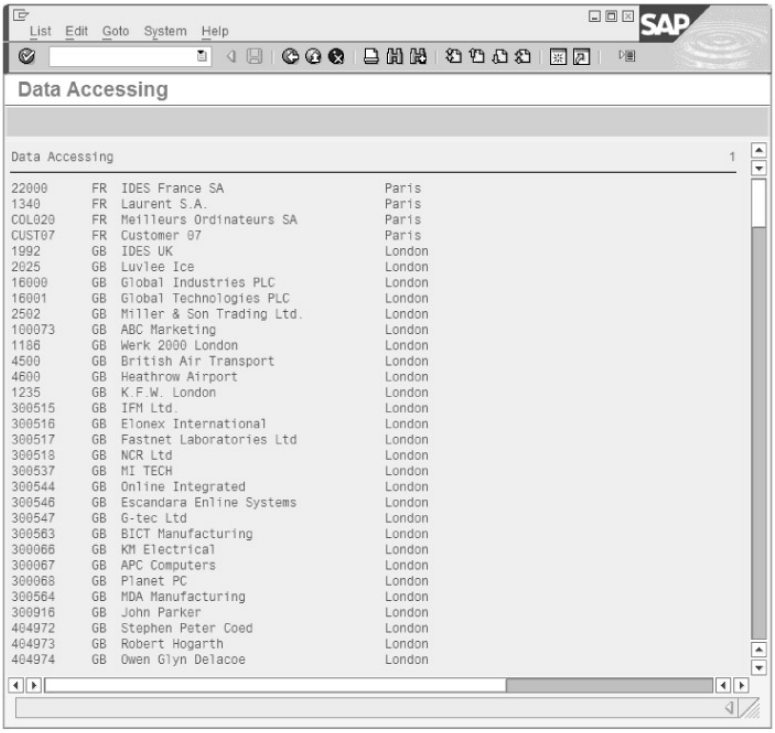


Figure 8.24: Displaying results after specifying the tabular conditions

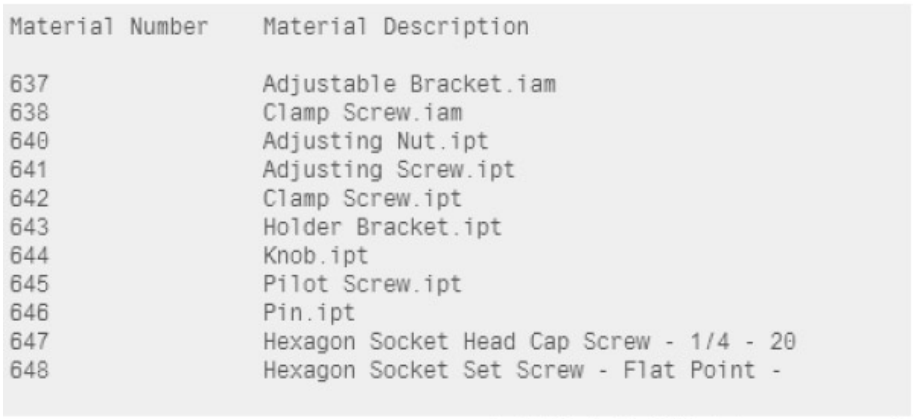
Listing 8.17 shows another example of specifying a tabular condition:

Listing 8.17: Specifying a tabular condition

```
REPORT ZDATA_ACCESS.

*/Specifying and using tabular condition

DATA: TABLE_A TYPE TABLE OF MARA,
       TABLE_B TYPE SORTED TABLE OF MAKT
                           WITH UNIQUE KEY TABLE LINE,
     KOG LIKE LINE OF TABLE_B.
SELECT MATNR ERNAM
INTO CORRESPONDING FIELDS OF TABLE TABLE_A
FROM MARA
WHERE ERNAM = 'RUDISILL'.
SELECT MATNR MAKTX

INTO CORRESPONDING FIELDS OF TABLE TABLE_B
```

```
FROM MAKT
FOR ALL ENTRIES IN TABLE_A
WHERE MATNR = TABLE_A-MATNR.
WRITE: 'Material Number', ' ', 'Material
Description'.
skip.
LOOP AT TABLE_B INTO KOG.
   AT NEW MATNR.
       WRITE: / KOG-MATNR.
   ENDAT.
WRITE: KOG-MAKTX.
ENDLOOP.
```

In Listing 8.17, the data of the MAKT database table is selected on the basis of the corresponding fields of the MARA table, where the value of the ERNAM field is RUDISILL. Figure 8.25 shows the output of Listing 8.17:



```
Material Number    Material Description

637                Adjustable Bracket.iam
638                Clamp Screw.iam
640                Adjusting Nut.ipt
641                Adjusting Screw.ipt
642                Clamp Screw.ipt
643                Holder Bracket.ipt
644                Knob.ipt
645                Pilot Screw.ipt
646                Pin.ipt
647                Hexagon Socket Head Cap Screw - 1/4 - 20
648                Hexagon Socket Set Screw - Flat Point -
```


Figure 8.25: Displaying results from multiple tables after specifying conditions

The *GROUP BY* Clause

The GROUP BY clause collects several lines from a database table into a single line. It allows you to categorize and summarize the lines of a database table on the basis of a single column or group of columns. Note that the columns used in the GROUP BY clause cannot be specified for aggregate functions in the SELECT statement. However, the columns that are not used in the GROUP BY clause can be included in the SELECT statement for aggregate functions.

You can specify the columns in the GROUP BY clause either statically or dynamically.

Now, let's explore these two ways of specifying columns in the GROUP BY clause, one by one.

Specifying Columns Statically

The following syntax is used to specify the columns in the GROUP BY clause statically:

```
SELECT <database_tab_lines> <s1> [AS <a 1>] <s 2>
[AS <a 2>] ...
         <agg> <sm> [AS <a m>] <agg> <s n> [AS <a n>] ...
...
GROUP BY <s1> <s 2>....
```

In this syntax, <s1>, <s2>...<sn> represents column names of a database table; <a1>, <a2>...<an> represent alias names for these column names; the <agg> expression represents aggregate functions; and <am> and <an> represent alias names for the columns that are involved in aggregate functions.

To use the GROUP BY clause, you must specify all the relevant columns in the SELECT statement. You should include only those column names in the GROUP BY clause whose values are repeated in the table. The alias names of the columns, specified in the SELECT statement, are not allowed in the GROUP BY clause.

All columns of the SELECT clause that are not listed in the GROUP BY clause must be included in aggregate functions.

This defines how the values stored in these columns are calculated when the lines are summarized.

Specifying Columns Dynamically

The following syntax is used to specify the columns in the GROUP BY clause dynamically:

```
... GROUP BY (<internal_tab>) ...
```

In this syntax, <internal_tab> represents an internal table with line type C and a maximum length of 72 characters.

Now, let's consider an example of using the GROUP BY clause, as shown in Listing 8.18:

Listing 8.18: An example of using the GROUP BY clause

```
REPORT ZDATA_ACCESS.

*/Using GROUP BY Clause
DATA: MATNR TYPE MARI-MATNR,
       MINIMUM TYPE P DECIMALS 2,
       MAXIMUM TYPE P DECIMALS 2.
SELECT MATNR MIN(MENGE) MAX(MENGE)
INTO (MATNR, MINIMUM, MAXIMUM)
FROM MARI
GROUP BY MATNR.
  WRITE: / MATNR, MINIMUM, MAXIMUM.
ENDSELECT.
```

In Listing 8.18, the data of the MARI database table is grouped based on the values of the MATNR field because the MATNR field is used in the GROUP BY clause. The minimum and maximum values of the MENGE field for each group are then obtained and placed in the summarized line, as shown in Figure 8.26:

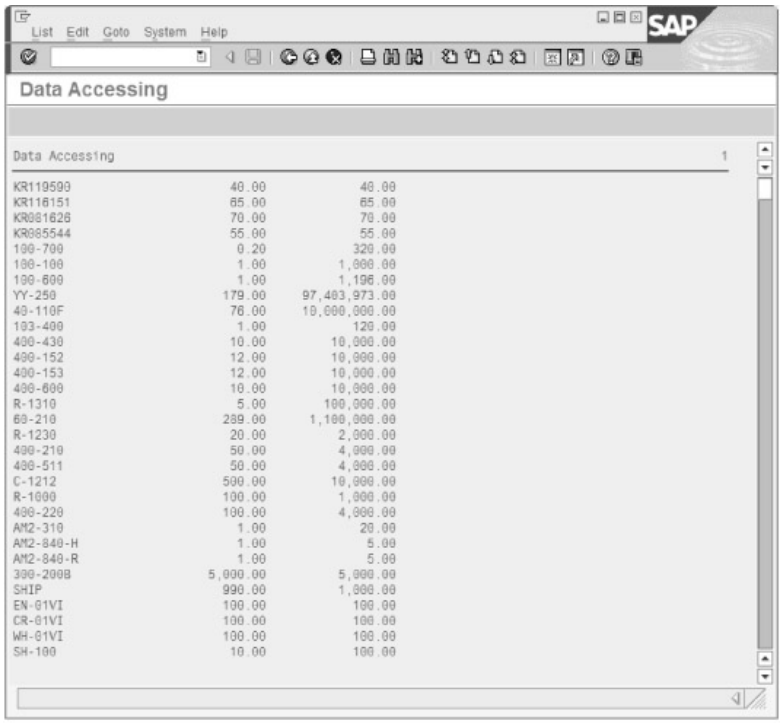


Figure 8.26: Showing the usage of the GROUP BY clause

The *HAVING* Clause

The HAVING clause is used to restrict the output to selected groups based on specified conditions. The HAVING clause can be used only in conjunction with the GROUP BY clause. The syntax to select lines is:

```
SELECT <database_tab_lines> <s1> [AS <a1>] <s2> [AS <a2>]
...
       <agg> <sm> [AS <am>] <agg> <sn> [AS <an>] ...
. . .
GROUP BY <s1> <s2> ....
HAVING <cond>.
```

In this syntax, the HAVING clause is used after the GROUP BY clause to specify one or more conditions in the <cond> expression. The fields specified in the conditions of the HAVING clause should be the same as those used in the SELECT clause. If you use an invalid column, a runtime error occurs.

You can also specify the aggregate expressions in the HAVING clause for the columns that do not appear in the GROUP BY clause. However, you cannot use aggregate expressions in the conditions specified in the WHERE clause. Moreover, in the HAVING clause, similar to the WHERE clause, the conditions can be specified in the form of an internal table with line type C and a length of 72 characters.

