Performance Problems in ABAP Programs: How to Fix Them

Werner Schwarz

Werner Schwarz joined SAP Retail Solutions in October 1998. Today, he is the development team contact for all performance-related issues concerning SAP's Retail Industry Business Unit. Werner's major tasks include information rollout and support of his colleagues regarding performance during both development and maintenance.

(complete bio appears on page 48)

Editor's Note:

SAP Professional Journal has published ABAP performance articles on a number of occasions. It is with particular pride that we have presented this special two-part series.

The first installment, "Performance Problems in ABAP Programs: How to Find Them" (May/June 2003), demonstrated how to use SAP's tools for performance analysis to identify the areas that dominate the runtime of a poorly performing program. This second installment profiles the most frequently encountered causes of runtime maladies and offers suggestions for addressing those problems, with a focus on both database accesses and ABAP statements.

Together, these two articles deliver a comprehensive approach to performance optimization.

Whatever the cause of your program's performance troubles, optimizing the runtime of the application requires that you first find out where the majority of the runtime is spent.

The first installment of this two-part article series[1] showed you how to use SAP's performance monitoring and tracing tools — Performance Trace (transaction *ST05*) and ABAP Runtime Analysis (transaction *SE30*) — to identify the critical actions or code sequences that are causing poor runtime in your program.

But then what? Once you've identified the code area in question, how do you go about improving your program's performance? Here in

[1] "Performance Problems in ABAP Programs: How to Find Them" (May/June 2003).

this second installment, I present the most commonly found performance bugs, and the recommended solutions. In my experience, the most typical sources of performance problems are inefficient database accesses and faulty ABAP coding (regarding internal table handling in particular). These typical problem sources will be the focus of this article.

Inefficient Database Accesses

In my experience, there are certain problems with database accesses that appear over and over again:

1. Lack of index support
2. More data read than necessary
3. Same data read multiple times
4. Improper use of buffering
5. Use of single accesses instead of an array access

These issues cause problems of varying severity. A missing index can really "kill" your program, and reading more data than you need to can have a noticeable impact on performance, so you'll need to address these issues right away. Overlooking buffering possibilities or performing single instead of array accesses, on the other hand, is less critical — while a correction will help, it will hardly reduce the total runtime by factors. Be sure to keep these prioritization considerations in mind when you've identified the problem in your own environment and are planning your next steps.

In the following sections I will discuss these five typical problems, in order of severity, and outline their possible solutions.

Database Access Problem #1: Lack of Index Support

As discussed in the first installment of this two-part series, to handle access requests the database uses an "execution plan," which determines whether the database performs a full table scan (where *all* records are read) or uses an index (where only *specific* records are read). An unfavorable execution plan — for example, using a full table scan for a large table or using an index that doesn't match the specified restrictions — can cause an extended runtime and result in severe performance problems for your program. So, how do you locate the problem and what can you do about it?

As described in the previous article, once you have identified the database accesses that are causing the highest load using Performance Trace (*ST05*), check the execution plans for these accesses (place the cursor on the line containing the query statement in question and select Explain, *Trace → Explain SQL*, or *F9*).

As part of the execution plan analysis, compare the fields in the *WHERE* clause of the query statement with the fields of the used index (if any). If the used index does not match (or no index is used at all), check all indexes on the table (to access the required information, click Explain or *DDIC info*). This comparison will help you determine if there is an index on the table that matches the selection. The following problems can occur here:

- Long runtime despite the use of a suitable index
- Wrong index is used
- No suitable index exists

Let's take a closer look at these three sources and how to handle them.

Long Runtime Despite the Use of a Suitable Index

If your analysis shows that the runtime of the query statement is too long even though a suitable index is being used, the reason is likely to be one of the following:

- There are missing index fields.
- "Wildcards" are used.
- The index has a bad storage quality.

The more index fields specified with an "equal" operation in the statement's *WHERE* clause, the more efficient the index. Note that the order of the index fields is also important: if an index field is not specified in the *WHERE* clause, index fields subsequent to that field cannot be used properly, even if they are specified in the *WHERE* clause.

Recall the example of a standard phone book from the previous article. If you know the city name, street address, and first name of someone you want to call, but you do not know the person's last name, the only information that will significantly reduce the number of entries you have to check is the city name. The other information is useless if you don't know the person's last name. Along these lines, let's say that your *WHERE* clause specifies three fields (city name, street address, first name) with an "equal" condition, and that all three fields are included in an index that is designed like a typical phone book. While at first glance this index appears to be a suitable match, it cannot be used properly if its second index field (the person's last name) is not part of the query statement's *WHERE* clause.

So what do you do if you find there is an index field missing from the query's *WHERE* clause? Sometimes you can find the information for a missing field in the business context, or you can easily retrieve it by adding a quick selection to a database table, for example. In such cases you should add this information to the *WHERE* clause, even if it doesn't reduce the size of the result set. The improved access time for the selection might outweigh the additional effort spent on retrieving the information. If you cannot find the information, your only alternative is to create a new index, which will improve access performance, but will also increase runtime for modifications and memory requirements, so be sure to keep this in mind.

Take a look at **Figure 1**, which demonstrates how

Figure 1 Selecting Rows from a Database Table: A Runtime Comparison

Example 1: Use a single SELECT statement with a WHERE clause without specifying a value for field BUKRS.*

```
select * from z_tabl into table itab where belnr = '0000000001'.
```

Runtime: 74 ms

Example 2: Select a (small) table to determine all possible values for field BUKRS, then call the SELECT statement using FOR ALL ENTRIES, passing the value of BUKRS to the statement's WHERE clause.*

```
* fill company code table (3 entries !)
select bukrs into table bukrs_tab from z_bukr.
* selection
  select * from z_tabl into table itab_res
      for all entries in bukrs_tab
      where bukrs = bukrs_tab-bukrs and belnr = '0000000001'.
```

Runtime: 1 ms

Analysis of runtime results

The second statement is considerably faster, despite the additional selection on table Z_BUKR and the use of FOR ALL ENTRIES to execute an array selection on Z_TABL. This is because the index fields BUKRS and BELNR can now be used for the selection on Z_TABL, which allows the database to make far better use of the index.

* Both selection examples use the same key index — including fields MANDT, BUKRS, BELNR, and GJAHR — with an index range scan execution plan.

retrieving missing information can significantly improve access time. This example first considers a selection on table *Z_TABL* (which has about 60,000 entries, including key fields *MANDT*, *BUKRS*, *BELNR*, and *GJAHR*) that specifies a value for the field *BELNR* (*0000000001*) and returns one record from the database. An ideal index for such a database selection would start with fields *MANDT* and *BELNR*. Suppose there is no such index, however. In this case, the table's key index must be used,[2] which contains the additional field *BUKRS* between fields *MANDT* and *BELNR*. The problem here is that we do not want to restrict the selection to field *BUKRS*, so we do not specify its value, but without a value specified for *BUKRS*, the database cannot make effective use of the index and thus has to read and check numerous entries that are not part of the result set, namely records with another value for *BELNR*. The result, as you can see, is that the selection takes 74 ms during the test run.

So how can this runtime be improved? Let's say that we know there are only a few different values for *BUKRS*. In this case, we can identify all of them and perform a number of selections on *Z_TABL*, each time with a different identified value for *BUKRS*. At first glance this might seem excessive — in contrast to the first option, where we performed only one selection, here we have to perform one selection to read all possible values for *BUKRS*, and then perform several additional selections on *Z_TABL*. However, because we added field *BUKRS* and its values, the database only reads and checks records that belong to the result set. Consequently, the complete runtime for determining all values for *BUKRS* (three in the example) and executing the array selection on *Z_TABL* is considerably faster (1 ms instead of 74).

Another situation that can degrade access performance despite the presence of a suitable index is the use of "wildcards," which means that instead of a value, a pattern is specified for a table field in the *WHERE* clause. For example, take a look at the following *SELECT* statement:

```
SELECT … WHERE MATNR LIKE 'A%'
```

Here, all records where field *MATNR* starts with the character *A* are selected.

Using a wildcard makes it very difficult for the database to use the field specified with the wildcard and the subsequent fields of the index. To remedy this problem, look for superfluous use of *LIKE* in the *WHERE* clause. When you come across a clause with a restriction such as *LIKE '%'*, simply omit the restriction completely. This will not adversely affect your program because it does not restrict the result set in the first place.

To be on the safe side, it's best to also modify *WHERE* clauses containing *LIKE* that do not use wildcards — for example, by replacing *WHERE MATNR LIKE 'ABCD'* with *WHERE MATNR = 'ABCD'* instead. Here, the "equal" operation serves the same purpose in the *WHERE* clause as the *LIKE*, only without the risks associated with using the *LIKE* operator.

> ✓ ***Tip***
>
> *Use wildcards only if it is really necessary and appropriate. However, keep in mind that if a field is specified with LIKE, the database cannot make good use of subsequent index fields.*

The storage quality of an index has a considerable influence on program performance as well. Efficient use of an index depends on how compactly the index information is stored in the database blocks and how quickly the information can be accessed. Frequent updates, inserts, or deletes in a database table can fragment the information so that it is spread out over many data blocks, so that even if a suitable index is used, many index blocks must be read to retrieve the index information, thus causing an extended runtime.

