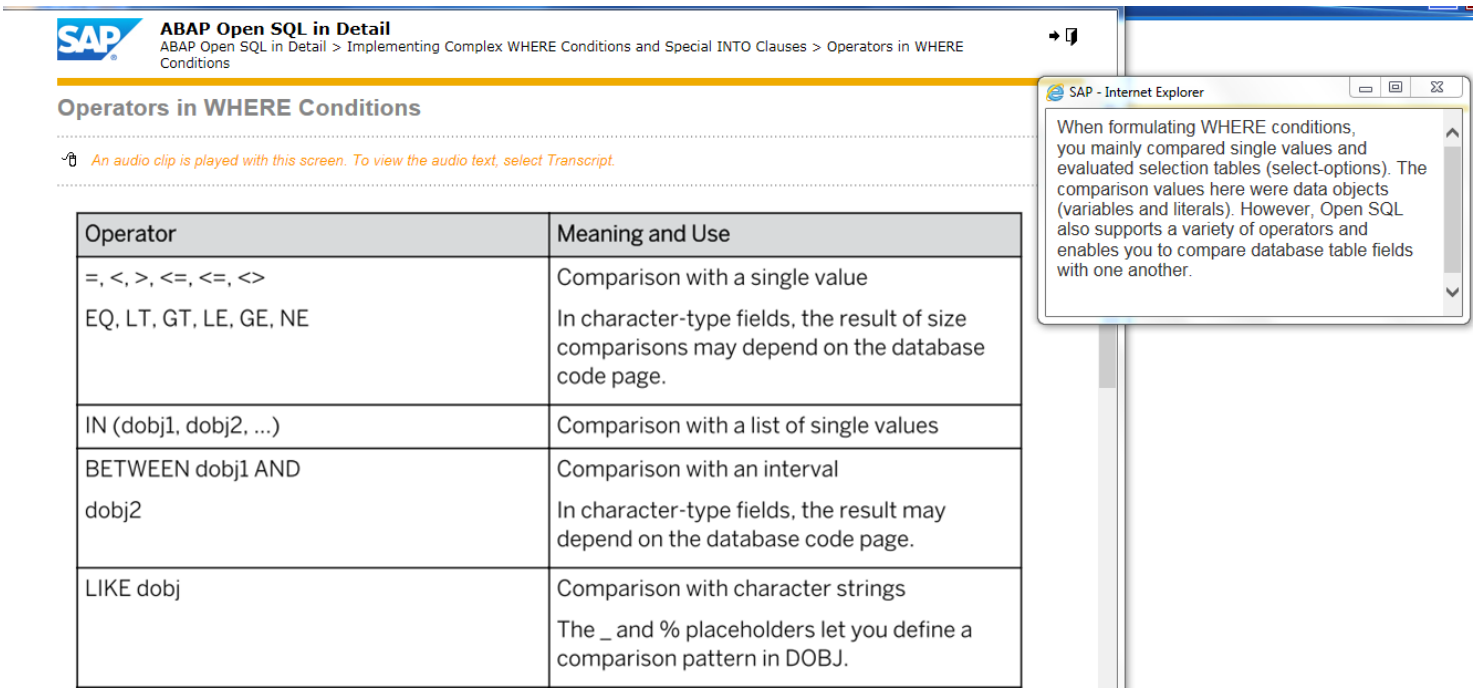


ABAP Open SQL in Detail

ABAP Open SQL in Detail > Implementing Complex WHERE Conditions and Special INTO Clauses > Operators in WHERE Conditions

Operators in WHERE Conditions

An audio clip is played with this screen. To view the audio text, select Transcript.

Operator	Meaning and Use
=, <, >, <=, <=, <> EQ, LT, GT, LE, GE, NE	Comparison with a single value In character-type fields, the result of size comparisons may depend on the database code page.
IN (dobj1, dobj2, ...)	Comparison with a list of single values
BETWEEN dobj1 AND dobj2	Comparison with an interval In character-type fields, the result may depend on the database code page.
LIKE dobj	Comparison with character strings The _ and % placeholders let you define a comparison pattern in DOBJ.

IS [NOT] NULL	Checks whether the database field has a null value
AND, OR	Link of logical expressions
NOT	Negation of a logical expression

Possible operators in WHERE conditions are listed in the following table:

Operator	Meaning and Use
=, <, >, <=, <=, <> EQ, LT, GT, LE, GE, NE	Comparison with a single value In character-type fields, the result of size comparisons may depend on the database code page.
IN (dobj1, dobj2, ...)	Comparison with a list of single values
BETWEEN dobj1 AND dobj2	Comparison with an interval In character-type fields, the result may depend on the database code page.
LIKE dobj	Comparison with character strings The _ and % placeholders let you define a comparison pattern in DOBJ.
IN seltab	Evaluation of a selection table (select-options)
IS [NOT] NULL	Checks whether the database field has a null value
AND, OR	Link of logical expressions
NOT	Negation of a logical expression

Operators in WHERE Conditions (2)

Example: SELECT with complex WHERE condition

```
DATA lt_customers TYPE TABLE OF scustom.
SELECT * FROM scustom INTO TABLE lt_customers
     WHERE country  IN ('DE','US')
       AND discount BETWEEN '005' AND '010'
       AND postcode LIKE '___5_'
       AND name     LIKE '%ra%'.
```

Country selection using list of single values

Range for discount

Postal code has five places and the number "5" in position 4

Name contains substring "ra" in any position

Wildcards for LIKE:

_ = Placeholder for a single character

% = Placeholder for substring of any length

SAP - Internet Explorer

The figure illustrates an example of a complex search for airline customers using name and address data.

The "_" and "%" placeholders correspond to the SQL standard in delimitation with LIKE. In other ABAP statements, the "+" and "*" characters are used for similar comparisons, for example, within comparison tables.

To search for the "_" or "%" characters, use the ESCAPE addition to define an escape symbol. A condition that the column contents must contain a "_" in any location is then formulated as LIKE '%! _%' ESCAPE '!'.

When the developer links conditions, the AND operators are evaluated before the OR operators. Set parentheses to change this order as needed.

Note: The selection of the operator can have a major impact on the usability of a condition for an index search string. The formulation of the WHERE condition influences the search strategy of the optimizer and the performance of the database access.

Table Field Comparison

All domestic connections (country of departure = country of arrival)

```
DATA lt_spfli TYPE TABLE OF spfli.

SELECT * FROM spfli
         INTO TABLE lt_spfli
         WHERE countryfr = spfli~countryto.
```

Table name must be specified

Both fields must have the same (technical type), otherwise the result is dependent on the database!

