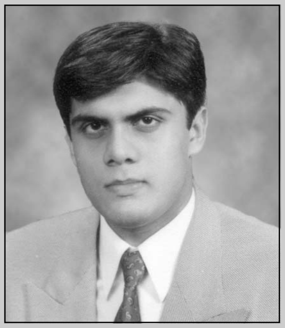
Optimize Database Access and Increase System Performance Through More Efficient ABAP Programming

Rehan Zaidi



Rehan Zaidi is an SAP Technical Consultant at Siemens Pakistan. He has been involved in ABAP development at both in-house and remote implementations of SAP R/3. Most of his work includes designing and writing country- and customer-specific applications for the SAP HR module, and he is also familiar with the BAPI, ALE, and Workflow components.

(complete bio appears on page 22)

I often find ABAP developers struggling to optimize their programs in order to meet their users' performance requirements. An ABAP program can be slow for a variety of reasons, but one of the most common is the presence of long-running SQL statements, which affects a program's overall runtime. These statements can also cause over-consumption of both database and network resources, and could potentially become a threat to your R/3 system's overall performance.

So what causes "expensive" SQL statements? What are the basic techniques for optimizing database access processing? What are the available utilities in an SAP R/3 system for achieving more streamlined and efficient data access? This article answers these questions by providing an overview of database access optimization for ABAP developers who have practical experience in database programming. I will start with a brief introduction to Open SQL and the SAP architecture, and then I will discuss the various techniques available to you for speeding up data access in ABAP programs. Using sample code along the way, I will also illustrate these techniques in action and point out the most common coding mistakes developers make.

Database Access in SAP R/3 — A Brief Background

In an R/3 system, data is stored in a relational database. A relational database consists of tables that are two-dimensional data structures defined by rows (data records) and columns (fields). The smallest possible combination of fields that can uniquely identify each table row is called a *key*. Each table must have at least one key, and each table has one key that is defined as its *primary key*. In a relational model, the relationship between various tables is expressed in terms of *foreign keys*. A combination of fields in a table is termed a foreign key if it is the primary key of another table.

Application programs can address this database by means of a standardized language known as SQL (Structured Query Language), which contains statements for reading, modifying, creating, and administering database tables. SQL is embedded in the ABAP language via two kinds of statements:

- Open SQL (a database-independent subset of standard SQL)
- Native SQL (the language of the database system being used)

Open SQL statements are fully integrated into ABAP and allow you to access data regardless of the database system that the R/3 installation is using. Open SQL consists of the Data Manipulation Language (DML) portion of standard SQL; in other words, it allows you to read and change data. **Figure 1** lists all Open SQL keywords and their respective functions.

Open SQL, along with other ABAP statements, can be used to simplify and speed up database access, as you will see later in this article. It supports buffering of certain tables and statements on the application server, thus preventing excessive database access. Buffers are partly stored in the working memory of the current work process, and partly in the shared memory for all work processes on an R/3 application server. In situations where an R/3 system is distributed across more than one application server, the data in the various buffers is synchronized at set intervals by the buffer management.

Open SQL statements do not access the database directly. They are checked at compile time and converted to the proper SQL statements for the specific database system.

Native SQL statements, in contrast to Open SQL statements, are only loosely integrated into ABAP, but allow access to all of the functions contained in the programming interface of the underlying database system. So, if you want to use specific features of a certain database that are beyond what Open SQL can do, you'll need to use Native SQL. Obviously, programs that use Native SQL become inextricably linked to the database system for which they are written, so when writing R/3 applications, Native SQL should be used as a last resort.

Figure 1 ***Open SQL Keywords and Their Functions***

Keyword	Function
SELECT	Read data from database tables.
INSERT	Insert lines into database tables.
UPDATE	Change the contents of lines in database tables.
MODIFY	Insert lines into database tables or change the contents of existing lines.
DELETE	Delete lines from database tables.
OPEN CURSOR, FETCH, CLOSE CURSOR	Read lines of database tables using a cursor.
COMMIT WORK	Confirm all changes made in a database logical unit of work (LUW*).
ROLLBACK WORK	Undo all changes in a database LUW.

* A sequence of database operations that must be executed either in its entirety or not at all.

SQL and the SAP Architecture

In order to understand how SQL statements affect the runtime of ABAP programs, you need to understand the underlying SAP system architecture.

SAP R/3 is based on a multi-tiered architecture. It consists of three layers:

- The presentation layer
- The application layer
- The database server layer

In a single R/3 installation, there can be many presentation servers (client frontends), and one or more application servers, but there can be only one database server. R/3 supports databases such as SAP DB, Oracle, SQL Server, DB2, and Informix. The ABAP programs containing the SQL statements run on the application server, while the data resides on the database server.

On the database server, large amounts of data are administered using a Relational Database Management System (RDBMS) that maintains the data and the relationships between that data (see **Figure 2**).

Figure 2 *A Typical Architecture of an SAP R/3 System*

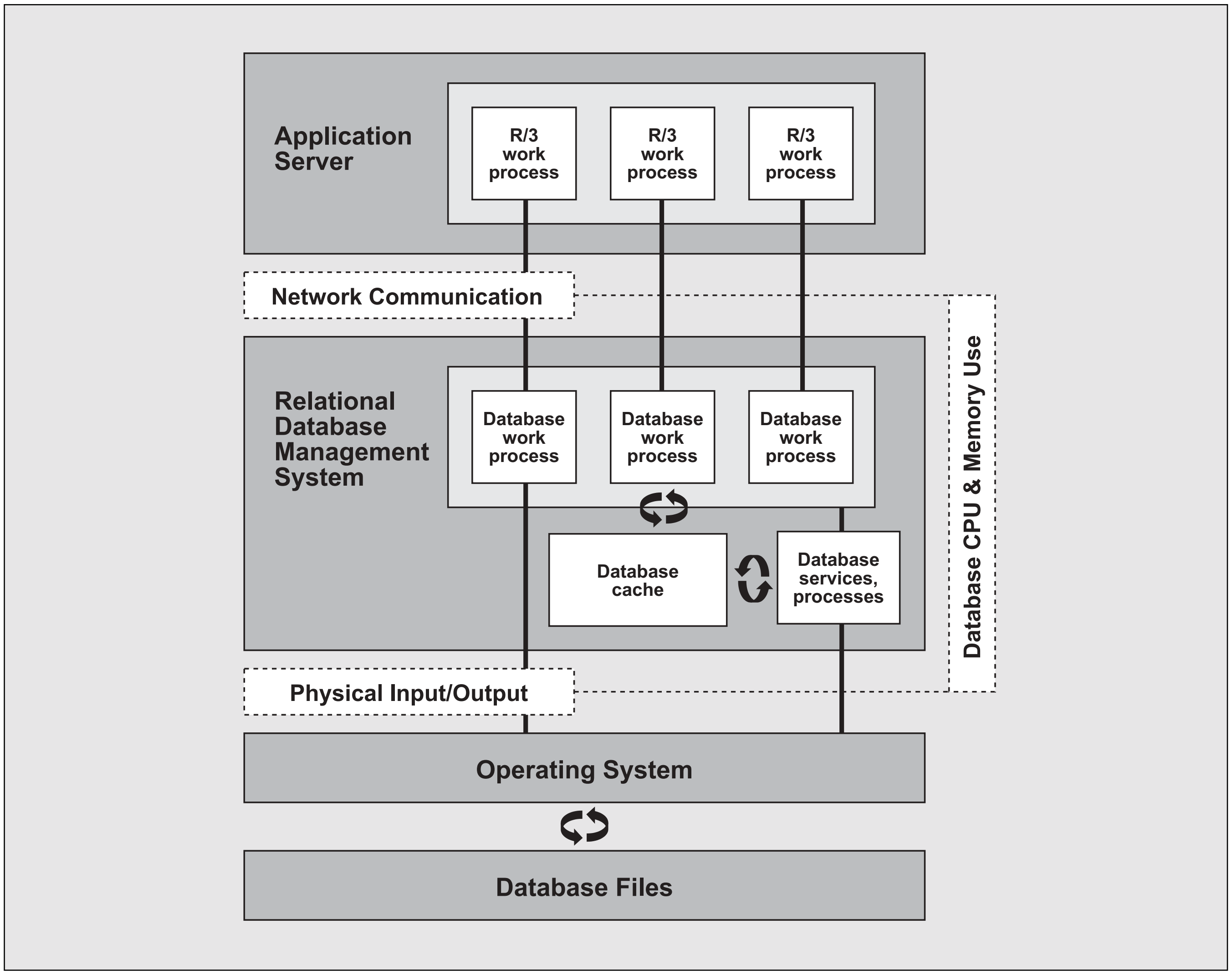
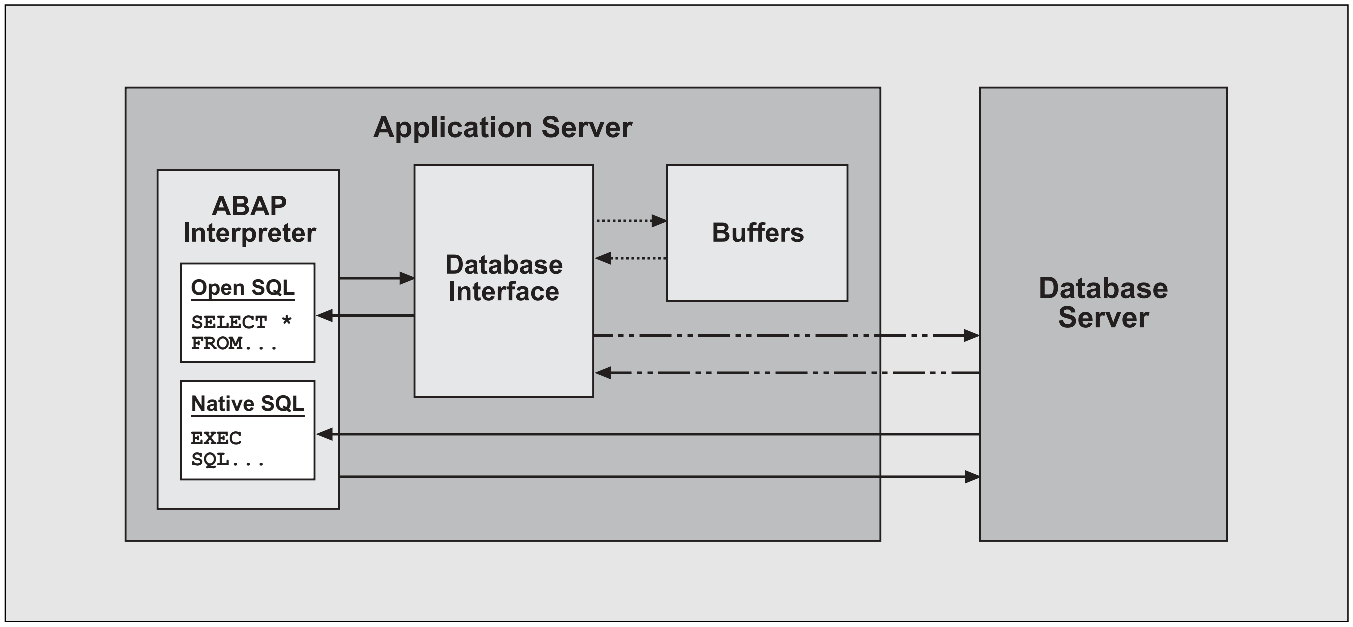


