

# BC490

## ABAP Performance & Tuning

*SAP NetWeaver*

Date \_\_\_\_\_  
Training Center \_\_\_\_\_  
Instructors \_\_\_\_\_  
Education Website \_\_\_\_\_

### Participant Handbook

Course Version: 62  
Course Duration: 2 Day(s)  
Material Number: 50084185



*An SAP course - use it to learn, reference it for work*

## Copyright

Copyright © 2008 SAP AG. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP AG. The information contained herein may be changed without prior notice.

Some software products marketed by SAP AG and its distributors contain proprietary software components of other software vendors.

## Trademarks

- Microsoft®, WINDOWS®, NT®, EXCEL®, Word®, PowerPoint® and SQL Server® are registered trademarks of Microsoft Corporation.
- IBM®, DB2®, OS/2®, DB2/6000®, Parallel Sysplex®, MVS/ESA®, RS/6000®, AIX®, S/390®, AS/400®, OS/390®, and OS/400® are registered trademarks of IBM Corporation.
- ORACLE® is a registered trademark of ORACLE Corporation.
- INFORMIX®-OnLine for SAP and INFORMIX® Dynamic ServerTM are registered trademarks of Informix Software Incorporated.
- UNIX®, X/Open®, OSF/1®, and Motif® are registered trademarks of the Open Group.
- Citrix®, the Citrix logo, ICA®, Program Neighborhood®, MetaFrame®, WinFrame®, VideoFrame®, MultiWin® and other Citrix product names referenced herein are trademarks of Citrix Systems, Inc.
- HTML, DHTML, XML, XHTML are trademarks or registered trademarks of W3C®, World Wide Web Consortium, Massachusetts Institute of Technology.
- JAVA® is a registered trademark of Sun Microsystems, Inc.
- JAVASCRIPT® is a registered trademark of Sun Microsystems, Inc., used under license for technology invented and implemented by Netscape.
- SAP, SAP Logo, R/2, RIVA, R/3, SAP ArchiveLink, SAP Business Workflow, WebFlow, SAP EarlyWatch, BAPI, SAPPHIRE, Management Cockpit, mySAP.com Logo and mySAP.com are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world. All other products mentioned are trademarks or registered trademarks of their respective companies.

## Disclaimer

THESE MATERIALS ARE PROVIDED BY SAP ON AN "AS IS" BASIS, AND SAP EXPRESSLY DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR APPLIED, INCLUDING WITHOUT LIMITATION WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, WITH RESPECT TO THESE MATERIALS AND THE SERVICE, INFORMATION, TEXT, GRAPHICS, LINKS, OR ANY OTHER MATERIALS AND PRODUCTS CONTAINED HEREIN. IN NO EVENT SHALL SAP BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR PUNITIVE DAMAGES OF ANY KIND WHATSOEVER, INCLUDING WITHOUT LIMITATION LOST REVENUES OR LOST PROFITS, WHICH MAY RESULT FROM THE USE OF THESE MATERIALS OR INCLUDED SOFTWARE COMPONENTS.

# About This Handbook

This handbook is intended to complement the instructor-led presentation of this course, and serve as a source of reference. It is not suitable for self-study.

## Typographic Conventions

American English is the standard used in this handbook. The following typographic conventions are also used.

Type Style	Description
<i>Example text</i>	Words or characters that appear on the screen. These include field names, screen titles, pushbuttons as well as menu names, paths, and options. Also used for cross-references to other documentation both internal (in this documentation) and external (in other locations, such as SAPNet).
<b>Example text</b>	Emphasized words or phrases in body text, titles of graphics, and tables
EXAMPLE TEXT	Names of elements in the system. These include report names, program names, transaction codes, table names, and individual key words of a programming language, when surrounded by body text, for example SELECT and INCLUDE.
Example text	Screen output. This includes file and directory names and their paths, messages, names of variables and parameters, and passages of the source text of a program.
<b>Example text</b>	Exact user entry. These are words and characters that you enter in the system exactly as they appear in the documentation.
< <b>Example text</b> >	Variable user entry. Pointed brackets indicate that you replace these words and characters with appropriate entries.

## Icons in Body Text

The following icons are used in this handbook.

Icon	Meaning
	For more information, tips, or background
	Note or further explanation of previous point
	Exception or caution
	Procedures
	Indicates that the item is displayed in the instructor's presentation.

# Contents

<b>Course Overview .....</b>	<b>vii</b>
Course Goals.....	vii
Course Objectives .....	vii
<b>Unit 1: Architecture and Technical Overview .....</b>	<b>1</b>
Architecture and Technical Overview .....	2
<b>Unit 2: Analysis Tools .....</b>	<b>33</b>
Analysis Tools .....	34
<b>Unit 3: Basics for Indexes .....</b>	<b>95</b>
Basics for Indexes .....	96
<b>Unit 4: Accessing Single Tables .....</b>	<b>135</b>
Accessing Single Tables .....	136
<b>Unit 5: Accessing Multiple Tables .....</b>	<b>169</b>
Accessing Multiple Tables .....	170
<b>Unit 6: Buffering Strategies .....</b>	<b>207</b>
Buffering Strategies .....	208
<b>Unit 7: Summary and Additional Tips .....</b>	<b>287</b>
Summary and Additional Tips .....	288
<b>Unit 8: Special Topics (Optional) .....</b>	<b>311</b>
Special Topics (Optional) .....	312
<b>Index.....</b>	<b>335</b>



# Course Overview

## Target Audience

This course is intended for the following audiences:

- ABAP programmers
- ABAP consultants
- Performance officer

## Course Prerequisites

### Required Knowledge

- SAPTEC (SAP50)
- BC400 (ABAP Basics)
- Programming experience in ABAP
- Programming experience with relational databases

## Course Goals



This course will prepare you to:

- Diagnose frequent errors in ABAP programming that cause performance problems
- Improve ABAP programs proactively in non-production ABAP systems and reactively in production ABAP systems

## Course Objectives



After completing this course, you will be able to:

- Use SAP tools for analyzing ABAP performance
- Systematically analyze ABAP performance problems
- Solve ABAP performance problems that cause high database loads (database accesses) or high CPU loads (accesses of internal tables), and analyze performance problems in special areas

## **SAP Software Component Information**

The information in this course pertains to the following SAP Software Components and releases: SAP Netweaver AS ABAP 7.00

# Unit 1

## Architecture and Technical Overview

### Unit Overview

#### Contents:



- Course Goals
- Course Objectives
- Table of Contents
- Course Overview Diagram
- Main Business Scenario



### Unit Objectives

After completing this unit, you will be able to:

- Describe the architecture of SAP NW AS ABAP
- Name important monitoring tools for the purposes of administration and performance analysis

### Unit Contents

Lesson: Architecture and Technical Overview .....	2
Exercise 1: Architecture and System Overview .....	25

# Lesson: Architecture and Technical Overview

## Lesson Overview

### Contents:



- Course Goals
- Course Objectives
- Table of Contents
- Course Overview Diagram
- Main Business Scenario



## Lesson Objectives

After completing this lesson, you will be able to:

- Describe the architecture of SAP NW AS ABAP
- Name important monitoring tools for the purposes of administration and performance analysis

## Business Example

### This course will prepare you to:

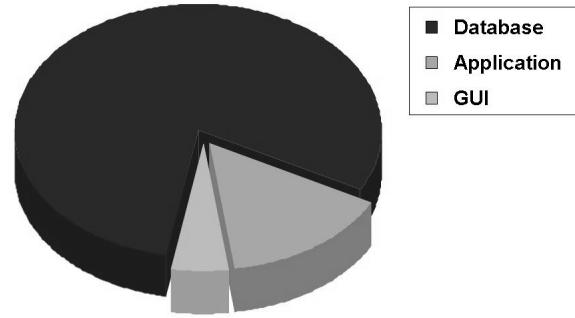
- Diagnose frequent errors that cause performance problems in ABAP programming
- Improve ABAP programs proactively in non-production R/3 Systems and reactively in production R/3 Systems



**Note:** You can obtain other documents on the subject of performance for use in the course on the Service Marketplace <http://service.sap.com/performance> or help portal at <http://help.sap.com> or via the Developer Network (SDN) on the Service Marketplace.



- On average, an ABAP command is executed within microseconds.
- On average, a database access is executed in milliseconds.
- For business applications, performance is chiefly dependent on the database access strategy.