All flights with full business class,
but available seats in economy class

```
DATA lt_flight TYPE TABLE OF sflight.

SELECT * FROM sflight AS a
         INTO TABLE lt_flight
         WHERE  seatsocc_b = a~seatsmax_b
         AND    seatsocc   < a~seatsmax.
```

Alias for the table → shorter `WHERE` condition

SAP - Internet Explorer

In addition to variables and constants, you can also enter fields from the same database tables after the comparison operators (for example, =, EQ, <>, NE, <, LT) in the WHERE conditions of open SQL statements. Entering fields from the same database tables after the comparison operators makes it possible to compare two database fields with each other in the WHERE condition and only read the records whose contents fulfill the condition.

To avoid the confusion of the second table field with a data object, the name of the database table must be specified with the field label connected with the "~" character. Alternatively, use an alias for the table name, which you define with the AS addition after the table name in the FROM clause.

Aliases for table names play a subordinate role here. They are used to abbreviate the WHERE condition. However, you see aliases again in the formulation of joins, where they are crucial if the same database table appears several times in the FROM clause.

Caution: The fields that are compared must have the same basic type and length (ideally, they are based on the same domain). Otherwise, the result depends on how the respective database system stores the different types and how it handles spaces at the end of values.

Do not confuse this with single line and multiline result sets, as listed in the following table:

Statement	Result-Set	Target Area
SELECT SINGLE...	single-line	single-line
SELECT... INTO TABLE...	multiline	multiline
SELECT... ENDSELECT.	multiline	single-line

SAP - Internet Explorer

When you specify data objects as targets of SELECT statements, you need to differentiate whether the target area that is specified after INFO is single line or multiline.

Statement	Result-Set	Target Area
SELECT SINGLE...	single-line	single-line
SELECT... INTO TABLE...	multiline	multiline
SELECT... ENDSELECT.	multiline	single-line

Single Line Target Area

Filling any list of elementary fields

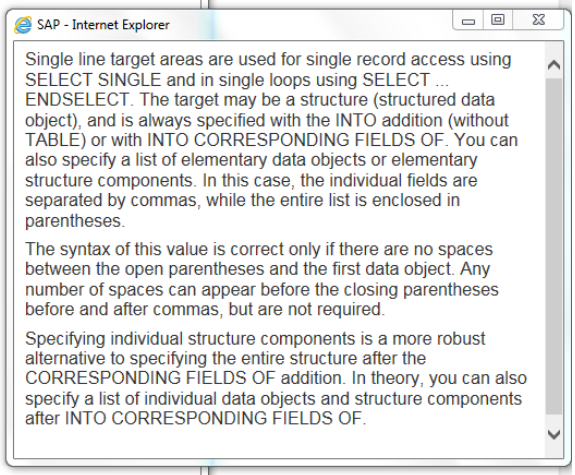
```
DATA: lv_max TYPE sflight-seatsmax,
      lv_occ TYPE sflight-seatsocc.

SELECT SINGLE seatsmax seatsocc
              FROM sflight INTO (lv_max, lv_occ)
              WHERE ... .
```

Fields comma-separated, no spaces behind open parenthesis

Targeted filling of individual components of a structure

```
DATA ls_struct TYPE sflight.

SELECT SINGLE seatsmax seatsocc
   FROM sflight
   INTO (ls_struct-seatsmax, ls_struct-seatsocc)
   WHERE ... .

SELECT SINGLE seatsmax seatsocc
   FROM sflight
   INTO CORRESPONDING FIELDS OF ls_struct
   WHERE ... .
```

Explicit components as an alternative to `CORRESPONDING FIELDS OF ls_struct`

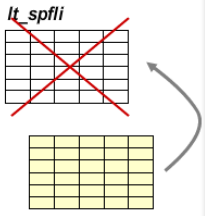


Multiline Target Area

```
DATA lt_spfli TYPE TABLE OF spfli.
```

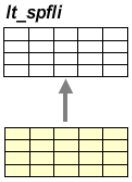
Filling an internal table (overwrite)

```
SELECT * FROM spfli
       INTO TABLE lt_spfli
       WHERE ... .

SELECT cityfrom cityto FROM spfli
       INTO CORRESPONDING FIELDS OF
       TABLE lt_spfli
       WHERE ... .
```



Appending lines to an internal table

```
SELECT * FROM spfli
       APPENDING TABLE lt_spfli
       WHERE ... .

SELECT cityfrom cityto FROM spfli
       APPENDING CORRESPONDING FIELDS OF
       TABLE lt_spfli
       WHERE ... .
```



SAP - Internet Explorer

When SELECT statements have multiline target areas, you must specify an internal table as the data object. Specify the table, for example, after the INTO TABLE addition or after INTO CORRESPONDING FIELDS OF TABLE.

Alternatively, replace INTO TABLE and INTO CORRESPONDING FIELDS OF TABLE with APPENDING TABLE and APPENDING CORRESPONDING FIELDS OF TABLE, respectively. If you use INTO, the contents of the internal table are replaced completely with the selected data. If you use APPEND, the data is added to any existing lines in the table.

Hint: If you use APPENDING for sorted and hashed tables, no append is performed. Specifically, the addition is not treated as an index access to the internal table, which means it can also be used for hashed tables. In sorted tables, the new lines are inserted in the existing lines in accordance with the sort sequence.

ABAP Open SQL in Detail

ABAP Open SQL in Detail > Implementing Complex WHERE Conditions and Special INTO Clauses > Explicit Cursor

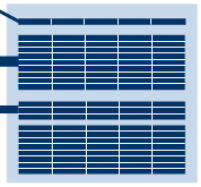
Explicit Cursor

Packaged (packages with variable package size)

```
DATA: lt_book   TYPE TABLE OF sbook,
      ls_book   TYPE sbook,
      lv_cursor TYPE cursor,
      lv_number TYPE i.

OPEN CURSOR lv_cursor FOR
     SELECT * FROM sbook
            WHERE ... .

 FETCH NEXT CURSOR lv_cursor
       INTO ls_book.
  ...
 DO.
   lv_number = ... .
   FETCH NEXT CURSOR lv_cursor
         INTO TABLE lt_book
         PACKAGE SIZE lv_number.
    IF sy-subrc <> 0.
      EXIT.
    ENDIF.
    ...
 ENDDO.

CLOSE CURSOR lv_cursor.
```

The cursor variable "remembers" how much of the result set was already processed

The DB access starts here; the data selection (result set) is defined)



The DB access ends here and the cursor variable is initialized

The importance of cursor variable, which is a variable with a special CURSOR type, is highlighted as follows:

- The cursor variable identifies database access in the subsequent statements.
- The cursor variable stores the position in the result set of database access up to which processing is already complete.