[2] Note that there is always an index on a table's key fields.

If you suspect this situation, ask your database administrator to check the storage quality of the index and to reorganize it if necessary. A compact index stored in fewer index blocks can be accessed substantially faster.

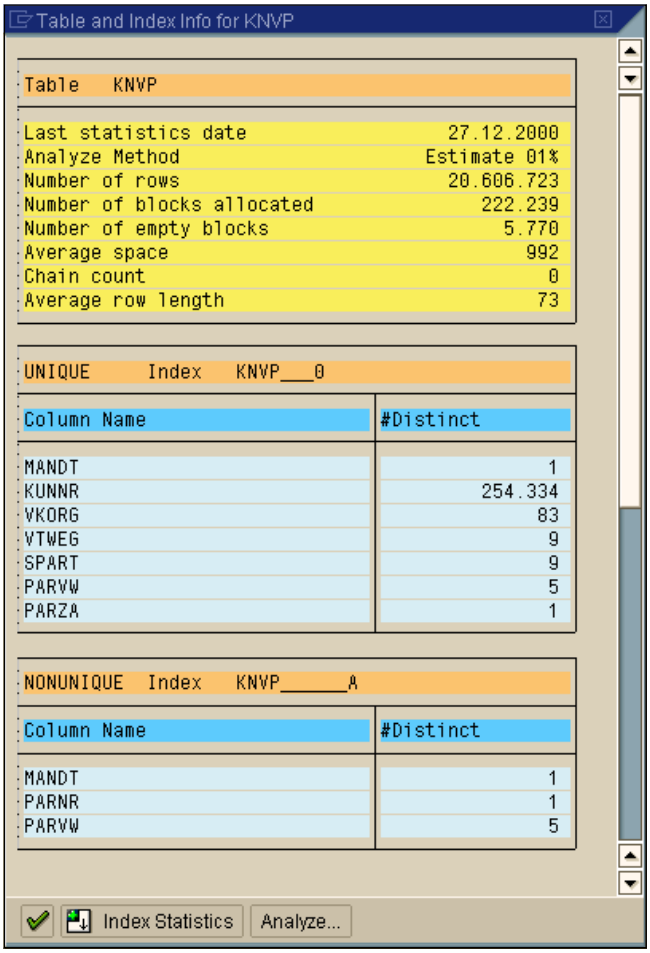
Wrong Index Is Used

If a suitable index exists, but is not used, it may be that the suitable index has a low "selectivity." An index's selectivity is determined by the number of records it can exclude from all possible records in a table. The more different values exist for the fields of an index, the more you can reduce the number of records to be read when you specify these fields in the query statement's *WHERE* clause.

The database decides on an execution plan based on the selectivity of all available indexes, which is documented in the database statistics. For some databases, like Oracle, you can access this statistics information directly from the execution plan display in the SAP transaction. Simply click on the table name in the execution plan display (from transaction *ST04* or *ST05*), which launches a dialog that displays information on that table and all of its indexes. **Figure 2** shows this information for table KNVP. There, in the first line of the top frame, you see the creation date of the statistics (*27.12.2000*). In the second frame, the column *#Distinct* tells you for each field (*MANDT*, *KUNNR*, *VKORG*, etc.) how many different values were found in the records of the table. The higher the number of values found for a field (like *KUNNR* in Figure 2), the more selective the field, and the more selective the field, the better it can be used to reduce the amount of data that must be evaluated. Consequently, an index containing such a selective field becomes "more attractive" if that field is specified in the query statement's *WHERE* clause.

If the selectivity of an index's fields is very low, the number of records that will have to be read using that index might be very high, in which case the database might consider using a full table scan or a different index instead. It's a good idea to first make sure that the database table statistics are up-to-date. If the statistics are old (perhaps from a time when the table contained only a few records), this information might differ considerably from reality, and current statistics might trigger a different — and better — execution plan.

Figure 2 Table and Index Information



> ✓ ***Tip***
>
> *Keeping statistics up-to-date by a regular job is required in order for the database to determine suitable execution plans.*

If the table does, in fact, have the latest statistics, and the index's fields still show a low selectivity, you can also try to improve the quality of the statistics. In general, not all records in a table are considered when statistics are being created, as this would take too long for large tables. Instead, a "spot check" is performed — when you specify the statistics to be recorded, you determine a percentage of records to be read.[3]

If a high percentage of a table's records have the same value for a certain field, and only a few records contain different values for that field, it is likely that the spot check will find only the value that appears in the majority of the records. In this case it might be better to change the settings so that the statistics analysis is run on the entire table, even if it might take a little longer, so that you can be sure to get an accurate picture of the data distribution.

If none of the approaches described so far — updating the statistics and improving their quality — cause the database to use the desired index, you can try to force the database optimizer to use it. Note, however, that this cannot be accomplished by changing the order of the restrictions in the query statement's *WHERE* clause. Instead, you have to change the database optimizer's settings or add "database hints" to the SQL statement. However, be aware that these options are extremely database-dependent and they are only useful for solving a specific problem in a specific situation. For this reason, a database hint should not be included in a standard program, which is designed to be database-independent. You should also be aware that not all databases support the use of database hints.

I do not provide a detailed description of these options, as they are beyond the scope of this article. For more details on database hints, see CSN note 129385 and look through the detailed information for the database in question.

[3] This information is displayed under *Analyze Method* in the top frame of Figure 2 (*Estimate 01%*).

No Suitable Index Exists

If there is no suitable index for the statement in question, you have two remaining options:

- Modify the query statement so that an existing index can be used.
- Create a new index that is suitable.

Sometimes it is possible to add further restrictions to the query statement's *WHERE* clause that enable the database to use an index that already exists. If the *WHERE* clause contains index fields that are located at the end of an index, but not fields from the beginning of an index, that index cannot be used optimally. If you manage to change the selection by adding more restrictions with respect to the missing index fields, the existing index may be used (refer back to the example in Figure 1). Of course, this only makes sense if the selection still fits to your business requirements and if the additional overhead is minimal.

As a last resort you can create a new index for the database table that suits your query statement. Remember, however, that while an additional index will improve the performance of read accesses, it will also increase the required database memory and, since the index must be maintained, it will increase the runtime of any database operations that modify the table (such as *INSERT*, *UPDATE*, and *DELETE*).

> ✓ ***Tip***
>
> *To avoid excessive performance drain, you should create an additional index for only those accesses that are performed frequently, such as program queries or transactions that are performed every day (and not those that come from a report that is executed only once a month, for example).*

Listing 1: Checks That Eliminate Extraneous Records After the Data Selection

```
SELECT * FROM MARA.
       …
       CHECK MARA-MATART = 'HAWA'.
       CHECK MARA-MATKL  = 'ZKL'.
       CHECK MARA-BSTME  = 'KART'.
       …
ENDSELECT.
```

Database Access Problem #2: More Data Read Than Necessary

When looking at an SQL trace, you can sometimes identify heavy loads caused by the database reading a large number of records from a table. In such cases, you should determine whether this large amount of data is really required. The reading of unnecessary data can happen for several reasons, such as carelessness, faulty design, or a problem with the *FOR ALL ENTRIES* statement. Sometimes you might find additional "checks" in an ABAP program that eliminate irrelevant records from the result set immediately after performing the selection, like the ones shown in **Listing 1**.