Figure 3 *Executing an Open SQL Statement*



The RDBMS forms the link between the users and the actual data, and provides services that are called by the work processes of the application server. The RDBMS consists of a large memory area containing caches for storing frequently used data. The data is stored permanently on the database server's hard disk and is managed either by the file system or directly by a physical layer of the RDBMS.

Now let's examine in detail what happens when a program containing an Open SQL statement is executed. Suppose you write the following `SELECT` statement in your program:

```
SELECT * FROM SPFLI
  WHERE carrid = 'LH'
  AND   connid = '1234'.
  ...
ENDSELECT
```

This code is executed on the application server (see **Figure 3**). The statement is passed on to the component of the application server known as the *database interface*. If the data requested by the `SELECT` statement is present in the buffers of the application server, it is made available to the program from there. Otherwise, the database interface converts this Open SQL statement into the corresponding Native SQL statement of the underlying RDBMS, and then sends the statement to the database server. The database server determines the rows requested by the Native SQL statement (the *result set*) and sends them back to the application server. The processing of the program then continues.

Keeping the above architecture in mind, the important factors that determine performance are the following:

- **Physical data input/output:** As mentioned earlier, the data is stored on the hard disk of the database server. Reading data from the database server (or writing data to it) involves a large amount of physical data input and output, which can become a bottleneck when large amounts of data are involved.
- **CPU and memory usage:** The more data that is read from the database, and the more frequently it is read, the more CPU resources and memory are

required, and this has an adverse effect on the speed of the program.

- **Network load:** The database server generally resides separately from the application server. Communication between work processes and the database server take place through a network connection. As increasing amounts of data are transferred from the database server to the application server, the network load increases considerably, which adversely affects the speed of the ABAP program.

So how can you prevent your SQL statements from causing these factors to negatively affect your program's performance? By following these five "golden rules" to ensure their efficiency:

1. Optimize the size of the result set (i.e., the number of rows that the statement returns).
2. Optimize the amount of data to be transferred from the database to the application server.
3. Optimize the number of data transfers.
4. Optimize the time required to search the database and retrieve the result set.
5. Optimize the load on the database.

After exploring each of these, I will also demonstrate how to use a special utility called SQL Trace, which shows you how your program interacts with the database. It shows you which Native SQL statements are created from your Open SQL statements and which parameters are passed to the database. It also provides information on how the database determines the result set by showing you the execution plan and, most important, tells you the execution time of each SQL statement. With all these capabilities, SQL Trace is a valuable tool for scrutinizing your program's performance with respect to database accesses.

Let's now take a closer look at each of the five golden rules.

Rule #1: Optimize the Size of the Result Set

The amount of time it takes to access the result set depends on the number of rows found in the database. More rows means a longer transfer period from the database to the application server. To keep this time (and thus its effect on performance) minimal, you need to keep the size of the result set minimal, and the only way to do this is to restrict the data that is selected. By restricting the data up front, you additionally avoid having to later expend resources filtering out unwanted data with additional ABAP coding.

For example, do not select a large data set and then reject unwanted data with `CHECK`. This method is shown in the following example, where all data records from table SFLIGHT are transferred to the application server and discarded immediately, except for those where CARRID equals "LH" and CONNID equals "3577":

☒
```
SELECT * FROM SFLIGHT.
  CHECK SFLIGHT-CARRID = 'LH'
  AND   SFLIGHT-CONNID = '3577'.
  ...
  ...
ENDSELECT.
```

Open SQL provides the `WHERE` and `HAVING` clauses to achieve this purpose more efficiently by allowing the database server to make the selection before the data transfer, so no unwanted data is transferred to the application server. Wherever possible, include all selection conditions in the `WHERE` clause (using `AND`) and use the relational operator `EQ` ("=") to check for equality between fields:

☑
```
SELECT * FROM SFLIGHT
  WHERE CARRID = 'LH'
  AND   CONNID = '3577'.
  ...
  ...
ENDSELECT.
```

Another way of restricting the number of selected rows is to use the `HAVING` clause, which is used in conjunction with the `GROUP BY` clause. It allows you to select groups of rows (`GROUP BY`) that fulfill a certain condition (`HAVING`). Aggregate expressions are used to formulate these conditions. Consider the following example:

```
DATA:  cnt    TYPE I.
SELECT count(*) INTO cnt FROM sflight
       WHERE carrid = 'AU'
       GROUP BY connid
       HAVING SUM( seatsocc ) > 100.
   ...
ENDSELECT.
```

This coding selects the number of flights per connection from airline "AU" whose connections have, in sum, at least 100 passengers. To perform this selection, it not only selects all lines from database table SFLIGHT with the value "AU" for CARRID, but also groups all lines with identical values for CONNID. These groups are then restricted further by the condition that the sum of the contents of the column SEATSOCC for a group must be greater than 100.

If you want to read a single line of a database table, or you are sure that the `WHERE` clause lists all the primary key fields of the table, it is better to use the `SELECT SINGLE` statement. A `SELECT SINGLE` statement is the fastest method if all key fields are checked for equality and are listed in the sequence in which they appear in the table. Suppose table `my_tab` consists of the primary key fields `f1`, `f2`, and `f3`. The `SELECT` statement should look as follows:

```
SELECT SINGLE * FROM my_tab
  WHERE f1 = field1
    AND f2 = field2
    AND f3 = field3.
```

To sum up, keep the result set as minimal as possible. Avoid reading the entire contents of a database table and do not access rows that are not actually needed. Read all lines of a table only in very rare cases.

Rule #2: Optimize the Amount of Data Transferred

Data is transferred in *blocks* from the database server to the application server. The exact size of a block depends on the settings of your R/3 system, but typically the size is about 32 KB. The greater the number of blocks to be transferred, the greater the network load. To optimize programs, the number of blocks must be as minimal as possible.

An ABAP programmer should always try to avoid reading the complete data record (`SELECT *`) and instead read only the data that is actually needed (`SELECT F1, F2, ... FROM`). Suppose you require only the two fields CITYFROM and CITYTO from the table SPFLI, and you want to select only the instances where CARRID equals "SQ" and CONNID equals "0026". In this case, you should avoid the following coding:

```
SELECT * FROM SPFLI
  WHERE CARRID = 'SQ'
  AND   CONNID = '0026'.
    ...
ENDSELECT.
```

A much better approach is to restrict the selection to the fields that are really needed:

```
DATA: FROM LIKE SPFLI-CITYFROM,
        TO LIKE SPFLI-CITYTO.
SELECT CITYFROM CITYTO INTO (FROM,TO)
         FROM SPFLI
        WHERE CARRID = 'SQ'
        AND   CONNID = '0026'.
    ...
ENDSELECT.
```

Here only the fields CITYFROM and CITYTO are transferred to the application server and stored in the variables `FROM` and `TO`, respectively.