Rollover for more information

Rollover for more information

SAP - Internet Explorer

Packages of different sizes can be processed sequentially with the syntax illustrated in the figure.

In this variant of database read access, the access itself is detached from the transfer of the result set to the application program.

The OPEN CURSOR statement defines and executes the access, but does not pass any data on to the application program.

The FETCH NEXT CURSOR statement retrieves the desired number of lines (PACKAGE SIZE addition) from the result set and copies them to an internal table.

The CLOSE CURSOR statement completes the database access.

After OPEN CURSOR, the cursor variable points to a position before the first line.

The sequential FETCH statements not only allow you to read different numbers of records, but also allow you to specify a different data object as the target (structures or internal tables) after each

Ordered Datasets

Sorting by primary key (ascending):

```
SELECT *
   FROM sflight INTO ... WHERE ...
   ORDER BY PRIMARY KEY .

SELECT mandt carrid connid fldate ...
   FROM sflight INTO ... WHERE ...
   ORDER BY PRIMARY KEY .
```

ORDER BY must appear after the WHERE condition

All primary key fields (including MANDT) must appear in the field

Sorting by any key (ascending):

```
SELECT *
   FROM sflight INTO ... WHERE ...
   ORDER BY connid seatsocc.

SELECT ... connid ... seatsocc ....
   FROM sflight INTO ... WHERE ...
   ORDER BY connid seatsocc.
```

Sequence of fields determines priority

All sort fields must be in the field list

Sorting by any key (ascending or descending):

```
SELECT *
   FROM sflight INTO ... WHERE ...
   ORDER BY connid DESCENDING seatsocc ASCENDING .
```

The restrictions on ORDER BY are as follows:

- You cannot sort pooled tables and cluster tables by any field.
- You can only sort by columns that appear after the SELECT statement, which means the sort columns must be a part of the result set.
- You cannot sort by fields with type LCHAR, LRAW, STRING, or RAWSTRING.

SAP - Internet Explorer

In Open SQL, you can have the database sort the dataset by specific criteria directly as part of the SELECT operation. To do so, use the ORDER BY clause for the SELECT statement.

In the simplest form of the ORDER BY PRIMARY KEY clause, the table is sorted by its complete primary key in ascending order.

To use ORDER BY PRIMARY KEY, the prerequisites are as follows:

- The FROM clause contains a single database table (no views or joins).
- The field list after SELECT contains all the key fields of the table (including the client, which is often forgotten).

The syntax ORDER BY COL1, COL2 ... sorts the data by any existing columns. The sequence of the specified columns determines their sort priority.

ORDER BY Addition

```
TYPES: BEGIN OF lty_s_flightocc,
         connid   TYPE sflight-connid,
         fldate   TYPE sflight-fldate,
         seatsocc TYPE sflight-seatsocc,
       END OF lty_s_flightocc.
DATA lt_flightocc TYPE TABLE OF lty_s_flightocc.

SELECT connid fldate seatsocc
       FROM sflight
       INTO TABLE lt_flightocc
       WHERE seatsocc > 200
       ORDER BY connid DESCENDING seatsocc ASCENDING.
```

Result set sorted first by CONNID, then by SEATSOCC within the CONNID groups

ORDER BY must appear after the WHERE condition

MANDT	CARRID	CONNID	FLDATE	SEATSOCC
001	AA	0014	20081217	250
001	AA	0014	20090121	243
001	AA	0017	20081230	243
001	AA	0017	20090102	250
001	AA	0017	20090106	300
~~001~~	~~AA~~	~~0017~~	~~20090117~~	~~143~~
001	AA	0017	20090119	300
~~001~~	~~AA~~	~~0018~~	~~20081203~~	~~143~~
~~001~~	~~AA~~	~~0018~~	~~20081207~~	~~155~~
~~001~~	~~AA~~	~~0018~~	~~20090106~~	~~165~~
001	AA	0019	20081205	244
001	AA	0019	20090114	312

lt_flightocc

CONNID	FLDATE	SEATSOCC
0019	20081205	244
0019	20090114	312
0017	20081230	243
0017	20090102	250
0017	20090106	300
0017	20090119	300
0014	20090121	243
0014	20081217	250

SAP - Internet Explorer

The figure shows an example of the ORDER BY addition with free column selection.

If you do not use an addition, the system sorts retrieved datasets in ascending order. You can use the optional DESCENDING and ASCENDING clauses after a given field to define the sort direction.

Note: Sorting can be a computationally expensive process if the result set is large and the sort fields are not also index fields.

Condensed Datasets

```
TYPES: BEGIN OF lty_s_flight,
         carrid TYPE sflight-carrid,
         connid TYPE sflight-connid,
       END OF lty_s_flight.
DATA lt_flight TYPE TABLE OF lty_s_flight.

SELECT DISTINCT  carrid connid
       FROM sflight
       INTO TABLE lt_flight
       WHERE seatsocc > 200.
```

DISTINCT: Duplicates are removed from result set

WHERE condition: Entries are ignored in result set

MANDT	CARRID	CONNID	FLDATE	SEATSOCC
001	AA	0014	20081217	250
001	AA	0014	20090121	243
001	AA	0017	20081230	243
001	AA	0017	20090102	250
001	AA	0017	20090106	300
~~001~~	~~AA~~	~~0017~~	~~20090117~~	~~143~~
001	AA	0017	20090119	300
~~001~~	~~AA~~	~~0018~~	~~20081203~~	~~143~~
~~001~~	~~AA~~	~~0018~~	~~20081207~~	~~155~~
~~001~~	~~AA~~	~~0018~~	~~20090106~~	~~165~~
001	AA	0019	20081205	244
001	AA	0019	20090114	312

lt_flight

CARRID	CONNID
AA	0014
AA	0017
AA	0019

SAP - Internet Explorer

The SELECT DISTINCT statement in Open SQL instructs the database to aggregate the result set of a SELECT statement such that it does not contain any duplicate entries. All fields of the result set are used for the comparison. You can either load the result of SELECT DISTINCT into an internal table (array fetch) or process it sequentially (SELECT loop).

For SELECT DISTINCT, the applicable restrictions are as follows:

- The field list cannot contain any columns with type LCHAR, LRAW, STRING, or RAWSTRING.
- You must use SELECT DISTINCT * to access pooled tables and cluster tables. You cannot select individual columns.