**Figure 1: Graph: Performance and Cause**

The graph above illustrates where, in practice, the greatest optimization potential lies in business applications. Optimal database access is a key factor, but the logic of the application is also crucial (buffering, internal tables, etc.).



**The SAP System**

**The Database System**

**Figure 2: Architecture and Technical Overview (1)**

## ABAP Application Server (AS ABAP)

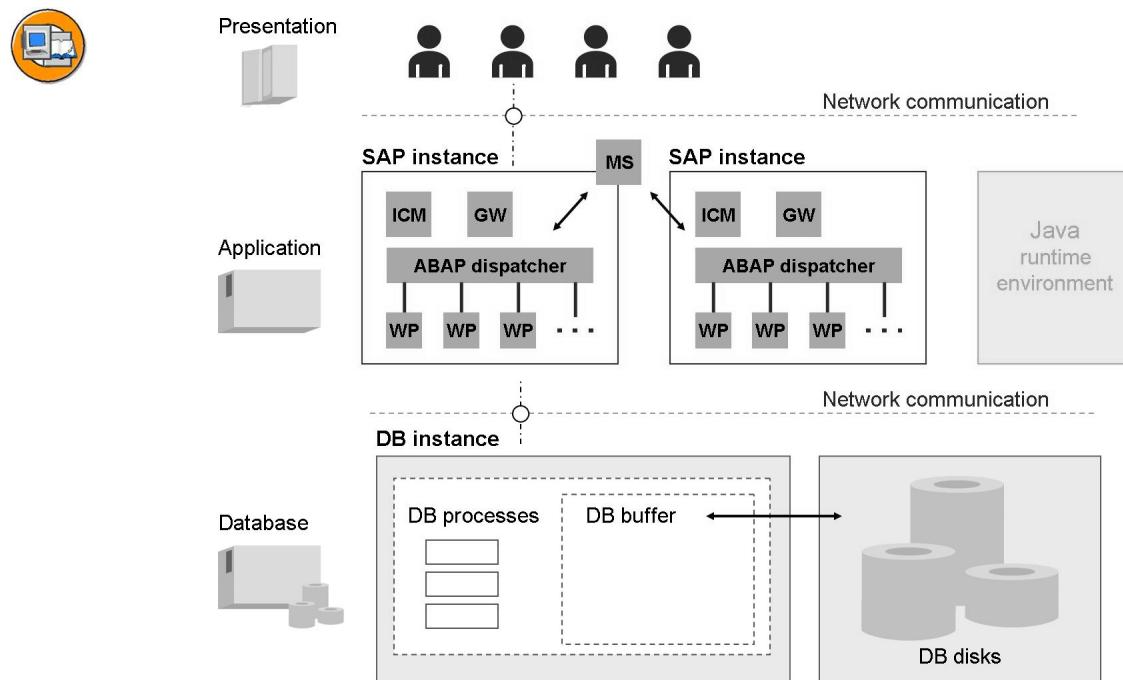


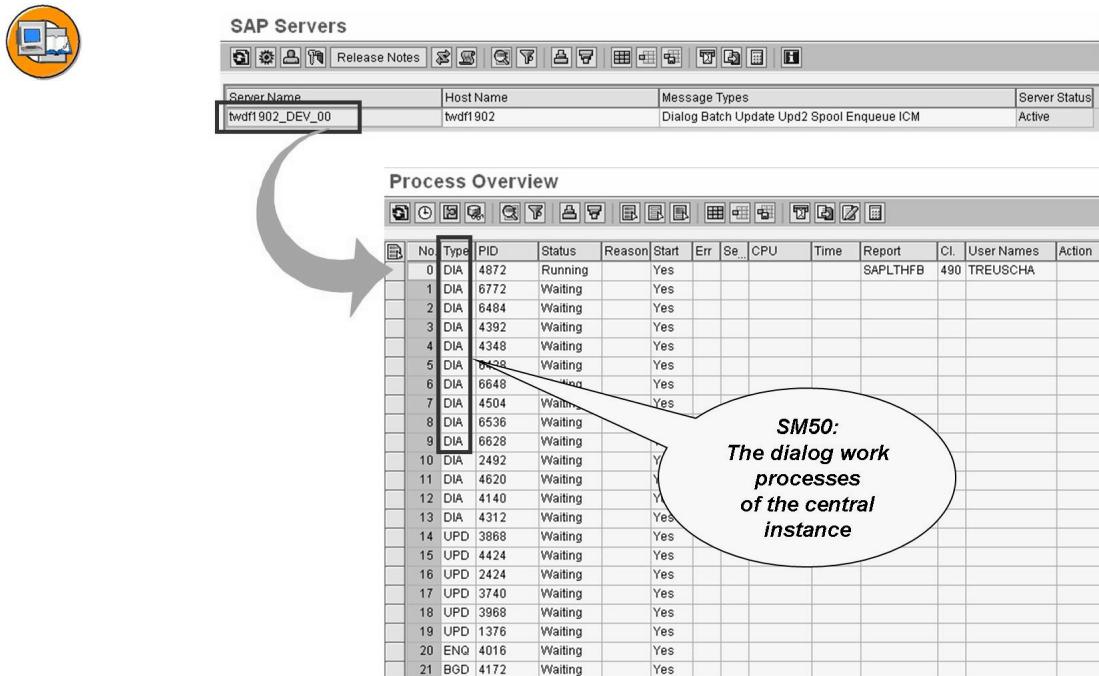
Figure 3: Architecture of the ABAP Application Server

This course addresses performance-critical aspects of ABAP programming on the SAP NW application server ABAP. We will begin with a standard scenario for a user logged on to the SAP NW AS ABAP system (hereinafter referred to simply as the SAP system) via the SAPGUI. The SAP system is implemented as a multitier client/server architecture (presentation layer, application layer, database layer and possibly a middleware level such as a web server). The presentation and application layers of the SAP system are scalable, which means additional frontends and application servers can be added to deal with hardware bottlenecks. By contrast, the database layer – the central data repository – is not scalable. (A database can have multiple DB instances. However, the SAP environment does not support distribution of the database as regards data retention.)

ABAP performance analysis has two main areas of focus. One goal is to reduce the runtime of programs on the application server. Another goal is to reduce the database load. Reducing the database load is particularly important, because the database server is a central resource.

Data is transferred between a frontend and an application server in blocks of 2 to 10 KB each, while the blocks transferred between an application server and the database server are usually exactly 32 KB in size.

In practice, an additional layer may be used between the presentation and application layers. This middleware layer contains software components, for instance for internet access (external ITS, WebServer, ICM). Performance considerations in this environment (HTTP protocol) are not covered in the course as they are irrelevant to ABAP development. (Trace and debug options in browser-based applications are addressed later.)



**Figure 4: Overview of the SAP system: Instances and Processes**

Transactions SM51 and SM50 provide information about the instances installed on the SAP system as well as corresponding work processes. The graphic above shows a system with an instance known as the “central instance”. The central instance runs the enqueue work process and a message server process. You can branch to the process overview by double-clicking the instance name.