If you want to limit a result set to only a certain number of lines, you should always use the `UP TO n ROWS` addition in the `FROM` clause. For example, you should avoid coding like:

```
SELECT * FROM SCARR.
   IF sy-dbcnt = 5.
      EXIT.
   ENDIF.
   ...
ENDSELECT.
```

A more efficient alternative is the following, since it transfers only the number of rows you really want to the application server:

```
SELECT * FROM SCARR UP TO 5 ROWS.
   ...
ENDSELECT.
```

In situations where you want to perform calculations on data, it is advisable to use the aggregate functions that the `SELECT` clause provides. Instead of reading individual entries from the database and performing the calculations yourself on the application server, have aggregate functions perform the calculations on the database side, which is much quicker, especially if you transfer the data solely to compute the result.

The aggregate functions that Open SQL provides are `MAXIMUM`, `MINIMUM`, `AVERAGE`, `SUM`, and the `COUNT` of all values of a particular column as well as the number of rows that meet the selection criterion, i.e., `COUNT(*)`. Let's look at the following coding example:

```
CNT = 0.
SELECT * FROM SPFLI
    WHERE CARRID = 'LH'.
  CNT = CNT + 1.
ENDSELECT.
```

The coding determines the number of connections for airline "LH" and stores the result in variable `CNT` by selecting the required rows from table SPFLI and explicitly counting each row as it is transferred to the application server. Not a very efficient method.

Now look at the following coding:

```
SELECT COUNT(*) INTO CNT FROM SPFLI
  WHERE CARRID = 'LH'.
```

This line of code does exactly the same thing with respect to the final result: counting the number of rows in the result set. However, rather than transferring the data to the application server and counting the rows there, here the counting is performed on the database server side. No need to transfer any data from the result set to the application server, except when the final result is stored in `CNT`. This means that no unnecessary data is transferred, and thus the network load is kept to a minimum.

Avoiding unnecessary transfers is not just a matter of `SELECT` statements. It can also be employed when changing existing data. For example, if you want to change only some columns in certain rows of a database table, use the `UPDATE` ... `SET` statement. First select the desired rows using the `WHERE` clause, then set the desired columns to the new value. This is shown in the following example, which sets the plane type to "A380" and decreases the price for flight "AA1234" by 200:

```
UPDATE SFLIGHT SET
     PLANETYPE = 'A380'
         PRICE = PRICE - '200.00'
  WHERE CARRID = 'AA'
  AND   CONNID = '1234'.
```

Note that using the `UPDATE` ... `SET` statement to make changes in the existing data is much more efficient than using a separate work area to make these changes, since a separate work area would require additional data transfer via the network.

In summary, transfer only data that you really need and move computations to the database side whenever possible.

Rule #3: Optimize the Number of Data Transfers

Remember that for the execution of every SELECT statement, the database interface accesses the database server (via the network), the database server determines the result set (either from scratch or from a cache), and the database server then sends the result set back to the database interface. These activities consume both network and database system resources, thus you should make every effort to minimize the number of times the database is accessed — e.g., by avoiding redundant access to data.

Suppose there is a table ZTAB containing 50 entries and, in your application, you wish to read the table 1,000 times for values based on field f1. One alternative would be to use a SELECT statement each time, as follows:

☒
```
SELECT SINGLE * FROM ztab
  WHERE f1 = fieldvalue.
```

Since table ZTAB in this example is relatively small, a better approach would be to make a replica of the database table in your program and then access the desired line in that table via a READ statement. In order to do so, use the INTO clause and provide a table instead of using the implicit work area:

☑
```
SELECT * FROM ztab INTO TABLE itab.
```

Then retrieve the data from that table via a simple READ statement:

☑
```
READ TABLE ztab WITH KEY
  f1 = fieldvalue.
```

This method is much quicker since activity happens only on the application server side and there is no need to access the database.

Of course, with a large table, the above strategy would violate Rule #1 ("Optimize the Size of the Result Set"), so avoid such a coding practice when the number of rows is very large. For example, if table ZTAB contains 10,000 entries and you need to read it only 100 times, it is better to use the SELECT SINGLE statement instead of populating an internal table.

In some situations, you may find that a SELECT statement is executed many times with the same set of values passed in the WHERE clause. As an example, take a look at the following statement:

☒
```
SELECT SINGLE * FROM t512t
  WHERE spras = sy-langu
  AND   lgart = w1.
```

Suppose the above statement is written in a loop and there is a chance that the value of w1 is identical to what it was in the previous iteration (loop cycle). In that case, since the value of sy-langu remains the same throughout, the SELECT is called despite the fact that the work area t512t already contains the desired data.

A better way is to omit the unnecessary SELECT in this case, as shown below:

☑
```
IF w1 <> t512t-lgart
      OR t512t-spras <> sy-langu.
  SELECT SINGLE * from t512t
    WHERE spras = sy-langu
    AND   lgart = w1.
* else data already there
ENDIF.
```

Here, an IF statement is used to call the SELECT only when the value has really changed.

Nested SELECT loops are a serious problem. In a relational data model, it is quite common for logically interconnected data to be split up into several tables. In situations where a developer has to access the interconnected data, many developers often employ a nested SELECT statement to enable this, with disastrous effects on the overall performance.

As an example, consider the tables SPFLI and SFLIGHT, which are interconnected by the fields CARRID and CONNID. Suppose you wish to find all the carriers and their respective details. You can use the following simple piece of code:

```
☒ DATA: CARR  TYPE SPFLI-CARRID,
        CONN  TYPE SPFLI-CONNID,
        SEATS TYPE SFLIGHT-SEATSOCC.
  SELECT CARRID CONNID INTO (CARR,
         CONN) FROM SPFLI.
      SELECT SEATSOCC INTO SEATS FROM
             SFLIGHT
        WHERE carrid = CARR
        AND   connid = CONN.
        ...
      ENDSELECT.
      ...
  ENDSELECT.
```

This construct, without a doubt, provides the right result. But it is slow and unnecessarily consumes a lot of system resources. The inner SELECT statement (on table SFLIGHT) is executed for every data record of the outer SELECT statement (on table SPFLI), which can become a hassle if the outer SELECT yields a large number of rows. Suppose the number of rows from table SPFLI is 100 — the inner SELECT is then also processed 100 times. The total number of database accesses (SELECTs) amounts to 101 (1 for the outer SELECT, and then 1 for each of the 100 rows of the result set). Since every SELECT consumes a certain amount of time (and resources), the large number of SELECTs could slow the program significantly.[1]

There are two methods available to you for solving this problem:

- ABAP JOINs — Nesting of SELECT loops can be avoided by using the JOIN command in a SELECT statement.[2] It enables you to link tables based on certain conditions between their related fields. The condition between the outer and inner tables is specified using an ON clause. So, an equivalent statement to the previous nested loop example that uses a JOIN instead is:

```
☑ SELECT P~CARRID P~CONNID
         F~SEATSOCCINTO
         (CARR, CONN, SEATS)
    FROM SFLIGHT AS F INNER
         JOIN SPFLI AS P
      ON  F~CARRID = P~CARRID
      AND F~CONNID = P~CONNID.
    ...
  ENDSELECT.
```

  The above program fragment results in a single SELECT loop, which means 1 database access instead of the 101 in the previous example, thus increasing the overall program performance. Moreover, less data is transferred from the database server to the application server. To determine the result of a SELECT statement, the database system creates a temporary table containing the lines that meet the ON condition. Any WHERE condition is then applied to the temporary table. After that, only the data that is actually needed is transferred to the application server.

- Dictionary Views — An alternative is to define JOINs as views in the database. Views are statically and globally defined in the ABAP Dictionary and can be used by all ABAP programs. Views are implemented in the ABAP Dictionary as INNER JOINs. The fields that are common to both tables are transferred only once from the database server to the application server. Views are more efficient than implementing ABAP JOINs since they utilize the buffering advantages offered by the SAP system.

[1] Note that the result set is transferred in blocks, so not every loop through a SELECT ... ENDSELECT results in a database access. If this were the case, the number of accesses would increase to 100 * <average size of result set for inner SELECT>. Of course, however, there are more than 101 network transfers, depending on the block size and the size of the result set in bytes.

[2] For simplicity, I have used the INNER JOIN. However, you can also use a LEFT [OUTER] JOIN. In addition to what the INNER JOIN does, the OUTER JOIN also includes in the result set the fields in the left-hand table for which there is no corresponding match in the right-hand table. The fields that are supposed to be filled by data from the right-hand table are assigned NULL values.