Aggregate Expressions

An audio clip is played with this screen. To view the audio text, select Transcript.

An aggregate expression uses an aggregate function to perform calculations on a specified column in the SELECT statement. Aggregate expressions determine values from multiple rows in a column of a database table. The calculation is performed in the database system.

The aggregate functions supported by the ABAP Open SQL are as follows:

Function	Meaning of Result and Conditions	Data Type of Result
MIN(col)	Minimum value in the col column within the result set	Like the col column
MAX(col)	Maximum value in the col column within the result set	Like the col column
SUM (col)	Sum of the contents of the col column in the result set-col column must be numeric	Like the col column
AVG(col)	Average (arithmetic mean) value of the contents of the col column in the result set-col column must be numeric	Float (F)
COUNT (*)	Number of lines in the result set	Integer (I)

SAP - Internet Explorer

When an INTO clause is used with aggregate expressions, the data object in the INTO clause must provide a structure component or table column with the appropriate type for each aggregate expression in the SELECT statement.

When aggregate functions are used, there is a major difference between cases in which the field list consists exclusively of aggregate expressions, and cases in which the field list also contains field labels that are not arguments to an aggregate function.

Field Lists Only with Aggregate Expressions

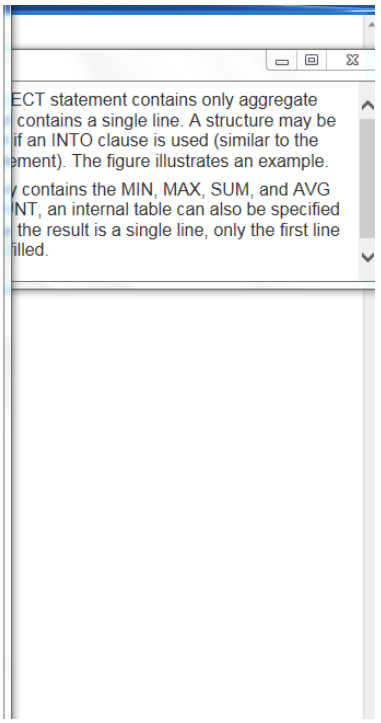
```
TYPES: BEGIN OF lty_s_flightocc,
         cntall TYPE i,
         minocc TYPE sflight-seatsocc,
         maxocc TYPE sflight-seatsocc,
         sumocc TYPE sflight-seatsocc,
       END OF lty_s_flightocc.
DATA ls_flightocc TYPE lty_s_flightocc.

SELECT COUNT(*) MIN( seatsocc ) MAX( seatsocc ) SUM( seatsocc )
       FROM sflight
       INTO ls_flightocc.
```

Field list only contains aggregate functions → One-line result

MANDT	CARRID	CONNID	FLDATE	SEATSOCC
001	AA	0014	20081217	250
001	AA	0014	20090121	243
001	AA	0017	20081230	243
001	AA	0017	20090102	250
001	AA	0017	20090106	300
001	AA	0017	20090117	143
001	AA	0017	20090119	300
001	AA	0018	20081203	143
001	AA	0018	20081207	155
001	AA	0018	20090106	165
001	AA	0019	20081205	244
001	AA	0019	20090114	312

ls_flightocc

CNTALL	MINOCC	MAXOCC	SUMOCC
12	143	312	2748



Aggregate Functions with the DISTINCT Addition

```
TYPES: BEGIN OF lty_s_flightocc,
         cntall TYPE i,
         cntcon TYPE i,
         sumocc TYPE sflight-seatsocc,
       END OF lty_s_flightocc.
DATA ls_flightocc TYPE lty_s_flightocc.

SELECT COUNT(*) COUNT(DISTINCT connid) SUM(DISTINCT seatsocc)
       FROM sflight
       INTO ls_flightocc.
```

Identical values are only taken into account once during totaling

Number of distinct CONNID values

MANDT	CARRID		CONNID	FLDATE	SEATSOCC
001	AA	1	0014	20081217	250
001	AA		0014	20090121	243
001	AA		0017	20081230	~~243~~
001	AA		0017	20090102	~~250~~
001	AA	2	0017	20090106	300
001	AA		0017	20090117	143
001	AA		0017	20090119	~~300~~
001	AA		0018	20081203	~~143~~
001	AA	3	0018	20081207	155
001	AA		0018	20090106	165
001	AA		0019	20081205	244
001	AA	4	0019	20090114	312

ls_flightocc

CNTALL	CNTCON	SUMOCC
12	4	1812

SAP - Internet Explorer

In addition to COUNT(*), the COUNT function is used in combination with a DISTINCT clause, in the form COUNT (DISTINCT col).

This aggregate expression returns the number of unique, distinct values in the specified column.

You can use the DISTINCT addition for all other aggregate functions as well, for example, AVG (DISTINCT col). Values that occur multiple times in the column in the selection result are only included once in the aggregate calculation. The table is retrieved before the aggregate is calculated.

Note: In the example, the aggregate SUM (DISTINCT seatsocc) is only intended as an illustration, not as a typical example of the technique. In general, the DISTINCT clause is most useful when applied together with the COUNT function, and is not commonly combined with the MIN, MAX, SUM, and AVG functions.

Special case 1: Aggregate functions only

```
SELECT COUNT(*) MIN(seatsmax) MAX(seatsmax) SUM(seatsmax)
       FROM sflight
       INTO ls_flightocc
       WHERE carrid = 'XX'.
```

DB table has no flights with CARRID = 'XX'

ls_flightocc

CNTALL	MINOCC	MAXOCC	SUMOCC
0	0	0	0

The access still returns a result (initial values) and sy-subrc = 0, sy-dbcnt = 1

Special case 2: Only aggregate function COUNT(*)

```
SELECT COUNT (*)
   FROM sflight
   WHERE carrid = 'AA'
   AND   connid = '0017'
   AND   fldate = '20090101'.

IF sy-subrc = 0.
   . . .
ENDIF.
```

Field list only contains aggregate function COUNT(*) → INTO clause can be omitted

sy-subrc = 4 if no data is found
sy-subrc = 0 if at least 1 record fulfills the conditions

When writing SELECT statements that consist only of aggregate expressions, keep in mind the following considerations:

- Special case 1

  The dataset is empty before aggregate calculation. Unlike SELECT statements, these database accesses return a result even if no matching records are found in the database. In this case, the result of COUNT is zero and the other aggregate functions return initial values. SY-SUBRC is set to 0 and SY-DBCNT to 1.