Listing 8.19 shows an example of using the HAVING clause:

Listing 8.19: Example of using the HAVING clause

```
REPORT ZDATA_ACCESS.

*/Using GROUP BY with HAVING Clause

DATA KOG TYPE MARI.
SELECT ZEILE
INTO KOG-ZEILE
FROM MARI
WHERE USNAM = 'BUROW'
GROUP BY ZEILE
HAVING SUM(MENGE) > 10.
    WRITE: / KOG-ZEILE.
ENDSELECT.
```

In Listing 8.19, the data of the MARI table is displayed only when:

- The value in the USNAM field is BUROW
- The data is grouped by the values of the ZEILE field
- The sum of the values in the MENGE field is greater than 10

Figure 8.27 shows the data of the MARI table after using the GROUP BY and HAVING clauses:

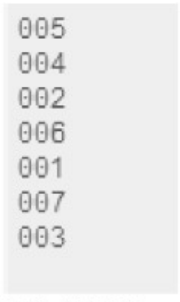


Figure 8.27: Displaying results after using the GROUP BY and HAVING clauses

The *ORDER BY* Clause

The ORDER BY clause sorts the result set according to the content of its columns. You can sort the selection by any column (not necessarily the primary key) and specify the columns either statically or dynamically.

The syntax to sort the selection set in ascending order by the primary key is:

```
SELECT <database_tab_lines> *
...
... ORDER BY PRIMARY KEY.
```

The sorting method, shown in this syntax, is possible only if you use an asterisk (*) in the SELECT clause to select all the columns. Moreover, it works only if you specify a single database table in the FROM clause. You cannot use views or joins because neither has a defined primary key.

The following syntax is used to sort the lines in the selection by specifying one or more columns in the ORDER BY clause:

```
SELECT ...
... ORDER BY <s1> [ASCENDING|DESCENDING]
<s2> [ASCENDING|DESCENDING] ...
```

This syntax shows that the lines are sorted by the <s1>, <s2>,... columns in the ORDER BY clause in either ASCENDING or DESCENDING order. The default is the ascending order. The sort order depends on the sequence in which you list the columns.

You can use either the names of the columns specified in the SELECT clause or their alias names. By using alias names for aggregate expressions, you can use them as sort fields.

The syntax to specify the columns in the ORDER BY clause dynamically is:

```
SELECT . . .
. . .
ORDER BY (<internal_tab>).
```

In this syntax, <internal_tab> is an internal table, with line type C and a maximum length of 72 characters, containing the column names <s1>, <s2>, ... <sn>.

Listing 8.20 shows an example of using the ORDER BY clause:

Listing 8.20: Example of using the ORDER BY clause

```
REPORT ZDATA_ACCESS.

*/Using ORDER BY with DESCENDING Clause

DATA: BEGIN OF KOG,
USNAM TYPE MARI-USNAM,
MAX TYPE I,
END OF KOG.
SELECT USNAM MAX(MENGE) AS MAXI
INTO CORRESPONDING FIELDS OF KOG
FROM MARI
GROUP BY USNAM
ORDER BY USNAM DESCENDING.
WRITE: / KOG-USNAM, '|', KOG-MAXI.
ENDSELECT.
```

In Listing 8.20, the lines of the MARI database table are grouped on the basis of the USNAM field. The SAP system finds the maximum value of the MENGE field for each group. The ORDER BY clause is then used to sort the USNAM field in descending order. It is to be noted that an alias, MAXI, is created to store the maximum value of the MENGE field. Figure 8.28 shows the output of Listing 8.20:

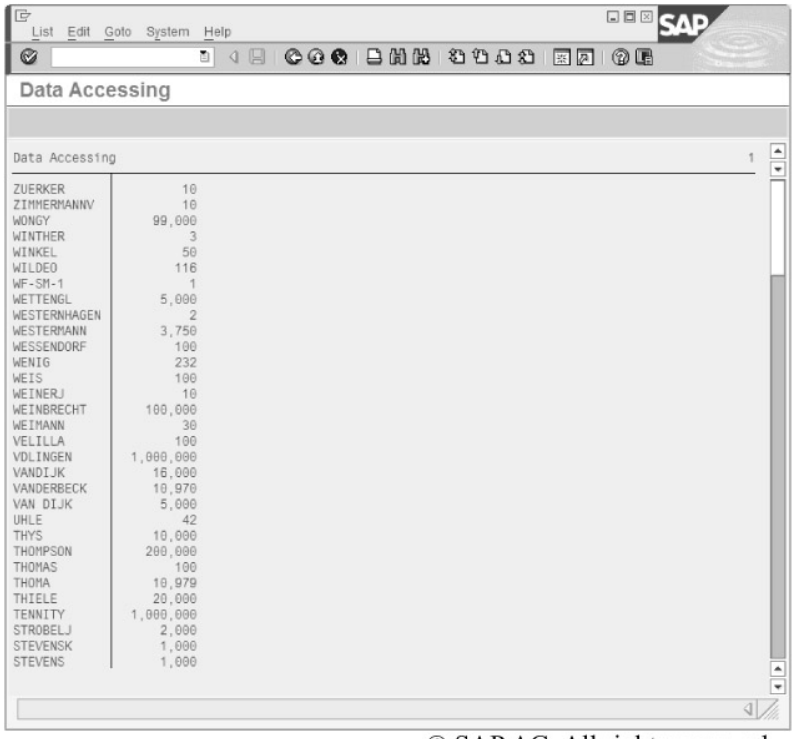


Figure 8.28: Displaying results after using the ORDER BY clause

Subqueries

A subquery is a type of the SELECT statement used within the WHERE clause of another SELECT statement, that is, a SELECT statement nested inside another SELECT statement. In this kind of nesting, the outer SELECT statement forms an outer query, while the inner SELECT statement forms an inner query or subquery. The following syntax is used to implement a subquery:

```
(SELECT <result>
FROM <source>
[WHERE <sql_condition>]
[GROUP BY <group_fields>]
[HAVING <group_condition>])
```

In this syntax, you can see that a subquery is provided in the parentheses ( ). A subquery can use all the clauses of the SELECT statement except the INTO and ORDER BY clauses. Also note that a subquery cannot be used in the ON condition of the FROM clause. The level of nesting of subqueries is fixed to 10, which means that the WHERE and the HAVING clauses of a SELECT statement can have 10 levels of nested SELECT statements. When a nested subquery in the WHERE clause uses the fields from the outer query, it is known as a correlated query.

A subquery can be specified by using the following logical expressions in the <sql_condition> condition of the WHERE clause:

- **Using the ALL, ANY, or SOME clause along with relational operators**—The syntax to specify the ALL, ANY, or SOME clause is:

  ```
  . . . col operator [ALL|ANY|SOME] subquery . . .
  ```

In this syntax, operator stands for a relational operator, such as =, <=, and >=. The result set can have single or multiple lines. The ALL clause is used when the comparison is evaluated to true for all the lines of a subquery, while the ANY or SOME clause is used when the comparison is evaluated to true for at least one line of the subquery. The equality operator (= or EQ), in conjunction with the ANY or SOME clause, has the same effect as the IN operator to check a value.

If the result set of the subquery contains only one line, the comparison can be carried out without the specification of the ALL, ANY, or SOME clause. However, if the result set for the subquery contains multiple lines, an unhandled exception occurs when the statement is executed.

- **Using the EXISTS or NOT EXISTS clause**—The syntax to specify the `EXISTS` or `NOT EXISTS` clause is:

```
... [NOT] EXISTS subquery ...
```

In this syntax, the `EXISTS` or `NOT EXISTS` clause is used when the result set of the subquery contains at least one line or no lines.

- **Using the IN or NOT IN clause**—The syntax to specify the `IN` or `NOT IN` clause is:

```
... col [NOT] IN subquery ...
```

In this syntax, the `IN` or `NOT IN` clause is used to check whether or not the value for the `col` column is stored in the subquery.

Examples of Subqueries

Listing 8.21 shows an example when the result set of a subquery contains only one line:

Listing 8.21: The result set of a subquery containing one line

```
REPORT ZDATA_ACCESS.

*/Creating subquery by using equal to (=) operator

DATA: Tab1 TYPE TABLE OF MAKZ,
       KOG LIKE LINE OF Tab1.
SELECT * INTO TABLE Tab1 FROM MAKZ
 WHERE MATNR = (select MATNR
                    FROM MAKV
                    WHERE MATNR = 'T-COP').
LOOP AT Tab1 INTO KOG.
 WRITE: / KOG-MATNR, KOG-WERKS, KOG-KUPPL, KOG-DATUB.
ENDLOOP.
```

In Listing 8.21, the subquery returns a single line to the outer query, where the `MATNR` field of the value of the MAKV table is T-COP. Consequently, the outer query displays all the lines of the MAKZ table, where the value of the `MATNR` field is T-COP, as shown in Figure 8.29:

T-COP	1000	T-COP	31.12.9999
T-COP	1000	T-COP1	31.12.9999
T-COP	1000	T-COP2	31.12.9999
T-COP	1000	T-COP3	31.12.9999
T-COP	1000	T-COP	31.12.9999
T-COP	1000	T-COP1	31.12.9999
T-COP	1000	T-COP2	31.12.9999
T-COP	1000	T-COP3	31.12.9999
T-COP	1000	T-COP	31.12.9999
T-COP	1000	T-COP1	31.12.9999
T-COP	1000	T-COP2	31.12.9999
T-COP	1000	T-COP3	31.12.9999

Figure 8.29: Displaying result set when the subquery returns a single line

Listing 8.22 shows a correlated subquery:

Listing 8.22: A correlated subquery

```
REPORT ZDATA_ACCESS.

*/Creating a subquery by using EXISTS keyword

DATA: Tab1 TYPE TABLE OF MAKZ,
       KOG LIKE LINE OF Tab1.
SELECT * INTO TABLE Tab1 FROM MAKZ
   WHERE EXISTS ( select *
                  FROM MAKV
                  WHERE MATNR = 'T-COP').
```

```
LOOP AT Tab1 INTO KOG.
 WRITE: / KOG-MATNR, KOG-WERKS, KOG-KUPPL, KOG-DATUB.

ENDLOOP.
```

Listing 8.22 shows that the outer query contains the EXISTS keyword in the WHERE clause, according to which only those records of the MAKZ table for which the MATNR field of the MAKV table contains the value T-COP are displayed, as shown in Figure 8.30:

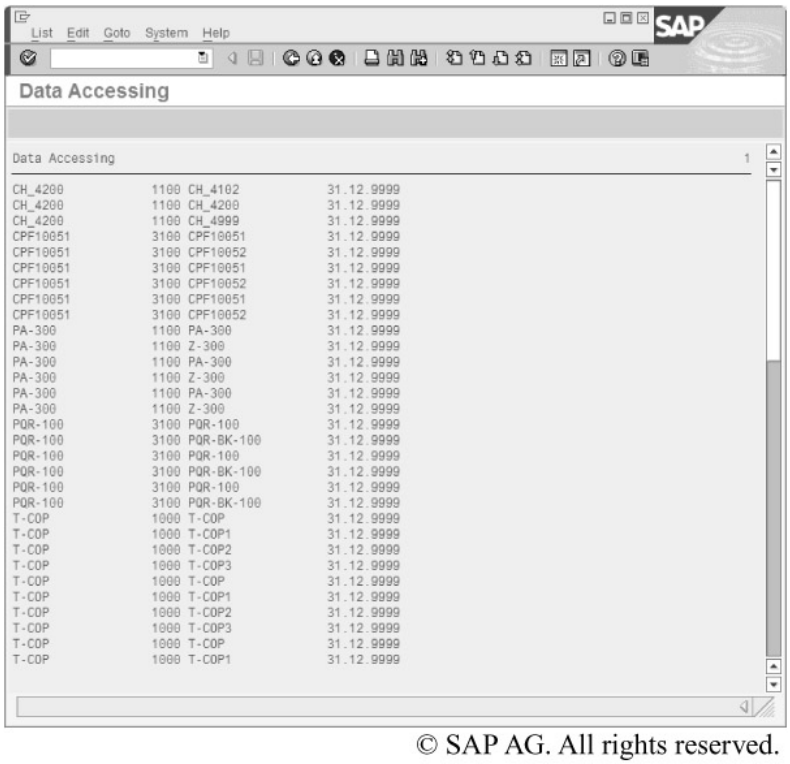


Figure 8.30: Displaying the result of the correlated subquery

Listing 8.23 shows an example when the outer query contains the NOT IN operator:

Listing 8.23: Example of an outer query containing the NOT IN operator

```
REPORT ZDATA_ACCESS.

*/Creating a subquery by using NOT IN

DATA: Tab1 TYPE TABLE OF MAKZ,
       KOG LIKE LINE OF Tab1.
SELECT * INTO TABLE Tab1 FROM MAKZ
    WHERE MATNR NOT IN (select MATNR
           FROM MAKV
           WHERE MATNR = 'T-COP' OR MATNR = 'PA-300').
LOOP AT Tab1 INTO KOG.
 WRITE: / KOG-MATNR, KOG-WERKS, KOG-KUPPL, KOG-DATUB.

ENDLOOP.
```

Listing 8.23 shows that the outer query contains the NOT IN operator in the WHERE clause. As a result, the outer query displays all the records of the MAKZ table where the value of the MATNR field of the MAKV table is not T-COP or PA-300. Figure 8.31 shows the output of Listing 8.23:

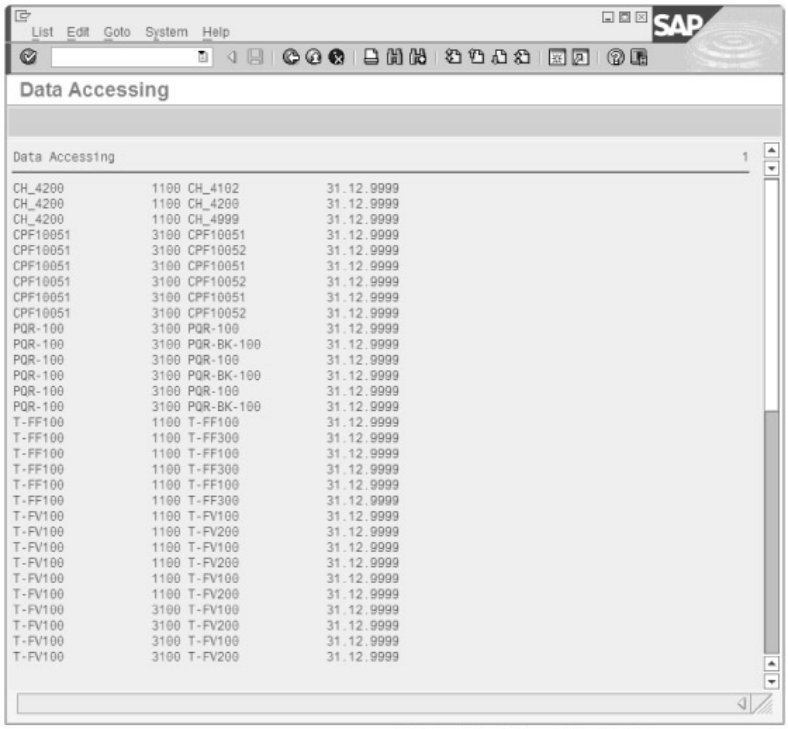


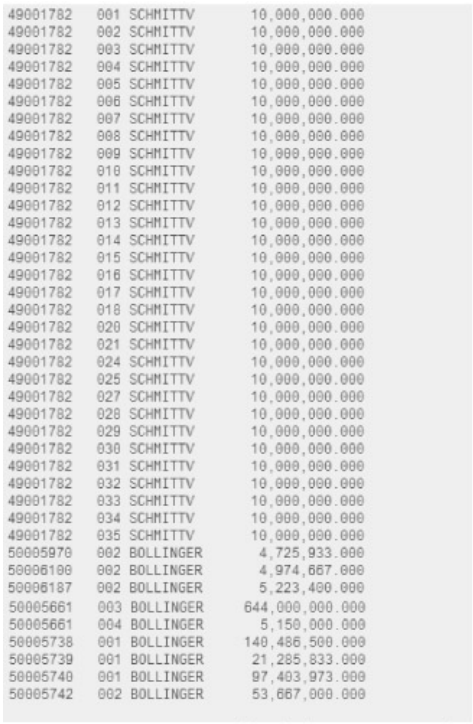
Figure 8.31: Displaying the result of a subquery when the outer query contains the NOT IN operator

Listing 8.24 shows an example of a subquery containing an aggregate function:

Listing 8.24: Example of a subquery containing an aggregate function

```
REPORT ZDATA_ACCESS.

*/Subquery on the basis of an aggregate function
Tables: MARI.
DATA: MARI_TAB TYPE TABLE OF MARI,
       KOG LIKE LINE OF MARI_TAB.
SELECT *
  INTO TABLE MARI_TAB FROM MARI
  WHERE MENGE > (select avg( MENGE ) FROM MARI
                    WHERE USNAM = 'SCHMITTV').
LOOP AT MARI_TAB INTO KOG.
  WRITE: / KOG-MBLNR, KOG-ZEILE, KOG-USNAM, KOG-MENGE.

ENDLOOP.
```

In Listing 8.24, the outer query shows all the records of the MARI table, where the values of the MENGE field are greater than their average value, and the value in the USNAM field is SCHMITTV. Figure 8.32 shows the result of Listing 8.24:



49001782	001	SCHMITTV	10,000,000.000
49001782	002	SCHMITTV	10,000,000.000
49001782	003	SCHMITTV	10,000,000.000
49001782	004	SCHMITTV	10,000,000.000
49001782	005	SCHMITTV	10,000,000.000
49001782	006	SCHMITTV	10,000,000.000
49001782	007	SCHMITTV	10,000,000.000
49001782	008	SCHMITTV	10,000,000.000
49001782	009	SCHMITTV	10,000,000.000
49001782	010	SCHMITTV	10,000,000.000
49001782	011	SCHMITTV	10,000,000.000
49001782	012	SCHMITTV	10,000,000.000
49001782	013	SCHMITTV	10,000,000.000
49001782	014	SCHMITTV	10,000,000.000
49001782	015	SCHMITTV	10,000,000.000
49001782	016	SCHMITTV	10,000,000.000
49001782	017	SCHMITTV	10,000,000.000
49001782	018	SCHMITTV	10,000,000.000
49001782	020	SCHMITTV	10,000,000.000
49001782	021	SCHMITTV	10,000,000.000
49001782	024	SCHMITTV	10,000,000.000
49001782	025	SCHMITTV	10,000,000.000
49001782	027	SCHMITTV	10,000,000.000
49001782	028	SCHMITTV	10,000,000.000
49001782	029	SCHMITTV	10,000,000.000
49001782	030	SCHMITTV	10,000,000.000
49001782	031	SCHMITTV	10,000,000.000
49001782	032	SCHMITTV	10,000,000.000
49001782	033	SCHMITTV	10,000,000.000
49001782	034	SCHMITTV	10,000,000.000
49001782	035	SCHMITTV	10,000,000.000
50005970	002	BOLLINGER	4,725,933.000
50006100	002	BOLLINGER	4,974,667.000
50006187	002	BOLLINGER	5,223,400.000
50005661	003	BOLLINGER	644,000,000.000
50005661	004	BOLLINGER	5,150,000.000
50005738	001	BOLLINGER	140,486,500.000
50005739	001	BOLLINGER	21,285,833.000
50005740	001	BOLLINGER	97,403,973.000
50005742	002	BOLLINGER	53,667,000.000

Figure 8.32: Displaying the result when an aggregate function is used in a subquery

Inserting Data into a Database Table

In Open SQL, the `INSERT` statement is used to insert one or more lines into a database table. The following syntax shows the use of the `INSERT` statement:

```
INSERT INTO <target_database_tab> <database_tab_lines>.
```

In this syntax, `<target_database_tab>` represents a database table and `<database_tab_lines>` represents the lines that have to be inserted in the table. You can specify the `<target_database_tab>` database table either statically or dynamically.

The syntax to specify a database table statically is:

```
INSERT INTO <target_database_tab> [CLIENT SPECIFIED]
<database_tab_lines>.
```

The syntax to specify a database table dynamically is:

```
INSERT INTO <target_database_tab>) [CLIENT SPECIFIED]
<database_tab_lines>.
```

In both syntaxes, the `<target_database_tab>` expression represents the name of a database table defined in the ABAP Dictionary. The `CLIENT SPECIFIED` clause is used to disable automatic client-handling feature of Open SQL.

The syntax to insert a row of a single line into a database table is:

```
INSERT INTO <target_database_tab> VALUES <work_area>.
```

In this syntax, `<work_area>` is a work area created for the `<target_database_tab>` database table. The `<work_area>` work area has the same len gth and alignment as the line structure of the `<target_database_tab>` table.

If the database table does not contain a line with the same primary key as specified in the work area, the operation is completed successfully and the value of the `SY-SUBRC` variable is set to 0. Otherwise, the line is not inserted and the value of the `SY-SUBRC` variable is set to 4. The syntax to insert lines individually by using the `INSERT` statement is:

```
INSERT <target_database_tab> FROM <work_area >.
```

In this syntax, the `FROM` clause is used instead of the `VALUES` clause, allowing you to insert data without using the `INTO`

clause. Another way to write the INSERT statement is as follows:

```
INSERT <target_database_tab>.
```

In this syntax, the INSERT statement inserts a line into the <target_database_tab> database table by using a table work area. The table work area is defined by using the TABLES statement. Note that the name of the table work area is the same as that of the database table. Note that in this case, you cannot specify the name of a data base table dynamically.