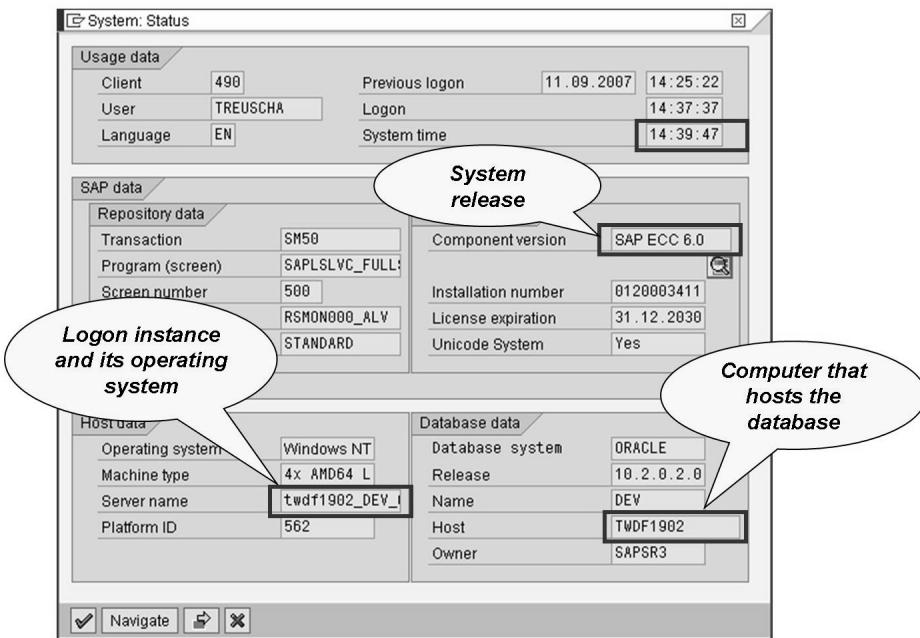
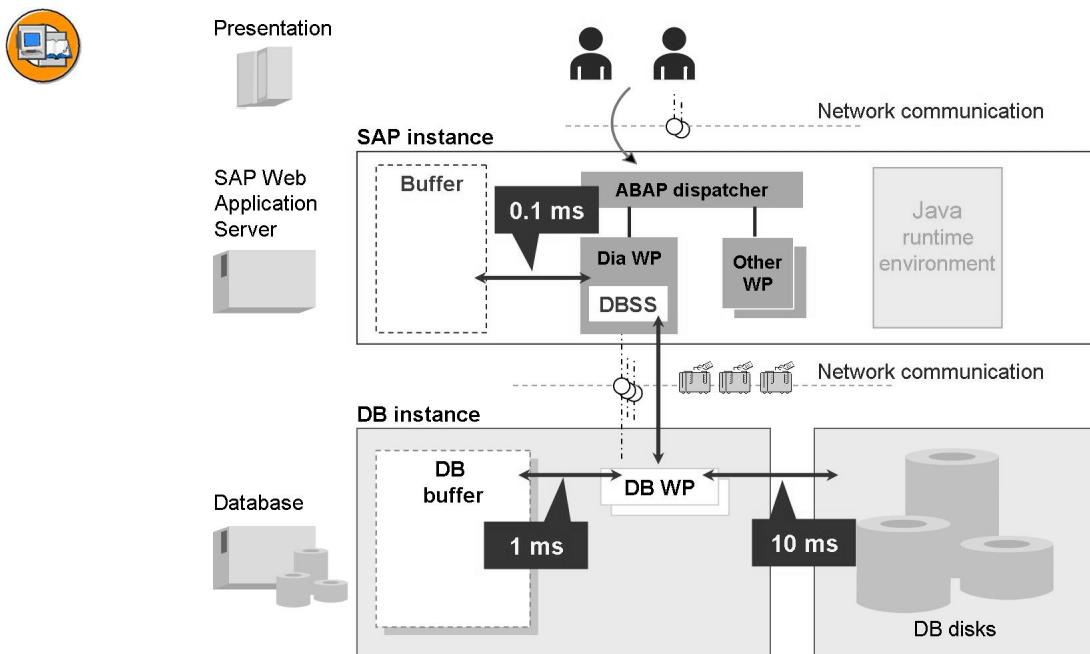


Figure 5: Overview of the SAP System: System Status

If you want to find out which instance you logged onto, you can branch to the system status (*System -> Status*). The question is naturally immaterial in SAP systems with only one instance. The system status can also be helpful for other questions. It displays the database in use (including database release information) and the operating system used of the computer on which the instance is running.



**Figure 6: Access Times to the Sap Buffer and the Database**

If the data is in the SAP table buffers on the application server, the transfer takes roughly 0.1 ms/record, depending on the hardware. If the records are read from the database data buffer, approx. 1 ms/record is needed. If the data has to be read from disk, approx. 10 ms/record are required. (All times listed here are approximate and may vary in practice. Times always depend on the hardware and the architecture of the system. On a smaller, central system where the application and DB are on the same machine, DB access times should be considerably better as there is no need to communicate via a network.)

In a smaller SAP system (release 4.6), an SAP work process allocates approx. 5-10 MB of memory; the SAP table buffers allocate approx. 120 MB (40 MB for single record buffers, 80 MB for generic table buffers); and the data buffers of the database generally allocate several gigabytes of memory. Modern databases for larger installations typically occupy several terabytes of hard disk space.

You can use the “Quicksizer” tool to determine what hardware is required for specific customer requirements. For more information, go to <http://service.sap.com/quicksizer>.

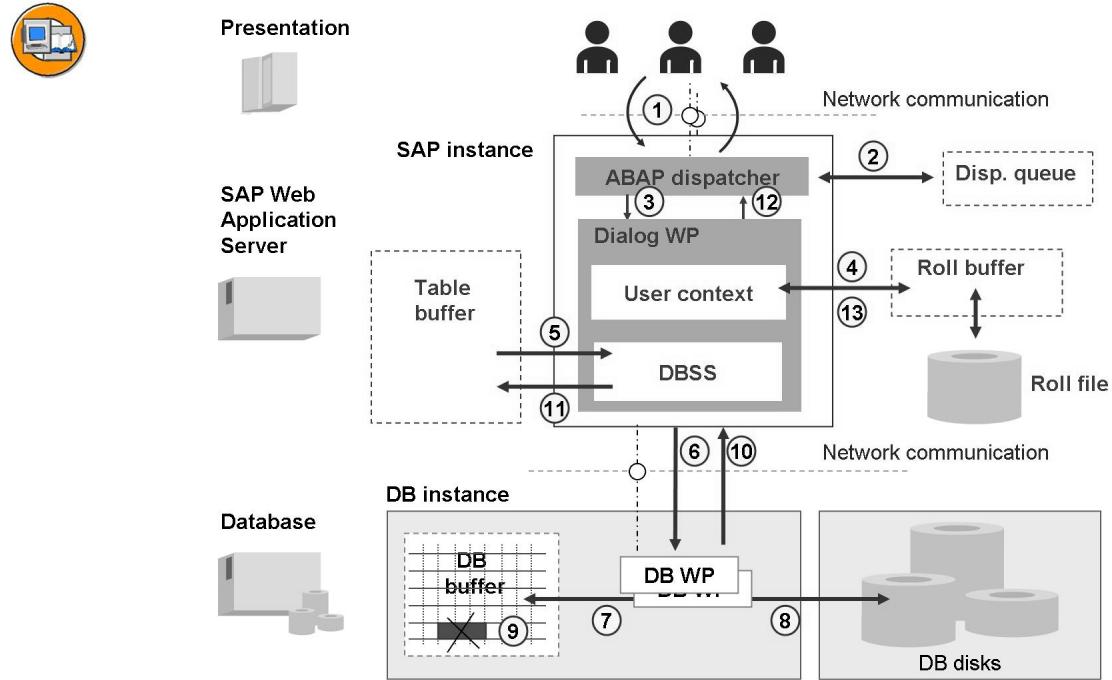
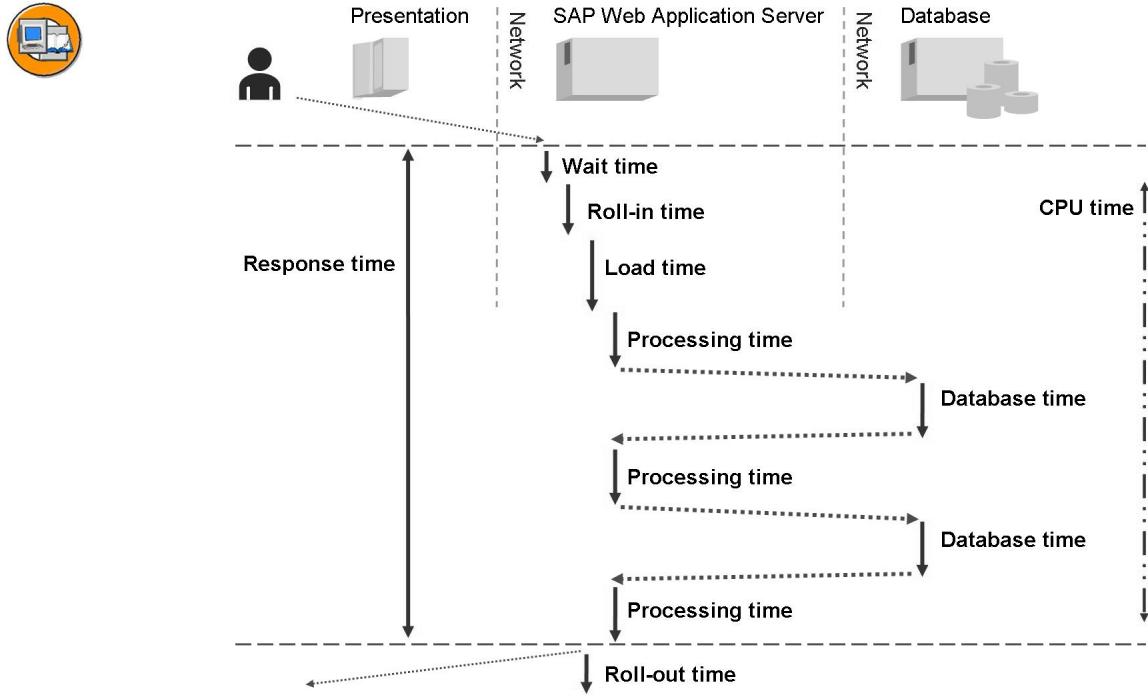


