Reading Sample

This chapter describes methods for analyzing individual ABAP programs, for example, using tools such as performance trace and ABAP runtime analysis, debugger, and code inspector. It also provides information on optimizing ABAP code.

- "Optimization of ABAP Programs"
- Contents
- Index
- The Authors

Thomas Schneider

SAP Performance Optimization Guide: Analyzing and Tuning SAP Systems

892 Pages, 2018, $89.95
ISBN 978-1-4932-1524-9

www.sap-press.com/4370

Chapter 5
Optimization of ABAP Programs

This chapter explains how to perform a detailed performance analysis of programs and transactions that you've already identified as expensive. In other words, you've performed a workload analysis, consulted users, and discovered that the performance of these programs isn't satisfactory.

To begin the analysis of expensive ABAP programs, examine the *single statistical records*, which will give you an overview of the response times of a transaction. For more in-depth analysis, use SAP performance trace for detailed analysis of database accesses, remote function calls (RFCs), and lock operations (enqueues). If you still can't find the problem after using these methods, you can use ABAP trace and ABAP debugger as additional analysis methods. You should also routinely use the Code Inspector, which has numerous statistical performance checks.

When Should You Read This Chapter?

You should read this chapter if you've identified a program or transaction as being critical for performance and you now want to perform a detailed analysis of it.

5.1 Performance Trace

Runtime analysis of ABAP programs

Performance trace is a powerful tool for analyzing the runtime of ABAP programs. It enables you to record a program runtime for the following operations: database access (i.e., SQL user statements), RFCs, enqueue operations, and accesses to SAP buffers. Performance trace is a tool that was developed by SAP and is thus identical for all database systems, except in the fine details.

To navigate to the initial performance trace screen, select **System • Utilities • Performance Trace** or use Transaction ST05. In this view, you'll find buttons to start, stop, and evaluate performance trace. You can also find checkboxes for selecting the trace modes **SQL-Trace**, **Enqueue Trace**, **RFC Trace**, **HTTP Trace** (as of SAP Basis version 7.10), and **Buffer Trace**. Only the field for the SQL trace is marked by default. For standard analysis of a program, we recommend that you activate the SQL trace, enqueue trace, HTTP trace, and RFC trace.

5.1.1 Activating a Performance Trace

You can start and stop a performance trace by clicking the **Trace On** and **Trace Off** buttons in Transaction ST05. You can only create one performance trace per application server at a time. In the **State of Trace** field, you can see whether a trace is already activated and which user has activated the trace. When you start a trace, a selection screen appears where you can enter users for whom the trace should be activated. The name with which you logged on is usually the user name entered here. Use a different name if you want to trace the actions of another user. The user who activates the trace doesn't have to be the same one whose actions are being traced.

Information on activating a trace

Keep in mind the following points when activating a trace:

- Ensure that the user whose actions are to be recorded only carries out one action during the trace; otherwise, the trace won't be clear. You should also ensure that no background jobs or update requests are running for this user.
- The performance trace is activated in the application server. For each database operation, data is written to a trace file in the file system on the application server. You must therefore ensure that you've logged on to the same application server as the user to be monitored. This is particularly important if you want to record an update request or a background job and are working in a system with distributed updating or distributed background processing. In this case, you won't know where the request will be started, and, as a result, you'll have to start the trace on all application servers with update or background work processes.

- The SQL trace only displays accesses to the database. SQL statements that can be satisfied from data in the SAP buffer don't appear in the trace. If you want to analyze SAP buffer accesses, activate the SAP buffer trace.
- However, buffer load processes are also recorded in the SQL trace. Because you're normally not interested in recording the buffer load process in the SQL trace, first execute a program once without activating the trace to allow the buffers to be loaded (i.e., the SAP buffers and database buffers). Then run the program again with the SQL trace activated, and use the results of this trace for evaluation.
- During the trace, look at the following monitors: the work process overview (for general monitoring), the operating system monitor of the database server (for monitoring possible CPU bottlenecks on the database server), and the database process monitor for direct monitoring of the executed SQL statements. It makes no sense to watch these monitors during the trace if you're logged on as the user being traced. The SQL statements of the monitors would appear in the trace and thus make the trace unreadable.
- The default trace file name is set with the SAP profile parameter `rstr/file`. In the initial screen, you can assign a different name to the trace file. Writing to the trace file is cyclical in the sense that when the file is full, the oldest entries are deleted to make room for new entries. The size of the trace file (in bytes) is specified by the SAP profile parameter `rstr/max_diskspace`, for which the default value is 16,384,000 bytes (16 MB).
- You can also store recorded traces. For this purpose, select **Performance Trace • Save Trace** or **Performance Trace • Display Saved Trace** to retrieve a saved trace again.

5.1.2 Evaluating an SQL Trace

To evaluate a performance trace, select **Display Trace** in the initial screen. A selection screen is displayed, and you can specify the part of the trace which you want to analyze in the **Trace Type** field. In this and the following sections, we'll discuss the evaluation of each of the trace types separately. In practice, you can analyze all trace modes together. Table 5.1 lists other fields you can use to restrict SQL trace analysis.

Field	Explanation
Trace Filename	Name of the trace file. Normally this name should not be changed.
Trace Type	The default trace mode setting is **SQL-Trace**. To analyze an RFC trace, enqueue trace, HTTP trace, and buffer trace, select the corresponding checkboxes.
Trace Period	Period in which the trace runs.
User Name	User whose actions have been traced.
Object Name	Names of specific tables to which the display of trace results is to be restricted. Note that, by default, the tables `D010*`, `D020*`, and `DDLOG` aren't shown in the trace results. These tables contain the ABAP coding and the buffer synchronization data.
Execution time	Restricts the display to SQL statements that have a certain execution time.
Operation	Restricts the trace data to particular database operations.

Table 5.1 Fields in the Dialog Box for Evaluating a Trace

Executing trace

Next, click the **Execute** button. The basic SQL trace list is displayed. Figure 5.1 shows an example of a basic trace list. Table 5.2 explains the fields displayed in an SQL trace.

Field	Explanation
hh:mm:ss.ms	Time stamp in the form *hour:minute:second:millisecond*.
Duration	Runtime of an SQL statement in microseconds (µs). If the runtime is more than 150,000 µs, the corresponding row is red to identify that the SQL statement has a long runtime. However, the value 150,000 µs is a somewhat random boundary.
Program Name	Name of the program from which the SQL statement originates.
Object Name	Name of the database table or database view.

Table 5.2 Fields in an SQL Trace

Field	Explanation
Operation	The operation executed on the database, for example, `PREPARE` (preparation "parsing") of a statement), `OPEN` (open a database cursor), `FETCH` (transfer of data from the database), and so on.
Curs	Database cursor number.
Records	Number of records read from the database.
RC	Database system-specific return code.
Statement	Short form of the executed SQL statement. The complete statement can be displayed by double-clicking the corresponding row.

Table 5.2 Fields in an SQL Trace (Cont.)