Figure 4 *A Comparison of ABAP Dictionary Views and ABAP JOINs*

Dictionary View	Explicit JOIN
Only INNER JOIN possible	INNER JOIN and OUTER JOIN permissible
Buffering advantages can be utilized	Buffering bypassed by ABAP JOIN
Globally defined — can be used in more than one program	Defined locally in a program, so only this program can utilize it
JOIN condition statically defined	JOIN condition can be varied in program according to the user's requirement

Figure 4 shows a comparison of the Dictionary View method and the ABAP `JOIN` method.

Another method for reducing the number of data transfers and for programming complex database operations is to use subqueries in the `WHERE` clause. Subqueries give you an efficient way to access more than one database table in the same Open SQL statement. The data from the subquery is not transferred to the application server, but instead the subquery is evaluated in the database system.

For example, consider the following coding:

```
DATA wa TYPE sflight.
SELECT * FROM sflight AS s INTO wa
  WHERE EXISTS ( SELECT * FROM spfli
    WHERE carrid = s~carrid ).
  ...
ENDSELECT.
```

In contrast to the nested `SELECT` example on the previous page, here only those rows where the field CARRID of table SPFLI equals the field CARRID of table SFLIGHT are transferred to `wa`.

When using the statements `INSERT`, `UPDATE`, and `DELETE` for more than one row of a database table, it is more efficient to use an internal table followed by a single Open SQL statement, instead of calling an Open SQL statement for each row. An example of this is the statement:

```
UPDATE dbtab FROM TABLE itab.
```

This statement enacts a mass update of several lines in a database table. Here, the key for selecting the lines to be updated and the values to be changed are taken from the lines of the internal table `itab`.

To sum up, minimize the number of data transfers. Each data transfer adds to a program's execution time.

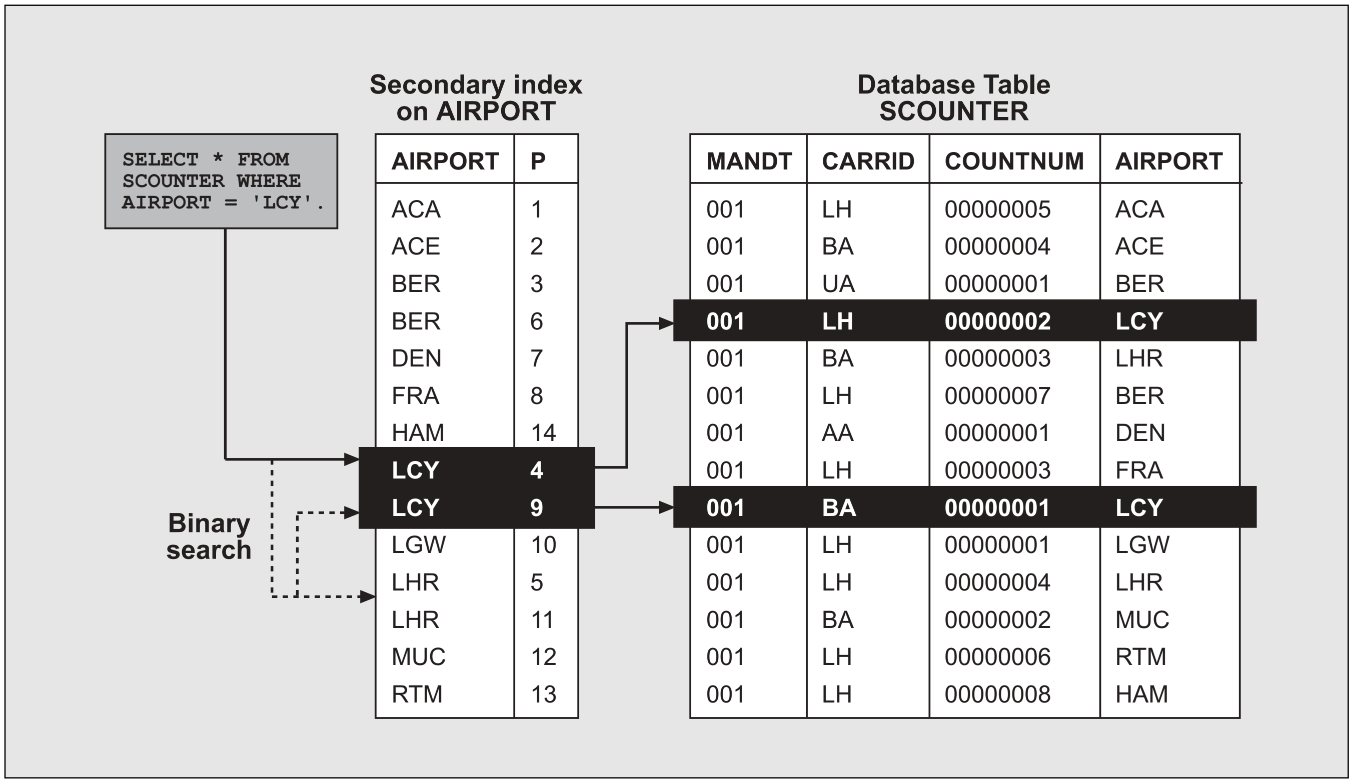
Rule #4: Optimize the Cost of Database Searches

The time required to search for the result set depends on the number of rows in the database table. Databases provide a mechanism known as *indexing* to speed up the selection of records from a large database table. Make every effort to formulate your `SELECT` clauses according to the table indexes.

Think of an index as a copy of a portion of a database table. The fields of the index are stored in a sorted form, which enables faster access to the records of the table (using algorithms such as binary search). The values listed in the index represent the search criteria of a `WHERE` clause. The index contains a pointer that links the index entry to the corresponding entry in the database table (see **Figure 5**).

Suppose we write the following Open SQL statement:

Figure 5 ***Linking the Index Entry to the Database Table***



```
SELECT * FROM
SCOUNTER WHERE
AIRPORT = 'LCY'.
```

Secondary index on AIRPORT

AIRPORT	P
ACA	1
ACE	2
BER	3
BER	6
DEN	7
FRA	8
HAM	14
LCY	**4**
LCY	**9**
LGW	10
LHR	5
LHR	11
MUC	12
RTM	13

Database Table SCOUNTER

MANDT	CARRID	COUNTNUM	AIRPORT
001	LH	00000005	ACA
001	BA	00000004	ACE
001	UA	00000001	BER
001	**LH**	**00000002**	**LCY**
001	BA	00000003	LHR
001	LH	00000007	BER
001	AA	00000001	DEN
001	LH	00000003	FRA
001	**BA**	**00000001**	**LCY**
001	LH	00000001	LGW
001	LH	00000004	LHR
001	BA	00000002	MUC
001	LH	00000006	RTM
001	LH	00000008	HAM

☑
```
SELECT * FROM SCOUNTER
  WHERE AIRPORT = 'LCY'.
```

Let us assume that an index called "AIRPORT" exists for the table SCOUNTER. When the above query is sent to the database server, the database optimizer locates the index that is best for accessing the data (in this example, the AIRPORT index). So, instead of searching the entire SCOUNTER table for the rows where the field AIRPORT is equal to "LCY," the optimizer carries out a binary search in the AIRPORT index, as shown in Figure 5. This is known as an *index-range scan*. Via the index, the optimizer finds the sequential numbers of the rows (in the actual table) that fulfill the query criteria (`WHERE AIRPORT = 'LCY'`). Each index row contains a pointer to the corresponding record of the table SCOUNTER. These pointers (or sequential numbers) are used as a link to the complete data record so that the fields not contained in the index can also be read. The result set is then determined and transported to the application server for further processing.

In an SAP R/3 system, each database table has a primary index that consists of the table's key fields. The index is created and administered by the R/3 system, and is automatically created in the database when the table is activated. The primary index is an example of a *unique index*.[3] When using the primary index, it is always fastest to mention all of the primary key fields with an `EQ` operator in the `WHERE` clause. If none of the primary index fields appear in the `WHERE` clause of your `SELECT` statement, the system searches the entire table to determine the result set. This is known as a *full-table scan*, which can become a substantial bottleneck if there are a large number of rows in that table. For

[3] An index of a table is a unique index if the fields contained in the index uniquely identify each record of the table (i.e., there are no two records in the table that have the same index).

such situations, SAP allows you to create secondary indexes (via transaction SE11) that can restrict the number of entries searched.

It is worth creating secondary indexes only when you want to select table entries based on fields not contained in the primary index and the search is very slow.

Conditions to Keep in Mind When Using Indexes

✓ The index search is very fast if you check all index fields for equality and combine them with the AND operator. For example, in the following SELECT statement, the primary index (CARRID, CONNID) is used:

```
☑ SELECT SINGLE * FROM SFLIGHT
    WHERE CARRID = 'LH'
    AND   CONNID = '1234'.
```

Even if not all index fields are mentioned in the WHERE clause, a quick search can be achieved by specifying the fields in the same sequence in which they appear in the index definition.