  Rollover for more information

- Special case 2

  The SELECT statement only contains the COUNT (*) function. If the field list only contains the COUNT (*) aggregate function, you can omit the INTO clause completely. If no data is found in the database, SY-SUBRC is set to 4 and SY-DBCNT to 0

SAP - Internet Explorer

When writing SELECT statements that consist only of aggregate expressions, keep in mind the following considerations.

Information

Field Lists with Aggregate Expressions and Field Names

```
TYPES: BEGIN OF lty_s_flightocc,
         carrid TYPE sflight-carrid,
         connid TYPE sflight-connid,
         sumocc TYPE sflight-seatsocc,
       END OF lty_s_flightocc.
DATA lt_flightocc TYPE TABLE OF lty_s_flightocc.

SELECT carrid connid SUM(seatsocc)
       FROM sflight
       INTO TABLE lt_flightocc
       GROUP BY carrid connid.
```

Sum of SEATSOCC

One result for each group with same CARRID and CONNID

MANDT	CARRID	CONNID	FLDATE	SEATSOCC
001	AA	0014	20081217	250
001	AA	0014	20090121	243
001	AA	0017	20081230	243
001	AA	0017	20090102	250
001	AA	0017	20090106	300
001	AA	0017	20090117	143
001	AA	0017	20090119	300
001	AA	0018	20081203	143
001	AA	0018	20081207	155
001	AA	0018	20090106	165
001	AA	0019	20081205	244
001	AA	0019	20090114	312

lt_flightocc

CARRID	CONNID	SUMOCC
AA	0014	493
AA	0017	1236
AA	0018	463
AA	0019	556

Rollover for more information

For the GROUP BY clause, the restrictions are as follows:

- Individual columns are listed after SELECT (SELECT * is not allowed)
- No pooled tables or cluster tables
- The fields after GROUP BY cannot have type STRING or RAWSTRING

SAP - Internet Explorer

If the field list of a SELECT statement contains field labels in addition to aggregate expressions, the result is always multiline. As a result, it must be loaded into an internal table (array fetch) processed sequentially (SELECT loop). The SELECT statement must contain the GROUP BY clause and all fields that are not arguments of an aggregate function in the field list must be listed after the GROUP BY clause.

When the GROUP BY clause is specified, the database does not apply the aggregate functions to all the retrieved records together, but instead sorts them into groups first. A group contains all the records that have the same contents in the columns specified after GROUP BY. The aggregate functions are evaluated separately for each of these groups. Each group then corresponds to one line in the selection result.

HAVING Addition

```
TYPES: BEGIN OF lty_s_flightocc,
         carrid TYPE sflight-carrid,
         connid TYPE sflight-connid,
         sumocc TYPE sflight-seatsocc,
       END OF lty_s_flightocc.
DATA lt_flightocc TYPE TABLE OF lty_s_flightocc.

SELECT carrid connid SUM(seatsocc)
       FROM sflight
       INTO TABLE lt_flightocc
       WHERE fldate > '20090101'
       GROUP BY carrid connid
       HAVING SUM(seatsocc) < 500.
```

HAVING condition: Entries are removed ***after*** aggregate formation

WHERE condition: Lines are ignored during aggregate formation

MANDT	CARRID	CONNID	FLDATE	SEATSOCC
~~001~~	~~AA~~	~~0014~~	~~20081217~~	~~250~~
001	AA	0014	20090121	243
~~001~~	~~AA~~	~~0017~~	~~20081230~~	~~243~~
001	AA	0017	20090102	250
001	AA	0017	20090106	300
001	AA	0017	20090117	143
001	AA	0017	20090119	300
~~001~~	~~AA~~	~~0018~~	~~20081203~~	~~143~~
~~001~~	~~AA~~	~~0018~~	~~20081207~~	~~155~~
001	AA	0018	20090106	165
~~001~~	~~AA~~	~~0019~~	~~20081205~~	~~244~~
001	AA	0019	20090114	312

lt_flightocc

CARRID	CONNID	SUMOCC
AA	0014	243
~~AA~~	~~0017~~	~~993~~
AA	0018	165
AA	0019	312

SAP - Internet Explorer

When you use GROUP BY, you can specify a logical expression after the HAVING expression to restrict the result set further.

In contrast to the WHERE condition, the logical expression after HAVING can also contain aggregate functions (in SAP Web AS 6.10 and later). Apart from the aggregate functions, the logical expression after HAVING can only contain fields that are specified after GROUP BY.

If you use field labels after HAVING that are not listed after GROUP BY, this does not cause a syntax error, but results in a catchable runtime error (CX_SY_OPEN_SQL_DB).

The aggregate functions after HAVING are different from the aggregate functions after SELECT. In the example, you could specify a condition COUNT (*) > 1 after HAVING to only evaluate groups that contain more than one flight. You do not have to use the aggregate COUNT (*) function after SELECT in this case.

The Problem with Nested Selects

```
SELECT * FROM t1 WHERE ...
  SELECT * FROM t2 WHERE ...
    SELECT * FROM t3 WHERE ...
      SELECT * FROM t4 WHERE ...
        SELECT * FROM t5 WHERE ...
        ...
        ENDSELECT.
      ENDSELECT.
    ENDSELECT.
    SELECT * FROM t6 WHERE ...
      SELECT * FROM t7 WHERE ...
        SELECT * FROM t8 WHERE ...
        ...
        ENDSELECT.
      ENDSELECT.
    ENDSELECT.
  ENDSELECT.
ENDSELECT.
```

Results in:
- Lots of data packages
- Lots of identical accesses
- A large transfer effort

SAP - Internet Explorer

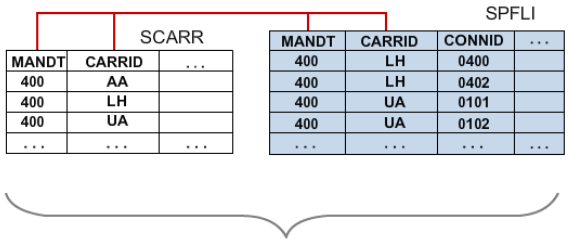
In a relational database system such as Open SQL, attempts to access data stored in a primary table require reads from the associated secondary tables as well. The obvious programmatic method to perform those accesses is to loop over the records in the primary table with nested SELECTs (see figure 191 for an example).

However, secondary tables may themselves have associated tables that need to be read in turn. There is no theoretical or practical limit to the nesting depth; the nested SELECT model can indefinitely chain tables as long as the tables are linked by common data columns.

For a number of reasons, nested SELECTS are the most resource-intensive method possible for reading multiple linked database tables, and good program design will avoid this method for any but the smallest and simplest of data access procedures. Nested SELECTS place a high load on the database server and produce large amounts of network traffic, because many partially-filled data packages are transferred and the same data may be read many times in a row (identical accesses).