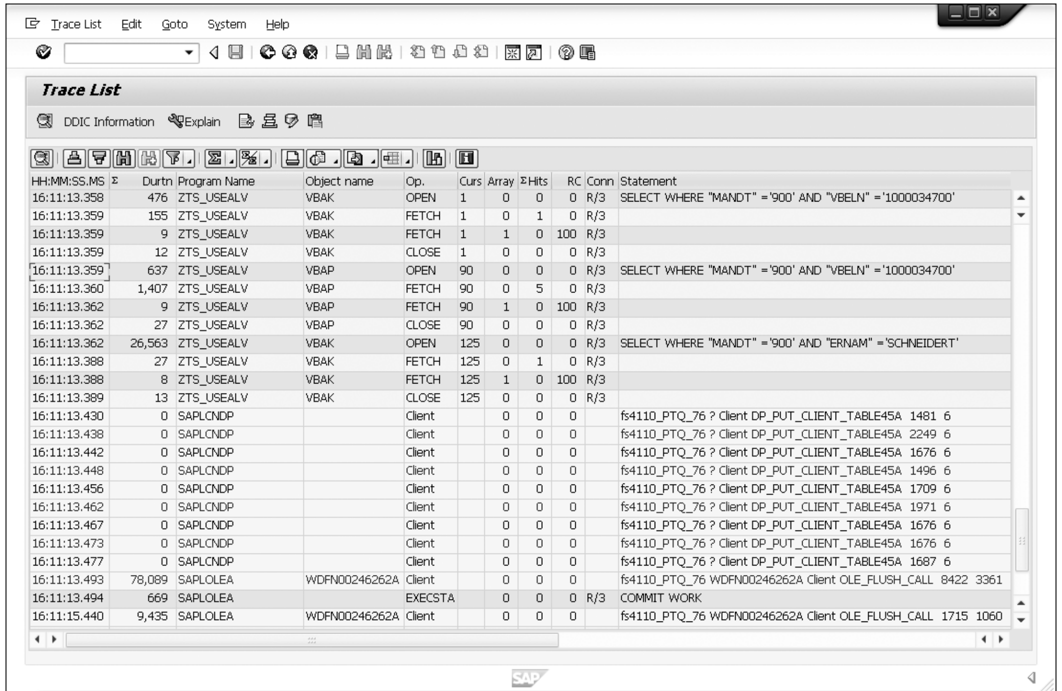


Figure 5.1 Basic Performance Trace List with Entries from SQL Trace and RFC Trace

Direct read

An SQL statement that appears in Figure 5.1 accesses table VBAK. The fields specified in the WHERE clause are key fields in the table. The result of the request can therefore only be either one record (Rec = 1) or no record (Rec = 0), depending on whether a table entry exists for the specified key. SQL statements in which all fields of the key of the respective table are specified as "same" are called *fully qualified accesses* or *direct reads*. A fully qualified database access should not take more than about 2 to 10 ms. However, in individual cases, an access may last up to 10 times longer, such as when blocks can't be found in the database buffer and must be retrieved from the hard drive.

The database access consists of two *database operations*, an OPEN/REOPEN operation and a FETCH operation. The REOPEN operation transfers the concrete values in the WHERE clause to the database. The FETCH operation locates the database data and transfers it to the application server.

Sequential read

A second access in Figure 5.1 takes place in table VBAP. Not all key fields in the WHERE clause are clearly specified with this access. As a result, multiple records can be transferred. However, in our example, five records are transferred (Rec = 5). The data records are transferred to the application server in packets, in one or more fetches (*array fetch*). An array fetch offers better performance for applications than transferring individual records in a client/server environment.

The second access takes place via an efficient index, thus the duration of execution also remains significantly less than 10 ms. The third access (again in table VBAK) takes place via a field for which there is no efficient index. Thus, the duration of this statement is significantly longer than that of the previous statement.

Maximum number of records

The maximum number of records that can be transferred in a FETCH operation is determined by the SAP database interface as follows: every SAP work process has an I/O buffer for transferring data to or from the database. The SAP profile parameter dbs/io_buf_size specifies the size of this buffer. The number of records transferred from the database by a fetch is calculated as follows:

Number of records = dbs/io_buf_size ÷ *length of 1 record in bytes*

The number of records per fetch depends on the SELECT clause of the SQL statement. If the number of fields to be transferred from the database is

restricted by a SELECT list, more records fit into a single fetch than when SELECT * is used. The default value for the SAP profile parameter dbs/io_buf_size is 33,792 bytes and shouldn't be changed unless recommended explicitly by SAP.

Guideline Value for Array Fetch

The guideline response time for optimal array fetches is less than 10 ms per selected record. The actual runtime greatly depends on the WHERE clause, the index used, and how effectively the data is stored.

Declare, prepare, open operations

Other database operations that may be listed in the SQL trace are DECLARE, PREPARE, and OPEN. The DECLARE operation defines what is known as a *cursor* to manage data transfer between ABAP programs and a database, and also assigns an ID number to the cursor. This cursor ID is used for communication between SAP work processes and the database system.

Prepare operation

In the subsequent PREPARE operation, the database process determines the access strategy for the statement. In the **Statement** field, the statement is to be seen with a variable (INSTANCE =:A0, not shown in Figure 5.1). To reduce the number of relatively time-consuming PREPARE operations, each work process of an application server retains a certain number of already parsed SQL statements in a special buffer (*SAP cursor cache*). Each SAP work process buffers the operations DECLARE, PREPARE, OPEN, and EXEC in its SAP cursor cache. After the work process has opened a cursor for a DECLARE operation, the same cursor can be used repeatedly until it's displaced from the SAP cursor cache after a specified time because the size of the cache is limited.

The database doesn't receive the concrete values of the WHERE clause (MANDT = 100, etc.) until the OPEN operation is used. A PREPARE operation is only necessary for the first execution of a statement, as long as that statement hasn't been displaced from the SAP cursor cache. Subsequently, the statement, which has already been prepared (parsed), can always be reaccessed with OPEN or REOPEN.

First and subsequent executions

Figure 5.1 shows the SQL trace for the second run of the same report. Because the DECLARE and PREPARE operations are executed in the report's first run, our example shows only the OPEN operation.

Network problems

If you've identified an SQL statement with a long runtime, you should activate the trace again for further analysis. It's useful to perform the trace at a time of high system load and again at a time of low system load. If you find that the response times for database accesses are high only at particular times, this indicates throughput problems in the network or in database access (e.g., an I/O bottleneck). For detailed information, see Chapter 2, Section 2.2.2. If, on the other hand, the response times for database access are poor in general (not only at particular times), the cause is probably an inefficient SQL statement, which should be optimized.

When evaluating database response times, remember that the processing times of SQL statements are measured on the application server. The runtime shown in the trace includes not only the time required by the database to furnish the requested data but also the time required to transfer data between the database and the application server. If there is a performance problem in network communication, the runtimes of SQL statements will increase.

[+]

Trace Comparison

You can best recognize network problems between the database and the application server by comparing traces as follows: First, execute the same SQL trace at least twice—once on the application server that is on the same computer as the database and is directly connected to the database, and once on an application server that is connected to the database via the TCP/IP network. Compare the two SQL traces. If there are significantly higher response times (greater by 50% or more) on the application server connected via the network, you have a network problem. Perform this test at a time of low system load, and repeat it several times to rule out runtime differences due to the buffer-load process on the database and application servers. This test works only when your application server is connected to the database via IPC.

5.1.3 Other Functions in the SQL Trace

Compression

Using the **Summary** function of the SQL trace, you can get an overview of the most expensive SQL accesses. If you navigate via **Trace List • Compress Trace according to SQL Statements**, a list is displayed, which shows the data listed in Table 5.3 for each statement. Sort the list according to the runtimes

of the SQL statements. The SQL statements with the longest runtimes should be optimized first.

Field	Explanation
Executions	Number of executions
Redundancy	Number of redundant (identical) executions
Duration	Runtime for all executions of an SQL statement in microseconds
Object Name	Name of the table
Statement	Short form of the executed SQL statement

Table 5.3 Fields in the Compressed Summary of an SQL Trace

Identical selects

Inefficient programs are often characterized by the fact that they read identical data from the database several times in succession. To help you identify these identical SQL statements, choose **Trace list • Display Identical Selects** to see a list of identical selects that tells you how often each identical select was executed. By using this function in conjunction with the compressed data, you can roughly see how much of an improvement in performance can be gained by avoiding identical SQL statements via more skillful programming.