✓ Avoid using the NOT operator in the WHERE clause, since this will prevent the database optimizer from using the index. The database system supports only SQL statements that are defined in positive terms — for example, EQ and LIKE (not NE and NOT LIKE). If possible, invert the logical expression instead.

The database optimizer halts when it sees an inner OR in an expression of an SQL statement. Try to reformulate such conditions into forms relevant to the index, such as an IN condition. For example, you should avoid the following SELECT statement, which has an inner OR:

```
☒ SELECT * FROM SPFLI
    WHERE   CARRID = 'AU'
    AND ( CITYFROM = 'FRANKFURT'
    OR    CITYFROM = 'NEW YORK' ).
```

It's better to rephrase the WHERE condition in a way that the OR is outmost — i.e., the WHERE condition consists solely of subexpressions that are combined by OR and contain only AND or IN:

```
☑ SELECT * FROM SPFLI
    WHERE ( CARRID = 'AU'
    AND     CITYFROM = 'FRANKFURT' )
    OR    ( CARRID = 'AU'
    AND     CITYFROM = 'NEW YORK'  ).
```

While defining secondary indexes, fields that are highly selective[4] should be included. It is best to place such fields at the beginning of the index so that there are fewer rows on which to perform the binary search, and thus less time is required.

✓ Try not to create more than five indexes for a single database table. First, whenever table fields are changed, the indexes also need to be updated, which places a load on the database server. Second, there are more chances for the database optimizer to choose a "wrong" index. To ensure that this does not happen, the fields of the indexes should be as disjunctive as possible.

✓ An index should consist of no more than four fields of a table. The index has to be updated each time you modify its fields in a database operation, which then places an extra load on the database server.

✓ In extreme situations — e.g., if your SQL statement is rather complex — the database optimizer might not use any index and will try to search the entire table for the result set. This can be avoided by providing the optimizer with HINTS, which (as the name indicates) provide the optimizer with an indication of the index to be used for a particular table.

[4] A field is highly selective if a small proportion of table rows can be selected by using it (i.e., a field that significantly restricts the set of results in a selection).

Open SQL allows you to specify `HINTS` in the `SELECT` statement by using the `%_HINTS` clause. For more information on specifying `HINTS`, refer to OSS Note 129385.

In summary, keep the cost of searching the database to a minimum. A slow search could result in increased runtime and thus decreased performance.

Rule #5: Optimize the Load on the Database

As mentioned earlier, there may be many presentation or application servers, but there is just one database server in a single R/3 installation. If you aren't careful, the database server can quickly become overloaded, so make every effort to reduce the load on the database.

There are several different ways to reduce the load, including the following:

- Use table buffering.
- Avoid rereading the same data.
- Use the ABAP `SORT` clause instead of `ORDER BY`.
- Avoid the `SELECT DISTINCT` statement.
- Use logical databases.

In the following sections, I will describe these methods in more detail.

Use Table Buffering

SAP provides a buffering facility for tables in order to increase performance when records are being read. Since the buffers reside on the application server, it takes considerably less time to access a table that is buffered locally rather than reading it from the database server. In fact, SAP testing indicates that buffering improves accessing time by a factor of 10 to 100, an improvement I have also noticed in my own experiences. For example, reading an entry from table T001 can take between 8 and 600 milliseconds, while reading the same entry from the buffer takes between 0.2 and 1 millisecond.

The R/3 system manages and synchronizes the buffers on the application servers. If an ABAP program requests data from a particular table, the database interface first checks whether the requested data is already available in the buffers of the application server. In that case, the data is read directly from the local buffer. Otherwise, it is read from the database and then copied into the buffer in order to satisfy the next access of this data.

Now let's take a closer look at what happens when a record is read from a buffered table. Assume a program requests some data from a table. If the table is defined as *buffered* in the ABAP Dictionary, the system checks the local buffer on the application server to see whether any records of the table are already there. If the data is present in the buffer, it is passed on to the program. If not, the request is sent to the database server, the data is then read from there, and the buffers are updated according to the *type of buffering* used. The buffering type defines which table records are loaded into the buffer of the application server when a table record is accessed.

There are three types of buffering:

- **Full buffering:** In this case, all records of the table are loaded into the buffer when a single record is accessed from the buffered table.
- **Generic buffering:** When a record from the buffered table is accessed, all records whose generic key fields match those of the accessed record are loaded into the buffer. The generic key field is the leftmost portion of the table's primary key. (Remember that a key is a combination of fields that uniquely identifies a table row, and that each table has one key that is its primary key.) The generic key field must be defined when the buffering type is selected.

- **Single record buffering:** In this case, only the records that are actually accessed are loaded into the buffer.

When a buffered record changes, the buffer on the application server is updated and the database interface logs this change in the table DDLOG. If the installation contains more than one application server, the other application server(s) periodically read the contents of table DDLOG to update their buffers. This is known as *synchronization* and typically occurs every 60 seconds.

Tables that are often read and rarely changed are ideal candidates for buffering. Tables whose contents are frequently changed should not be buffered, since the buffers need to be updated each time the rows are overwritten. This may increase both the load on the application server and on the network.

Not all statements can be supplied from the buffer, so when working with buffered tables, it is important to know which statements will bypass the buffer. When using one of the following statements, you cannot take advantage of the maximum speed buffering can provide:

- The `DISTINCT` addition, or any aggregate expressions in the `SELECT` clause
- The `ORDER BY` clause or a `GROUP BY / HAVING` clause
- Any `WHERE` clause that contains a subquery or an `IS NULL` expression
- Any `JOIN`s
- A `SELECT ... FOR UPDATE`
- All Native SQL statements

If you want your statement to bypass the buffer in any case — e.g., to be sure to get the very latest data records (and not some likely outdated ones) — you can simply use the `BYPASSING BUFFER` addition in the `SELECT` clause.

Avoid Rereading the Same Data

If you read the same data over and over again, you increase both the number of database accesses and the load on the database. In addition, on some database systems, a “dirty” read may occur, which means that the second time the data is read from the database table, it may be different from the data read the first time. In order to avoid all this, read the data once and store it in an internal table (refer back to Rule #3, “Optimize the Number of Data Transfers”).

Use the ABAP SORT Clause Instead of ORDER BY

In order to sort the selected lines of a database table, programmers usually use the `ORDER BY` clause in the `SELECT` statement. Two of its common forms are `ORDER BY PRIMARY KEY` and `ORDER BY f1 … fn`. In the latter form, the database optimizer might not choose the correct index to form the result set because the optimizer does not support the `ORDER BY f1 … fn` form in the `SELECT` statement. In this case, sorting the result set places an extra load on the database server, particularly when the hit list is very large.

Since the database server is usually the bottleneck, and other application servers might have statements waiting for execution, sometimes it’s better for the overall performance to move the sort effort from the database server to the application server. You can shift the task to sort by reading the result set into an internal table and then sort it using the ABAP command `SORT`:

```
☑ DATA flights TYPE TABLE OF sflight.
  SELECT * FROM sflight INTO TABLE
    flights.
  SORT flights BY currency price.
```

Of course, the success of this measure depends largely on the size of the result set. For very large result sets, this might not be a feasible solution and you would want to let the database server sort it.

Avoid the SELECT DISTINCT Statement

Similar to the `ORDER BY` clause, it is better to avoid the `SELECT DISTINCT` variation of the `SELECT` statement in cases where some of the fields to be distinct are not part of an index. The database server has to do more work in that case since it cannot immediately identify duplicates. If you really need distinct rows, an efficient alternative may be to read all of the data into an internal table (`itab`) on the application server, sort this table using the `SORT` command, and then use the `DELETE ADJACENT DUPLICATES` command to delete all redundant rows transferred from the database server. This can greatly reduce the load on the database server. Of course, it depends on the number of duplicates, since effects from the transfer of a larger amount of data can outweigh the savings on the database side.

For example, to determine all destinations for Lufthansa from Frankfurt, instead of using:

```
☒ SELECT DISTINCT cityto FROM spfli
      INTO TABLE itab
    WHERE carrid = 'LH'
    AND cityfrom = 'FRANKFURT'.
```

the following will likely be better for overall performance:

```
☑ SELECT cityto FROM spfli
      INTO TABLE itab
    WHERE carrid = 'LH'
    AND cityfrom = 'FRANKFURT'.
  SORT itab BY cityto.
  DELETE ADJACENT DUPLICATES
      FROM itab COMPARING cityto.
```

Here, the burden of rejecting duplicates is moved from the database server to the application server. Of course, this comes with additional costs, as a result of transferring data to the application server that is eventually discarded (contradicting Rule #1, "Optimize the Size of the Result Set"). So use that method only if the number of duplicates is small in comparison to the number of records in the result set. If the number of duplicates is large, it is better to let the database do the job.