In this lesson, you will learn several techniques that enable you to largely avoid nested SELECTs, and to reduce their negative impact on system performance when their use is unavoidable.

ABAP-Joins



MANDT	CARRID	...
400	AA	
400	LH	
400	UA	
...	...	...

MANDT	CARRID	CONNID	...
400	LH	0400	
400	LH	0402	
400	UA	0101	
400	UA	0102	
...	...	...	...

INNER JOIN

MANDT	CARRID	CONNID
400	LH	0400
400	LH	0402
400	UA	0101
400	UA	0102
...		...

LEFT OUTER JOIN

MANDT	CARRID	CONNID
400	AA	
400	LH	0400
400	LH	0402
400	UA	0101
400	UA	0102
...		...

Restrictions for an outer join are as follows:

- You can only have a table or a view to the right of the join operator. You cannot have another join expression.
- Only AND can be used as a logical operator in an ON condition.
- Every comparison in the ON condition must contain a field from the table on the right.
- None of the fields in the table on the right can appear in the WHERE conditions of the left outer join.

For more information about implementing join functions, see the ABAP documentation.

Click Next to continu

To read data that is distributed across multiple tables, you must create a link between the functionally dependent tables. The link between the tables is known technically as a join, and the corresponding database operator is join.

To implement a join, you can use either database views in the ABAP dictionary or ABAP joins.

Database views and ABAP joins can only be used with transparent tables. If you are working with pooled tables or cluster tables, you have to use other techniques.

You can derive the logic of the inner join and outer join from the intended result set.

An inner join produces the result set that only considers the records from the outer table for which suitable data records exist in the inner table (as in the example).

A left outer join produces the result set that contains all the records from the outer table, regardless of whether or not suitable records exist in the inner table. If no suitable records exist in the inner table, the fields of the inner (the other table that is used in the left outer join condition) table are set to zero values in the result set. The tables involved in a join are called base tables. A projection (column selection) or a selection (line selection) can be applied on the result of a join.

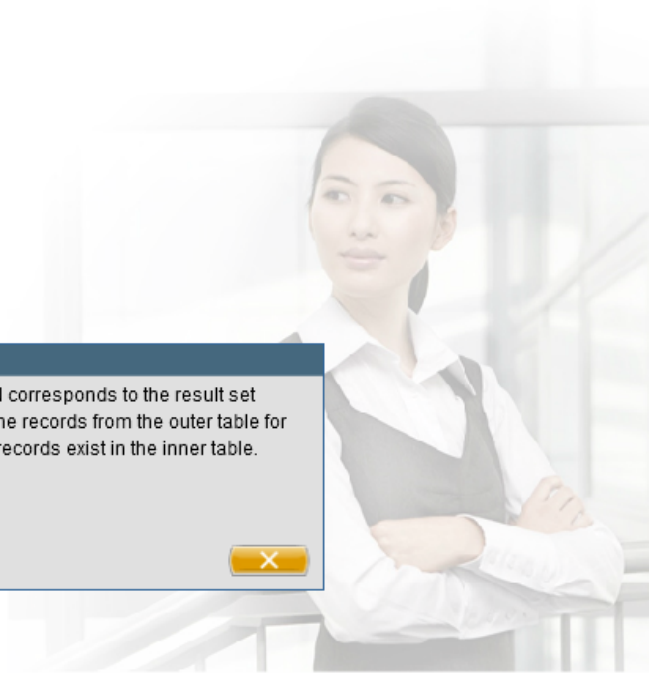
Not all of the databases supported by SAP support the standard syntax for ON conditions. Therefore, use restricted syntax to ensure that only joins that return the same result set on all database systems are allowed.

Test Your Knowledge

Choose the correct answer.

_____ corresponds to the result set that only considers the records from the outer table for which suitable data records exist in the inner table.

- LEFT OUTER JOIN
- ✓ INNER JOIN
- OUTER JOIN



Feedback

Correct. INNER JOIN corresponds to the result set that only considers the records from the outer table for which suitable data records exist in the inner table.

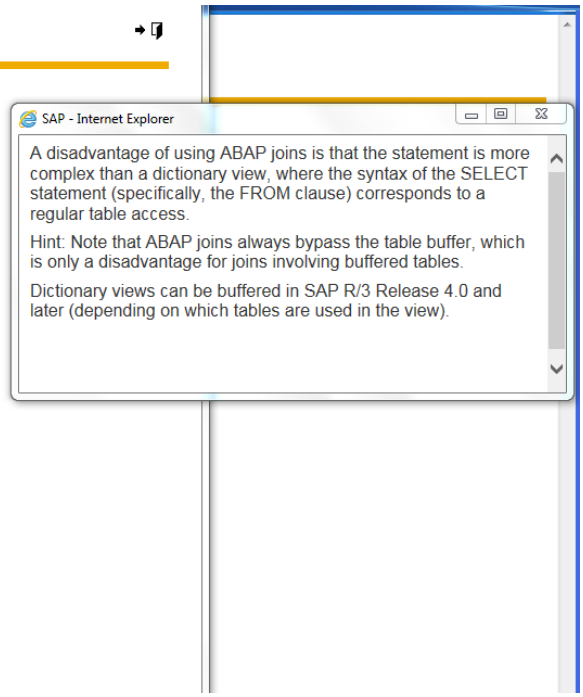
Click Next to conti

Example – ABAP Inner Join

Example: ABAP INNER Join

```
SELECT <fieldlist> INTO <target>
  FROM <dbtab1> [AS <alias1>]
  INNER JOIN <dbtab2> [AS <alias2>]
  ON <alias1>~<dbtab1-field1> = <alias2>~<dbtab2-field1>
  AND <alias1>~<dbtab1-field2> = <alias2>~<dbtab2-field2>
  AND ...
  WHERE ...
  ...
ENDSELECT.
```

```
REPORT zselect_view.

SELECT f~carrid b~connid ... INTO (...)
  FROM scarr AS f INNER JOIN spfli AS b
  ON f~carrid = b~carrid
  WHERE ...
 ...
ENDSELECT.
```



Properties and Advantages of Database Views

Attributes and benefits of database views are as follows:

- You can use views in other programs as well.
- You can use views for lists and search functions; for example, SE84 and SE81 find existing views quickly.
- You can buffer views (technical settings) like database tables.
- Fields common to both tables (join fields) are only transferred from the database to the application server once.
- The view is implemented in the ABAP Dictionary as an inner join. This means no data is transferred if the inner table does not contain any entries that correspond to the outer table.
- If you do not want to use an inner join to read from a text table, use an ABAP left outer join. For example, if you have a situation where the results of an inner join do not contain any records because no entry is available in a certain language, you can use an ABAP left outer join instead (see the figure “Example – Inner or Outer Join”).