Other functions

After this preliminary evaluation using the SQL trace, you have all of the information necessary for a more detailed analysis:

- **Program name and transaction of the executed program**
  Using the **ABAP Display** button in the trace list, you can jump directly to the code location that executes the SQL statement.
- **Table name**
  The **DDIC Info** button gives a summary of the most important dictionary information for this table.
- `WHERE` clause in the SQL statement.
- **Analysis**
  Detailed analysis of the SQL statement, for example, `EXPLAIN` function.

For more information on these functions, see Chapter 11.

Procedure for quality control

For customer-developed ABAP programs, perform (at least) the following checks as a form of program quality control:

1. For each customer-developed ABAP program, perform an SQL trace either on the production system or on a system with a representative volume of test data.
2. From the basic trace list display, create a compressed summary to find the SQL statements with the longest runtimes: **Trace List • Compress Trace according to SQL Statements**.
3. Display a list of identical accesses to find SQL statements that are executed several times in succession: **Trace List • Display Identical Selects**.
4. Use these lists to decide whether the program should be approved or whether it needs to be improved by the responsible ABAP developer.
5. Save a copy of the lists along with the program documentation. If program performance diminishes at a later date (whether due to a modification or due to the growing data volume), perform another SQL trace, and compare it to the earlier one. Monitor performance in this way after each significant program modification.

These measures can be used not only to monitor customer-developed programs but also to regularly monitor frequently used, standard SAP transactions that are critical to performance. If the runtimes of particular SQL statements increase over time, you may need to archive the corresponding table.

5.1.4 Evaluating a Buffer Trace

Table 5.4 contains and explains the fields that are displayed in a buffer trace. The last three columns are only displayed if you click the **More Info** button. You can find more detailed explanations on SAP table buffering in Chapter 12.

Field	Explanation
hh:mm:ss.ms	Time stamp in the form *hour:minute:second:millisecond*
Duration	Runtime of an SQL statement in microseconds

Table 5.4 Fields of a Buffer Trace

Field	Explanation
Program Name	Name of the program that has triggered the buffer call
Object Name	Name of the database table or database view
Operation	Name of the function that was executed on the buffer
Records	Number of records that were read from the buffer
RC	Return code: ■ 0: Function executed correctly ■ 64: No further records available ■ 256: Records not available in the buffer ■ 1024: Records not available in the buffer, buffer is loaded (for parallel read accesses)
Statement	In the `OPEN` operation, the buffering type is indicated: ■ G: Generic buffering ■ P: Single-record buffering ■ R: Full buffering ■ S: Export/import buffer ■ C: CUA buffer ■ O: OTR buffer ■ E: Export/import buffering in shared memory

Table 5.4 Fields of a Buffer Trace (Cont.)

5.1.5 Evaluating a Remote Function Call Trace

Table 5.5 lists the fields displayed in an RFC trace.

Field	Explanation
hh:mm:ss.ms	Time stamp in the form *hour:minute:second:millisecond*.
Duration	RFC runtime in microseconds.
Program Name	Name of the program from which the RFC originates.
Object Name	Name of the recipient, for example, the SAP instance or computer that was called.

Table 5.5 Fields in an RFC Trace

Field	Explanation
Operation	For entry **Client**: RFC sent, that is, the instance on which the trace is executed is the client (sender). For entry **Server**: RFC received, that is, the instance on which the trace is executed is the server (recipient).
Records	Not used.
RC	Return code (for successful execution: 0 [zero]).
Statement	Additional information on the RFC, including the names of the sender and recipient; name of the RFC module; and amount of data transferred. You can view all information on the RFC by double-clicking the corresponding row in the RFC trace.

Table 5.5 Fields in an RFC Trace (Cont.)

Figure 5.1 shows an example of an RFC (`OLE_FLUSH_CALL`).

Detailed analysis in the RFC trace

Like the SQL trace, the RFC trace provides several detailed analysis functions:

- By double-clicking a row of the RFC trace or by clicking the **Details** button, you can obtain complete information on the RFC, including the names and IP addresses of the sender and recipient, the name of the RFC module, and the transferred data quantity.
- By clicking the **ABAP Display** button, you can view the source text of the corresponding ABAP program.

5.1.6 Evaluating an HTTP Trace

Table 5.6 contains and explains the fields that are displayed in an HTTP trace. You'll find more detailed descriptions of HTTP calls in Chapter 8.

Field	Explanation
hh:mm:ss.ms	Time stamp in the form *hour:minute:second:millisecond*.
Duration	Runtime of the HTTP call in microseconds.
Program Name	Name (logical path) of the executed HTTP call.
Object Name	Name of the recipient, for example, the SAP instance called.
Object Name	For entry **Client**: HTTP sent, that is, the instance on which the trace is executed is the client (sender). For entry **Server**: HTTP received, that is, the instance on which the trace is executed is the server (recipient).
RC	HTTP return code (200 for successful execution, 401 for failed authentication, etc.).
Statement	Additional information on the HTTP, including the names of the sender and recipient, name of the HTTP call, and amount of data transferred. You can view all information on the HTTP call by double-clicking the corresponding row in the HTTP trace.

Table 5.6 Fields of an HTTP Trace

Table 5.6 Fields of an HTTP Trace (Cont.)

5.1.7 Evaluating an Enqueue Trace

Table 5.7 contains and explains the fields that are displayed in an enqueue trace. The last three columns are only displayed if you click the **More Info** button. You can find more detailed explanations of the SAP enqueue concept in Chapter 10.

Field	Explanation
hh:mm:ss.ms	Time stamp in the form *hour:minute:second:millisecond*.
Duration	Runtime of the enqueue/dequeue statement in microseconds.
Program Name	Name of the program from which the enqueue statement originates.
Object Name	Name of the enqueue object. To see details of the corresponding object, call the ABAP Data Dictionary (using Transaction SE12) and enter the object name in the field under **Lock Objects**. To view the properties of the objects, select **Display**.

Table 5.7 Fields in an Enqueue Trace

Field	Explanation
Operation	Operation ENQUEUE: Enqueue(s) set. Operation DEQUEUE: Individual enqueues are released. Operation DEQ ALL with specific object entry in the **Object** column: All enqueues for the object in question are released. Operation DEQ ALL with no object entry in the **Object** column (entry {{{{{{{{{): All enqueues for the transaction in question are released (end of a transaction). Operation ENQPERM: At the end of the dialog part of a transaction, enqueues are passed on to update management.
Records	Number of enqueues that are set or released.
RC	Return code (for successful execution: 0 [zero]).
Statement	More detailed information on the enqueue: Entries **Excl** or **Shared**: Exclusive or "shared" locks; name of locked unit, (e.g., "MARC 900SD000002": Material SD000002 is locked in table MARC, in client 900). The entries in this row correspond with the properties of the enqueue object defined in the ABAP Data Dictionary.

Table 5.7 Fields in an Enqueue Trace (Cont.)

5.2 Performance Analysis with ABAP Trace (Runtime Analysis)

You should use an ABAP trace, also referred to as ABAP runtime analysis, when the runtime of the programs to be analyzed consists mainly of CPU time. During an ABAP trace, not only the runtime of database accesses (SELECT, EXEC SQL, etc.) is measured but also the time required for individual modularization units (MODULE, PERFORM, CALL FUNCTION, SUBMIT, etc.), internal table operations (APPEND, COLLECT, SORT, READ TABLE), and other ABAP statements.

User interfaces are available in SAP GUI (Transaction SAT) and in the Eclipse-based ABAP development environment—the underlying tracing technology is identical.