Figure 7: The Dialog Step

During dialog processing, a transaction step corresponds to a change of screens. During an update or spool process, each request counts as a transaction step. Background processing can consist of one or more transaction steps. The graphic above describes the most important processes within a dialog or transaction step in general:

1. Send the user request to the AS
2. Place the user request in the dispatcher queue if all work processes are occupied
3. Assign the user request to a work process
4. Roll the user context into the work process
5. Attempt to satisfy the SQL statement from the AS buffers
6. Send the SQL statement to the database server if the result was not found in the AS buffer
7. Attempt to satisfy the SQL statement from the database buffer
8. Load the missing data blocks from the disk into the DB buffer
9. Displace blocks from the DB buffer if necessary
10. Send the results to the AS
11. Change table buffer due to DB changes
12. Send the results of the request to the presentation server
13. Roll user context out of the work process



**Figure 8: Average Response Time**

The average response time should be under one second. This is the time from the receipt of a user request to the sending of a response as measured on the application server; this does not include network time between the presentation server and the application server.

The graphic shows the components of the response time.

Measurement of the response time starts as soon as the dispatcher receives the request.

This time, also called the AVERAGE RESPONSE TIME, is composed of the following:

- WAIT TIME  
Time spent by the request in the dispatcher queue
- ROLL-IN TIME  
Time required to copy the user context into the work process
- LOAD TIME  
The time required to load and generate program objects (coding, screens, and so on).
- PROCESSING TIME  
This time results from the calculation. The processing time is calculated as the response time minus the following times: wait time, roll-in time, load time, database time, enqueue time, roll-wait time
- DATABASE TIME  
Time between sending an SQL statement and receipt of the results (includes network transmission time)
- ROLL-OUT TIME  
Time required to roll the user context into the roll buffer
- CPU TIME  
Time spent by the CPU in processing the transaction step (as measured by the operating system; not an additive component of the response time).



**Note:** For more information on the distribution and interpretation of the response time, see the following OSS Notes: 8963, 203924, 364625, 376148, 99584



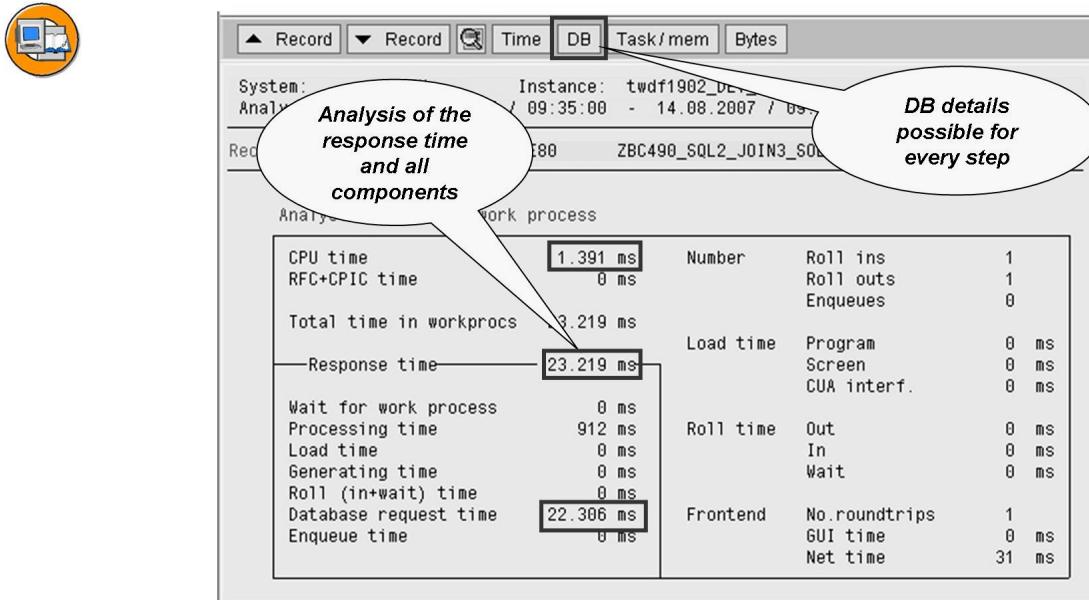
**STAD:**  
*transaction steps in detail*

**Execution of a report with multiple accesses to the list (screen 0120)**

Time	Access	Object	User
16:57:04	*	RSSTAT26	D 0120 0
16:57:04	*	RSSTAT26	D 0010 0
16:57:04	*	RFC	TREUSCHA
16:57:04	*	SAPMSEU0	R 3004
16:57:04	*	SAPMSEU0	D 0500 1
16:57:04	*	SAPMSEU0	D 0700 1
16:57:04	*	RFC	TREUSCHA
16:57:04	*	RFC	D 0500 1
16:57:04	*	RFC	D 0500 1
16:57:04	*	RFC	R 0
16:57:39	twdf1052_DEV_00	SE80	TREUSCHA
16:57:39	twdf1052_DEV_00	SE80	D 0120 1
16:57:39	twdf1052_DEV_00	SE80	TREUSCHA
16:57:40	twdf1052_DEV_00	SE80	D 0120 1
16:57:40	twdf1052_DEV_00	SE80	TREUSCHA
16:57:40	twdf1052_DEV_00	SE80	D 0120 1
16:57:40	twdf1052_DEV_00	SE80	TREUSCHA
16:57:52	twdf1052_DEV_00	STAD	D 0120 0
		RSSTAT26	TREUSCHA

**Figure 9: STAD - Transaction Step Analysis (1)**

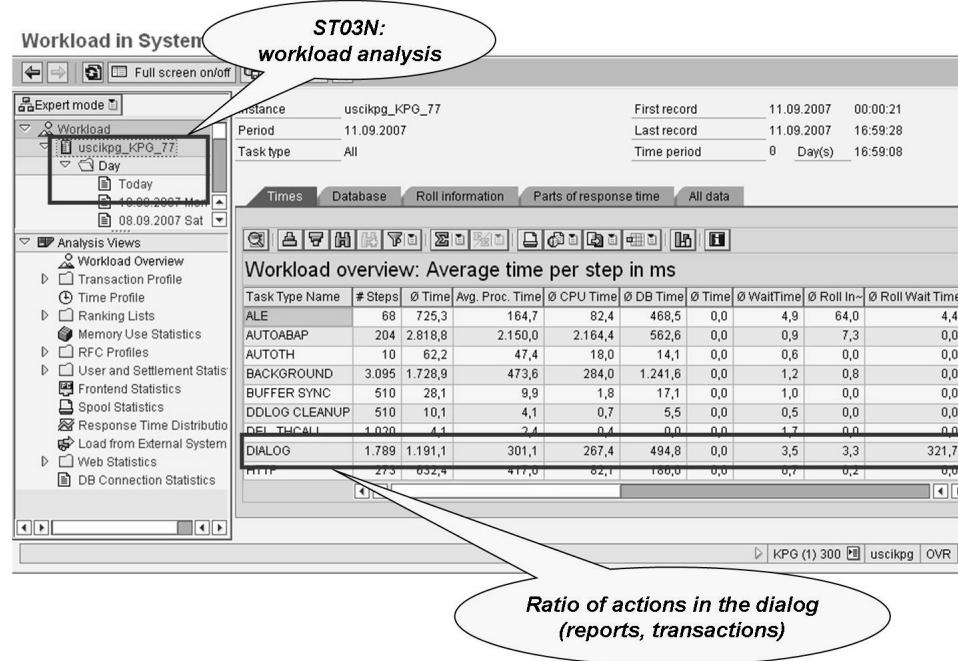
Details are logged in the system for each dialog or transaction step. You can view these “statistics records” using transaction **STAD** or **STAT** (for releases prior to 4.6). Note the numerous filter options for entry. In our case it makes sense to filter the search for the user and time, making it easier to observe and analyze particular user-specific records.