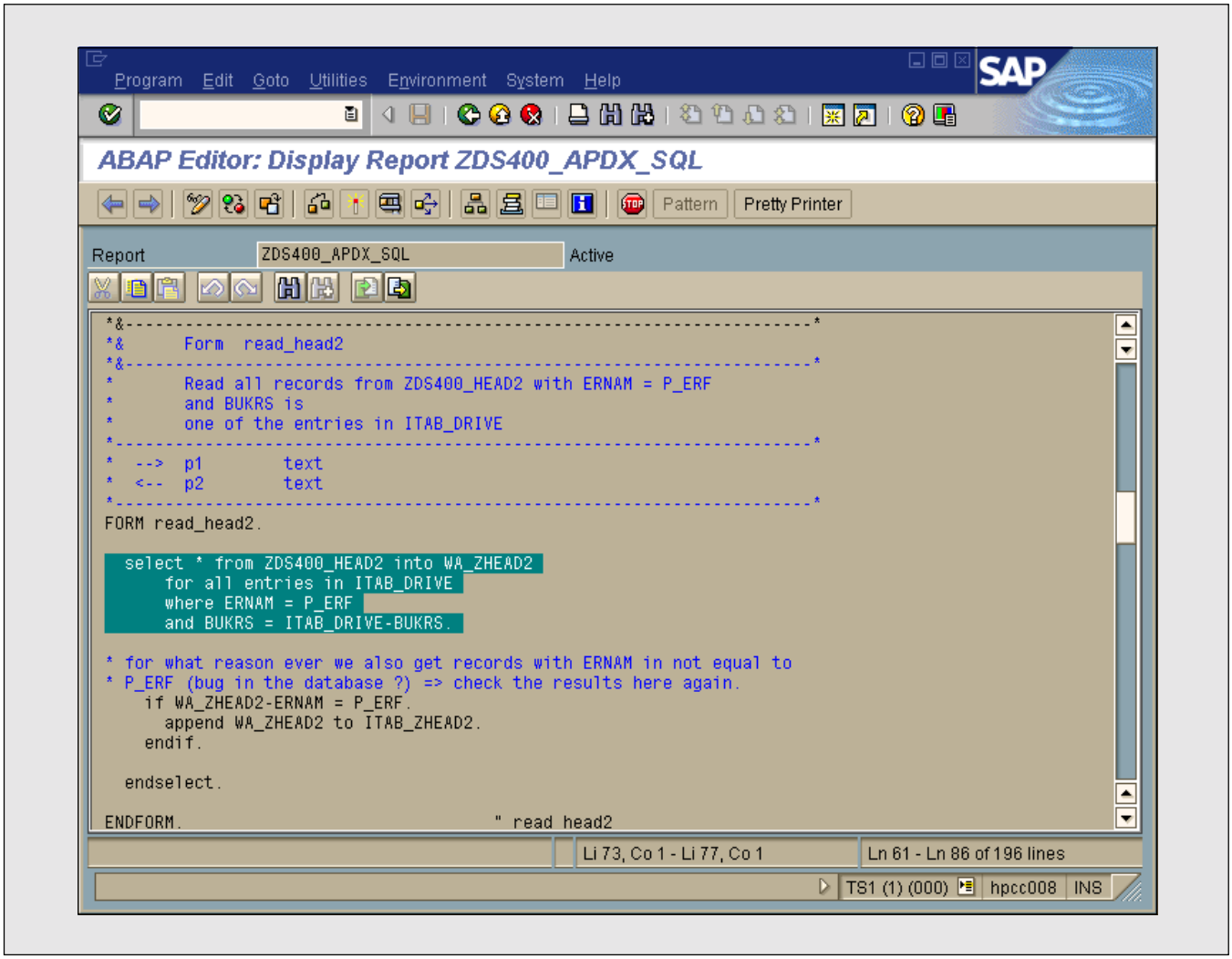
It is best to locate and remove such checks and instead include all known restrictions in the *WHERE* clause, because this avoids unnecessary reading of data on the database, the transfer of this unnecessary data to the application server, and the unnecessary execution of coding in the ABAP program to delete this data. Unfortunately, the checks are not always as obvious as the ones shown in Listing 1, so you might not find them without a further analysis of the program and business logic.

In some cases, however, belated checks are useful because the additional restrictions are difficult for the database to evaluate with the selection. Sometimes, surprisingly enough, it is even faster to read more records than required and eliminate the needless data later, if this makes the database selection simpler and faster. For example, it may pay to skip a restriction for a field completely instead of adding hundreds of possible values to a range table whose restriction would eliminate only very few records. Generally, though, you should include all known restrictions in the *WHERE* clause.

Another frequent problem derives from the *FOR ALL ENTRIES* feature of the *SELECT* statement. *FOR ALL ENTRIES* enables you to perform a selection on a database table in one ABAP statement that contains varying values for one or several fields in the *WHERE* clause. The different values are stored in an internal table that is specified in the query statement.

But what happens if this internal table is empty? You might assume that the selection will simply skip all restrictions in the *WHERE* clause that refer to that internal table and keep all others that are independent of this table. But this is not the case! Instead, the database selection skips all restrictions in the *WHERE* clause, keeping only the client specification.

Figure 3 The Query Statement Source Code



Have a look at the highlighted code in **Figure 3**. Here, the internal table *ITAB_DRIVE* is empty. Consequently, the statement handed over to the database (see **Figure 4**) contains only the restriction referring to the client (*"MANDT" = :A0*) — the restriction for *ERNAM* (*ERNAM = P_ERF*) was completely skipped. As a result, far more data is read than expected, which might cause functional problems in addition to performance problems, as the result set also contains entries that do not fulfill the restriction concerning the value for *ERNAM*. To avoid this behavior, for any internal table that is to be used in a *FOR ALL ENTRIES* query, be sure to check if it contains any entries by inserting a check like:

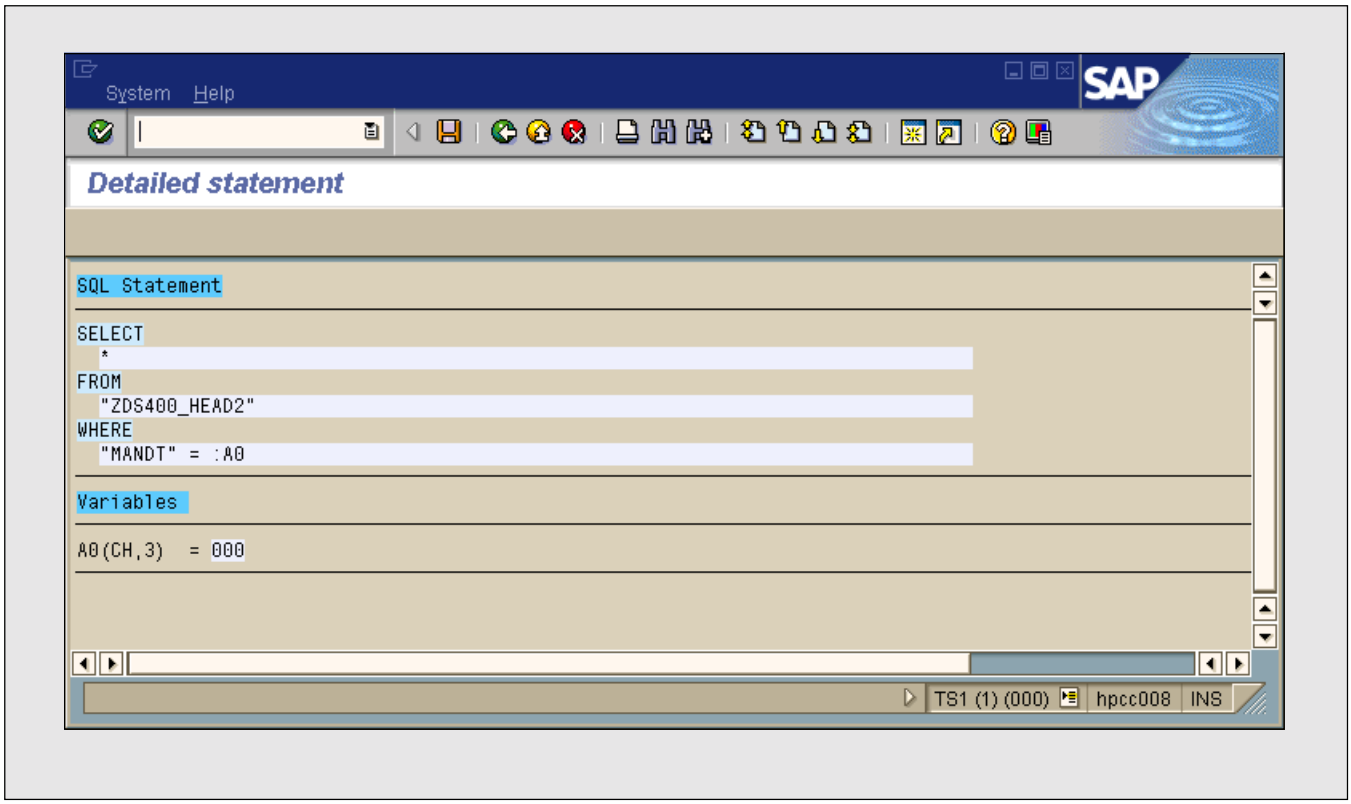
```
IF NOT ITAB[] IS INITIAL
```

before the query is executed. Alternatively, you can have the context guarantee that the internal table has at least one entry.

Database Access Problem #3: Same Data Read Multiple Times

Usually, it makes little sense to perform the same database selection repeatedly within a program. An

Figure 4 *The Statement Received by the Database*



exception may be reading data from the database after locking it with an SAP *ENQUEUE* lock. But in general, these identical selections put an unnecessary additional load on the database and increase the program's runtime. If you identify identical selections in the SQL trace, as detailed in the previous article, it's best to determine where the selections come from and try to correct them.

To avoid identical selections, save the results of the database selection in an internal table and make the information accessible to all function modules, forms, and methods that need it. In other words, create function modules ("read" modules) specifically for performing all relevant database selections on a table, and then buffer the results in internal tables that can be accessed by all these modules. Of course, you also have to maintain all queries in your buffered information that have an empty result set — i.e., all queries that did not yield a record. Then, instead of having all modules in your application perform the selection directly, replace the *SELECT* statements by a function call to the suitable read module. If the information on the requested selection is already in the buffer, the result can be passed to the caller without performing a database access. Otherwise, the data is read from the database and the buffer is updated.