5.2.1 Activating an ABAP Trace

For SAP GUI transactions

You activate an ABAP trace for an SAP GUI transaction as follows:

1. Use the following menu path to access the initial screen of the ABAP trace: **System • Utilities • Runtime Analysis • Execute**. Alternatively, select Transaction SAT.
2. In the upper part of the screen, under **Measure**, enter a transaction, program name, or function module, and then select **Execute** to start the measuring process. You can also click the **In Parallel Mode** button to access a process list, where you can activate the ABAP trace for the currently active work process.
3. The system starts to measure the runtime and creates a file with the resulting measurement data.
4. When you want to return to the initial runtime analysis screen, simply exit the transaction, function module, or program as normal, or start the runtime analysis again.

For web browser or external systems

If a runtime measurement is to be performed for a transaction that is started via a web browser or an external system, follow these steps:

1. Start the ABAP runtime analysis transaction as previously described.
2. Select the **Schedule** button under **User/Service**. The **Overview of Scheduled Measurements** screen is displayed.
3. Select **Schedule Measurement**, and enter the following data in the dialog box:
   - User, client, and server
   - Process type, for example, dialog, background processing, or HTTP
   - Object type (e.g., transaction, report, or URL) and object name
   - Maximum number of scheduled measurements
     - Note here that an end-to-end transaction may involve multiple measurements. To ensure that all parts are recorded, you should specify multiple measurements here.
   - Expiry date, time, and description
4. Close the window with the confirmation **Schedule Measurement**. The measurement is then scheduled and appears in the overview list.

5. Start the transaction to be analyzed. While the transaction is running, the columns **Started** and **Status** indicate whether measurement files are created.
6. When the transaction has ended, you evaluate the measurement files as described in the next section. If a measurement is still open because, for example, the web browser wasn't properly closed, you can stop a measurement by clicking **Stop Active Measurement** (on the **Evaluate** tab, as discussed later).

For BSP and Web Dynpro ABAP applications

To start a runtime analysis for Business Server Pages (BSPs) and Web Dynpro ABAPapplications, call Transaction SICF (Service Maintenance). Select the service to analyze in the navigation tree, and activate the runtime analysis via **Edit • Runtime Analysis • Activate**.

As of version 6.40, you can restrict the procedure to a user name during activation and specify a variant to use for recording. Here, it's also possible to specify the measurement accuracy.

Information on Measurement Accuracy

In the initial screen, a traffic light on the **Measure** tab indicates whether the runtime measurement can determine reliable times. If the traffic light is red, it means this isn't possible because of nonsynchronized CPUs in a computer with multiple processors. Therefore, you must perform the time measurements with low accuracy (**Settings • Measurement Accuracy • Low**). You can find further operating system-specific information in SAP Notes 20097 and 87447. If you want to perform a runtime analysis in parallel mode, refer to SAP Note 729520.

When you're activating an ABAP trace, filter functions enable you to restrict the trace to a particular function module or group of ABAP statements or to adjust an aggregation. Section 5.2.3 explains how to use these options.

Activating traces in Eclipse

To activate a trace in an Eclipse-based ABAP development environment, select the **ABAP Profiling** perspective in Eclipse first. This perspective contains the **ABAP Trace Requests** and **ABAP Traces** views. In the **ABAP Trace Requests** view, select a system and choose **Create Trace Requests** in the context menu. You can now define in a dialog box which actions are supposed

to be recorded. The options are the same as described for the SAP GUI interface.

You can activate the runtime analysis either in SAP GUI or in Eclipse. The analyses recorded can be evaluated with both interfaces.

Rules when activating traces

When you activate an ABAP trace, bear the following points in mind (which are similar to those that apply to an SQL trace):

- Because you're normally not interested in recording the buffer load process in the trace, you should first execute a program once without activating the trace, thereby allowing the buffers to be loaded (i.e., the SAP and database buffers). Then run the program again with the ABAP trace activated, and use the results of this trace for evaluation.
- Perform the trace at a time of low system load to ensure that the measured times aren't influenced by a temporary system overload (e.g., a CPU overload).
- To enable runtime analysis, the system requires the SAP profile parameters `abap/atrapath` and `abap/atrasizequota`. These parameters are set when the system is installed. The profile parameter `abap/atrapath` indicates in which directory the trace files are written. You can restrict the maximum size of all ABAP trace files by using the parameter `abap/atrasizequota`. The trace files are deleted after 30 days if you don't change the deletion date (**Evaluate** tab).

5.2.2 Evaluating an ABAP Trace

Evaluation views

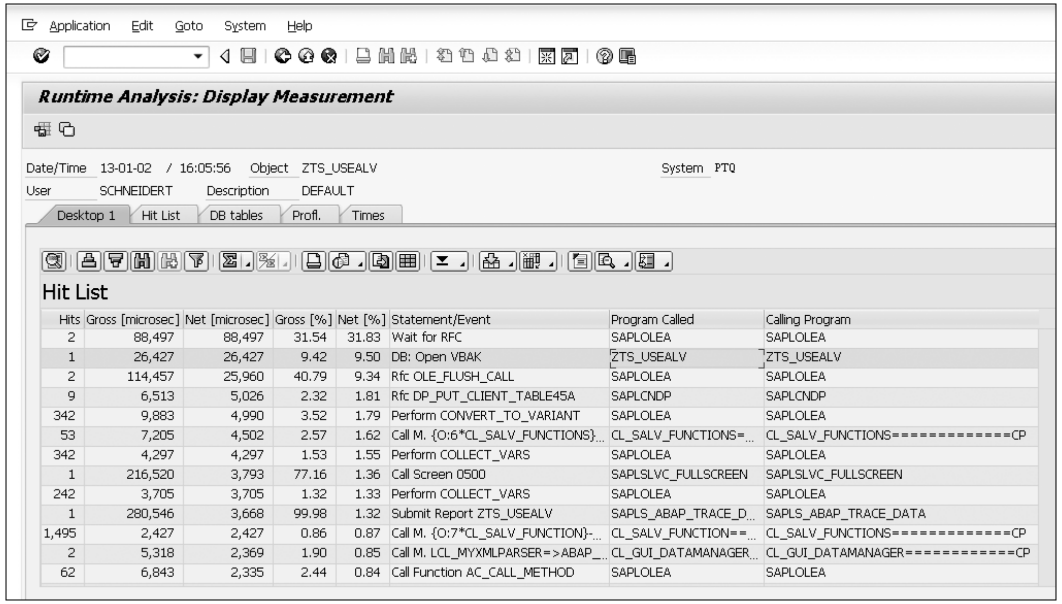
To display the results of an analysis, select the desired file under the **Evaluate** tab in Transaction SAT. In Eclipse, select the file in the **ABAP Traces** view.

The runtime analysis presents different views of the measurement results in list form or as graphics.

- Click the **Hit List** button to view a list that displays the execution time for each statement. This list is sorted in decreasing order of net times (see Figure 5.2).
- Via the **Hierarchy** button, you obtain a presentation of the chronological sequence of the recorded parts of the program.

- Other buttons will display specific evaluations that, for example, categorize database tables or modularization units.

If you've generated an ABAP trace, first display the **Hit List**. Sort the hit list according to net time to get an overview of statements with the highest net runtimes. If the ABAP trace was recorded in aggregate form, only the **Hit List** is available as an evaluation option.