Rollover for more information

Click Next to continue.

Subselects and Subqueries

Subquery returns single value

Select all flights in SFLIGHT with a maximum of passengers

```
SELECT * FROM sflight
         INTO TABLE lt_flights
         WHERE seatsocc =
                    (SELECT MAX(seatsocc) FROM sflight).
```

Subquery returns single-column, multiline result

Select airlines with departure city "Frankfurt"

```
SELECT * FROM scarr
         INTO TABLE lt_carriers
         WHERE carrid IN
                     (SELECT DISTINCT carrid FROM spfli
                              WHERE cityfrom = 'FRANKFURT').
```

Subquery returns any result

Select airlines with departure city "Frankfurt"

```
 SELECT * FROM scarr
          INTO TABLE lt_carriers
          WHERE EXISTS (SELECT carrid FROM spfli
                                WHERE carrid = scarr~carrid
                                AND cityfrom = 'FRANKFURT').
```


SAP - Internet Explorer

A subquery is a query within a SELECT, UPDATE, or DELETE statement. It is formulated in the WHERE or HAVING clause to check whether the data in various database tables or views possess certain attributes.

A SELECT statement with a subquery has a more restricted syntax than a SELECT statement without a subquery.

If the subquery returns exactly one value, use the usual comparison operators apart from LIKE and BETWEEN. If you use a subquery with a comparison operator instead of with EXISTS, then the SELECT clause of the subquery can only contain a single column, which can be a field in the database table or an aggregate expression.

In this example, the subquery is supposed to return several lines, each with one value. If you want to compare all the returned values, use IN. Subqueries whose WHERE condition contains fields from the main query are called correlated subqueries. If subqueries are nested, each subquery can use all the fields from the higher-level subqueries in the hierarchy. Query designers should formulate positive subqueries whenever possible; negative formulations may result in performance-degrading database reads if no adequate index is available.

Hint: In many cases, you can also use a join to obtain the same result as that produced with a subquery. The SQL code for a join is easier for other developers to read and understand, but no blanket statements can be made about the relative performance of subqueries and joins, as the table design, query logic, and amount of data to be retrieved all can have major impacts on the performance of both methods. For performance-critical systems, it is advisable to prototype the query using each method and running performance profiles with real-world data to ascertain which approach will produce faster output.

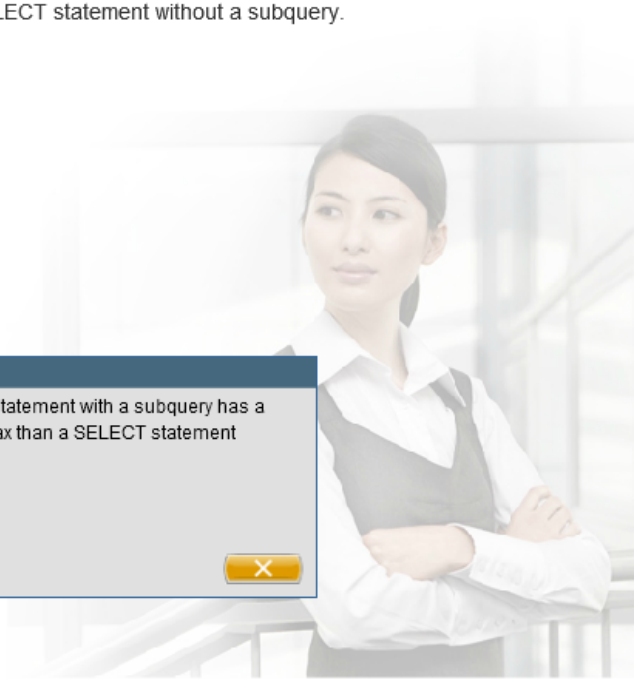
Test Your Knowledge

Determine whether this statement is true or false.

A SELECT statement with a subquery has a more restricted syntax than a SELECT statement without a subquery.

- ✓ ◉ True
- ○ False

Submit



Feedback

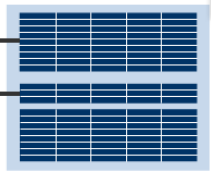
Correct. A SELECT statement with a subquery has a more restricted syntax than a SELECT statement without a subquery.

Parallel Cursors

```
SELECT * FROM sflight
         INTO ls_flight
         ORDER BY carrid connid fldate.

CLEAR lt_bookings.

SELECT * FROM sbook
         INTO TABLE lt_bookings
         WHERE   carrid = ls_flight-carrid
         AND     connid = ls_flight-connid
         AND     fldate = ls_flight-fldate
         AND      class = 'Y'
         AND cancelled <> 'X'
         ORDER BY carrid connid
                  fldate bookid.

   ...

ENDSELECT.
```

SELECT loop of over flights in SFLIGHT

Read corresponding uncanceled bookings in economy class



SAP - Internet Explorer

You know how to use the Open SQL statements OPEN CURSOR, FETCH NEXT CURSOR, and CLOSE CURSOR to program explicit cursor handling. You also know how cursors are opened in several database tables at the same time. This knowledge enables you process the contents of multiple database tables in parallel without using nested SELECTs.

You want the program to process all flights from database table SFLIGHT, and you want to read the non-canceled bookings in the economy class for each flight from database table SBOOK and process the bookings together with the flight. The obvious solution is a nested SELECT, but that solution is likely to perform slowly.

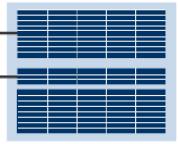
Due to the large data volume involved, do not buffer the bookings completely in an internal table.