> ✓ ***Tip***
>
> *Buffering read modules are already implemented for numerous database tables. Before creating your own module (or performing database selections in your code) check to see if an appropriate module is already available. A function group usually has several modules with different interfaces using the same buffer (e.g., ME_EKKO_ARRAY_READ and ME_EKKO_SINGLE_READ).*

Database Access Problem #4: Improper Use of Buffering

Instead of explicitly buffering data in the user context of your program, as described in the previous section, you can use table buffering on the R/3 instance provided by the system. In the data dictionary (DDIC), you can define buffer settings for a table in the technical settings. The effect is that all records (or a generic area) from the table are buffered in the shared memory of the instance. Accordingly, this data is available for all word processes of the instance, so it needn't be read from the database again. This is an advantage compared to the read modules discussed before, as the buffered data in a read module is only visible in the same user context of the transaction or program. However, buffering via technical settings is not useful for tables that are too large or frequently changed.

> ✓ ***Note!***
>
> *In general, it makes no sense to create the buffering read modules described previously for tables that are already buffered via their technical settings. The usage and maintenance of the buffer is completely hidden from the application. The code in the ABAP program is still a normal SQL statement, but internally it is satisfied from the buffer — if possible — and not from the database.*

Some specific selections always bypass the buffer, which results in "real" database accesses, even for buffered tables. These statements are:

- *SELECT ... FOR UPDATE*
- *SELECT* using *DISTINCT* or one of the aggregate functions (*COUNT*, *MIN*, *MAX*, *SUM*, etc.)
- *SELECT* with a *GROUP BY/HAVING* clause
- *SELECT* with an *ORDER BY* clause (other than *PRIMARY KEY*)
- *SELECT* with a *WHERE* clause that contains *IS [NOT] NULL* or any subquery
- *JOIN* commands in Open SQL (*SELECT … JOIN ON*)
- Any Native SQL statement
- *SELECT … BYPASSING BUFFER*

If you find accesses to buffered tables in the SQL trace, check why the buffer is bypassed. It might be because the *SELECT* statement in the program is one of the statements listed above. But you should also know that if you use "single record" buffering, the *SELECT* statement must specify the table key completely, and you must use the *SELECT SINGLE* option. Similarly, to use "generic" buffering, the complete generic key must be given. In some cases, the buffer is bypassed accidentally because the developers were not aware of these conditions.

If it is useful, and not incompatible with business requirements, you should consider modifying such statements so that the buffer is used or adapting the settings for the table buffer (but I strongly suggest you don't do this for SAP standard tables!).

> ✓ ***Tip***
>
> *Think about buffering via technical settings when defining your own tables:*
>
> - *In general, small tables that are rarely modified and accessed mostly by primary key are ideal candidates for buffering.*
> - *If a table is buffered but you need to bypass the buffer and read directly from the database, you can use the BYPASSING BUFFER option. It can add to a better understanding of your program code if you indicate that the bypass happens on purpose, even if the statement bypasses the buffer anyway.*

Figure 5 Single Selections vs. an Array Selection: A Runtime Comparison

Example 1: An array selection*

```
select * from ZDS400_HEAD into table i_result for all entries in I_HEAD
where EBELN = I_HEAD-EBELN.
```

Example 2: A sequence of single selections*

```
loop at I_HEAD.
select * from ZDS400_HEAD appending table i_result
        where EBELN = I_HEAD-EBELN.
endloop.
```

Analysis of runtime results

Number of entries in I_HEAD	10	1,000	10,000
Array selection	1 ms	75 ms	744 ms
Single selections	4 ms	334 ms	3,299 ms
Ratio of single to array	4	4.45	4.43
Conclusion	The array access is considerably faster (about four times faster) than the sequence of single selections.		

* Both examples read entries from a database table corresponding to the entries in internal table I_HEAD.

Database Access Problem #5: Use of Single Accesses Instead of an Array Access

In general, the database can process array accesses (i.e., reading several records with one request) faster than a sequence of single requests.

Look at the example in **Figure 5**, which shows two ways of reading a number of records from the database. In the first example, there is only one statement to the database that specifies all restrictions concerning field *EBELN* by using an internal table *I_HEAD* and *FOR ALL ENTRIES*.[4] In the second example, a sequence of single database selections is performed in a loop, each specifying only one value for *EBELN*. I compared the runtime of both for reading different numbers of entries. As the results show, using the array access is about four times faster than using a sequence of single reads on the database.

When analyzing the SQL trace, you sometimes find sequences of single selections on a database table. Quite often, these selections come from a loop over an internal table. What can you do to improve this?

Consider implementing a single array access that reads all relevant records from the database table into an internal table in one step before entering the loop. During the processing of the entries in the loop, the required record is read from this internal table instead of the database. You can combine this with buffering read modules for the database table (as described earlier). Here, the array access before the loop fills the buffer, and in the loop a call to the module rereads a single record from the buffer.

[4] Actually this single Open SQL statement is split into several database accesses by the Open SQL layer, but for clarity and simplicity I disregard this detail here.

The Keys to Avoiding Database Access Problems

As the database is a single resource in the system, optimizing database accesses and reducing the load on the database are essential when optimizing the performance of both a specific ABAP program and the whole system. The following key optimization techniques can generally be performed in the database or through local changes in the program, without making any changes to a program's overall design or functionality:

- ☑ Perform selections with a suitable index.
- ☑ Avoid reading data unnecessarily.
- ☑ Use the proper system buffering mechanisms.
- ☑ Use array accesses instead of performing a sequence of single selections.

For example:

```
LOOP AT ITAB.
  SELECT SINGLE FROM DBTAB
      WHERE FIELD = ITAB-FIELD.
Processing…
ENDLOOP.
```

would become:

```
CALL FUNCTION 'DBTAB_ARRAY_READ'
  TABLES ITAB …

LOOP AT ITAB.
CALL FUNCTION 'DBTAB_SINGLE_READ'
    EXPORTING ITAB-FIELD
    …
Processing…
ENDLOOP.
```

As you can see in the runtime comparison in Figure 5, there is hardly a benefit when executing, for example, an array access with 10,000 entries compared to a sequence of 10 array accesses with 1,000 entries each. Consequently, it is okay to split the processing of a large workload into several smaller packages, as this will hardly increase the total runtime for the database selections.

✓ ***Note!***

Be aware that reading and processing the entire data volume in a single step is not always the best solution. If a large amount of records must be read (e.g., tens of thousands or even more), the amount of memory required to store all the data and the time that is needed to maintain it might outweigh any positive effects. In such cases, think about processing the data in smaller packages using array selections with a suitable number of records.

Faulty ABAP Coding

If it turns out that the major part of your program's runtime is spent in the ABAP program itself, rather

than with database access requests, ABAP Runtime Analysis (transaction *SE30*) is the proper tool for investigation. My previous article covered the use of this tool in detail, especially how to use it to identify the module that is the most promising candidate for your tuning efforts. Here, I discuss the typical reasons behind an extended runtime in an ABAP module and what you can do to resolve such issues.

> ✓ ***Note!***
>
> *The examples used in the following discussion assume that there is only one module (or only very few modules) with a far higher net time than all others.*

Zeroing in on the Problem

Let's say that you have identified a module that consumes a large part of the total runtime of your program. There are two possible reasons for this excessive runtime:

- The long runtime is simply caused by a large number of calls.
- There are only a few calls to the module, but each call is consuming an excessive amount of runtime.

In the first case, verify that there is a justifiable reason for the large number of calls. There may be some calls that are unnecessary and can be eliminated. This decision depends on the business process, however, so the only advice I can offer is to avoid needless operations and make sure that you are knowledgeable of the particular business process involved. You can also try to improve performance by applying some minor changes (e.g., use work areas instead of header lines or use field symbols) that might reduce the runtime per call only slightly, but the total effect can still be considerable as it multiplies with the number of calls.

In the second case, perform a detailed analysis of the module with ABAP Runtime Analysis (see the sidebar "Performing a Detailed Analysis" on the next page).

You will often find operations on internal tables (mostly standard tables) at the top of the runtime list. In my experience, faulty internal table handling is the most common cause of severe runtime problems in ABAP, and is the problem source I will focus on here.