Hits	Gross [microsec]	Net [microsec]	Gross [%]	Net [%]	Statement/Event	Program Called	Calling Program
2	88,497	88,497	31.54	31.83	Wait for RFC	SAPLOLEA	SAPLOLEA
1	26,427	26,427	9.42	9.50	DB: Open VBAK	ZTS_USEALV	ZTS_USEALV
2	114,457	25,960	40.79	9.34	Rfc OLE_FLUSH_CALL	SAPLOLEA	SAPLOLEA
9	6,513	5,026	2.32	1.81	Rfc DP_PUT_CLIENT_TABLE45A	SAPLCNDP	SAPLCNDP
342	9,883	4,990	3.52	1.79	Perform CONVERT_TO_VARIANT	SAPLOLEA	SAPLOLEA
53	7,205	4,502	2.57	1.62	Call M. {O:6*CL_SALV_FUNCTIONS}...	CL_SALV_FUNCTIONS=...	CL_SALV_FUNCTIONS============CP
342	4,297	4,297	1.53	1.55	Perform COLLECT_VARS	SAPLOLEA	SAPLOLEA
1	216,520	3,793	77.16	1.36	Call Screen 0500	SAPLSLVC_FULLSCREEN	SAPLSLVC_FULLSCREEN
242	3,705	3,705	1.32	1.33	Perform COLLECT_VARS	SAPLOLEA	SAPLOLEA
1	280,546	3,668	99.98	1.32	Submit Report ZTS_USEALV	SAPLS_ABAP_TRACE_D...	SAPLS_ABAP_TRACE_DATA
1,495	2,427	2,427	0.86	0.87	Call M. {O:7*CL_SALV_FUNCTION}-...	CL_SALV_FUNCTION==...	CL_SALV_FUNCTIONS============CP
2	5,318	2,369	1.90	0.85	Call M. LCL_MYXMLPARSER=>ABAP_...	CL_GUI_DATAMANAGER...	CL_GUI_DATAMANAGER===========CP
62	6,843	2,335	2.44	0.84	Call Function AC_CALL_METHOD	SAPLOLEA	SAPLOLEA

Figure 5.2 ABAP Runtime Analysis: Aggregated Hit List of Call Locations Sorted according to Net Time

Gross and net time

The runtime analysis establishes the gross and/or net times of individual program calls in microseconds (µs). Gross time is the total time required for the call. This includes the times of all modularization units and ABAP statements in this call. The net time is the gross time minus the time required for the called modularization units (`MODULE`, `PERFORM`, `CALL FUNCTION`, `CALL SCREEN`, `CALL TRANSACTION`, `CALL DIALOG`, `SUBMIT`) and separately specified ABAP statements if logging was activated for them. For "elementary" statements such as `APPEND` or `SORT`, the gross time is the same as the net time. If the gross and net times for a call differ, this call contains other calls or modularization units. For example, if a subroutine shows a gross time of 100,000 µs

and a net time of 80,000 µs, this means 80,000 µs are used for calling the routine itself, and 20,000 µs are assigned to other statements in the routine, which were logged separately. The Eclipse-based evaluation interface uses the terms **Total Time** and **Own Time** as synonyms.

Figure 5.2 shows the analysis of a sample program. In this program, data records were read from tables `VBAK` and `VBAP` and were displayed via the ABAP List Viewer (ALV; SAP class `CL_SALV_TABLE`). The **Hit List** contains the ALV display function (program SAPLOLEA) with 39% net time and access to table `VBAK` at the fore with 8%.

> **Resource Requirement Due to Runtime Analysis**
>
> The runtime analysis involves a lot of work; generating the analysis can as much as double the runtime of a program (compared to a program run without runtime analysis activated). The runtime analysis takes this into account and displays correspondingly adjusted runtimes in the lists. However, if you look at the statistical record created while the runtime analysis was active, you'll see that it's clearly distorted when compared to a program run without runtime analysis. In contrast, letting a performance trace run simultaneously doesn't involve much extra load—less than 5%.

5.2.3 Using Variations

When you execute the ABAP trace function, you can use variants to adjust how the trace is carried out. We highly recommend the use of variants. It's particularly advisable to try out these options when analyzing a complex program because data quantities of several megabytes can be generated very quickly, much of which is often completely irrelevant to the analysis. Using variants, you can determine more precisely what you want to analyze.

You can see the currently selected variant in the initial screen of the runtime analysis under **Measurement**. The `DEFAULT` variant is already set in the system. You can save your personal settings as your own variant.

Settings

The **Display Variants** or **Change Variants** button takes you to the screen where you can enter the settings for a variant:

- **Aggregation type**
  On the **Duration and Type** tab, you can, among other things, set the type of aggregation. Aggregation is always relevant when a statement is called numerous times in a program, for example, an SQL statement within a loop. If aggregation isn't activated, an entry will be written to the trace measurement file each time the SQL statement is called. If aggregation is activated, only one entry will be recorded, in which the runtimes for each execution are added together. On the **Duration and Type** tab, you're offered two options:
  - **No Aggregation**
    An entry is written to the measurement file each time the statement is called.
  - **Aggregation According to Call Location**
    The runtimes for individual executions of a statement are added together in one entry. However, if a statement appears several times in a program at several locations in the program text, one entry is written for each time it appears in the program text. This option is activated in the default variant.

  In general, activating aggregation dramatically reduces the size of the measurement file, and, in many cases, an analysis of long program sequences is only possible with aggregation. When you use aggregation, some evaluation functions (e.g., the hierarchy list) are lost and therefore no longer available for evaluation.
- **Trace on remote systems**
  Moreover, you specify on the **Duration and Type** tab whether RFCs and update calls are supposed to be recorded as well. If you activate the corresponding option, this information is forwarded to the corresponding processes. To be able to record an RFC in an external system, that system must accept external trace calls. You configure this using the `rstr/accept_remote_trace` parameter, which you must set to `true`. The trace file is then written to the external system. If you use the end-to-end (E2E) trace in SAP Solution Manager, this function is activated automatically, and the E2E trace automatically reads all trace files.
- **Filter operations**
  The settings on the **Statements** tab determine which operations will be monitored in the runtime analysis.

[+]

Tip

If you want to analyze operations on internal tables such as `Append`, `Loop`, or `Sort`, activate the **Read Operations** and **Change Operations** checkboxes under **Int. Tables**. These settings aren't activated in the default variant.

- **Filter program parts**
  On the **Program Parts** tab, you can set which parts of the program to analyze. If you select the **Restrict to Program Parts** option, you can restrict the trace to selected classes, function groups, and programs, and their subcomponents, methods, function modules, and form routines.

[+]

Procedure for More Complex Programs

How should you carry out an analysis of more complex programs? We recommend that you first carry out an analysis of the entire program with aggregation according to call location and without analyzing operations on internal tables (default variant settings). The objective of this analysis is to find the modularization units with the highest runtimes. After this initial analysis, sort the hit list according to net times and identify the modularization units or statements with high runtime.

If you can't deduce recommendations for optimizing the program from this first analysis, perform a more detailed analysis, setting variants to limit the analysis to these modularization units. Simultaneously, activate the trace for operations from internal tables and deactivate the aggregation.

5.2.4 Using Timeline Views

The Eclipse-based interface provides an exclusive view, which displays the program flow on a timeline. This view indicates the number of nesting levels of the program and the calls with a high net time.

To use the timeline view, follow these steps:

1. Choose the **ABAP Profiling** perspective and **ABAP Traces** view in the Eclipse-based ABAP development environment. Select a file, and open

the **Call Timeline** view. The system displays the time line of the runtime analysis (see Figure 5.3).

2. You can use navigation tools such as zoom settings and the preview to navigate in the lower part of the view. Calls with a high net time are displayed as long horizontal bars in the view.
3. Select one of the critical calls. A window shows the following information about the call:
   - Operation
   - Calling program and program that is called
   - Gross and net time
4. Using the functions in the context menu, you can navigate to other views, such as **Hit List** and **Call Hierarchy**.
5. To improve the display of the overview, define your own coloring schema. For this purpose, select **Edit Coloring Schemas** from the context menu. Assign specific colors to the program groups. For example, assign programs with the "Z" schema to the color "Red" to identify program parts that begin with a prefix. Figure 5.3 shows a colored timeline view. You can store coloring schemas as local files and upload them to the system.