The following syntax is used to insert several lines into a database table:

```
INSERT <target_database_tab> FROM TABLE <internal_tab>
[ACCEPTING DUPLICATE KEYS].
```

In this syntax, lines from the <internal_tab> internal table are transferred to a database table, <target_database_tab>. The value of the SY-SUBRC variable is set to zero when all the lines have been transferred successfully. However, a runtime error occurs if it fails to transfer one or more lines because of duplicate entries in the same fields. This runtime error can be prevented by using the ACCEPTING DUPLICATE KEYS clause. When the ACCEPTING DUPLICATE KEYS clause is used, the lines that can cause runtime errors are discarded and the value of the SY-SUBRC variable is set to 4.

Note The SY-DBCNT system variable displays the number of lines that are inserted into the database table.

Note Use an internal table with the INSERT statement to insert a large number of lines into a database table rather than inserting one line at a time.

Examples of Data Insertion

In this section, all the listings use the ZEMPLOYEE_DATA table, which is a user-defined database table created in the ABAP Dictionary. The ZEMPLOYEE_DATA table initially has 10 lines, as shown in Figure 8.33:

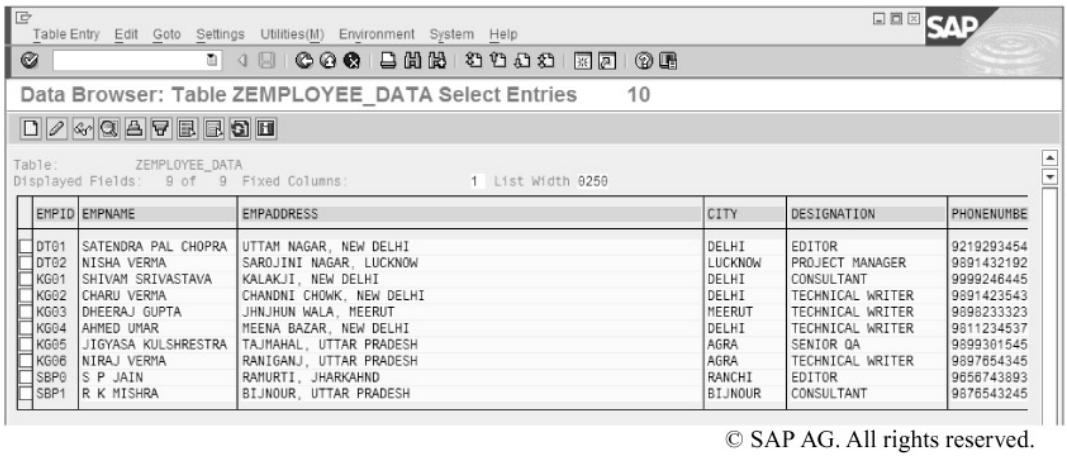


Figure 8.33: Displaying the lines stored in the ZEMPLOYEE_DATA table

Now, let's insert some lines into the ZEMPLOYEE_DATA table, as shown in Listing 8.25:

Listing 8.25: Inserting Lines in the ZEMPLOYEE_DATA table

```
REPORT ZINSERT_DATA.

*/Inserting three lines into the ZINSERT_DATA table

TABLES ZEMPLOYEE_DATA.
DATA KOG TYPE ZEMPLOYEE_DATA.
KOG-EMPID = 'KG07'.
KOG-EMPNAME = 'MEHTAB ALAM'.
KOG-EMPADDRESS = 'MUKHNANDAN VIHAR, PATNA'.
KOG-CITY = 'PATNA'.
KOG-DESIGNATION = 'TECHNICAL WRITER'.
KOG-PHONENUMBER = '9981145622'.
```

```
KOG-SALARY = 13000.
KOG-JOININGDATE = '20062006'.
KOG-MAILID = 'MEHTAB.ALAM@YAHOO.COM'.
INSERT INTO ZEMPLOYEE_DATA VALUES KOG.
KOG-EMPID = 'KG08'.
KOG-EMPNAME = 'VIKASH SUMAN'.
KOG-EMPADDRESS = 'BHARATPUR, PATNA'.

KOG-CITY = 'PATNA'.
KOG-DESIGNATION = 'TECHNICAL WRITER'.
KOG-PHONENUMBER = '9981718812'.
KOG-SALARY = 13000.
KOG-JOININGDATE = '20062006'.
KOG-MAILID = 'VIKASH.SUMAN@YAHOO.COM'.
INSERT ZEMPLOYEE_DATA FROM KOG.
KOG-EMPID = 'KG09'.
KOG-EMPNAME = 'RAM KUMAR'.
KOG-EMPADDRESS = 'CHANDAN NAGAR, PATNA'.
KOG-CITY = 'PATNA'.
KOG-DESIGNATION = 'TECHNICAL WRITER'.
KOG-PHONENUMBER = '9911424281'.
KOG-SALARY = 13500.
KOG-JOININGDATE = '20060723'.
KOG-MAILID = 'RAM.KUMAR@YAHOO.COM'.
INSERT ZEMPLOYEE_DATA.
```

Listing 8.25 shows the insertion of three lines into the KOG work area, which are then inserted into the table ZEMPLOYEE_DATA. It must be noted that each line is first inserted in the work area and then into the table. This process is repeated for each individual insertion. In this case, this process is repeated three times because we have inserted three lines in the ZEMPLOYEE_DATA table. As shown in Listing 8.25, any of the following syntaxes can be used to insert lines into a table:

```
INSERT INTO ZEMPLOYEE_DATA VALUES KOG.
```

or

```
INSERT ZEMPLOYEE_DATA FROM KOG.
```

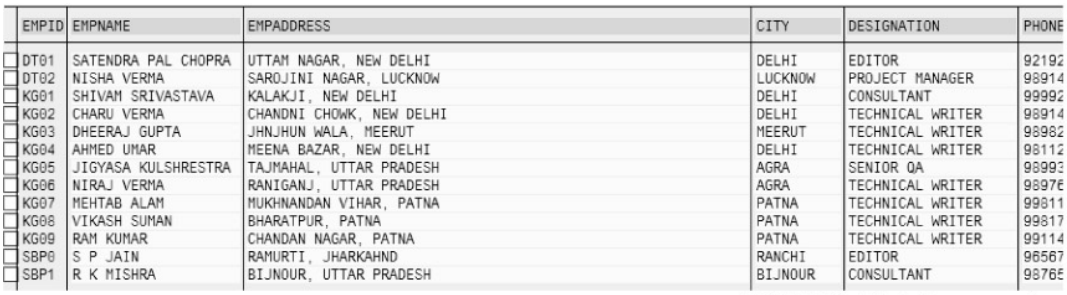
or

```
INSERT ZEMPLOYEE_DATA
```

Listing 8.25 shows the application of all these. three syntaxes to insert lines into the ZEMPLOYEE_DATA table.

Figure 8.34 shows the output of Listing 8.25:

In Figure 8.34, you can see that the lines corresponding to the values MEHTAB ALAM, VIKASH SUMAN, and RAM KUMAR in the EMPNAME field are added to the ZEMPLOYEE_DATA table.



EMPID	EMPNAME	EMPADDRESS	CITY	DESIGNATION	PHONE
DT01	SATENDRA PAL CHOPRA	UTTAM NAGAR, NEW DELHI	DELHI	EDITOR	92192
DT02	NISHA VERMA	SAROJINI NAGAR, LUCKNOW	LUCKNOW	PROJECT MANAGER	98914
KG01	SHIVAM SRIVASTAVA	KALAKJI, NEW DELHI	DELHI	CONSULTANT	99992
KG02	CHARU VERMA	CHANDNI CHOWK, NEW DELHI	DELHI	TECHNICAL WRITER	98914
KG03	DHEERAJ GUPTA	JHNJHUN WALA, MEERUT	MEERUT	TECHNICAL WRITER	98982
KG04	AHMED UMAR	MEENA BAZAR, NEW DELHI	DELHI	TECHNICAL WRITER	98112
KG05	JIGYASA KULSHRESTRA	TAJMAHAL, UTTAR PRADESH	AGRA	SENIOR QA	98993
KG06	NIRAJ VERMA	RANIGANJ, UTTAR PRADESH	AGRA	TECHNICAL WRITER	98976
KG07	MEHTAB ALAM	MUKHNANDAN VIHAR, PATNA	PATNA	TECHNICAL WRITER	99811
KG08	VIKASH SUMAN	BHARATPUR, PATNA	PATNA	TECHNICAL WRITER	99817
KG09	RAM KUMAR	CHANDAN NAGAR, PATNA	PATNA	TECHNICAL WRITER	99114
SBP0	S P JAIN	RAMURTI, JHARKAHND	RANCHI	EDITOR	96567
SBP1	R K MISHRA	BIJNOUR, UTTAR PRADESH	BIJNOUR	CONSULTANT	98765

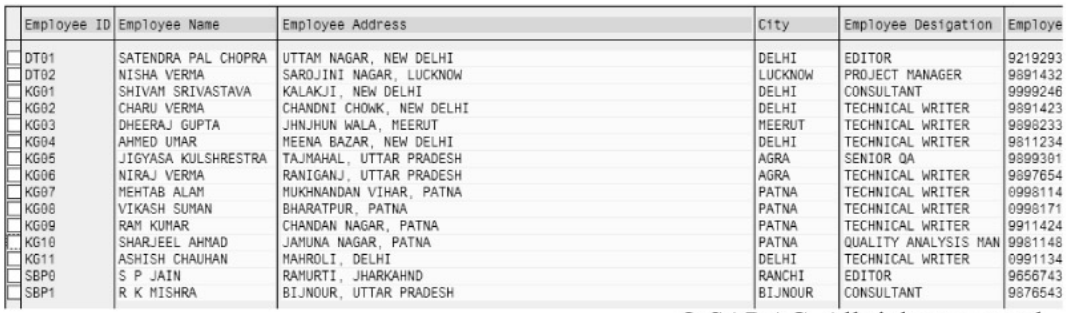
Figure 8.34: Displaying three more lines inserted into the ZEMPLOYEE_DATA table

Listing 8.26 shows how to transfer data from an internal table to a database table:

Listing 8.26: Transferring data from an internal table to a database table

```
REPORT ZDATA_ACCESS.
*/Inserting two lines into the ZEMPLOYEE_DATA table with the
help of an internal table-MYTABLE
DATA: MYTABLE TYPE HASHED TABLE OF ZEMPLOYEE_DATA
WITH UNIQUE KEY EMPID,
KOG LIKE LINE OF MYTABLE.

KOG-EMPID = 'KG10'.
KOG-EMPNAME = 'SHARJEEL AHMAD'.
KOG-EMPADDRESS = 'JAMUNA NAGAR, PATNA'.
KOG-CITY = 'PATNA'.
KOG-DESIGNATION = 'QUALITY ANALYSIS MANAGER'.
KOG-PHONENUMBER = '9981148113'.
KOG-SALARY = 30000.
KOG-JOININGDATE = '20070506'.
KOG-MAILID = 'SHARJEEL.AHMAD@YAHOO.COM'.
INSERT KOG INTO TABLE MYTABLE.
INSERT INTO ZEMPLOYEE_DATA VALUES KOG.

KOG-EMPID = 'KG11'.
KOG-EMPNAME = 'ASHISH CHAUHAN'.
KOG-EMPADDRESS = 'MAHROLI, DELHI'.
KOG-CITY = 'DELHI'.
KOG-DESIGNATION = 'TECHNICAL WRITER'.
KOG-PHONENUMBER = '9911343421'.
KOG-SALARY = 13000.
KOG-JOININGDATE = '20081001'.
KOG-MAILID = 'ASHISH.CHAUHAN@YAHOO.COM'.
INSERT KOG INTO TABLE MYTABLE.

INSERT ZEMPLOYEE_DATA FROM TABLE MYTABLE ACCEPTING DUPLICATE
KEYS.

IF SY-SUBRC = 0.
...
ELSEIF SY-SUBRC = 4.
...
ENDIF.
```

Listing 8.26 inserts two lines together into the KOG work area, which are then inserted into the MYTABLE internal table. Finally, the content of the MYTABLE internal table is inserted into the ZEMPLOYEE_DATA database table. The value of the SY-SUBRC variable is checked with the help of the IF...ELSEIF construct to verify whether the operation has been successful. Figure 8.35 shows the result of Listing 8.26:



Employee ID	Employee Name	Employee Address	City	Employee Desigation	Employe
DT01	SATENDRA PAL CHOPRA	UTTAM NAGAR, NEW DELHI	DELHI	EDITOR	9219293
DT02	NISHA VERMA	SAROJINI NAGAR, LUCKNOW	LUCKNOW	PROJECT MANAGER	9891432
KG01	SHIVAM SRIVASTAVA	KALAKJI, NEW DELHI	DELHI	CONSULTANT	9999246
KG02	CHARU VERMA	CHANDNI CHOWK, NEW DELHI	DELHI	TECHNICAL WRITER	9891423
KG03	DHEERAJ GUPTA	JHNJHUN WALA, MEERUT	MEERUT	TECHNICAL WRITER	9898233
KG04	AHMED UMAR	MEENA BAZAR, NEW DELHI	DELHI	TECHNICAL WRITER	9811234
KG05	JIGYASA KULSHRESTRA	TAJMAHAL, UTTAR PRADESH	AGRA	SENIOR QA	9899301
KG06	NIRAJ VERMA	RANIGANJ, UTTAR PRADESH	AGRA	TECHNICAL WRITER	9897654
KG07	MEHTAB ALAM	MUKHNANDAN VIHAR, PATNA	PATNA	TECHNICAL WRITER	0998114
KG08	VIKASH SUMAN	BHARATPUR, PATNA	PATNA	TECHNICAL WRITER	0998171
KG09	RAM KUMAR	CHANDAN NAGAR, PATNA	PATNA	TECHNICAL WRITER	9911424
KG10	SHARJEEL AHMAD	JAMUNA NAGAR, PATNA	PATNA	QUALITY ANALYSIS MAN	9981148
KG11	ASHISH CHAUHAN	MAHROLI, DELHI	DELHI	TECHNICAL WRITER	0991134
SBP0	S P JAIN	RAMURTI, JHARKAHND	RANCHI	EDITOR	9656743
SBP1	R K MISHRA	BIJNOUR, UTTAR PRADESH	BIJNOUR	CONSULTANT	9876543

Figure 8.35: Displaying two more lines inserted in the ZEMPLOYEE_DATA table

In Figure 8.35, you can see that the lines corresponding to the values "SHARJEEL AHMAD" and "ASHISH CHAUHAN" in the Employee Name field are newly added in the ZEMPLOYEE_DATA table.

Updating Data in a Database Table

The UPDATE statement of Open SQL used to update the data in a database table is:

```
UPDATE <target_database_tab> <database_tab_lines>.
```

In this syntax, <target_database_tab> represents a database table and <database_tab_lines> represents the lines that have to be updated in the table. You can specify a database table either statically or dynamically.

The syntax to update a database table, which is specified statically, is:

```
UPDATE <target_database_tab> [CLIENT SPECIFIED] <database_
tab_lines>.
```

The syntax to update a database table, which is specified dynamically, is:

```
UPDATE (<target_database_tab>) [CLIENT SPECIFIED] <database_
tab_lines>.
```

In both syntaxes, the <target_database_tab> field contains the name of a database table defined in the ABAP Dictionary. The CLIENT SPECIFIED clause is used to disable the automatic client-handling feature of Open SQL.

The following syntax is used to change certain columns in the database table:

```
UPDATE <target_database_tab> SET <si> ... [WHERE <cond>].
```

In this syntax, the WHERE clause is used to specify a condition on the basis of which the table lines are updated. If you do not specify a WHERE clause, all the lines are changed. The <s1>, <s2>...<sn> expressions are different SET statements that can be replaced by any one of the expressions listed in Table 8.8:

Table 8.8: Three different expressions of the SET statement

Expression	Description
<s i> = <f>	Sets the value of the <si> column to the value <f> for all lines of the result set
<s i> = <s i> + <f>	Sets the value of the <si> column, which is increased by the value of <f> for all lines of result set
<s i> = <s i> - <f>	Sets the value of the <si> column, which is decreased by the value of <f> for all lines of result set

In Table 8.8, <s i> stands for the <s 1>, <s 2>...<s n> field names, and <f> represents a data object or a column of the database table.

Note If you use the SET statements, you cannot specify the database table dynamically.

The syntax is used to overwrite a single line in a database table with the content of a work area:

```
UPDATE <target_database_tab> FROM <work_area> .
```

The content of the work area, <work_area>, overwrites the content of a line in the <target_database_tab> database table that has the same primary key. The work area <work_area> must be a data object with at least the same length and alignment as the line structure of the database table. The data is placed in the database table according to the line structure of the table and regardless of the structure of the work area. It is a good idea to define the work area with reference to the structure of the database table.

If the database table contains a line with the same primary key as specified in the work area, the operation is completed successfully and the value of the SY-SUBRC variable is set to 0; otherwise, the line is not inserted and the value of the SY-SUBRC variable is set to 4.

A shortened form of using the UPDATE statement is as follows:

```
TABLES <target_database_tab>
UPDATE <target_database_tab>.
```

In this syntax, the content of the <target_database_tab> database table is updated or overwritten by the content of a table work area defined by using the TABLES statement. Note that the name of the table work area and the name of the database table must be the same. However, it is not possible to specify the name of the database table dynamically.

The following is the syntax used to overwrite several lines in a database table with the content of an internal table:

```
UPDATE <target_database_tab> FROM TABLE <internal_tab>.
```

This syntax shows that the content of the internal table <internal_tab> overwrites the lines in the <target_database_tab> database table on the basis of the same key.

Sometimes, a record of an internal table may not get updated in the database table because the database table does not contain the corresponding record. In such situations, the entire operation of updating records is not terminated and the system continues to process the next record of the internal table.

If all the lines of the internal table are processed, the value of the SY-SUBRC variable is set to 0; otherwise, it is set to 4. If the internal table is empty, the values of the SY-SUBRC and SY-DBCNT variables are set to 0.

As a best practice, use an internal table in cases where you need to update a large number of lines in a database table.

Examples of Updating Data in Tables

Listing 8.27 shows how to update the values of columns in a table:

Listing 8.27: Updating the values of two columns of a table

```
REPORT ZDATA_ACCESS.

*/Updating the values of EMPADDRESS and CITY are updated where
EMPID is KG01
UPDATE ZEMPLOYEE_DATA SET EMPADDRESS = 'SAROJINI NAGAR,
LUCKNOW' CITY = 'LUCKNOW'
WHERE EMPID = 'KG01'.
```

In Listing 8.27, the value of the EMPADDRESS and CITY fields are updated. The value of the EMPADDRESS field is set to SAROJINI NAGAR, LUCKNOW and that of the CITY field is set to LUCKNOW. The value of the EMPID field is KG01. Figure 8.36 shows the updated record, where the value of the EMPID field is KG01:

Employee ID	Employee Name	Employee Address	City	Employee Designation	Employee
DT01	SATENDRA PAL CHOPRA	UTTAM NAGAR, NEW DELHI	DELHI	EDITOR	92192934
DT02	NISHA VERMA	SAROJINI NAGAR, LUCKNOW	LUCKNOW	PROJECT MANAGER	98914321
KG01	SHIVAM SRIVASTAVA	SAROJINI NAGAR, LUCKNOW	LUCKNOW	CONSULTANT	99992464
KG02	CHARU VERMA	CHANDNI CHOWK, NEW DELHI	DELHI	TECHNICAL WRITER	98914235
KG03	DHEERAJ GUPTA	JHNJHUN WALA, MEERUT	MEERUT	TECHNICAL WRITER	98982333
KG04	AHMED UMAR	MEENA BAZAR, NEW DELHI	DELHI	TECHNICAL WRITER	98112345
KG05	JIGYASA KULSHRESTRA	TAJMAHAL, UTTAR PRADESH	AGRA	SENIOR QA	98993015
KG06	NIRAJ VERMA	RANIGANJ, UTTAR PRADESH	AGRA	TECHNICAL WRITER	98976543
KG07	MEHTAB ALAM	MUKHNANDAN VIHAR, PATNA	PATNA	TECHNICAL WRITER	09981145
KG08	VIKASH SUMAN	BHARATPUR, PATNA	PATNA	TECHNICAL WRITER	09981718
KG09	RAM KUMAR	CHANDAN NAGAR, PATNA	PATNA	TECHNICAL WRITER	99114242
KG10	SHARJEEL AHMAD	JAMUNA NAGAR, PATNA	PATNA	QUALITY ANALYSIS MAN	99811481
KG11	ASHISH CHAUHAN	MAHROLI, DELHI	DELHI	TECHNICAL WRITER	09911343
SBP0	S P JAIN	RAMURTI, JHARKAHND	RANCHI	EDITOR	96567438
SBP1	R K MISHRA	BIJNOUR, UTTAR PRADESH	BIJNOUR	CONSULTANT	98765432

Figure 8.36: Displaying the updated record where EMPID is KG01

Listing 8.28 shows how to update the field values by using the MOVE and UPDATE statements:

Listing 8.28: Using the MOVE and UPDATE statements

```
REPORT ZDATA_ACCESS.

*/Updating the records of a table by using the MOVE statement

TABLES ZEMPLOYEE_DATA.
DATA KOG TYPE ZEMPLOYEE_DATA.
MOVE 'SBP0' TO KOG-EMPID.
MOVE 'A K SINHA' TO KOG-EMPNAME.
MOVE 'SAROJINI NAGAR, DELHI' TO KOG-EMPADDRESS.
MOVE 'DELHI' TO KOG-CITY.
UPDATE ZEMPLOYEE_DATA FROM KOG.
MOVE 'DT02' TO ZEMPLOYEE_DATA-EMPID.
MOVE 'SOMYA SRIVASTAVA' TO ZEMPLOYEE_DATA-EMPNAME.
MOVE 'SITA RAM BAZAR, DELHI' TO ZEMPLOYEE_DATA-EMPADDRESS.
MOVE 'DELHI' TO ZEMPLOYEE_DATA-CITY.
UPDATE ZEMPLOYEE_DATA.
```

In Listing 8.28, the values in the EMPNAME, EMPADDRESS, and CITY fields of the ZEMPLOYEE_DATA table are updated by the MOVE statement. The value of the EMPNAME field is updated to "SOMYA SRIVASTAVA", that of the EMPADDRESS field to "SITA RAM BAZAR, DELHI", and that of the CITY field to "DELHI", where EMPID is "DT02". In addition, the value of the EMPNAME field is updated to "A K SINHA", that of the EMPADDRESS field to "SAROJINI NAGAR, DELHI", and that of the CITY field to "DELHI", where EMPID is "SBP0". Figure 8.37 shows the updated records for the EMPID field:

Employee ID	Employee Name	Employee Address	City	Employee Desigation	Employee
DT01	SATENDRA PAL CHOPRA	UTTAM NAGAR, NEW DELHI	DELHI	EDITOR	921929345
DT02	SOMYA SRIVASTAVA	SITA RAM BAZAR, DELHI	DELHI		000000000
KG01	SHIVAM SRIVASTAVA	SAROJINI NAGAR, LUCKNOW	LUCKNOW	CONSULTANT	999924644
KG02	CHARU VERMA	CHANDNI CHOWK, NEW DELHI	DELHI	TECHNICAL WRITER	989142354
KG03	DHEERAJ GUPTA	JHNJHUN WALA, MEERUT	MEERUT	TECHNICAL WRITER	989823332
KG04	AHMED UMAR	MEENA BAZAR, NEW DELHI	DELHI	TECHNICAL WRITER	981123453
KG05	JIGYASA KULSHRESTRA	TAJMAHAL, UTTAR PRADESH	AGRA	SENIOR QA	989930154
KG06	NIRAJ VERMA	RANIGANJ, UTTAR PRADESH	AGRA	TECHNICAL WRITER	989765434
KG07	MEHTAB ALAM	MUKHNANDAN VIHAR, PATNA	PATNA	TECHNICAL WRITER	999811456
KG08	VIKASH SUMAN	BHARATPUR, PATNA	PATNA	TECHNICAL WRITER	999817188
KG09	RAM KUMAR	CHANDAN NAGAR, PATNA	PATNA	TECHNICAL WRITER	991142428
KG10	SHARJEEL AHMAD	JAMUNA NAGAR, PATNA	PATNA	QUALITY ANALYSIS MAN	998114811
KG11	ASHISH CHAUHAN	MAHROLI, DELHI	DELHI	TECHNICAL WRITER	999113434
SBP0	A K SINHA	SAROJINI NAGAR, DELHI	DELHI		000000000
SBP1	R K MISHRA	BIJNOUR, UTTAR PRADESH	BIJNOUR	CONSULTANT	987654324

Figure 8.37: Displaying updated records for employee ID DT02 and SBP0

In Figure 8.37, you can see the updated values in the record, where Employee ID is DT02 or SBP0. It must be noted that when values are updated in the specified fields of the ZEMPLOYEE_DATA table, the other fields whose values are not specified in the MOVE statement remain blank.

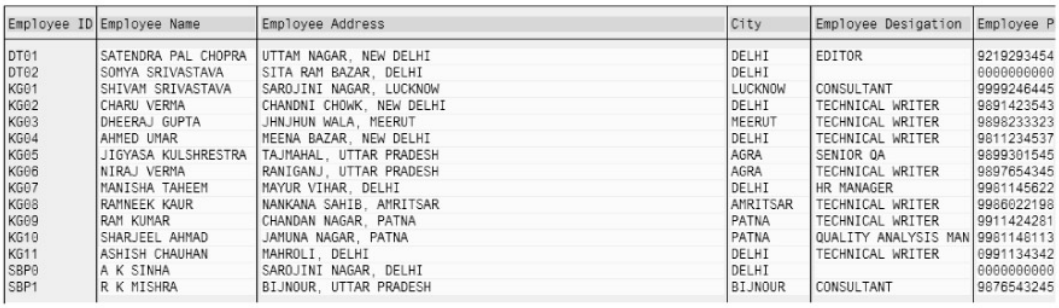
Listing 8.29 shows how a database table is updated by using the lines of a hashed internal table:

Listing 8.29: Updating a database table by using a hashed internal table

```
REPORT ZDATA_ACCESS.

*/Updating the values of a table, using a hashed table

DATA: MYTABLE TYPE HASHED TABLE OF ZEMPLOYEE_DATA
WITH UNIQUE KEY EMPID,
KOG LIKE LINE OF MYTABLE.
KOG-EMPID = 'KG07'.
KOG-EMPNAME = 'MANISHA TAHEEM'.
KOG-EMPADDRESS = 'MAYUR VIHAR, DELHI'.

KOG-CITY = 'DELHI'.
KOG-DESIGNATION = 'HR MANAGER'.
KOG-PHONENUMBER = '9981145622'.
KOG-SALARY = 30000.
KOG-JOININGDATE = '20080101'.
KOG-MAILID = 'MANISHA.TAHEEM@YAHOO.COM'.
INSERT KOG INTO TABLE MYTABLE.
KOG-EMPID = 'KG08'.
KOG-EMPNAME = 'RAMNEEK KAUR'.
KOG-EMPADDRESS = 'NANKANA SAHIB, AMRITSAR'.
KOG-CITY = 'AMRITSAR'.
KOG-DESIGNATION = 'TECHNICAL WRITER'.
KOG-PHONENUMBER = '9986022198'.
KOG-SALARY = 10000.
KOG-JOININGDATE = '20080911'.
KOG-MAILID = 'RAMNEEK.KAUR@GMAIL.COM'.

INSERT KOG INTO TABLE MYTABLE.

UPDATE ZEMPLOYEE_DATA FROM TABLE MYTABLE.
```

In Listing 8.29, two lines of the ZEMPLOYEE_DATA table are updated. First, we insert two lines together into the KOG work area, which are then inserted into the MYTABLE internal table. Finally, the content of the MYTABLE internal table modifies the lines of the ZEMPLOYEE_DATA database table, where EMPID is KG07 or KG08. Figure 8.38 shows the result of

Listing 8.29:



Employee ID	Employee Name	Employee Address	City	Employee Desigation	Employee P
DT01	SATENDRA PAL CHOPRA	UTTAM NAGAR, NEW DELHI	DELHI	EDITOR	9219293454
DT02	SOMYA SRIVASTAVA	SITA RAM BAZAR, DELHI	DELHI		0000000000
KG01	SHIVAM SRIVASTAVA	SAROJINI NAGAR, LUCKNOW	LUCKNOW	CONSULTANT	9999246445
KG02	CHARU VERMA	CHANDNI CHOWK, NEW DELHI	DELHI	TECHNICAL WRITER	9891423543
KG03	DHEERAJ GUPTA	JHNJHUN WALA, MEERUT	MEERUT	TECHNICAL WRITER	9898233323
KG04	AHMED UMAR	MEENA BAZAR, NEW DELHI	DELHI	TECHNICAL WRITER	9811234537
KG05	JIGYASA KULSHRESTRA	TAJMAHAL, UTTAR PRADESH	AGRA	SENIOR QA	9899301545
KG06	NIRAJ VERMA	RANIGANJ, UTTAR PRADESH	AGRA	TECHNICAL WRITER	9897654345
KG07	MANISHA TAHEEM	MAYUR VIHAR, DELHI	DELHI	HR MANAGER	9981145622
KG08	RAMNEEK KAUR	NANKANA SAHIB, AMRITSAR	AMRITSAR	TECHNICAL WRITER	9986022198
KG09	RAM KUMAR	CHANDAN NAGAR, PATNA	PATNA	TECHNICAL WRITER	9911424281
KG10	SHARJEEL AHMAD	JAMUNA NAGAR, PATNA	PATNA	QUALITY ANALYSIS MAN	9981148113
KG11	ASHISH CHAUHAN	MAHROLI, DELHI	DELHI	TECHNICAL WRITER	0991134342
SBP0	A K SINHA	SAROJINI NAGAR, DELHI	DELHI		0000000000
SBP1	R K MISHRA	BIJNOUR, UTTAR PRADESH	BIJNOUR	CONSULTANT	9876543245

Figure 8.38: Displaying the two updated lines in the ZEMPLOYEE_DATA table

Deleting the Data From a Database Table

In Open SQL, the `DELETE` statement is used to delete one or more lines from a database table. The following syntax shows the use of the `DELETE` statement:

```
DELETE [FROM] <target_database_tab> <database_tab_lines>.
```

In the preceding syntax, `<target_database_tab>` represents a database table and `<database_tab_lines>` represents the lines that have to be deleted from the table. You can specify the `<target_database_tab>` database table either statically or dynamically.

The following syntax is used to specify the database table statically:

```
DELETE [FROM] <target_database_tab> [CLIENT SPECIFIED]
<database_tab_lines>.
```

The following syntax is used to specify the database table dynamically:

```
DELETE [FROM] (<target_database_tab>) [CLIENT SPECIFIED]
<database_tab_lines>.
```

In both syntaxes, the `<target_database_tab>` expression represents the name of a database table defined in the ABAP Dictionary. The `CLIENT SPECIFIED` clause is used to disable the automatic client-handling feature of Open SQL.

The following syntax is used to select the lines to be deleted by using a condition:

```
DELETE FROM <target_database_tab> WHERE <cond> .
```

This syntax deletes all the lines from the `<target_database_tab>` database table that satisfy the conditions in the `WHERE` clause. The `FROM` clause must occur between the `DELETE` clause and the name of the `<target_database_tab>` table.