> ✓ ***Note!***
>
> *Runtime problems with internal tables usually occur only when processing large tables, which is why it is hard to find these problems with small test cases. Thus problems tend to appear only after the program is used in a production environment. See the appendix to this article on page 49 for some tips on how to detect these bugs using smaller test cases.*

> ✓ ***Tip***
>
> *Avoid using internal tables that are too large. Whenever possible, it's generally better to process smaller, independent packages, which can save both time and memory.*

The Internal Table Handling Problem

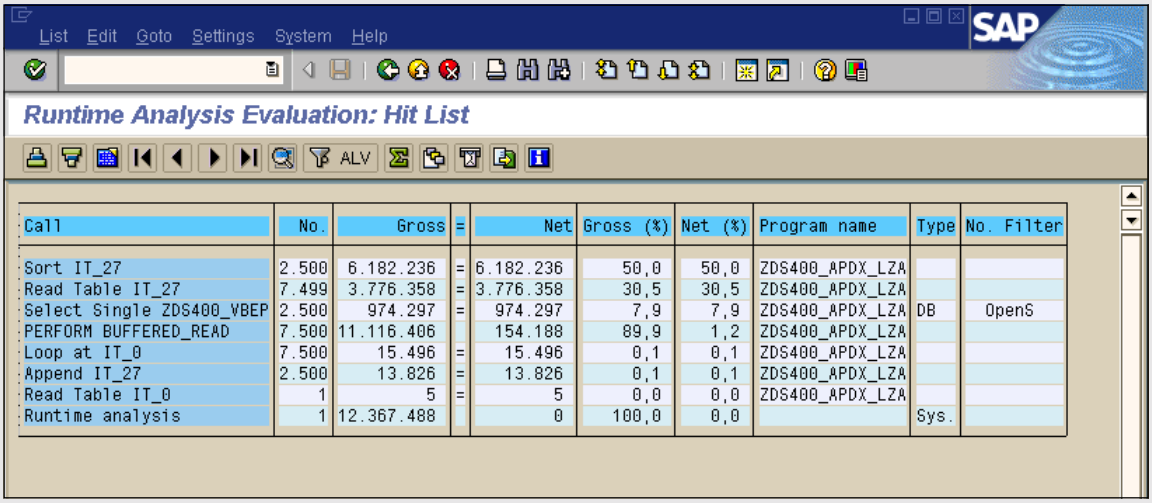
As I just mentioned, in most cases, performance problems in ABAP code are a result of sub-optimal handling of internal standard tables. Since standard

Performing a Detailed Analysis

To perform a detailed analysis, define a new restriction variant in ABAP Runtime Analysis (transaction SE30) that traces only the information of the suspect module, including all accesses on internal tables (refer to my previous article in the May/June 2003 issue for details). When you configure this new variant, specify the module in question on the "Program (parts)" tab, and on the "Statements" tab be sure to activate the checkboxes for read and change operations on internal tables.

> ✓ ***Tip***
>
> *In general, aggregated information is sufficient for this analysis.*

Rerun the program using the new variant and look at the hit list, sorted by net time (see the screenshot below). In the example below, you can see that the total runtime of the program was about 12 seconds (gross time). The read and sort operations on the internal table IT_27 required a total of about 10 seconds. Consequently, any tuning measures for this program must focus on optimizing these operations as they consume about 80% of the total runtime.



Call	No.	Gross	=	Net	Gross (%)	Net (%)	Program name	Type	No. Filter
Sort IT_27	2.500	6.182.236	=	6.182.236	50,0	50,0	ZDS400_APDX_LZA		
Read Table IT_27	7.499	3.776.358	=	3.776.358	30,5	30,5	ZDS400_APDX_LZA		
Select Single ZDS400_VBEP	2.500	974.297	=	974.297	7,9	7,9	ZDS400_APDX_LZA	DB	OpenS
PERFORM BUFFERED_READ	7.500	11.116.406		154.188	89,9	1,2	ZDS400_APDX_LZA		
Loop at IT_0	7.500	15.496	=	15.496	0,1	0,1	ZDS400_APDX_LZA		
Append IT_27	2.500	13.826	=	13.826	0,1	0,1	ZDS400_APDX_LZA		
Read Table IT_0	1	5	=	5	0,0	0,0	ZDS400_APDX_LZA		
Runtime analysis	1	12.367.488		0	100,0	0,0		Sys.	

You might find the hit list display a little confusing — instead of displaying the same name for the internal table used in the coding, it lists a technical name, like IT_27. To find out which table is being referenced, navigate to the program code using the button in the application toolbar, where you can view the corresponding statement containing the internal table name.

tables are not the only type of internal table, in addition to using a table better, performance problems can often be solved simply by using another table type. There are three types of internal tables:

- Standard
- Sorted
- Hashed

While standard tables can be used for any kind of access, sorted and hashed tables are designed for specific uses, and the appropriate access methods are optimized for performance. It's a good idea to consider using a sorted or hashed table if you expect the table will contain more than only a few entries. These tables have certain restrictions, however, and if the design of your application conflicts with these restrictions, you might be forced to use a standard table. Not to worry though — with a little effort, you can use standard tables in such a way that there are almost no performance disadvantages.

In an ideal world, all developers would be familiar with the performance aspects of different internal table types and would take them into consideration when designing programs. However, in the real world, experience shows that this is not always the case. Because improper use of internal tables can have a significant impact on runtime, my goal over the next few sections is to familiarize you with these important elements so that you can use them properly. I provide a short overview of the advantages and limitations of sorted and hashed tables, and the corresponding runtime implications. I will also show you how to optimize access times when you must use standard tables.

Hashed Tables

A hashed table must be defined with a unique key. Using this key is the only way you can access a single record in the table. The table is maintained in such a way that a hash function computes the index position from the key, and the index points directly to the data record.[5] This is why you always have to specify the complete key of the record you want to read. Thus if all read operations in your program are accesses to single records using a fully specified key, a hashed table is an appropriate table type.

Due to the hash algorithm, the response time for a key access to a hashed table record is constant, regardless of the number of table entries, making a hashed table ideal for processing large amounts of data — provided that you access single records only, and all with a fully specified key.

You should not use a hashed table if the access key is not always fully specified, as it is not possible to access ranges in a hashed table. Such an access would end up as a full scan of all table entries. Index operations like *LOOP … FROM* or *INSERT … INDEX* also cannot be executed. Furthermore, adding a new entry to a hashed table takes a little longer than appending a new entry to a standard table, and the maintenance of the hash key requires a certain amount of memory.

Sorted Tables

The best way to guarantee a sorted sequence of the entries, regardless of whether the key is unique or non-unique, is to use a sorted table. Sorted tables are designed and optimized for fast access to ranges if they can make use of the order of the entries. A sorted table is best for range accesses involving table key or index accesses. It is particularly useful for partial sequential processing in a loop if you specify (at least the leading part of) the table key in the *WHERE* condition. In contrast to a hashed table, you need not specify the complete key for read accesses to a sorted table. If you perform a read access with an incomplete key, the runtime system selects the first entry that fulfills the condition.

The runtime system uses a binary search for key access to a sorted table if the table key can be used. Therefore, the average runtime is logarithmically proportional to the number of table entries for these accesses. For large tables, this is considerably faster than a key access to a standard table. However, the system cannot use a binary search for accesses with a key that doesn't correspond to the table key. The average runtime for these accesses is exactly the same as for a standard table; it is proportional to the number of table entries.

[5] I disregard collision handling here, since this is transparent to the user and does not add to the understanding of this discussion.

A sorted table is always sorted according to the key specified in the table definition. It is not possible to change the sorted sequence according to another key at runtime. If your program design requires many key accesses that do not correspond to the table key, you have two alternatives:

- Use a standard table, which can be rearranged according to other keys at runtime via the *SORT* statement. In order to achieve a logarithmically proportional access, the table must be sorted according to the key by which you want to access the table, and you must use the *BINARY SEARCH* statement for the read accesses. However, this is only feasible if the rearrangement is rarely performed, as sorting the table takes a considerable amount of time.

- Retain the sorted table, but also maintain other internal tables as "secondary indices" to optimize frequent read accesses for all keys other than the table key. Keep in mind that this will have an additional overhead in terms of space consumption. (I will describe this in detail a little later.)

A sorted table is usually filled using the *INSERT* statement. Entries are inserted according to the sort order defined by the table key. If you specify a table index with the *INSERT* statement, the runtime system checks the accuracy of the position to identify any entries that would breach the order when added to the table. If you do not specify an index, the runtime system will determine the suitable table index for the new entry. Consequently, inserting a new entry into a sorted table is a little slower than inserting a new entry into a standard table. Furthermore, keep in mind that there will always be an index maintained for the sorted table, which requires additional memory, so when you use numerous, very small internal tables, the possible runtime benefits might not balance the memory consumption costs.

Standard Tables

Many developers use standard tables exclusively. Standard tables are the most flexible table type and can be used for all purposes (reading all entries, performing index, range, and key accesses). However, standard tables are not tuned for specific access types. They are very fast when accessing a record via the table index, but there are only rare cases where this type of access is used.

When performing a *LOOP AT … WHERE* or a *READ ... WITH KEY* on a standard table, the runtime system must scan all table entries from the beginning, which makes the average runtime for such an access proportional to the number of table entries. In contrast, the runtime for a read access that specifies an index is independent of the table size.

As I mentioned earlier, it is possible to optimize accesses to a standard table (I will show you how in a moment), but it requires some effort when writing your program and is not as easy as using a hashed or sorted table. Therefore, using a hashed or sorted table should always be considered, too.