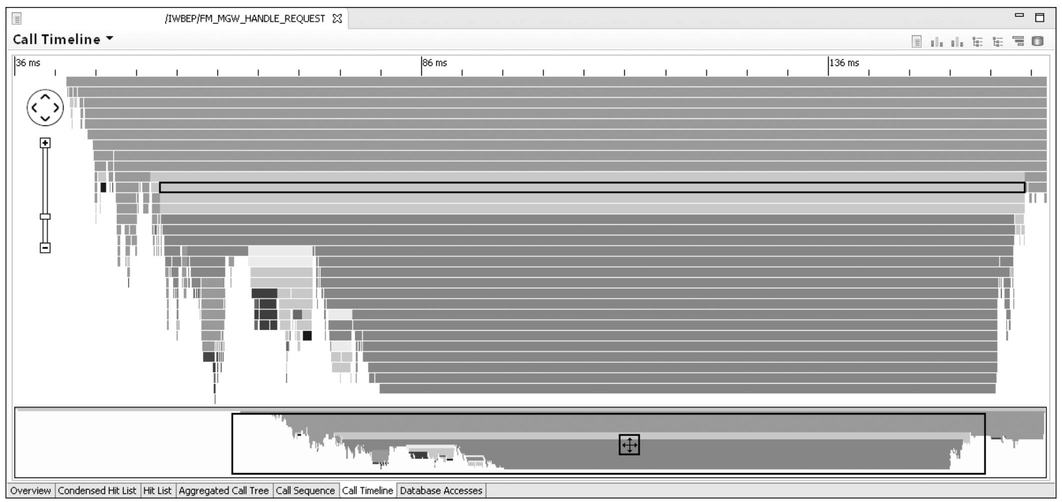


Figure 5.3 ABAP Runtime Analysis: Timeline View

5.3 Analyzing Memory Usage with ABAP Debugger and in the Memory Inspector

In addition to expensive SQL statements, one of the most important causes of performance problems is internal tables with many entries. Large internal tables consume massive amounts of memory and CPU, for example, during copy, sort, or search operations.

You can use the ABAP debugger to create an overview of all internal tables of a program. ABAP debugger is a tool for performing functional troubleshooting in programs. You can find more detailed descriptions of the debugger in SAP literature under ABAP programming. You'll find information on main memory usage both in the classic and the new debugger. You can set the debugger in the ABAP Editor (Transaction SE38) via **Utilities • Settings • Debugging**.

Performance analysis using the ABAP debugger isn't a standard procedure and is best performed by an ABAP developer.

[!]

Rules When Debugging

Take the following advice into account when working with the ABAP debugger. During the debugging process, the ABAP program may terminate and display the error message **Invalid interruption of a database selection**, or the system may automatically trigger a database commit. In either case, an SAP *Logical Unit of Work (LUW)* has been interrupted, and this may lead to inconsistencies in the application tables. Therefore, you should only debug on a test system or in the presence of someone who is very familiar with the program being analyzed and who can manually correct inconsistencies in the database tables if necessary. See "Debugging Programs in the Production Client" in SAP Online Help for the ABAP debugger.

Steps for performance analysis

You perform a performance analysis with the debugger as follows:

1. Start the program to be analyzed. Then open a second session. Here you can monitor the program to be analyzed in the work process overview (Transaction SM50). Enter the debugger from the work process overview by selecting the **Debugging** function. By using the debugger several times in succession, you can identify the parts of the program that cause

high CPU consumption. Often, these sections consist of `LOOP ... ENDLOOP` statements that affect large internal tables.

2. To display the current memory requirements (in the "classic" debugger), select **Goto • Other Screens • Memory Use**.
3. Check for cases of unnecessary memory consumption that may have been caused by a nonoptimal program or inefficient use of a program. As a guideline, bear in mind that a program being used by several users in dialog mode should not allocate more than 100 MB.
4. As of SAP NetWeaver AS ABAP 6.20, you can use the classic debugger to create a list of program objects located in the memory by selecting **Goto • Status Display • Memory Use**. Under **Memory Consumption**, the **Ranking Lists** tab contains a list of objects and their memory consumption.

   In SAP NetWeaver AS ABAP versions 4.6 und 6.10, you can obtain a memory consumption list by choosing **Goto • System • System Areas**. Enter "ITAB-TOP25" in the **Area** field. This way, you'll obtain a list of the 25 largest internal tables.
5. In the "new" debugger, you first display the memory analysis tool by clicking the button for the new tool and then selecting **Memory Analysis** from the tools on offer in the **Memory Management** folder. The initial screen then displays how much memory is allocated or used by the analyzed internal session. Click the **Memory Objects** button to go to the list of the largest memory objects, which can be the internal tables, objects, anonymous data objects, or strings.

Memory Inspector

Moreover, you can create and then analyze a memory extract, that is, an overview of the objects that occupy memory space. You can create a memory extract in any transaction by selecting **System • Utilities • Memory Analysis • Create Memory Extract** or simply enter function code "/HMUSA". The third option is to create a memory extract from program coding. Refer to SAP Help for a description of the system class `CL_ABAP_MEMORY_UTILITIES`.

Evaluating the memory extract

To evaluate the memory extract, start the Memory Inspector by selecting **System • Utilities • Memory Analysis • Compare Memory Extracts** in any transaction or via Transaction S_MEMORY_INSPECTOR. The Memory Inspector lists all memory extracts in the upper part of the screen. In the lower part of the screen, you can find details about the individual memory extract. Here, a distinction is made among the object types, programs, classes, dynamic memory request of a class, table bodies, strings, and types

of anonymous data objects. You're provided with different ranking lists, according to which you can sort the objects. For each memory object, you're provided with the values of bound allocated, bound used, referenced allocated, and referenced used memories. You can find a detailed description of the ranking lists and the displayed values in SAP Help.

Using the Memory Inspector

The Memory Inspector is particularly useful for examining transactions over a long period of time, as is the case in a customer interaction center. Here, users frequently enter a transaction at the beginning of their workday and exit it when they go home. In these "long-term" transactions, data often remains, and therefore memory consumption continuously increases.

Figure 5.4 shows an example of a memory extract. The dominator tree shows the hierarchical program structure and the memory used by the program parts. With a size of 494 MB, table `LT_MEM` is conspicuous. The next largest object is the `CL_GUI_ALV_GRID` class with a size of 250 KB. Below this class, 130 KB are used by table `MT_DATA`.

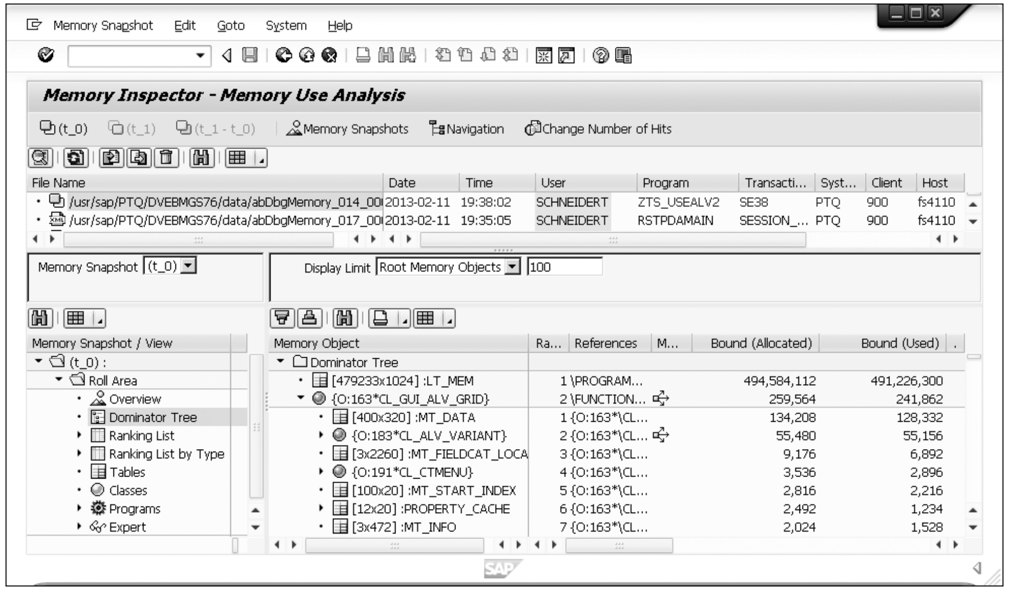


Figure 5.4 Memory Inspector: Dominator Tree

5.4 Code Inspector

The Code Inspector tool checks ABAP programs and other repository objects statically for problems. Its meaning for the quality analysis of ABAP programs goes far beyond the meaning for performance analysis, to which the following description is restricted. The Code Inspector performs a static analysis, that is, the code doesn't need to be executed and is therefore independent of test or production data. It has been generally available as of SAP Basis version 6.10.