**Figure 10: Transaction Step Analysis (2)**

If you double-click a particular record, the system displays the components of the transaction step (DB time, CPU time, load time, etc.). From here you can also do another analysis of DB and RFC accesses (if any). Note the buttons in the toolbar of this transaction.

To find out which programs and transactions generally account for the greatest load on the system, you can use the **workload analysis**. Oftentimes an administrator will notice a performance problem but not be able to categorize it immediately. This transaction is helpful in analyzing performance problems in greater detail.



**Figure 11: ST03 - Workload Analysis (1)**

In workload analysis (ST03N or ST03 depending on the release) it is possible to analyze all requests that occurred in a certain timeframe. You can display the average response time for the current day for all “task types” or an analysis of all relevant times per week or month. Transaction ST03 regards task types as actions subdivided into categories such as dialog, batch, RFC, etc. Be aware of the ratio of the response time to the other times such as CPU or DB time: the rule of thumb here is that the two times should not exceed roughly 40% of the response time.

The average database response time covering all actions should not exceed 600 ms. Such a value would suggest a fundamental system or database problem (the network transmission time is always taken into account in the database request time).

→ **Note:** For details on recommendations for analyzing the response time components, see OSS Notes: 8963, 364625, 203924

You can branch to the detail analysis in ST03 by double-clicking the “transaction profile”.

The ability to sort the list according to different criteria is a stupendous advantage. You can sort the list by steps per transaction, total response time and average response time.

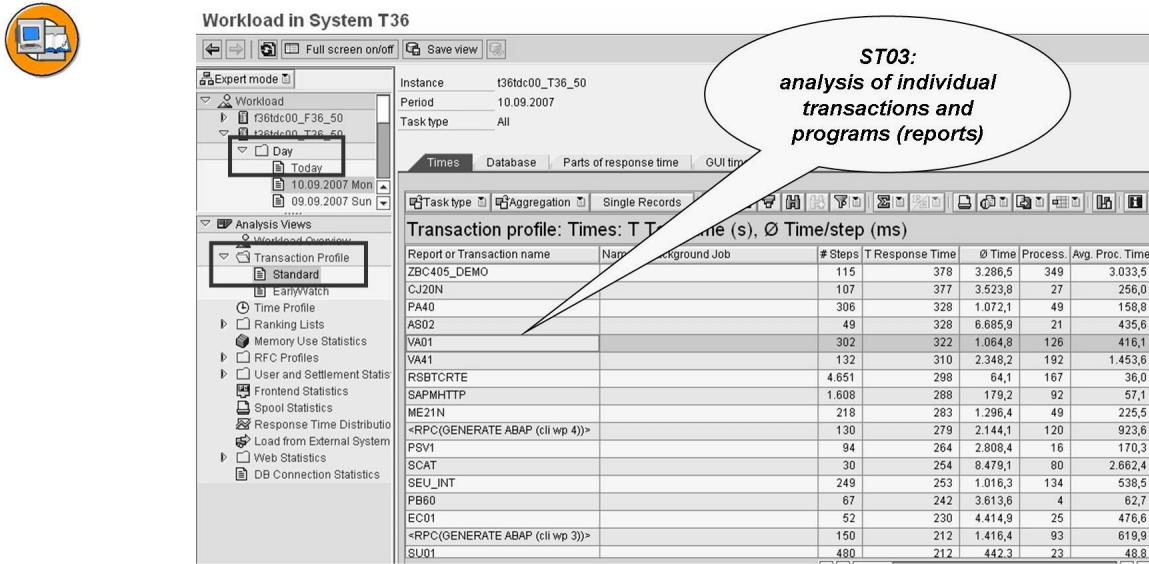


Figure 12: Workload Analysis (2)

To analyze a dialog transaction, sort the list by average response time in descending order and start your analysis at the top. Multiple transaction steps are usually performed during the execution of a transaction. The average response times per transaction step are usually higher for background transactions than for transactions running in dialog processing mode. A single transaction step may correspond to a program started as a background job. Pay special attention to reports that run parallel to work being done in dialog mode. The critical time threshold for a program running as a background job depends on the type of usage made of the SAP system.

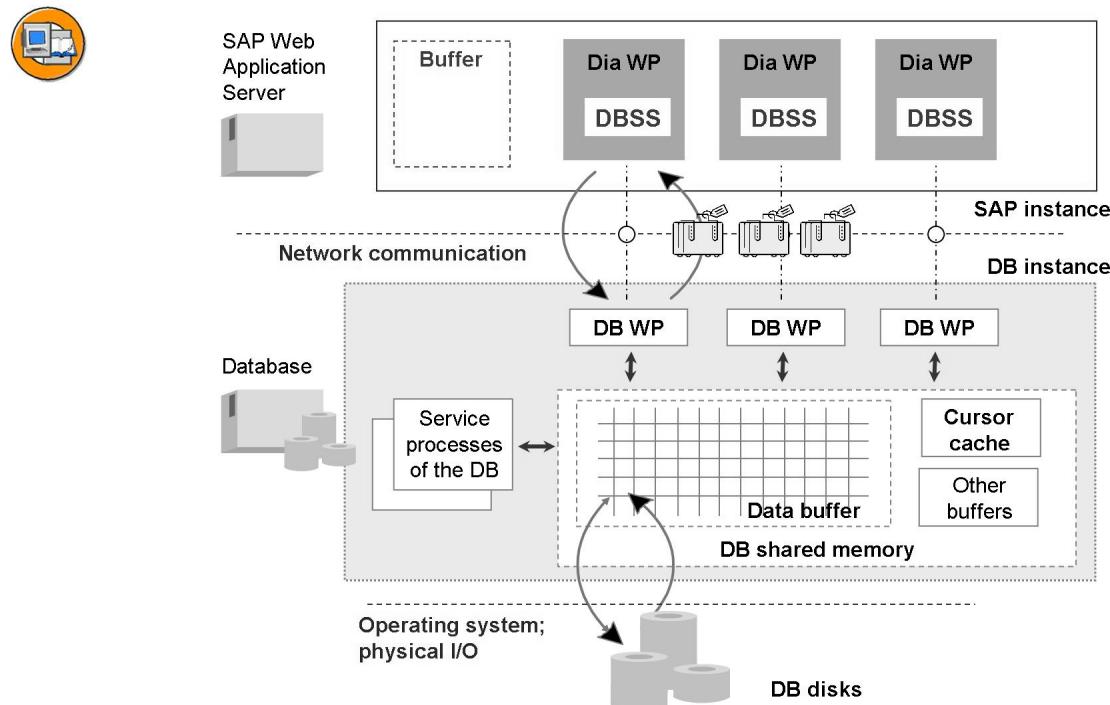


**Hint:** To find out which transactions and programs are “expensive” for the system, sort the list in descending order according to criteria such as total execution time or database access time; also compare the average response times with the database and CPU times. Anomalies stick out like a sore thumb.



Figure 13: Architecture and Technical Overview (2)

## The Database System



**Figure 14: Database architecture**

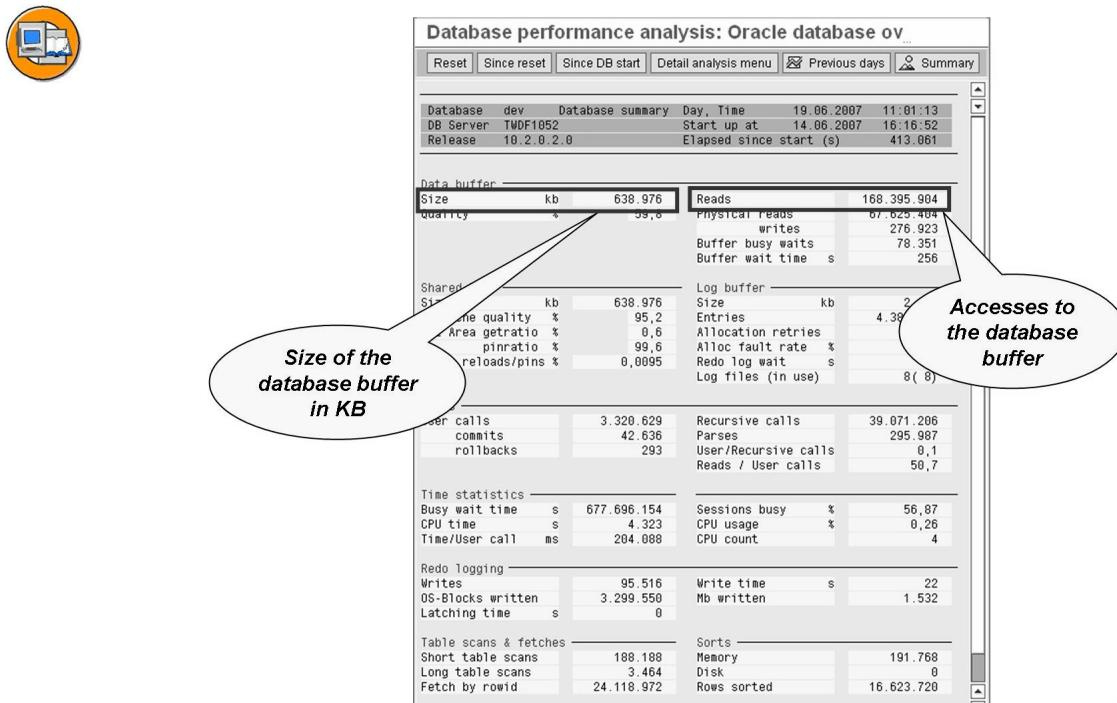
The DBMS (database management system) provides the database work process services that are used by the SAP work processes. Exactly one database work process is assigned to each SAP work process. There are different database processes for different purposes (for example, starting communications, changing database tables, locking mechanism, and archiving).