Choosing the Right Table Type: The "Nested Loops" Factor

You might think that using a sorted or hashed table instead of a standard table will not make a big difference in terms of runtime, but you should consider using them anyway, especially if you cannot be absolutely sure that the table will always be very small. There is almost no additional work for you when writing your program, and it can help you avoid tremendous runtime problems if your assumption about the size of the table is incorrect. Far too often I have heard developers say, "I never expected my program to be used in such an environment and with such data volumes!" Furthermore, maintaining your program can be easier with a sorted table, as the sorted sequence is always guaranteed. Assume you use a standard table and you are forced to adapt your code in order to use the *BINARY SEARCH* for a read access. You will then have to check all areas where the table is modified, so that you can be sure that the table is sorted appropriately. And this assumption can fail if you add (or someone else adds) new code to the program later on.

The average runtime for a single record access in an internal table depends on the table type:

- For a *hashed* table, it is independent of the number of entries and thus constant.
- For a *sorted* table, where the table key is used, it is logarithmically proportional to the number of entries. If the table key isn't used, it is proportional to the number of entries.
- For a *standard* table, it is usually proportional to the number of entries. There is one exception: if the table is sorted and the *BINARY SEARCH* addition is used, it is logarithmically proportional to the number of entries.

For small tables, this does not make a big difference in runtime. You will rarely observe performance problems that are caused by accesses to internal tables with only very few entries. Even for larger tables, the absolute runtime difference for a single access doesn't seem to be critical — and as the access time is proportional to table size, you might think that there is no real problem in sight.

This is only true if you focus on a single table access, however. If you take a more general look at your program, you will notice that table accesses are frequently performed within *loops*. It is important to take this into account because the number of accesses to the inner table multiplies by the number of repetitions in the outer loops — e.g., the number of loops over other tables — which leads to non-linear runtime for your program, and thus severe performance problems with a growing workload (i.e., table size).

Let's take a look at a simple example. Assume there are two internal tables, *I_HEAD* and *I_ITEM*, containing data on header and line item information for customer orders. For each order (let's say there are 200) there is exactly one entry in *I_HEAD*, identified by the order number *EBELN*, and there are several entries (depending on the number of line items in the order — let's say 20 for each on average) in *I_ITEM*, identified by order number *EBELN* and item number *EBELP*. Usually you would process each of the orders separately with its items in a nested loop as follows:

```
LOOP AT I_HEAD.
  …
  LOOP AT I_ITEM WHERE EBELN =
          I_HEAD-EBELN.
      …
  ENDLOOP.
  …
ENDLOOP.
```

Let's focus on *I_ITEM* accesses where only the entries in the table corresponding to the current order from *I_HEAD* need to be processed.

Assume *I_ITEM* is a standard table. In this case, the runtime system has to scan the entire table to find all records that fulfill the restriction in the *LOOP AT … WHERE* statement. There is no way to know whether there will be more lines to read after reading a certain line. This is essentially comparable to (and only barely faster than) looping over the entire table and checking each entry with an *IF* or *CHECK* statement. (In the example here, with every outer loop, only a few entries, 20 on average, are selected from a large table of 4,000 entries, so many entries are checked unnecessarily.) The runtime is therefore linear depending on the size of the table — in other words, if *N* describes the number of entries in the table, the time required to find the entries using the *LOOP AT … WHERE* statement is proportional to *N*.

To sum it up, the average runtime for the execution of the inner loop is proportional to the table size. At first glance, this doesn't seem bad for a single loop at *I_ITEM*, but keep in mind that this multiplies with the number of entries in *I_HEAD*. For each entry in *I_HEAD* you process a loop that is linear depending on the size of *I_ITEM*, which makes the total runtime for the nested loop proportional to *M*N*, where *M* is the number of entries in *I_HEAD*. If you double the number of orders, the size of *I_ITEM* and *I_HEAD* also doubles, which will extend the runtime for the entire nested loop by a factor of four. And increasing the number of orders by a factor of 10 worsens the runtime by 100. This quadratic runtime behavior has a considerably negative influence on the runtime of the whole program as tables increase in size.

Figure 6 Standard Table vs. Sorted Table: A Nested Loop Runtime Comparison

Example 1: A nested loop on a standard internal table

```
FORM standard.
Data: I_HEAD like standard table of EKKO with header line.
Data: I_ITEM like standard table of EKPO with header line.

  loop at I_HEAD.
     loop at I_ITEM where EBELN = I_HEAD-EBELN.
*               DO SOMETHING…
     endloop.
  endloop.

ENDFORM.                              " standard
```

Example 2: A nested loop on a sorted internal table

```
FORM sorted.
Data: I_HEAD like standard table of EKKO with header line.
Data: I_ITEM like sorted table of EKPO with unique key EBELN EBELP with header
      line.

  loop at I_HEAD.
     loop at I_ITEM where EBELN = I_HEAD-EBELN.
*               DO SOMETHING…
     endloop.
  endloop.

ENDFORM.                              " sorted
```

Analysis of runtime results

Number of entries in I_HEAD	50	100	200	400	800
Number of entries in I_ITEM	1,000	2,000	4,000	8,000	16,000
Runtime in standard table (ms)	40	157	632	2,531	11.239
Runtime in sorted table (ms)	3	6	11	23	46
Conclusion	The runtime increase is considerably smaller for sorted tables with a growing data volume than for standard tables.				

So what will happen if we change *I_ITEM* to a sorted table instead, sorted by *EBELN*? Well, the runtime system can then make use of sorting and find the first item for a specific customer order using a binary search, which is logarithmically proportional in time to the number of entries. Thus the runtime for the complete nested loop is proportional to *M*log(N)*.

If you increase the number of orders by a factor of 10, the runtime becomes *(10*M)*log((10*N))*, which is an increase by a factor of *10*log(10*N)/log(N)*. This converges to 10 for large values of *N*. This is a big improvement over the standard table approach with its quadratic behavior, and it will make a big difference for a large number of entries.

> ✓ ***Note!***
>
> *With a sorted table, the system only has to identify the first entry with the binary search — due to the ordering, the remaining entries can be accessed by simply reading the subsequent lines, which is considerably faster. Consequently, we take into account only the time to determine the first relevant entry from I_ITEM.*

The situation in this example is a typical range access on a sorted table, which is precisely what these tables are designed for. The runtime system can make use of the ordering to quickly identify the first relevant record, and when looping over *I_ITEM*, it can stop looking at records when the last matching record is reached. As soon as the value for *EBELN* changes, there cannot be any more matching records in *I_ITEM* after that position, so the loop can be left. As a result, table entries are not read unnecessarily.

Let's take a look at some example runtime comparisons to confirm what we have just deduced theoretically. Continuing with the example, **Figure 6** compares the runtimes of a nested loop on a standard table with a nested loop on a sorted table. As you can see, when using standard tables, the runtime increases by a factor of about four when doubling the workload. However, the increase in runtime is much more moderate (roughly doubling) when using a sorted table. And the absolute runtime is already considerably smaller even with smaller volumes.

The runtime problems become even more evident if there is a third table (*I_SCHED*) included in the nested loop, as you see in **Figure 7**. This enlarges the number of accessed records considerably. Note that in practice many nested loops consist of more than three levels, and they may not always be as obvious as they are in the example shown here. The time to process the nested loop is unacceptable even for quite small volumes, and it causes severe

Figure 7 Adding to the Nested Loop: A Runtime Comparison

Example 1: A nested loop on a standard internal table with table I_SCHED added

```
FORM standard.
Data: I_HEAD like standard table of EKKO with header line.
Data: I_ITEM like standard table of EKPO with header line.
Data: I_SCHED like standard table of EKET with header line.

  loop at I_HEAD.
     loop at I_ITEM where EBELN = I_HEAD-EBELN.
     loop at I_SCHED where EBELN = I_ITEM-EBELN AND EBELP = I_ITEM-EBELP.
*               DO SOMETHING…
     endloop.
     endloop.
  endloop.

ENDFORM.                              " standard
```

(continued on next page)

Figure 7 (continued)

Example 2: A nested loop on a sorted internal table with table I_SCHED added

```
FORM sorted.
Data: I_HEAD like standard table of EKKO with header line.
Data: I_ITEM like sorted table of EKPO with unique key EBELN EBELP with header
      line.
Data: I_SCHED like sorted table of EKET with unique key EBELN EBELP ETENR with
      header line.

  loop at I_HEAD.
     loop at I_ITEM where EBELN = I_HEAD-EBELN.
     loop at I_SCHED where EBELN = I_ITEM-EBELN AND EBELP = I_ITEM-EBELP.
*               DO SOMETHING…
     endloop.
     endloop.
  endloop.

ENDFORM.                              " sorted
```

Analysis of runtime results

Number of entries in I_HEAD	50	100	200	400	800
Number of entries in I_ITEM	1,000	2,000	4,000	8,000	16,000
Number of entries in I_SCHED	5,000	10,000	20,000	40,000	80,000
Runtime in standard table (ms)	4,224	16,648	73,005	316,304	1,280,566
Runtime in sorted table (ms)	31	67	148	303	613
Conclusion	The runtime increase is considerably smaller for sorted tables with a growing data volume than for standard tables.				

problems if the volumes get large.[6] As nested loops like these are used quite often, the effect will increase and the runtime problem will get more serious.