Use Logical Databases

SAP provides many logical databases for reading records from the database. Logical databases are ABAP programs that decouple Open SQL statements from application programs and have been optimized by SAP for the best database performance.

However, it is important for the ABAP developer to select the right logical database, otherwise it can have a negative effect on performance. The sequence of data that you want to read must correspond to the structure of the logical database chosen. A logical database has a hierarchical tree-like structure of tables where there is one root node table (the highest level) from which various levels of subnodes are derived (linked). Thus, for example, it is not wise to employ a logical database to read from a table's lowest subnode, because the system will have to read at least the key data from all tables (starting from the root) before finally reading the data from the required table. In this case, it is advisable to use a `SELECT` statement instead, which is more efficient because no unnecessary tables are read.

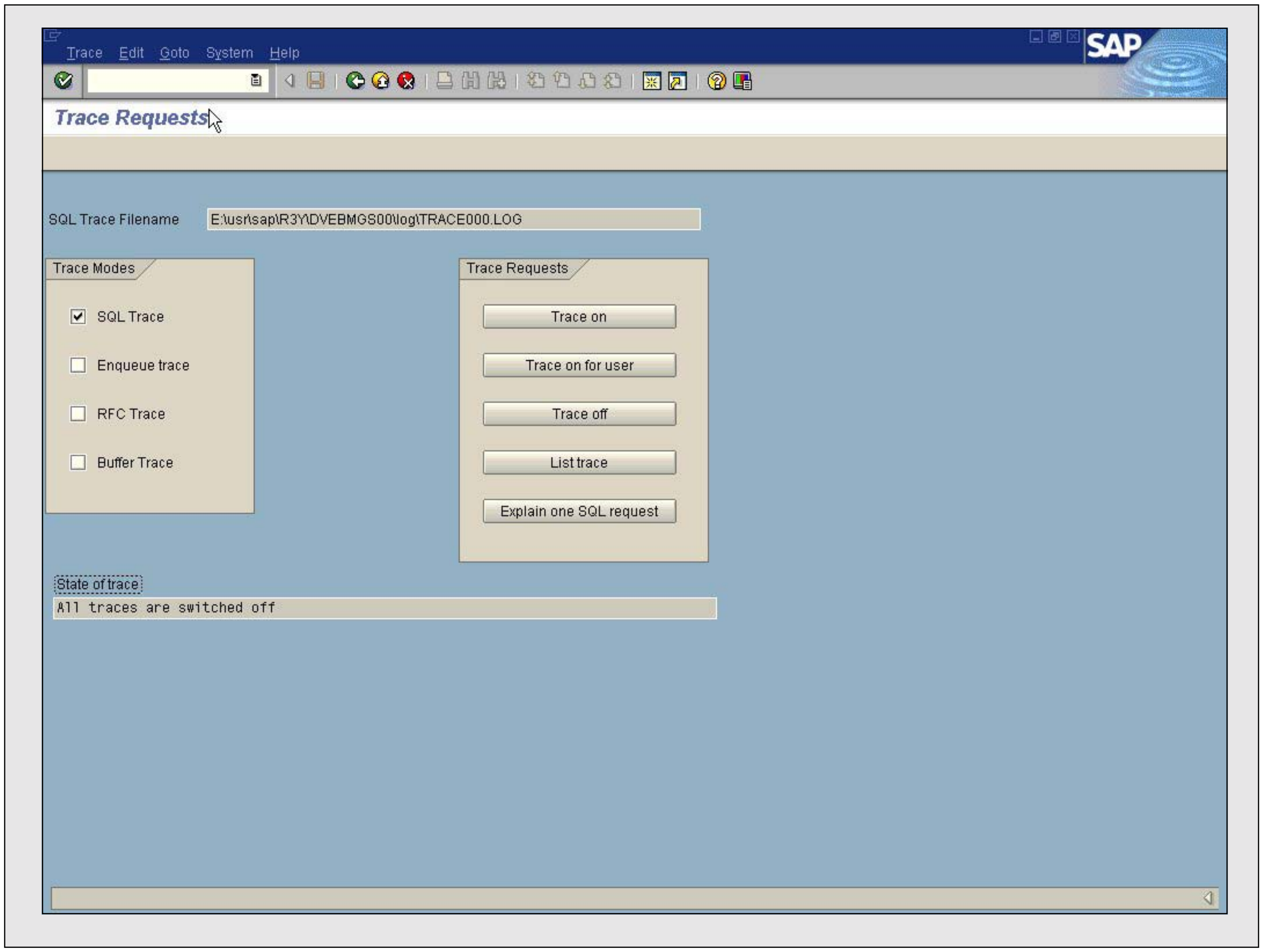
To sum up, reduce the load from the database. Since there is only one database server in an R/3 installation, any unnecessary load on it might result in a decline of the system's overall performance.

Using SQL Trace

If, after following the five "golden rules" and applying the techniques described so far, your program's performance is still too slow, or you simply want to analyze your program's database accesses and `COMMIT`s, you might want to use a special utility called SQL Trace (transaction ST05).

SQL Trace allows you to see how the Open SQL statements in your ABAP programs are converted to Native SQL statements, and the parameters with which the Native SQL statements are passed to the database. In addition, you can view the runtimes of

Figure 6 *The SQL Trace Main Screen*



SQL Trace Filename: E:\usr\sap\R3Y\DVEBMGS00\log\TRACE000.LOG

Trace Modes:
- [✓] SQL Trace
- [ ] Enqueue trace
- [ ] RFC Trace
- [ ] Buffer Trace

State of trace: All traces are switched off

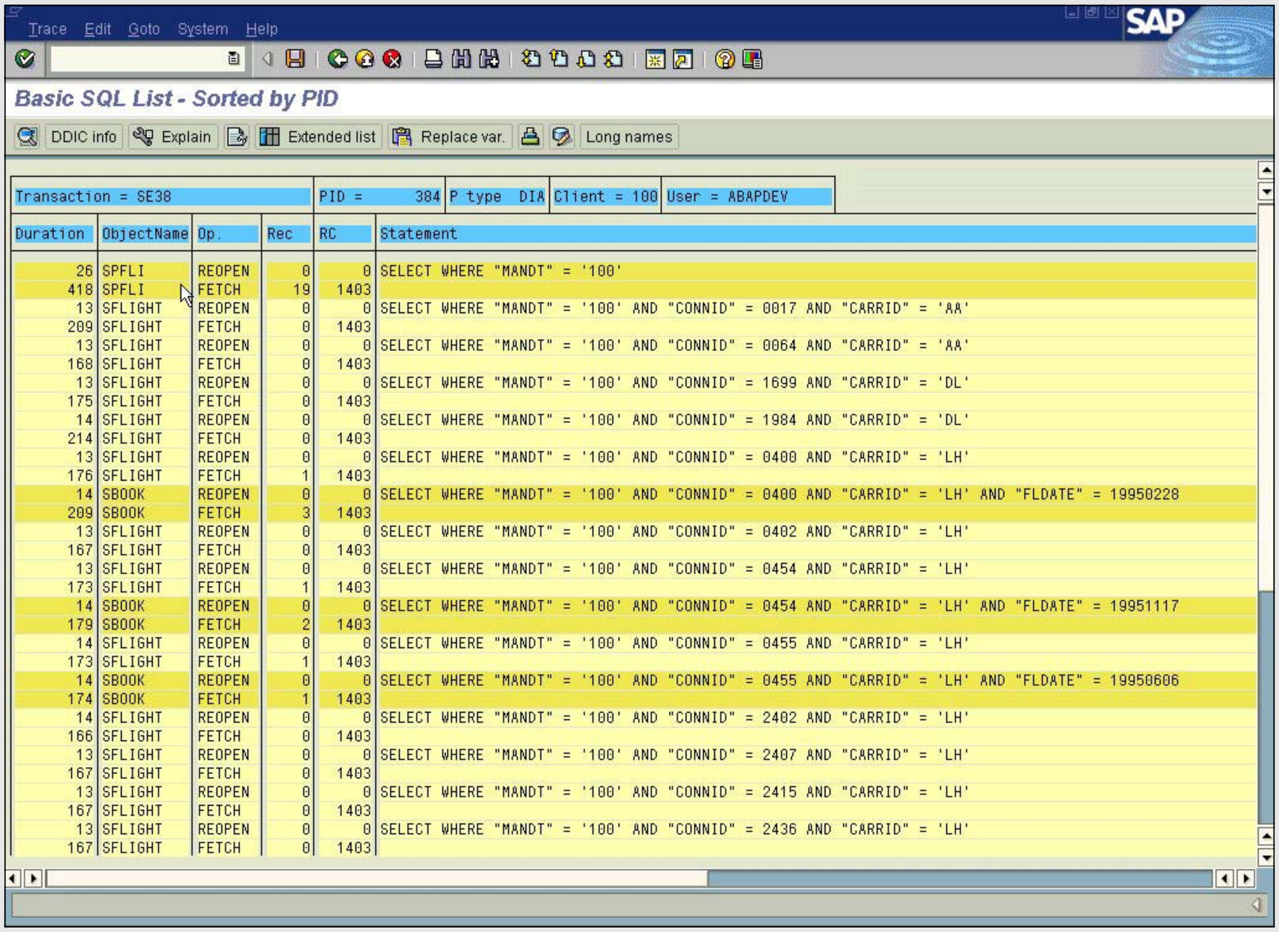
each individual SQL statement, which are not necessarily identical to the Open SQL statements of ABAP. Traces are stored on the application server as text files and any outdated traces are automatically overwritten by the system.