You call the Code Inspector from the development tools of the ABAP Workbench for programs, function modules, or ABAP classes (Transactions SE38, SE37, or SE24) using the menu path **Program/Function Module/Class • Check • Code Inspector** or directly using Transaction SCII.

Performance checks in the Code Inspector

Table 5.8 summarizes which checks are performed in the **Performance** category and where you can find further information on the checks. The checks reveal standard errors and problems. They can't indicate how seriously they will affect the performance. For this purpose, you require the runtime checks described previously. The verification of these checks should nevertheless be part of the quality control of ABAP programs.

Check	Description
Analysis of the `WHERE` clause	Checks that the SQL statements include a `WHERE` clause and that they have index support. Buffered tables, joins, and views aren't covered by this check. For more information, see Chapter 11, Section 11.2.
Analysis of table buffer accesses	Checks whether SQL statements to buffered tables will access the database and pass the buffer. For more information, see Chapter 12, Section 12.1.2.
`SELECT` statements with `CHECK`	Checks whether `SELECT`/`ENDSELECT` loops include a `CHECK` statement. This statement can often be integrated in the `WHERE` clause and thus reduce the read data quantity from the start. For detailed information, see Chapter 11, Section 11.2.
SQL statements in loops	Checks whether SQL statements are in loops. These result in an increased communication effort and can possibly be replaced by bundled accesses. For more information on this, see Chapter 11, Section 11.2.
Nested loops	Checks the system for nested loops where nonlinear runtime behavior can occur.
Copying large data objects	Sends alerts on high copy costs for large data objects, for instance, nested internal tables. However, this check only refers to structures with a width of more than 1,000 bytes; it can't consider how long a table will be at runtime.
Inefficient operations on internal tables	Sends alerts on inefficient read accesses to internal tables.
Inefficient parameter transfers	Examines whether there is a better way to transfer parameters when you call a form, function module, method, or event.
`EXIT` or no ABAP commands in `SELECT`/`ENDSELECT` loop	Checks whether `SELECT`/`ENDSELECT` loops include an `EXIT` statement or no ABAP coding at all. With this type of code, you often check the existence of records in a database table, which can be designed to be more efficient. For detailed information, see Chapter 11, Section 11.2.
Instance generation of Business Add-Ins (BAdIs)	As of SAP Basis version 7.0, you should use the statement `GET BADI` instead of the method call `CALL METHOD cl_exithandler=>get_instance` for performance reasons.
Check of table properties	Examines the technical settings such as transportability, buffering, and indices of database tables.

Table 5.8 Performance Checks in the Code Inspector

Table 5.8 Performance Checks in the Code Inspector (Cont.)

[»]

Check Variants in the Performance Environment

The Code Inspector offers *check variants* that bundle checks for specific areas. For performance analysis, the following variants are available:

- PERFORMANCE_CHECKLIST
  Minimum checklist with performance checks.
- PERFORMANCE_CHECKLIST_HDB
  Checklist with SAP HANA-specific performance checks.
- PERFORMANCE_CHECKLIST_XL
  Advanced checklist with performance checks.
- PERFORMANCE_DB
  Checklist of database-specific performance checks.

The following checklist can be used in the context of the migration from a traditional database to SAP HANA and comprise function checks (e.g., for code that might include programming on database-specific functions) and performance checks (code that might lead to performance issues:

- FUNCTIONAL_DB and FUNCTIONAL_DB_ADDITION
  Classic and advanced checklist for the migration to SAP HANA.

5.5 Tips and Tricks for High-Performance ABAP Programs

Frequent performance problems

In the SAP online help for the ABAP programming environment and in the ABAP runtime environment (Transaction SAT), the "Tips and Tricks" section provides detailed information on high-performance ABAP programming, which you can use to get familiar with the sample code introduced there.

There are three common programming errors that cause large memory or CPU requirements for programs:

- **Missing REFRESH or FREE statements**
  The ABAP statements REFRESH and FREE delete internal tables and release the memory that was allocated to them. If these statements are missing, memory resources may be unnecessarily tied up, and the operations being executed (READ or LOOP) will require an unnecessarily large amount of time.

[«]

What Effect Do CLEAR, REFRESH, FREE, and DELETE Have?

The ABAP statements CLEAR, REFRESH, FREE, and DELETE have the following effect on the allocated memory:

- A CLEAR or REFRESH statement causes the release of the table content and—if available—of the indexes. The header information of the table remains unchanged. Note that the CLEAR statement only deletes the header row in internal tables with a header row. Use this statement if the table is to be used again as the program proceeds.
- A FREE statement causes the complete release of the table content and—if available—of the indexes. Use this statement if the table isn't to be used again as the program proceeds.
- A DELETE statement in an internal table causes *no* release of memory! If you reduce an internal table with a large number of entries to very few entries using DELETE statements, the memory still remains allocated. When you copy such a table, the "memory image" is copied; that is, the copied table has the same size as the original table. To solve this problem, build a new small table instead of thinning out a large table using DELETE.

- **Inefficient reading in larger internal tables**
  The ABAP statement READ TABLE ... WITH KEY ... enables you to search internal tables. If you use this statement by itself for a standard table, the search is sequential. For large tables, this is a time-consuming process. You can significantly improve search performance by adding the clause ... BINARY SEARCH, thereby specifying a binary search. However, the table must be sorted (see ABAP Help for the statement READ TABLE).

  You can optimize the performance of operations on large tables by using sorted tables (SORTED TABLE) or hash tables (HASHED TABLE). If a READ statement is executed on a sorted table, the ABAP processor automatically performs a binary search. It's important that the key fields used for the search correspond to the sort criteria for the table. For a sorted table, the search effort increases logarithmically with the size of the table. For hash tables, constant access costs exist if the ABAP statement READ TABLE ... WITH TABLE KEY is used. However, efficient access to hash tables is only possible if you enter the complete key.

- **Nested loops**
  Nested loops are frequently used for processing dependent tables (e.g., header and position data):

```
LOOP AT HEADER INTO WA_HEADER.
  LOOP AT POSITION INTO WA_POSITION
              WHERE KEY = WA_HEADER-KEY.
    "Processing ...
  ENDLOOP.
ENDLOOP.
```

  If HEADER and POSITION are standard tables in this case, then for each entry in the HEADER table, the ABAP processor loops across all entries in the POSITION table and checks if the WHERE clause is fulfilled for all entries. This is especially time-consuming if the tables HEADER and POSITION contain many entries. However, if you use SORTED tables, the ABAP processor determines the data to be processed by performing a binary search and only loops across those areas that fulfill the WHERE clause. Sorting is only useful, however, if the WHERE clause contains the first fields of the sort key.