Optimizing Access Times for Standard Tables

After our examination so far, it's not surprising that read requests on internal standard tables, used in a nested loop, are often the reason for long-running programs. The next step to a solution should be clear by now: if possible, use a sorted or hashed table instead of the standard table. If you are lucky, there is nothing more to do but modify the table definition to make this change. Sometimes, however, it requires some more effort, due to operations that are not allowed for a hashed or a sorted table, in which case you have to change the program code accordingly.

But what do you do if one or more of the restrictions of a sorted or hashed table prevent their use? In this case, ensure that your standard table is sorted, according to the key that is used for read accesses, and use a *READ … WITH KEY … BINARY SEARCH* when accessing the table. The runtime is then

[6] Note that these times are just for the loops, as no real work is performed on the data at all.

comparable to a read request on a sorted table. For reading a range of entries from a sorted standard table, do not use a *LOOP AT … WHERE*, as the runtime system cannot use and benefit from the sort property. Instead, determine the index of the first entry that satisfies your selection criterion with *READ … BINARY SEARCH*. All following records are then read with a *LOOP … FROM INDEX*, and your program must verify for each entry that it still fulfills the condition and will terminate the loop as soon as one doesn't. That way you can avoid reading many entries unnecessarily.

Now two questions still remain:

- How can I ensure that the table is sorted appropriately?
- What do I do if I need different sorting sequences for different key accesses during the program?

You will find the answers to these questions in the following paragraphs.

Keeping Tables Sorted

The easiest way to sort a standard table, so that you can use the *BINARY SEARCH* addition, is through the explicit use of the *SORT* statement. Keep in mind, however, that the *SORT* statement takes its time and can be quite expensive, especially if the table is large. Even if the table is already sorted correctly, the statement will at least lead to a full scan of the entire table. Sorting a partly sorted table can even increase the required time considerably due to the sorting algorithm used.

Therefore, avoid any unnecessary sorting — in particular, do not sort the table before each *BINARY SEARCH* read request! Furthermore, if you fill your standard table, don't keep it sorted by calling *SORT* each time you append a new entry. Either fill the table completely and sort it then, or — if you need the table to be sorted before it is completely filled — instead of simply appending it, add each new entry by inserting it at the right index position to preserve the sorting sequence. The bottom line: perform a *SORT* on the internal table only rarely.

When performing a runtime analysis, you will sometimes find a *SORT* statement consuming considerable time simply because you must sort a large table. You might be able to diminish the problem by reducing the size of the table (e.g., by processing the entire workload in smaller packages). This can help a little, but you still have to sort the table.

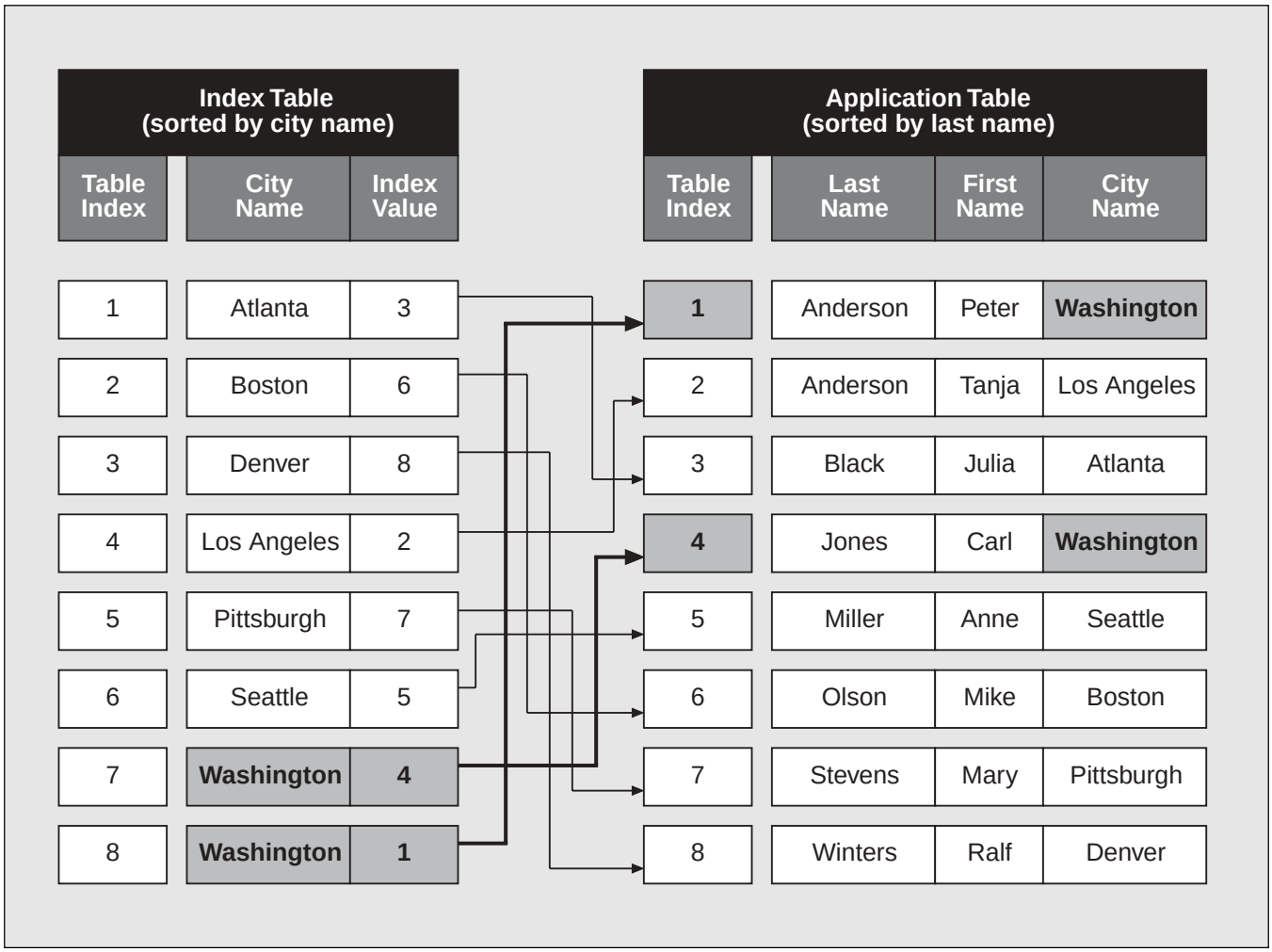
Under certain circumstances, however, you can guarantee the ordering of records by other means — i.e., without executing the *SORT* — causing less or no additional load.

Assume, for example, that the table is filled by a single database selection. Sometimes it is better for performance reasons to skip the *SORT* and instead use the *ORDER BY* clause in the database selection. Although conventional wisdom tells us to shift extra load from the database to the application server, in the following situations you can use the *ORDER BY* statement in the database selection without hesitation:

1. **If you are using *FOR ALL ENTRIES* and you want the result set to be sorted by the primary key:** As the Open SQL layer has to sort the result set by primary key anyway (in order to avoid duplicate entries), adding *ORDER BY PRIMARY KEY* to the selection does not cause any overhead at all.
2. **If the required sort order is the same as the order in the index used by the database:** In this case, the *ORDER BY* statement will not result in an additional overhead for the database, as the result set will be sorted in this order anyway.

> ✓ ***Note!***
>
> *Never rely on a certain ordering of the entries in an internal table if you don't ensure this by the use of SORT or ORDER BY, or by filling the table "manually" to preserve the required order.*

Figure 8 *An Index Table Example*



Using Tables As "Secondary Indices"

What can you do if a table is already sorted but the sorting order is different from the key for the read access? This can happen with sorted tables, where the table key is different, as well as with standard tables that are kept sorted. With the standard table, this is not a problem if all accesses according to one ordering are already processed before you need the table in a different order. In that case you simply re-sort the table using the *SORT* statement. But if the accesses alternate repeatedly, you would have to re-sort the table repeatedly, which is definitely not an optimal solution for this problem.

If you have alternating accesses with changing keys (that require different orderings of table entries), consider creating (and maintaining) other sorted or hashed tables as "secondary indices" that can be used to speed up access. These index tables simply contain the specific key that you want and the index position of the corresponding record in the original table. Since an index table has to support fast access for a (further) access key, it is either a sorted or hashed table with respect to this access key. With this index table, you can determine the table indices for all searched records in the original table. The records can then be accessed very quickly using the *READ … INDEX* statement.

The Keys to Avoiding Faulty Internal Table Handling

The most frequent reasons for an extended runtime in ABAP is the sub-optimal handling of internal tables, which can cause dramatic non-linear runtime behavior with growing data volume. The following key optimization techniques can often be applied through local changes in the program, without making any changes to a program's overall design or functionality:

- ☑ Use sorted or hashed tables, if appropriate.
- ☑ Use the READ operation with BINARY SEARCH on standard tables whenever possible (this requires the table to be sorted appropriately!).
- ☑ Avoid numerous SORTs when ensuring the suitable sorting sequence of a standard table.
- ☑ Use secondary index tables if you need to perform read accesses on an internal table with different keys.