In order to carry out a trace, proceed as follows:

1. In one session, start the SQL Trace using transaction ST05. The main screen appears, as shown in **Figure 6**.
2. Select the "Trace on" button in the "Trace Requests" frame of the main SQL Trace screen. The trace is activated for the user — i.e., all the database operations under your user ID are recorded.
3. In a second session, run the program that you want to observe. If you want to trace a single statement, use the ABAP Debugger to stop the program right before that statement, select the "Trace on" button, and then proceed using the debugger.
4. In the second session, select the "Trace off" button and then "List trace."

Figure 7 shows a typical SQL Trace results list

Figure 7 A Typical SQL Trace Results List



Basic SQL List - Sorted by PID

```
Transaction = SE38           PID =      384 P type  DIA Client = 100 User = ABAPDEV

Duration ObjectName Op.     Rec  RC    Statement

      26 SPFLI      REOPEN    0     0 SELECT WHERE "MANDT" = '100'
     418 SPFLI      FETCH    19  1403
      13 SFLIGHT    REOPEN    0     0 SELECT WHERE "MANDT" = '100' AND "CONNID" = 0017 AND "CARRID" = 'AA'
     209 SFLIGHT    FETCH     0  1403
      13 SFLIGHT    REOPEN    0     0 SELECT WHERE "MANDT" = '100' AND "CONNID" = 0064 AND "CARRID" = 'AA'
     168 SFLIGHT    FETCH     0  1403
      13 SFLIGHT    REOPEN    0     0 SELECT WHERE "MANDT" = '100' AND "CONNID" = 1699 AND "CARRID" = 'DL'
     175 SFLIGHT    FETCH     0  1403
      14 SFLIGHT    REOPEN    0     0 SELECT WHERE "MANDT" = '100' AND "CONNID" = 1984 AND "CARRID" = 'DL'
     214 SFLIGHT    FETCH     0  1403
      13 SFLIGHT    REOPEN    0     0 SELECT WHERE "MANDT" = '100' AND "CONNID" = 0400 AND "CARRID" = 'LH'
     176 SFLIGHT    FETCH     1  1403
      14 SBOOK      REOPEN    0     0 SELECT WHERE "MANDT" = '100' AND "CONNID" = 0400 AND "CARRID" = 'LH' AND "FLDATE" = 19950228
     209 SBOOK      FETCH     3  1403
      13 SFLIGHT    REOPEN    0     0 SELECT WHERE "MANDT" = '100' AND "CONNID" = 0402 AND "CARRID" = 'LH'
     167 SFLIGHT    FETCH     0  1403
      13 SFLIGHT    REOPEN    0     0 SELECT WHERE "MANDT" = '100' AND "CONNID" = 0454 AND "CARRID" = 'LH'
     173 SFLIGHT    FETCH     1  1403
      14 SBOOK      REOPEN    0     0 SELECT WHERE "MANDT" = '100' AND "CONNID" = 0454 AND "CARRID" = 'LH' AND "FLDATE" = 19951117
     179 SBOOK      FETCH     2  1403
      14 SFLIGHT    REOPEN    0     0 SELECT WHERE "MANDT" = '100' AND "CONNID" = 0455 AND "CARRID" = 'LH'
     173 SFLIGHT    FETCH     1  1403
      14 SBOOK      REOPEN    0     0 SELECT WHERE "MANDT" = '100' AND "CONNID" = 0455 AND "CARRID" = 'LH' AND "FLDATE" = 19950606
     174 SBOOK      FETCH     1  1403
      14 SFLIGHT    REOPEN    0     0 SELECT WHERE "MANDT" = '100' AND "CONNID" = 2402 AND "CARRID" = 'LH'
     166 SFLIGHT    FETCH     0  1403
      13 SFLIGHT    REOPEN    0     0 SELECT WHERE "MANDT" = '100' AND "CONNID" = 2407 AND "CARRID" = 'LH'
     167 SFLIGHT    FETCH     0  1403
      13 SFLIGHT    REOPEN    0     0 SELECT WHERE "MANDT" = '100' AND "CONNID" = 2415 AND "CARRID" = 'LH'
     167 SFLIGHT    FETCH     0  1403
      13 SFLIGHT    REOPEN    0     0 SELECT WHERE "MANDT" = '100' AND "CONNID" = 2436 AND "CARRID" = 'LH'
     167 SFLIGHT    FETCH     0  1403
```

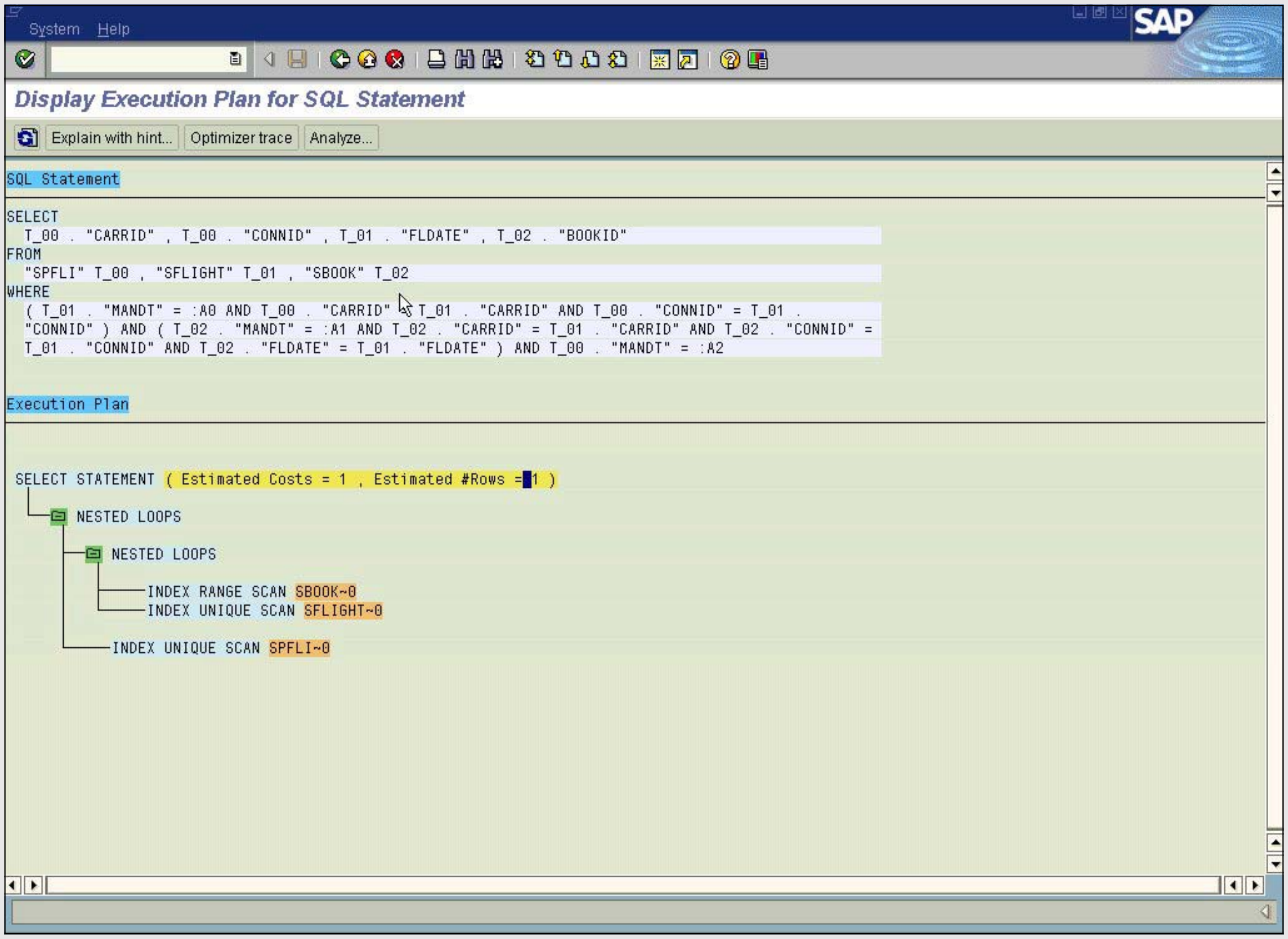
generated from the following Open SQL statements (i.e., the program run in step 3):

```
SELECT connid carrid INTO (con, car)
                FROM spfli.
  SELECT fldate INTO fld FROM
      sflight WHERE connid = con
              AND   carrid = car.
    SELECT bookid INTO id FROM
        sbook WHERE connid = con
              AND   carrid = car
              AND   fldate = fld.
    ENDSELECT.
  ENDSELECT.
ENDSELECT.
```

The format of the generated output depends on the database system used. As you can see in Figure 7, the columns in the list are:

- **Duration:** The execution time of the individual SQL operations in milliseconds. Longer times would be displayed red.
- **ObjectName:** This indicates the dictionary object whose data is accessed. It may be a table or a view.
- **Op (Operation):** The database operations involved. Typical examples are FETCH, (RE)OPEN, and RE(EXEC).