  As an alternative, you can use index operations that are much less time-consuming. To use index operations, it's necessary to sort the internal tables HEADER and POSITION by the KEY field as follows:

```
I = 1.
LOOP AT HEADER INTO WA_HEADER.
  LOOP AT POSITION INTO WA_POSITION FROM I.
    IF WA_POSITION-KEY <> WA_HEADER-KEY.
      I = SY-TABIX.
      EXIT.
    ENDIF.
    " ...
  ENDLOOP.
ENDLOOP.
```

Table 5.9 summarizes the most important performance properties of standard tables, sorted tables, and hash tables in ABAP. It provides information on how access costs increase when the size of the table increases (see also the "Scalability of a Program" section). Based on these properties, you can determine which table type is ideal for your specific application.

Operation	Standard Table	Sorted Table	Hash Table
Populate (mass operation)			
APPEND LINES ...	Depending on position and size due to movement costs	Depending on position and size due to movement costs (check required)	Depending on position and size due to movement costs
INSERT LINES ...	Depending on position and size due to movement costs	Depending on position and size due to movement costs (check required)	Depending on position and size due to movement costs
Populate (single record)			
APPEND	Constant	Constant (higher than standard as check required)	–
INSERT ... INTO ... INDEX	Depending on position and size due to movement costs	Depending on position and size due to movement costs (higher than standard as check required)	–
INSERT ... INTO ... TABLE	Constant	Logarithmic	Constant (higher than standard due to hash management)
Read (areas)			
LOOP ... ENDLOOP (all rows)	Linear	Linear	Linear

Table 5.9 Most Important Performance Properties of Standard Tables, Sorted Tables, and Hash Tables in ABAP

Operation	Standard Table	Sorted Table	Hash Table
LOOP ... WHERE ENDLOOP (complete key)	Linear	Logarithmic	Constant
LOOP ... WHERE ENDLOOP (incomplete key with initial section)	Linear	Logarithmic	Linear
LOOP ... ENDLOOP WHERE (incomplete key without initial section)	Linear	Linear	Linear
LOOP ... FROM ... TO	Constant	Constant	–
Read (individual rows)			
READ ... INDEX	Constant	Constant	–
READ ... WITH KEY (complete key)	Linear, logarithmic for binary search	Logarithmic	Constant
LOOP ... WHERE ENDLOOP (incomplete key with initial section)	Linear, logarithmic for binary search	Logarithmic	Linear
LOOP ... WHERE ENDLOOP (incomplete key without initial section)	Linear	Linear	Linear
Change (analogously, delete)			
MODIFY ... TRANSPORTING ... WHERE (complete key)	Linear	Logarithmic	Constant

Table 5.9 Most Important Performance Properties of Standard Tables, Sorted Tables, and Hash Tables in ABAP (Cont.)

Operation	Standard Table	Sorted Table	Hash Table
MODIFY ... TRANSPORTING ... WHERE (incomplete key with initial section)	Linear	Logarithmic	Linear
MODIFY ... TRANSPORTING ... WHERE (incomplete key without initial section)	Linear	Logarithmic	Linear
MODIFY ... FROM <wa> (index access)	Constant	Constant	–
MODIFY TABLE ... FROM <wa >	Linear	Logarithmic	Constant
COLLECT (table compression)	Linear	Logarithmic	Constant
SORT	Runtime intensive, depending on size and sorting	(already sorted)	Runtime intensive, depending on size and sorting

Table 5.9 Most Important Performance Properties of Standard Tables, Sorted Tables, and Hash Tables in ABAP (Cont.)

Scalability of a Program

Scalability of a program refers to the dependency of a program's runtime from the data quantity. Many operations depend linearly on the data quantity ($t = O(n)$); that is, the runtime increases linearly to the data quantity. Examples include the database selections in large tables without or with inappropriate index support and loops via internal tables in the program. Linear scalability is acceptable for the processing of medium data quantities. If they can't be avoided in programs that are supposed to process large data quantities, you must consider parallelization.

Of course, constant runtimes ($t = O(1)$) or a logarithmic dependency ($t = O(\log n)$) is better for performance than a linear scalability. Logarithmic dependencies occur, for example, for database selections in large tables with optimal index support or for read operations in internal tables with binary search. Because the logarithm function increases only very slowly, in real life, you don't need to differentiate between constant and logarithmically increasing runtimes.

Quadratic dependencies ($t = O(n \times n)$) and anything beyond are inacceptable for the processing of medium and large data quantities. However, problems with quadratic dependencies through intelligent programming can usually be traced back to dependencies of $t = O(n \times \log n)$. An example is the comparison of two tables that both grow with a factor of n. A comparison of the unsorted tables would result in a quadratic dependency, a comparison with sorted tables in dependency $t = O(n \times \log n)$. Because the logarithm function increases only very slowly, in real life, you don't need to differentiate between an increase of $t = O(n \times \log n)$ and a linear increase.

5.6 Summary

Performance trace

This chapter discussed monitors that enable detailed analysis of individual ABAP programs. *SQL trace* is the recommended tool for analyzing SQL statements in ABAP programs. Evaluating the trace enables you to identify network problems or throughput bottlenecks in the database. You'll find further information on optimizing SQL statements in Chapter 11.

You use RFC trace to analyze the performance of sent and received RFCs. As of SAP NetWeaver AS ABAP 7.10, you also have an HTTP trace at hand to record HTTP statements. For more information, see Chapter 7 and Chapter 8.

The enqueue trace is a means for selecting analyses of lock operations (enqueue/dequeue operations). For detailed information, see Chapter 10.

You trace inefficient table buffering with the buffer trace. For detailed information, see Chapter 12.

ABAP trace and ABAP debugger

For high CPU consumption problems, use an *ABAP trace*. In contrast to an SQL trace, an ABAP trace enables time measurements for operations on internal tables (`LOOP`, `READ`, `SORT`, etc.).

As an alternative, you can monitor CPU-consuming programs using the ABAP debugger, which you can call from the work process overview. However, only developers should perform this analysis.

Code Inspector

You should examine ABAP programs proactively using the Code Inspector, which implements static checks of the program and sends alerts on standard performance errors and problems.

Important Concepts

After studying this chapter, you should be familiar with the following concepts:

- Performance trace: SQL trace, buffer trace, RFC trace, HTTP trace, enqueue trace
- ABAP trace
- ABAP debugger
- Code Inspector
- Internal tables and their performance attributes

5.7 Questions

Appendix C, Section C.1, provides the answers to these questions.

1. What do you have to consider when you perform an SQL trace?
   a) There is only one trace file in each SAP system. Therefore, only one SQL trace can be created per SAP system.
   b) The user whose actions are being traced should not run multiple programs concurrently.
   c) You should perform the SQL trace on a second execution of a program because the relevant buffers will already have been loaded.
   d) SQL traces are useful on the database server but not on application servers, which yield inexact results due to network times.
2. When should you perform an ABAP trace?
   a) If a problem occurs with the table buffer
   b) For programs with high CPU requirements
   c) For analyzing I/O problems on hard drives

Contents

Index

D

E

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

SAP PRESS First-hand knowledge.

Rheinwerk Publishing

Thomas Schneider

SAP Performance Optimization Guide: Analyzing and Tuning SAP Systems

892 Pages, 2018, $89.95
ISBN 978-1-4932-1524-9

www.sap-press.com/4370

Dr. Thomas Schneider started his career at SAP AG in 1996. Among other things, he was in charge of the Center of Expertise for Performance in the Service & Support group and the responsible Support Alliance manager for key accounts. Since 2004 he has worked in the Research & Breakthrough Innovation group, where he first was responsible for IT Service & Application Management. Since 2009 he has focused on the Partner Development Infrastructure in the SAP Cloud organization as the responsible architect.