Figure 8 shows a simple example. As you can see, there is an application table that records personal data, sorted by the last name field. Consequently, any access using the last name field is fast because we can use a binary search. If we wanted to access a record using the city name, however, a complete table scan would be necessary since the ordering doesn't support searching on that field. To circumvent that problem we maintain an index table, which has the exact same number of entries as the application table, but each entry records only the new key (the city name) and the index position of the corresponding record in the application table. If the key is not unique (as is the case here) several entries will be found for the same key, each pointing to a record with the same data value in the application table. Of course, the index table is sorted according to its key. Now, to access records by city name (for example, all records where the city name is *Washington*), instead of scanning the complete application table, we can determine the index position of these records quickly from the index table. Since the index table is sorted by city name, we can use a binary search to quickly read the index records, which will tell us the position of the searched records in the application table (*4* and *1* in the example) so they can be accessed directly. Note that although you have to perform two read accesses for every record searched (one in the index table and one in the application table), the total runtime is still improved because you don't have to search the entire table.

There is, of course, an additional overhead that results from creating this index table (both in terms of storage space and time), but the faster access usually outweighs the additional costs if the application table is not extremely small. However, keep in mind that once the index table is built, any modifications (like inserting or deleting records) cause additional load — essentially, you will have to update or re-create the index table each time you make a change. If changes will happen only rarely, the maintenance overhead is still acceptable. Ideally, the application is first created completely, and then you create the index table, so that no further maintenance is necessary.

Under certain circumstances, it is even possible to avoid the maintenance or re-creation of the index tables with some additional effort. For example, if the only modifications on the table are deletions, you might be able to simply mark the record as invalid without really deleting it from the application table.

Then you would only have to perform an additional check if a record is still valid. This is likely to be much faster than maintaining the index table.

> ✓ ***Note!***
>
> *You can find runtime comparisons for the use of secondary index tables and other valuable performance information in the "Tips & Tricks" of transaction SE30. There you will find some examples on how to use faster alternatives for otherwise identical functionality (e.g., copying an internal table or deleting duplicate entries).*

Summary

As part of my work with SAP R/3 installations, I have investigated numerous performance problems. In my experience, I have found a relatively small number of reasons to be responsible for a large number of these problems — in particular, problems with database accesses and internal table handling in ABAP program code.

With respect to database accesses, selections without a suitable index and reading unnecessary data are the most common causes of severe runtime problems. A suitable modification of the selection or creation of a new index are usually simple and efficient ways to ensure appropriate index use. Aside from avoiding any unnecessary functionality, of course, unnecessary database accesses can often be avoided by adding checks on internal tables with *FOR ALL ENTRIES*, by using SAP's buffering mechanisms, or by buffering data in read modules.

Problems due to a long runtime in ABAP are usually caused by the improper use of internal tables in nested loops, which results in a dramatic increase in runtime as data volume grows. These problems can usually be avoided by using the appropriate table type for internal tables — i.e., sorted or hashed tables — or by keeping standard tables suitably sorted and performing all frequent accesses with a binary search.

In the end, of course, the best defense is a good offense. The issues described here are the ones I've found to be the most common causes of runtime problems in production systems — keeping them in mind during the specification and design of your programs will help you to avoid the costs of subsequent corrections.

Werner Schwarz joined SAP Retail Solutions in October 1998 after working as a developer at two other IT companies. Today, he is the development team contact for all performance-related issues concerning IBU Retail (Industry Business Unit Retail). Werner's major tasks include information rollout and support of his colleagues regarding performance during both development and maintenance. He can be reached at werner.schwarz@sap.com.

Appendix: Conducting Performance Tests with Smaller Data Volumes

Quite often, a program's excessive runtime does not become apparent until it reaches the production system. The reason for this is simple: the database tables in the test or integration system are usually small. Furthermore, tests in these systems are usually performed with only small data volumes because people, understandably, tend to focus on functional tests in the early stages and neglect performance tests. Compounding this is that performance problems, like a missing database index or non-linear runtime behavior, aren't immediately visible if the table has only a few entries — a full table scan would still be quite fast. On the other hand, no one wants to struggle with performance problems in the production environment. So what can you do to avoid this?

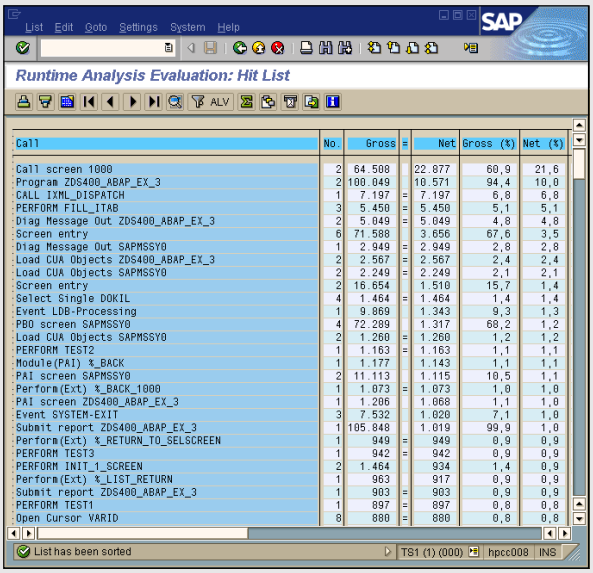
Ideally, before you use the program in the production system, perform adequate performance tests in a test system dedicated to performance testing, where data volumes for test cases and database table sizes are representative of the actual production environment. Although this is the most reliable way to conduct a performance test, it is not always feasible, as creating a realistic environment for such a test requires additional hardware and effort. If you can't set up an ideal test system with realistically sized data volumes, you can still conduct adequate performance investigations with smaller data volumes using the ABAP Runtime Analysis (transaction *SE30*) and Performance Trace (transaction *ST05*) tools.

You can use Performance Trace to find identical selections, sequences of single accesses instead of array accesses, or accesses that read unnecessary data. Even with small data volumes you can check whether there is an appropriate index for each selection. Performance Trace can aggregate all "equal" accesses on a table (i.e., not consider the actual values in the *WHERE* clause) that use the same execution plan, so that you can easily get a good overview of the different database accesses, check their execution plans, and determine the availability of a suitable index.

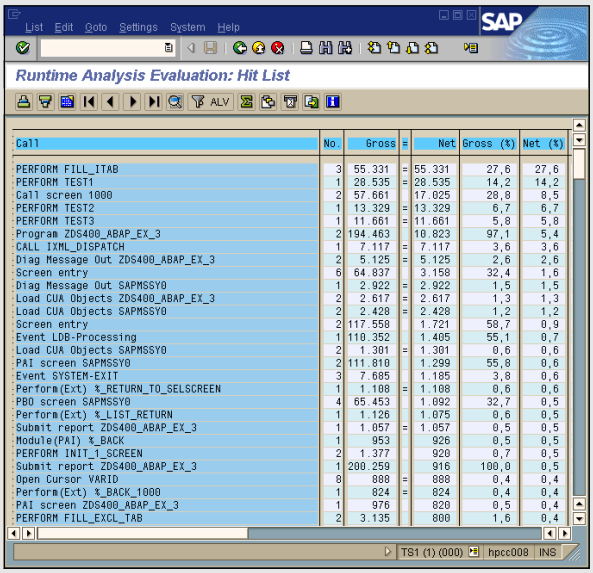
A non-linearity in the program coding usually has a noticeable effect only with large data volumes. Comparing the total runtime of two executions of the same program with only small data volumes in both cases (e.g., 10 or 50 items) is not likely to help you detect these critical areas. But ABAP Runtime Analysis provides you with detailed information on the runtime for each module, so you can compare the net runtime of both executions for each module independently. Obviously, it's almost useless to analyze all modules — just compare the runtime of, say, the 10 or 20 modules with the highest net time from the program execution with a data volume of 50 items.

Take a look at the figure below. The screenshots show the hit lists (both sorted by net time) resulting from two test runs of the same program with different data volumes — the first run contained 100 items and the second contained 1,000. The data analysis shown compares the runtimes of the top modules from the second run with the runtimes of those in the first run.

First run



Second run



Analysis of runtime results

Module	FILL_ITAB	TEST1	TEST2	TEST3
Runtime: Second run	55.331 ms	28.535 ms	13.329 ms	11.661 ms
Runtime: First run	5.450 ms	0.897 ms	1.163 ms	0.942 ms
Ratio of run 2 to run 1	10.2	31.8	11.5	12.4
Interpretation	Linear	Highly non-linear	Slightly non-linear	Slightly non-linear
Conclusion	Module TEST1 shows a highly non-linear runtime behavior while the others are either linear, or display an acceptably minor non-linearity, suggesting that module TEST1 is a candidate for further analysis.			