Listing 8.30 shows an example of deleting a single record from a database table:

Listing 8.30: Deleting a single record from a database table

```
REPORT ZDATA_ACCESS.

*/Deleting a record from the ZEMPLOYEE_DATA table, where EMPID
is KG08
TABLES: ZEMPLOYEE_DATA.
DELETE FROM ZEMPLOYEE_DATA WHERE EMPID = 'KG08'.
```

In Listing 8.30, a single line is deleted from the `ZEMPLOYEE_DATA` table, where the value of the `EMPID` field is KG08. Figure 8.39 shows the deletion of the specified record from the `ZEMPLOYEE_DATA` table:

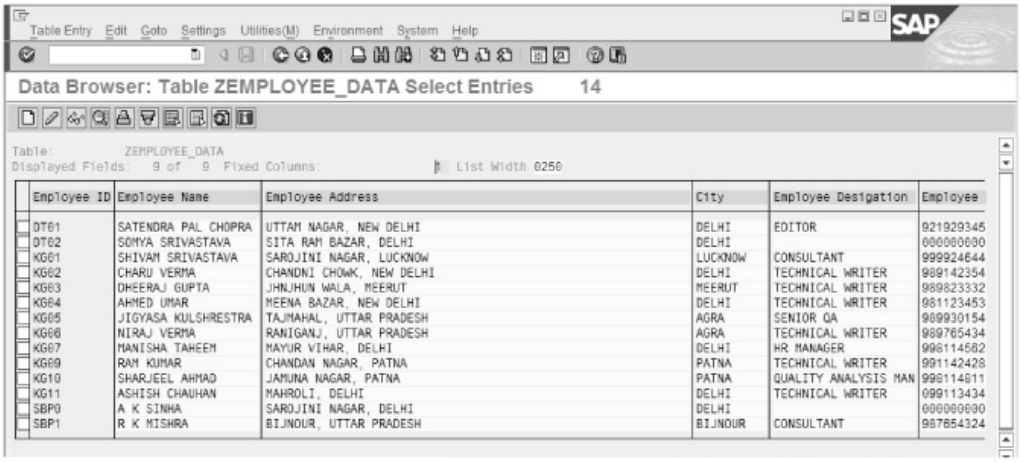


Figure 8.39: Showing the deletion of a record from the ZEMPLOYEE_DATA table

Instead of using a WHERE clause, you can also use the following syntax to delete the lines by using a work area:

```
DELETE <target_database_tab> FROM <work_area> .
```

This syntax deletes the line with the same primary key as that of the <work_area> work area. The FROM clause must not occur between the <target_database_tab> database table and <work_area> and between the DELETE clause and the <target_database_tab> database table. The <work_area> work area has the same length and alignment as the line structure of the <target_database_tab> table. Another way to use the DELETE statement is as follows:

```
TABLES <target_database_tab>.
DELETE <target_database_tab>.
```

In this syntax, the DELETE statement deletes one or more lines from the <target_database_tab> database table by using a table work area. The table work area is defined by using the TABLES statement. Note that the name of the table work area and the database table must be the same. It is not possible to specify the name of the database table dynamically.

The following syntax is used to delete several lines from a database table by using an internal table:

```
DELETE <target_database_tab> FROM TABLE internal_tab
<work_area>.
```

This syntax deletes all lines from the <target_database_tab> database table that have the same primary key in the <internal_tab> internal table. Sometimes, a record of an internal table may not be deleted from the database table because the database table does not contain the matching record. In such situations, the operation of deleting the records is not terminated and the SAP system continues to process the next record of the internal table.

If all the lines from the internal table have been processed, the value of the SY-SUBRC variable is set to 0; otherwise, it is set to 4. If the internal table is empty, the values of the SY-SUBRC and SY-DBCNT variables are set to 0.

As a best practice, use an internal table where you need to delete a large number of lines from a database table.

Examples of Deleting Data

Listing 8.31 shows the delete operation by using the WHERE clause:

Listing 8.31: Using the WHERE clause to specify a condition for the delete operation

```
REPORT ZDATA_ACCESS.

*/Deleting records from the ZEMPLOYEE_DATA table, where EMPID
is DT02 and CITY is
DELHI

TABLES: ZEMPLOYEE_DATA.
DELETE FROM ZEMPLOYEE_DATA WHERE EMPID = 'DT02' AND
```

```
CITY = 'DELHI'.
```

In Listing 8.31, the DELETE statement is used to delete all the lines from the ZEMPLOYEE_DATA table, where the value of the Employee ID field is DT02 and the value of the City field is "DELHI". Figure 8.40 shows that the specified record has been deleted:

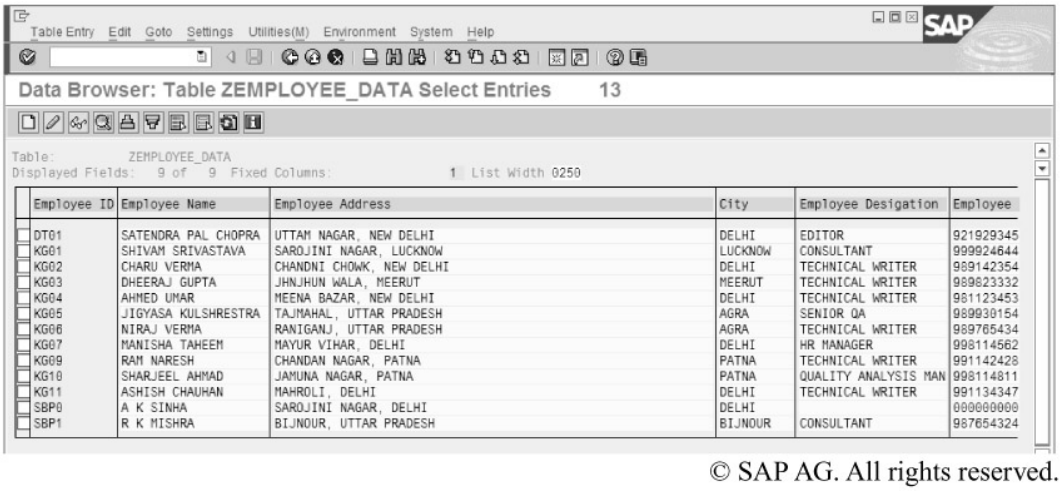


Figure 8.40: Deleting the record where EMPID is DT02 and city is DELHI

Listing 8.32 shows the deletion of records or lines by using the MOVE and DELETE statements:

Listing 8.32: Deleting the records by using the MOVE and DELETE statements

```
REPORT ZDATA_ACCESS.

*/Deleting the records using the MOVE and DELETE statements
TABLES ZEMPLOYEE_DATA.
DATA: BEGIN OF KOG,
EMPID TYPE ZEMPLOYEE_DATA-EMPID, CITY TYPE ZEMPLOYEE_
DATA-CITY,
END OF KOG.

MOVE 'SBP0' TO KOG-EMPID.
MOVE 'DELHI' TO KOG-CITY.
DELETE ZEMPLOYEE_DATA FROM KOG.

MOVE 'KG11' TO ZEMPLOYEE_DATA-EMPID.
MOVE 'DELHI' TO ZEMPLOYEE_DATA-CITY.
DELETE ZEMPLOYEE_DATA.
```

In Listing 8.32, records where the values of the Employee ID and City are SBP0 and DELHI, respectively, or the values of Employee ID and City are KG11 and DELHI, respectively, are deleted from the ZEMPLOYEE_DATA table. Figure 8.41 shows that records with Employee ID SBP0 and KG11 are deleted:

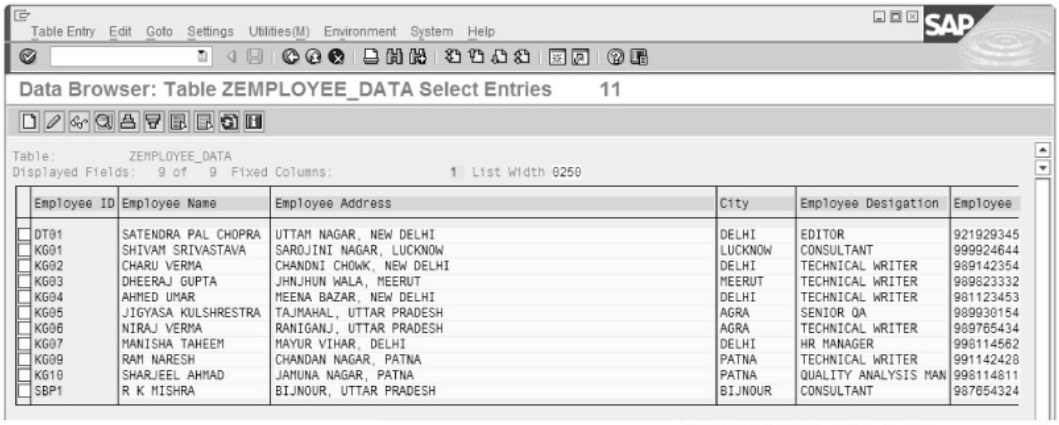


Figure 8.41: Deleting records by using the MOVE and DELETE statements

Listing 8.33 shows the deletion of records from a database table by using an internal table:

Listing 8.33: Deleting the records by using an internal table

```
REPORT ZDATA_ACCESS.

*/Deleting a record from the ZEMPLOYEE_DATA table, with the help
of an internal table-MYTABLE

DATA: MYTABLE TYPE HASHED TABLE OF ZEMPLOYEE_DATA
WITH UNIQUE KEY EMPID,
KOG LIKE LINE OF MYTABLE.
KOG-EMPID = 'KG07'.
KOG-EMPNAME = 'MANISHA TAHEEM'.
KOG-EMPADDRESS = 'MAYUR VIHAR, DELHI'.
KOG-CITY = 'DELHI'.
KOG-DESIGNATION = 'HR MANAGER'.
KOG-PHONENUMBER = '9981145622'.
KOG-SALARY = 30000.

KOG-JOININGDATE = '20080101'.
KOG-MAILID = 'MANISHA.TAHEEM@YAHOO.COM'.

INSERT KOG INTO TABLE MYTABLE.

DELETE ZEMPLOYEE_DATA FROM TABLE MYTABLE.
```

Listing 8.33 shows the deletion of a line from the ZEMPLOYEE_DATA table. First, a line is inserted into the KOG work area and then it is inserted into the MYTABLE internal table. Finally, the content of the MYTABLE table deletes the lines of the ZEMPLOYEE_DATA database table where the value of Employee ID is KG07. Figure 8.42 shows the output of Listing 8.33:

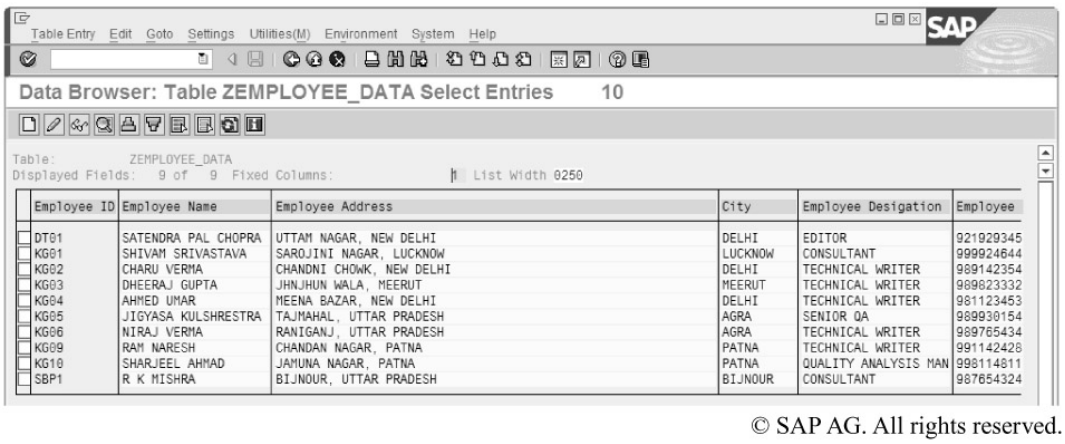


Figure 8.42: Deleting two lines from the ZEmployee_Data table

In Figure 8.42, notice that there is no line with Employee ID KG07.