The database allocates an area of the main memory, which contains the data buffer, the DB SQL cache, the redo log information, and so on. The database files are on the hard disk and are usually administered by a file system.

You can measure the performance of a database server by monitoring the **physical I/O** (read and write access to database files), **memory consumption**, **CPU consumption**, and **network communication**. The SAP system provides various measurement tools for this purpose. The performance of an optimally configured SAP system is determined largely by the physical I/O operations.

Data is usually transferred between the SAP application server and the database server in blocks of exactly 32 KB (see packages/suitcases in the graphic.) A key principle of ABAP programming is to **minimize the number of these packages**. This saves network and database resources!

The following graphic shows the classical database performance monitor. If you use a very high SAP release (>=SAP NW 7.0) you can obtain this view using ST04OLD. (in Oracle).

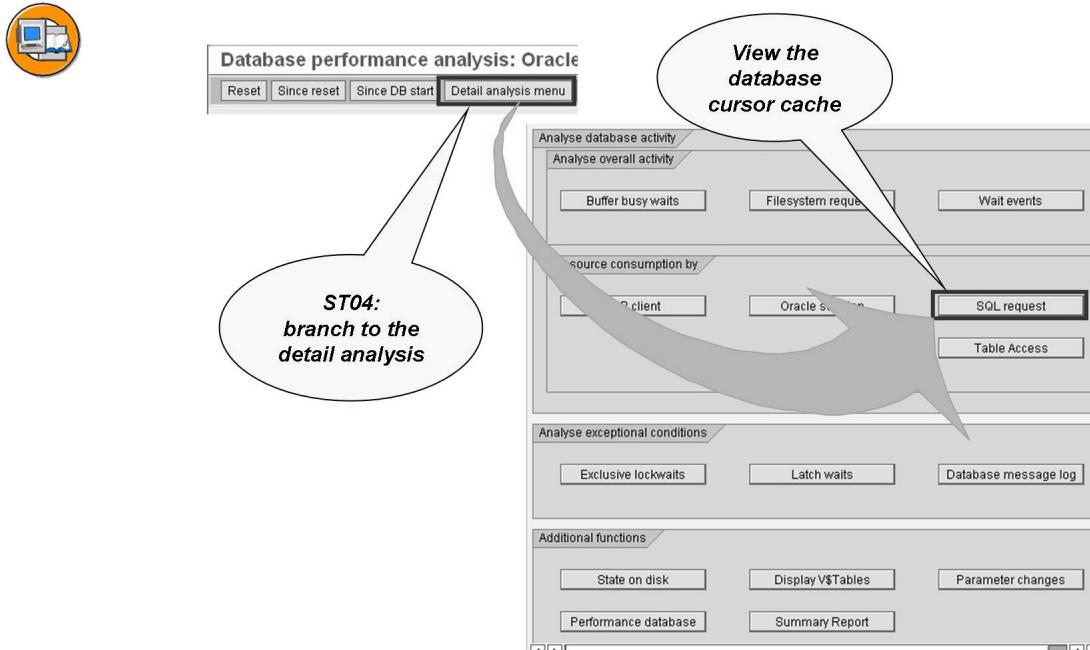


**Figure 15: The Database Performance Monitor - ST04 or ST04OLD Since SAP NW 7.0 SP12**

The initial screen of the database monitor shows an overview of important parameters:

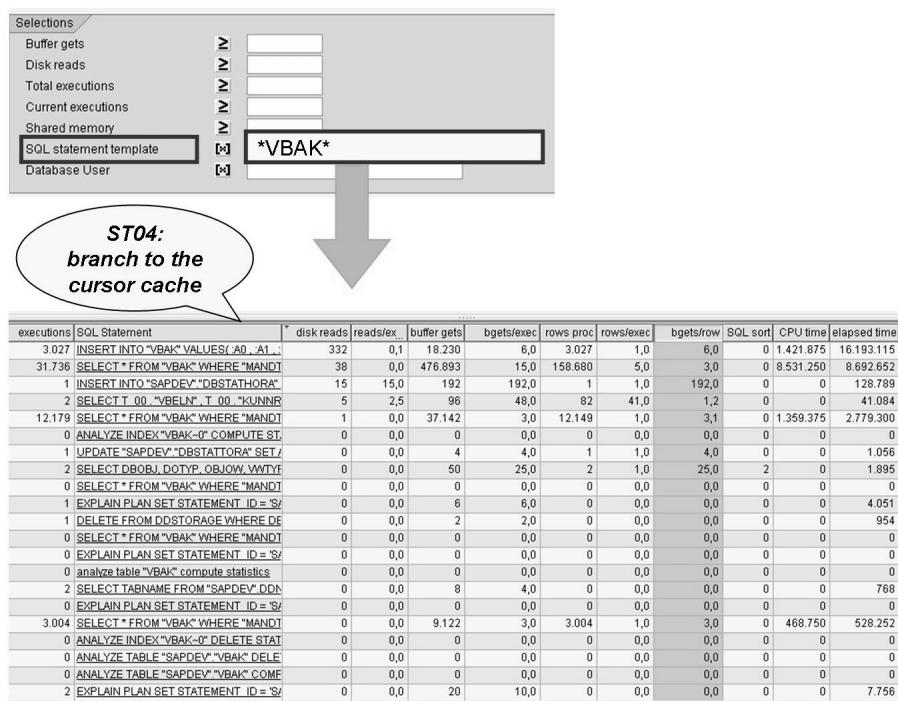
- Reads:  
Data blocks logically read from the database buffer
- Physical reads:  
Data blocks read physically from memory systems (hard disks)
- Data buffer size:  
Size of the data buffer of the database
- User calls:  
Number of SQL statements sent to the database
- Reads / User calls:  
Read accesses to the buffer per SQL statement

The ratio Reads / User calls is an indicator for whether it would be worthwhile to perform a database SQL cache (cursor cache) analysis. If the ratio is larger than 20:1, an analysis is urgently required. The SAP system and the database must have been running for several days prior to performing a database SQL cache analysis. Only then will the cache have salient data that accurately reflects practice. Later in the course other interesting key figures for this monitor will be introduced.



**Figure 16: Navigation to the database cursor cache**

The graphic above shows how to navigate to the cursor cache of an Oracle database. Note that most of the monitors in the basis environment are database-dependent.