Figure 8 *The Open SQL Statement Execution Plan*


System Help
Display Execution Plan for SQL Statement
Explain with hint... Optimizer trace Analyze...
SQL Statement
Execution Plan

```
SELECT
  T_00 . "CARRID" , T_00 . "CONNID" , T_01 . "FLDATE" , T_02 . "BOOKID"
FROM
  "SPFLI" T_00 , "SFLIGHT" T_01 , "SBOOK" T_02
WHERE
  ( T_01 . "MANDT" = :A0 AND T_00 . "CARRID" = T_01 . "CARRID" AND T_00 . "CONNID" = T_01 .
  "CONNID" ) AND ( T_02 . "MANDT" = :A1 AND T_02 . "CARRID" = T_01 . "CARRID" AND T_02 . "CONNID" =
  T_01 . "CONNID" AND T_02 . "FLDATE" = T_01 . "FLDATE" ) AND T_00 . "MANDT" = :A2
```

```
SELECT STATEMENT ( Estimated Costs = 1 , Estimated #Rows = 1 )
  └─ NESTED LOOPS
       ├─ NESTED LOOPS
       │    ├──INDEX RANGE SCAN SBOOK~0
       │    └──INDEX UNIQUE SCAN SFLIGHT~0
       │
       └────INDEX UNIQUE SCAN SPFLI~0
```

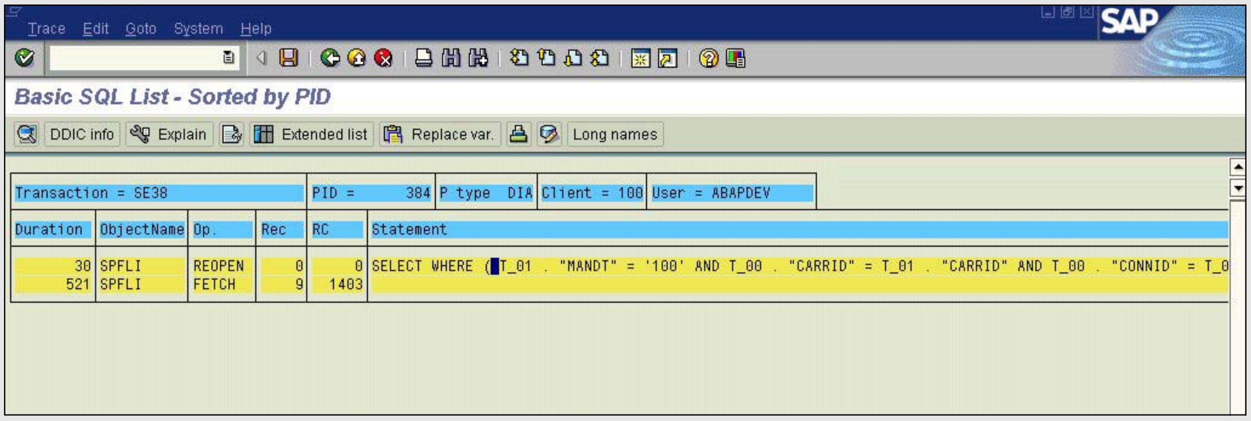
- **Rec (Records):** The number of records found or processed.

- **RC (Return Code):** This indicates whether an operation was successful or not. A return code of "0" indicates a successful operation.

- **Statement:** The statement being traced.

With nested `SELECT` statements (refer back to Rule #3, "Optimize the Number of Data Transfers"), an inner `SELECT` loop is executed for every outer `SELECT` statement, resulting in a large number of database accesses. In the example here, a `SELECT` statement is processed for every row in table SFLIGHT where the fields CONNID and CARRID equal those of table SPFLI, and also for each row of table SBOOK where the field FLDATE equals that of table SFLIGHT. Each `SELECT` statement has its own overhead in terms of time and system resources, which makes this query extremely inefficient. Figure 7 shows the large number of database accesses R/3 makes on the tables SPFLI, SFLIGHT, and SBOOK, along with the numerous database operations involved (in the "Operations" column) and their corresponding durations (in the "Durations" column).

Figure 9 *The Generated SQL Trace Output of the JOIN Statement*



```
Transaction = SE38          PID =      384 P type  DIA Client = 100 User = ABAPDEV

Duration ObjectName Op.     Rec  RC    Statement

      30 SPFLI      REOPEN    0     0 SELECT WHERE (T_01 . "MANDT" = '100' AND T_00 . "CARRID" = T_01 . "CARRID" AND T_00 . "CONNID" = T_0
     521 SPFLI      FETCH     9  1403
```

✓ *Tip*

*If you wish to know more about a specific statement, select it from the SQL Trace results list and press the "Display" button (), which takes you to the corresponding ABAP statement in the program. To find out whether an index (primary or secondary) is used by the database optimizer to access a particular table, choose the "Explain" button. The execution plan of this Open SQL statement then appears, listing the indexes that the database system would use to execute the Open SQL statement (shown in **Figure 8**).*

As you'll recall, there is a much faster way to get the results we're looking for — by using `JOIN`s instead of nested `SELECT`s (refer back to Rule #3, "Optimize the Number of Data Transfers"). Instead of running the previous code in step 3, run the following in its place:

```
SELECT a~carrid a~connid
       c~fldate b~bookid
       INTO (car, con, fld, id)
  FROM ( ( spfli AS a
    INNER JOIN sflight AS c
         ON  a~carrid = c~carrid
         AND a~connid = c~connid )
    INNER JOIN sbook   AS b
         ON  b~carrid = c~carrid
         AND b~connid = c~connid
         AND b~fldate = c~fldate ).
ENDSELECT.
```

The trace for this statement is displayed in **Figure 9**. As you can see, instead of the previous 19 fetches, it consists of just 9 fetches with a duration of only 521 milliseconds! Here, no unnecessary data is transferred to the program and the `JOIN` results in a single `SELECT` loop, so no unnecessary `SELECT`s are processed. Compare this with Figure 7, where the nested `SELECT` statement — which subdivides into numerous database operations, each with its own processing time — resulted in a gigantic results list (and Figure 7 shows only a part of this list!). Using the `JOIN` instead of nested `SELECT`s in this second code example is much faster.

The above example demonstrates that SQL Trace is a powerful tool for investigating and improving the speed of your application. With its help, you can easily check if certain measures add to an increase in speed and to what extent.

Note that, in general, some of the improvements you make may come at the expense of additional runtime costs on the application server side, similar to what can happen when you relocate the data set to make sorting more efficient (refer back to the discussion in the section "Use the ABAP SORT Clause Instead of ORDER BY"). Since SQL Trace addresses only the costs of accessing the database, it may be wise to use additional tools like ABAP Runtime Analysis (transaction SE30).[5]

Conclusion

This article explored the various means available to you for optimizing database access. There may be situations where the rules do not apply as they are presented in these pages. You should not take the techniques described here as hard-and-fast rules, but rather as guidelines. There may be scenarios where you have to give priority to one factor over another, such as minimizing network load rather than database resources. It is up to you and your organization to decide which guidelines best fit the particular problem you are facing. My hope is that this article will help you to make this decision wisely.

[5] While a discussion of Runtime Analysis is beyond the scope of this article, you can find more information in Axel Kurka's article "How Plausibility Checks Detect Bottlenecks in ABAP Programming" in the January-March 2001 issue of *SAP Insider* (available in the Article Archives at **www.SAPinsider.com**).

Rehan Zaidi studied Computer Science at the FAST Institute of Computer Science (now known as the National University of Computer and Emerging Sciences) in Karachi, Pakistan. Rehan is working as an SAP Technical Consultant at Siemens Pakistan. He has been involved in ABAP development at both in-house and remote implementations of SAP R/3. Most of his work includes designing and writing country- and customer-specific applications for the SAP HR module. Having used many elements of the ABAP Workbench, Rehan is also familiar with cross-application components such as BAPIs, ALE, and Workflow. He can be reached at rehan.zaidi@siemens.com.pk or saprehan@yahoo.com.