Modifying the Lines of Database Tables

The MODIFY statement is used to insert new lines or update the values of existing lines within a database table. If the key specified in the MODIFY statement is not present in any existing line of the database table, the MODIFY statement works as an INSERT statement; that is, the line is inserted. If the key specified in the MODIFY statement is present in an existing line of the database table, the MODIFY statement works as an UPDATE statement; that is, the line is updated. The following syntax of the MODIFY statement is used to insert or update one or more lines into a database table:

```
MODIFY <target_database_tab> <database_tab_lines>.
```

In this syntax, <target_database_tab> represents a database table and <database_tab_lines> represents the lines that have to be inserted or updated in the table.

Note For performance reasons, you should use the MODIFY statement only if you cannot distinguish between the INSERT and UPDATE statements in an ABAP program.

You can specify the <target_database_tab> database table either statically or dynamically.

The following is the syntax used to specify the database table statically:

```
MODIFY <target_database_tab> [CLIENT SPECIFIED] <database_
tab_lines>.
```

The following syntax is used to specify the database table dynamically:

```
MODIFY (<target_database_tab>) [CLIENT SPECIFIED] <database_
tab_lines>.
```

In both syntaxes, the <target_database_tab> expression represents the name of a database table defined in the ABAP Dictionary. The CLIENT SPECIFIED clause is used to disable the automatic client-handling feature of Open SQL.

The following syntax is used to insert or change a single line in a database table:

```
MODIFY <target_database_tab> FROM <work_area>.
```

In this syntax, <work_area> is a work area created for the <target_ database_tab> database table. If the database table does not contain a line with the same primary key as specified in the work area, a new line is inserted. If the database table contains a line with the same primary key as specified in the work area, the existing line is overwritten. The value of the SY-SUBRC system variable is always set to 0.

The preceding syntax can also be written in the following way:

```
MODIFY <target_database_tab>.
```

In this syntax, the MODIFY statement inserts or updates one or more lines in the <target_database_tab> database table by using a table work area. The table work area is defined by using the TABLES statement. Note that the name of the table work area and that of the database table must be the same. However, you cannot specify the name of the database table dynamically.

The following syntax is used to insert or update several lines in a database table by using an internal table:

```
MODIFY <target_database_tab> FROM TABLE <internal_tab>.
```

This syntax is used to insert lines from the <internal_tab> internal table to the <target_database_tab> database table. Note that the lines inserted do not exist in the <target_database_tab> database table. However, matching lines of an internal table with the database table are overwritten.

Now, let's explore the concept of cursors, which are used to read data from database tables.

Using Cursors to Read Data

Cursors are similar to data objects or variables used to retrieve the data from the result set obtained by using the SELECT statement. For this, you must open a cursor by using the OPEN CURSOR statement. Then retrieve the data from the result

set by using the FETCH NEXT CURSOR statement. Finally, close the cursor by using the CLOSE CURSOR statement. Therefore, we can say that using a cursor to read data involves the following processes:

- Opening and closing cursors
- Retrieving data

Now, let's discuss each process in detail.

Opening and Closing Cursors

The following syntax is used to open a cursor for a SELECT statement:

```
OPEN CURSOR [WITH HOLD] <c> FOR SELECT <result>
                                FROM <source>
                                [WHERE <condition>]
                                [GROUP BY <fields>]
                                [HAVING <cond>]
                                [ORDER BY <fields>].
```

In this syntax, the OPEN CURSOR statement is used to open the <c> cursor. You can use all the clauses of the SELECT statement (other than the INTO clause) to select one or more lines. However, you cannot specify the names of the columns individually in the SELECT statement and the columns cannot contain aggregate expressions.

You can open more than one cursor in a parallel way for a single database table. Moreover, an already opened cursor cannot be reopened. The CLOSE CURSOR statement is used to close a cursor. The following syntax shows the use of the CLOSE CURSOR statement:

```
CLOSE CURSOR <c>.
```

This syntax is used to close all cursors that are no longer required, because only a limited number of cursors can be opened simultaneously.

The WITH HOLD clause in the OPEN CURSOR statement allows you to prevent a cursor from closing when a database is committed in Native SQL. However, the WITH HOLD clause cannot be specified when the cursor is opened for a secondary database connection.

Retrieving Data

The FETCH NEXT CURSOR statement is used to retrieve data from a database table.

The following syntax is used to read the data into a target area in an ABAP program by using the FETCH NEXT CURSOR statement:

```
FETCH NEXT CURSOR <c> INTO <target>.
```

In this syntax, the FETCH NEXT CURSOR statement inserts a line of the database table into a target area, which is a variable. After the FETCH NEXT CURSOR statement retrieves the data, the cursor moves to the next line of the database table. The value of the SY-SUBRC system variable is set to 0 until all the lines of the selection have been read; otherwise, it is set to 4. Moreover, after the FETCH NEXT CURSOR statement is executed, the SY-DBCNT system variable is updated to reflect the number of lines that have been read.

Listing 8.34 shows an example of reading data using cursors:

Listing 8.34: Reading data by using cursors

```
REPORT ZDATA_ACCESS.

*/Declaring Cursors-c1 and c2

DATA: c1 TYPE cursor,
       c2 TYPE cursor.

DATA: work_area1 TYPE MARA,
       work_area2 TYPE MARA.
```

```
DATA: flag1(1) TYPE c,
       flag2(1) TYPE c.

*/Opening Cursors-c1 and c2
OPEN CURSOR: c1 FOR SELECT MATNR ERSDA ERNAM
                        FROM MARA
                        where ERNAM = 'RUDISILL',
                c2 FOR SELECT LAEDA AENAM
                        FROM MARA
                        WHERE ERNAM = 'RUDISILL'.
DO.
  IF flag1 NE 'X'.
*/Using the c1 cursor to fetch the data
  FETCH NEXT CURSOR c1 INTO CORRESPONDING FIELDS OF work_
  area1.

     IF sy-subrc <> 0.
*/Closing the c1 cursor
     CLOSE CURSOR c1.
     flag1 = 'X'.
ELSE.
     WRITE: / work_area1-MATNR, work_area1-ERSDA, work_
     area1-ERNAM.
ENDIF.
ENDIF.

IF flag2 NE 'X'.
*/Using the c2 cursor to fetch the data
      FETCH NEXT CURSOR c2 INTO CORRESPONDING FIELDS OF work_
      area2.
      IF sy-subrc <> 0.
*/Closing the c2 cursor
      CLOSE CURSOR c2.
      flag2 = 'X'.
ELSE.
      WRITE: / work_area2-LAEDA, work_area2-AENAM.
ENDIF.
ENDIF.
   IF flag1 = 'X' AND flag2 = 'X'.
EXIT.
ENDIF.
ENDDO.
```

In Listing 8.34, cursors are used to read data from the MARA table. The `DATA` statement defines two variables, `c1` and `c2`, which are used as cursors in the listing. The `DATA` statement also defines two work areas, `work_area1` and `work_area2`, which have the type similar to the fields of the MARA table. The `OPEN CURSOR` statement is used to open the `c1` and `c2` cursors. A `DO` loop is initiated, so that the required data is retrieved from the database table and stored in the work areas. The retrieved values finally are displayed on the output screen. The value of the `SY-SUBRC` system variable remains 0 until all the values are read. As soon as all the values are read, the value of the `SY-SUBRC` system variable is set to 4. Figure 8.43 shows the output of Listing 8.34:

```
637                30.07.2002 RUDISILL
23.01.2003 I021066
638                30.07.2002 RUDISILL
23.01.2003 I021066
640                30.07.2002 RUDISILL
23.01.2003 I021066
641                30.07.2002 RUDISILL
23.01.2003 I021066
642                30.07.2002 RUDISILL
23.01.2003 I021066
643                31.07.2002 RUDISILL
23.01.2003 I021066
644                31.07.2002 RUDISILL
23.01.2003 I021066
645                31.07.2002 RUDISILL
23.01.2003 I021066
646                31.07.2002 RUDISILL
23.01.2003 I021066
647                31.07.2002 RUDISILL
23.01.2003 I021066
648                31.07.2002 RUDISILL
23.01.2003 I021066
```


Figure 8.43: Using cursors to retrieve data

Now, let's learn how to save and undo the changes made by the DML operations by using the `COMMIT` and `ROLLBACK` statements, respectively.

Committing Database Changes

In Open SQL, the `COMMIT WORK` statement is used to save database updates and the `ROLLBACK WORK` statement is used to undo the database updates performed. The `COMMIT WORK` statement is declared by using the following syntax:

```
COMMIT WORK.
```

The syntax to use the `ROLLBACK WORK` statement to undo database updates is:

```
ROLLBACK WORK.
```

Note that the `COMMIT WORK` statement always concludes a database Logical Unit of Work (LUW) and starts a new one. However, the `ROLLBACK WORK` statement always reverts the changes to the initial position of the database LUW.

Summary

In this chapter, you have learned about the two types of SQL statements, Open SQL and Native SQL. You also have learned how to perform DML operations on the data stored in a database table, defined in the ABAP Dictionary, by using the `INSERT`, `UPDATE`, `MODIFY`, and `DELETE` Open SQL statements. Next, you have learned how to use cursors to read data from database tables. You also have learned how to use the `COMMIT WORK` and `ROLLBACK WORK` statements to confirm or revert the final changes made in a database.