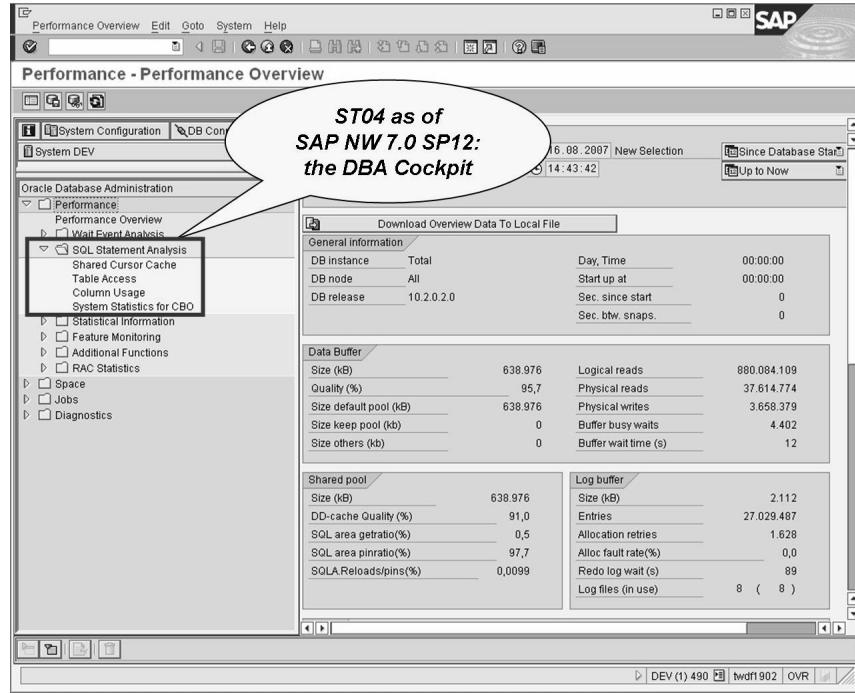


**Figure 17: The Database Cursor Cache**

The graphic shows how to navigate to the cursor cache using transaction ST04 (called ST04OLD as of SAP NW 70 SP12).

The database cursor cache provides an overview of all SQL statements that were executed by the database in a given time period. Note the filter options when entering the application. In the example, all commands sent to table VBAK are selected in the cursor cache. It is best to work with an “\*” before and after the name of the table you are searching for; otherwise you would have to enter the entire correct wording of the native SQL statement.

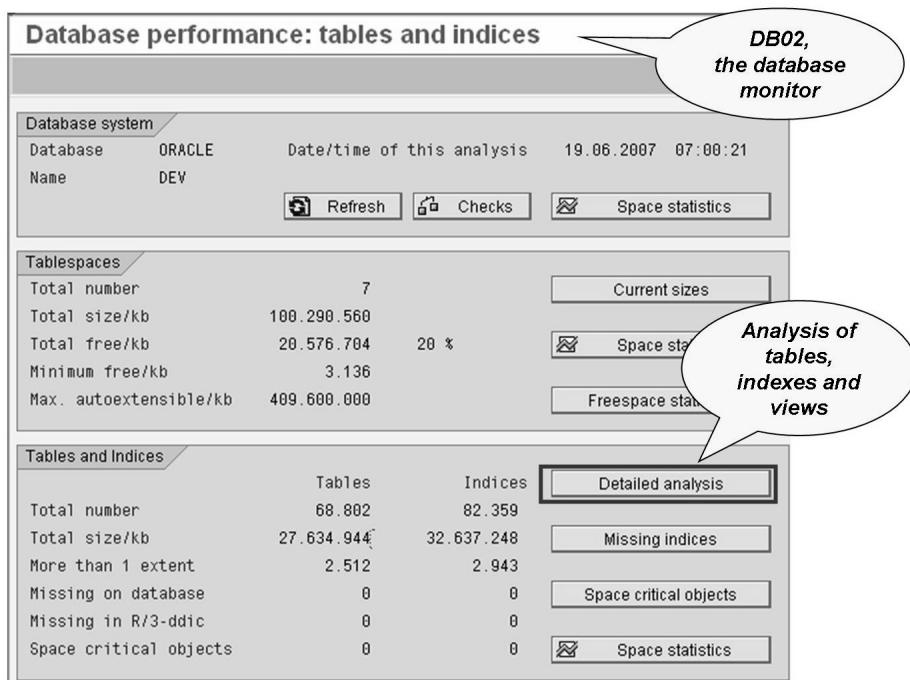
Sorting this list makes it possible to filter out the commands which are expensive and burdensome for the database. Additional threshold values for this list will be addressed later in the course.



**Figure 18: ST04 - the DBA Cockpit as of SAP NW 7.0 (SP12)**

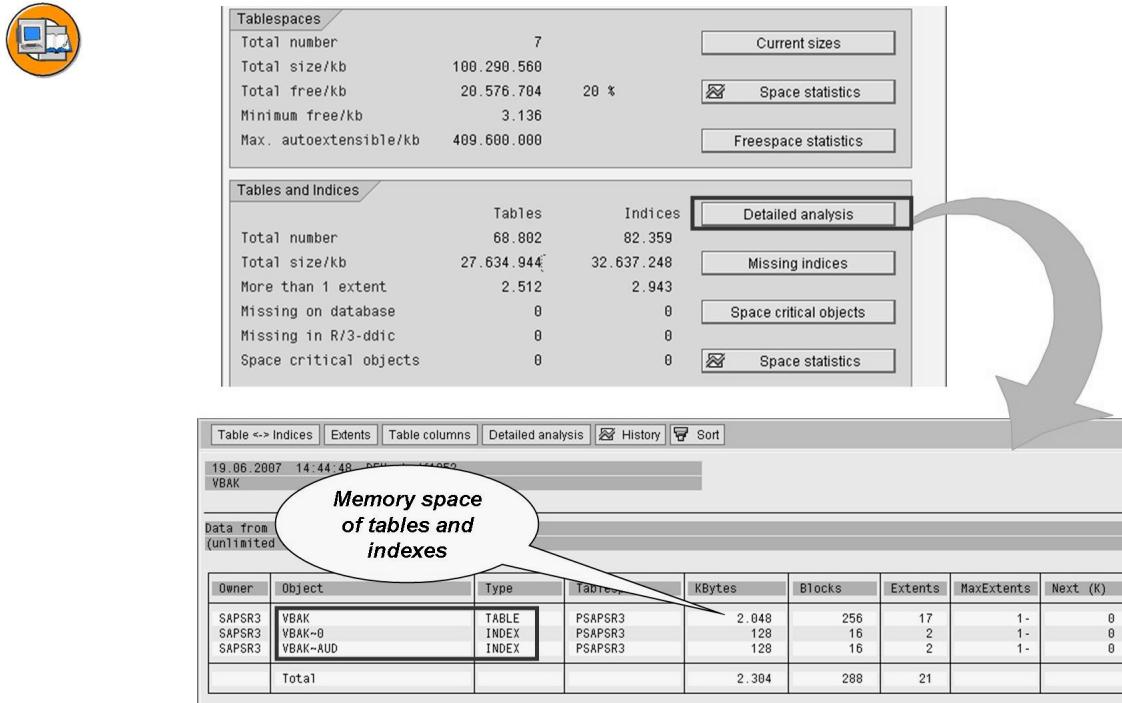
The initial screen used for transaction ST04 depends on the release status of your system. The new DBA Cockpit will be activated with SAP NW 7.0 with SP12. The basic functionality remains the same; what is new is the navigation tree at left, which allows you to go directly to various components, such as the cursor cache. The INFO icon at top left on the screen describes the functions of the cockpit.

→ **Note:** If you have SAP NW 7.0 SP12 and want to work with the classical transactions, you can call ST03OLD, ST04OLD or DB02OLD.



**Figure 19: Database Monitor DB02 (DB02OLD as of SAP NW 7.0 SP12)**

The database monitor provides information about the memory consumption of tables, indexes and views. Inconsistencies between ABAP dictionary and database, missing indexes or fragmentations can be analyzed with the monitor.



**Figure 20: Organization and Memory Consumption of Tables**

If you also want to display all indexes for a table, click the “Tables -> Indices” button in the toolbar. The “Table Columns” button displays the columns of the table with their “distinct values” (i.e. the different field values or entries for a field).

With the “detailed analysis” button you see the exact structure of indexes (provided that the desired index is selected).