Parallel Cursors Instead of Nested SELECTs

```
OPEN CURSOR lv_cursor_sflight FOR
     SELECT * FROM sflight
       ORDER BY carrid connid fldate.

OPEN CURSOR lv_cursor_sbook FOR
     SELECT * FROM sbook
       WHERE class = 'Y' AND cancelled <> 'X'
       ORDER BY carrid connid fldate bookid.
DO.
  FETCH NEXT CURSOR lv_cursor_sflight
        INTO ls_flight.
  IF sy-subrc <> 0.
    EXIT.
  ENDIF.

  CLEAR lt_bookings.
  IF ls_flight-seatsocc > 0.
    FETCH NEXT CURSOR lv_cursor_sbook
          INTO TABLE lt_bookings
          PACKAGE SIZE ls_flight-seatsocc.
  ENDIF.
  ...
ENDDO.

CLOSE CURSOR lv_cursor_sbook.
CLOSE CURSOR lv_cursor_sflight.
```

Two cursors for
- All flights
- Uncancelled bookings in economy class

seastocc = Number of uncancelled bookings in economy class
→ The next seatsocc bookings are bookings for the current flight



The concurrent cursor approach works as follows:

- Each fetch for the first cursor (cursor variable LV_CURSOR_SFLIGHT) places the next respective record from database table SFLIGHT into structure LS_FLIGHT (similar to a SELECT loop).
- Each fetch for the second cursor (cursor variable LV_CURSOR_SBOOK) reads the corresponding non-canceled economy bookings into internal table LT_BOOKINGS. However, these bookings are not actually selected based on their key. Instead, the next n records are simply read. The value of n corresponds to the value of the seatsocc field in the record for the flight. Because the selection results are sorted (ORDER BY clause), the next n bookings are the bookings for the current flight.

SAP - Internet Explorer

Rather than solving the problem with nested SELECTs, you can achieve the same result by addressing the two database tables with two concurrently open database cursors (see Figure 196).

For this technique to work correctly, the table and record design need to fulfill the following prerequisites:

- Both tables must be sorted by the same criteria (ORDER BY clause).
- Every record in the outer table must specify exactly (and correctly) how many corresponding records exist in the inner table.

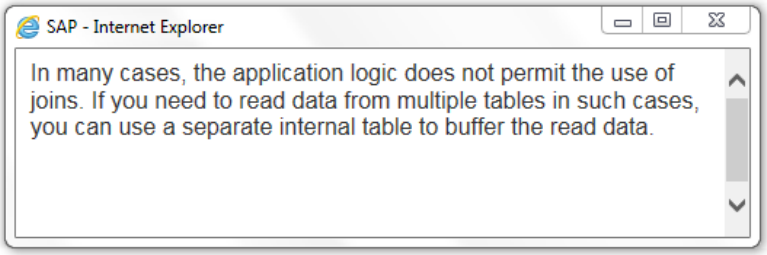
If these conditions for the data are not met, then the concurrent cursor program design will return incorrect results, or fail to work at all. It is therefore a less robust, though much better-performing, approach to the problem.

Hint: To circumvent the second criterion and improve the robustness of the solution, you can open two cursors for the same database table and have the outer cursor use the *count (*)* aggregate function to determine the number of records in each group.

Click Next to continue.

Motivation and Basic Idea

Avoiding database accesses in the first place is the simplest way to reduce database load. You can do this by taking table contents that have already been read and saving them in an internal table (with type SORTED or HASHED, if possible).



Read on Demand and Buffering

```
LOOP AT gt_bookings INTO gs_booking.
  PERFORM read_travelag USING    gs_booking-agencynum
                        CHANGING gs_travelag.
  ...
ENDLOOP .
```

```
FORM read_travelag USING    pv_agencynum
                   CHANGING cs_travelag.

  STATICS st_travelags TYPE HASHED TABLE OF stravelag
                       WITH UNIQUE KEY agencynum.

  IF pv_agencynum IS NOT INITIAL.
    READ TABLE st_travelags INTO cs_travelag
         WITH TABLE KEY agencynum = pv_agencynum.

    IF sy-subrc <> 0.
      SELECT SINGLE * FROM stravelag
                      INTO cs_travelag
                      WHERE agencynum = pv_agencynum.
      INSERT cs_travelag INTO TABLE st_travelags.
    ENDIF.
  ENDIF.
ENDFORM.
```

Static table: Retains the data Hashed table: High-performance read access

First check whether record is already buffered

Only read from the database if necessary

SAP - Internet Explorer

In the example, you want to read the matching data from STRAVELAG (travel agency master data) within a loop for the bookings from table SBOOK. Accomplish this with a SELECT SINGLE statement or a read routine. If you use SELECT SINGLE in this approach, execute identical SQL statements. Therefore encapsulate the reading of STRAVELAG records in a read routine. In the example, the table contents read from STRAVELAG are buffered in a static internal table. Before each database access, the system checks whether the corresponding table entry has already been read.

The read routine can also be a method of a local or global class instead of a subroutine. In this case, a private attribute of the corresponding class is used as buffer instead of the static internal table ST_TRAVELAGS.

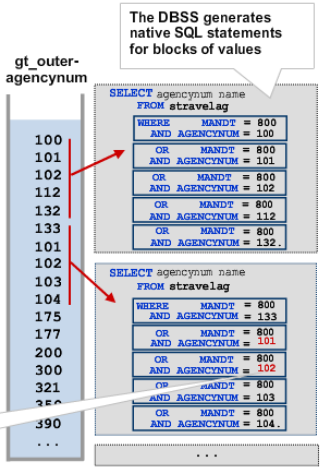
FOR ALL ENTRIES Addition

Join between itab and DB Table

```
* fill gt_outer with
* travel agency numbers

SELECT  agencynum name
  FROM stravelag
  INTO TABLE gt_travelags
  FOR ALL ENTRIES IN gt_outer
    WHERE agencynum =
          gt_outer-agencynum.
```

Caution!
If driving table gt_outer is empty, all entries in DB table STRAVELAG are read!

If gt_outer is not distinct, values will be read more than once from DB and removed from the final result



Rollover for more information

The figure illustrates an example of how the statement works.

Rollover for more information

Rollover for more information

If you want to read large data volumes, use FOR ALL ENTRIES only in exceptional cases.

SELECT ... FOR ALL ENTRIES was created in Open SQL at a time when it was not possible to perform database joins (this was not supported for all SAP-approved DBMS). The connection between the inner and outer database tables was created in ABAP.

Nowadays, this technique is often used when some data is already available in an internal table, but additional data is to be read from the database. In such cases, SELECT ... FOR ALL ENTRIES replaces a SELECT SINGLE statement inside a LOOP over the internal table, and normally shows better performance than such a LOOP.

The content of the internal table (driving table) is used as a restriction for the database access, and may depend on the database system. In general, the database interface takes a certain number of entries in the internal table (in the example it takes 5) and sends one native SQL statement to the database for each group. In the end, the results of the individual native SQL statements are combined to form the result of the Open SQL statement. Duplicate entries are automatically removed. Note that the size of the packages is controlled by a profile parameter.

Click Next to continue

Test Your Knowledge

Determine whether this statement is true or false.

The connection between the inner and outer database tables is created in LOOP.

- ○ True
- ○ False