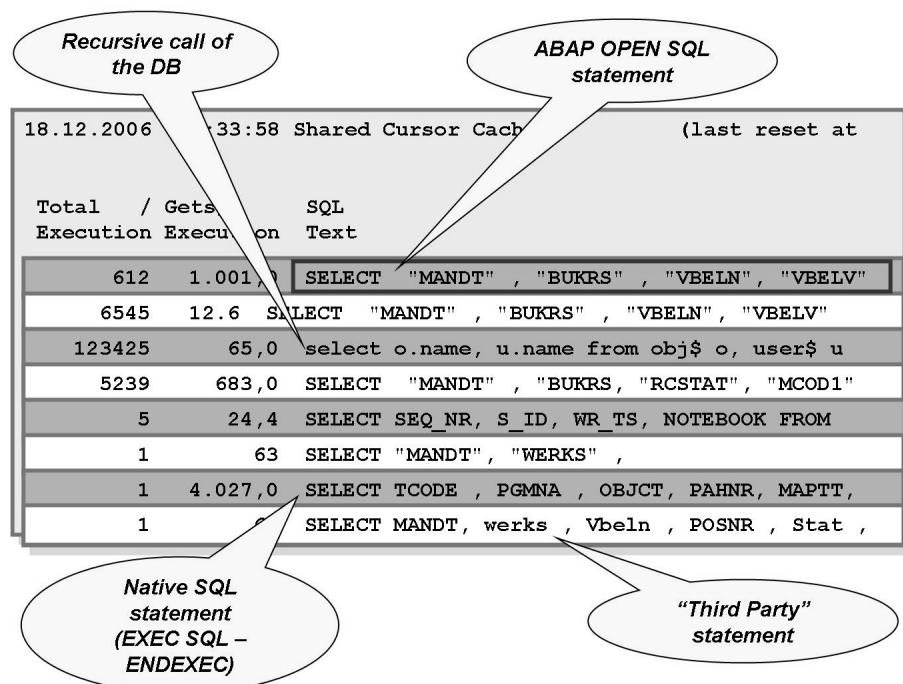


Figure 21: Various types of statements in cursor cache (ST04)

Not all statements in the cursor cache were sent by ABAP programs. ABAP statements can be recognized by the upper-case text and names of variable appearing in inverted commas. All statements originating from OPEN SQL cannot be optimized using the procedure described in this course.

These include statements from SAP tools or non-SAP software. Native SQL statements or recursive from the database itself can be recognized by the fact that their variable names are not in inverted commas or are in lowercase letters as part of the text.

# Exercise 1: Architecture and System Overview

## Exercise Objectives

After completing this exercise, you will be able to:

- Describe the architecture of the SAP training system
- Name important transactions for describing the structure of the system
- Name and operate transactions for workload and database analysis
- Navigate to the database cursor cache

## Business Example

A company which develops its own programs would like to assess the system load generated by those programs.

### Task 1: Architecture and System Overview

This exercise involves collecting information about the training system.

1. What is the system name (SID) of the present training system, what database is used and what operating system does the database run on?
2. How many instances does the system have? What instance are you logged on to?
3. How many work processes of the dialog type are there on your instance? Which dialog work processes show the greatest CPU time? (the CPU time allocated to the WP)

### Task 2: Workload Analysis

The object of this exercise is to identify programs and transactions that place the greatest strain on the system.

1. What is the average response time for dialog work processes (D-WPs) today?
2. Compare this time with the average DB and CPU times. How do you assess the results?
3. Which programs and transactions that ran today used the most resources in terms of total runtime? Which ones show the highest database access times? Are any user-defined “Z programs” among the ten most expensive?

*Continued on next page*

4. In the transaction profile, select a user-defined program (Z program) and re-analyze the response time for it and its components. What do you notice?

### Task 3: Database Performance Monitor

The object of this task is to analyze details about the database, such as the cursor cache.

1. How large is the database buffer of your database (database cache)? How many buffer and disk accesses have occurred since the database was started?

### Task 4:

The object of the following task is to write your own program and then identify it in the cursor cache.

1. Write a new program named **Z##DB\_ACCESS** that accesses the table of your choice. Do not read too much data. If the table is a very big one, restrict the volume using a suitable WHERE clause or limit the amount of data using an UP TO 10000 ROWS.
2. Start your program and then try to identify it in the cursor cache. Trace the program from the cursor cache back to the calling ABAP program.

# Solution 1: Architecture and System Overview

## Task 1: Architecture and System Overview

This exercise involves collecting information about the training system.

1. What is the system name (SID) of the present training system, what database is used and what operating system does the database run on?
  - a) The system ID can be found in the status bar. You can also navigate to *System -> Status*. At bottom left you will find information about the application server you are logged on to and the operating system you are using; adjacent to that is information on the database.
2. How many instances does the system have? What instance are you logged on to?
  - a) SM51 lists the instances in the system (that is, the active ones! Instances that have been closed are not displayed).

The instance name is comprised of the computer name (host), the system ID (the DB name) and the two digit instance number. The system status tells you which instance you are currently logged on to (in the Servername field).
3. How many work processes of the dialog type are there on your instance? Which dialog work processes show the greatest CPU time? (the CPU time allocated to the WP)
  - a) Navigate to SM50 (work process overview). It displays all work processes on your instance. The dialog work processes (D-WP) are indicated by the identifier DIA. If you click the CPU icon (clock icon), the system displays the CPU time allocated to the work processes. You will notice that the time decreases as you go down. The dispatcher distributes the user requests from top to bottom. The D-WP with number 0 is therefore always the first to receive work from the dispatcher. Your request will thus be distributed across the work processes from top to bottom in the list in SM50. If a D-WP is currently occupied, the request is redirected. The requests of a particular user are thus not necessarily tied to a particular D-WP. This behavior is known in SAP architecture lingo as work process multiplexing.

*Continued on next page*

## Task 2: Workload Analysis

The object of this exercise is to identify programs and transactions that place the greatest strain on the system.

1. What is the average response time for dialog work processes (D-WPs) today?
  - a) Navigate to the workload monitor (ST03). By clicking *Workload* in the tree display at left, you can select your instance and the current day.  
The average response time in ms is displayed on the right half of the screen under *Task type - dialog*.
2. Compare this time with the average DB and CPU times. How do you assess the results?
  - a) You will find the times you are searching for to the right of the average response time. The rule of thumb says both should be under 40% of the response time.
3. Which programs and transactions that ran today used the most resources in terms of total runtime? Which ones show the highest database access times? Are any user-defined “Z programs” among the ten most expensive?
  - a) In ST03 under *Analysis View*, switch to *Transaction Profile - Standard*. All executed programs and transactions are displayed on the right side of the screen. You can select columns and sort them according to various criteria.
4. In the transaction profile, select a user-defined program (Z program) and re-analyze the response time for it and its components. What do you notice?
  - a) The DB and CPU times should be under 40% here too. If the database portion is substantially above 40%, you should take a closer look at the code for database accesses. This will be addressed later in the course.

## Task 3: Database Performance Monitor

The object of this task is to analyze details about the database, such as the cursor cache.

1. How large is the database buffer of your database (database cache)? How many buffer and disk accesses have occurred since the database was started?
  - a) Navigate to ST04 (or ST04OLD since SAP NW 7.0 SP12) There, under *Overview*, you will find information about Logical Reads and Physical Reads. You can get more information about these fields via the F1 key.

*Continued on next page*

## Task 4:

The object of the following task is to write your own program and then identify it in the cursor cache.

1. Write a new program named **Z\_##\_DB\_ACCESS** that accesses the table of your choice. Do not read too much data. If the table is a very big one, restrict the volume using a suitable WHERE clause or limit the amount of data using an UP TO 10000 ROWS.
  - a) Proceed as usual in programming selects. You can use the instructor's demo programs for orientation.
2. Start your program and then try to identify it in the cursor cache. Trace the program from the cursor cache back to the calling ABAP program.

- a) You navigate to the database cursor cache via ST04 and *SQL Statement Analysis -> Shared Cursor Cache*.

In releases prior to 7.0 , you navigate via *ST04 -> Detail Analysis -> SQL Requests*. You can also use this classical DB Performance Monitor interface in releases under 7.0 via transaction code ST04OLD.

Then you have to select the time period you wish to display. Also filter the tables in the *SQL commands* field with \*<table>\*. In the button or toolbar there is a function for going to the calling ABAP program.



## Lesson Summary

You should now be able to:

- Describe the architecture of SAP NW AS ABAP
- Name important monitoring tools for the purposes of administration and performance analysis



## Unit Summary

You should now be able to:

- Describe the architecture of SAP NW AS ABAP
- Name important monitoring tools for the purposes of administration and performance analysis



# Unit 2

## Analysis Tools

### Unit Overview

#### Contents:



- Analyzing transaction steps
- SQL performance analysis
- ABAP runtime analysis
- Code Inspector
- System trace
- ABAP debugger



### Unit Objectives

After completing this unit, you will be able to:

- Determine whether a program causes poor performance due to a database or CPU problem
- Analyze database problems using the SQL trace
- Analyze performance problems with the Code Inspector
- Analyze performance problems with the system trace
- Identify and assess performance problems using runtime analysis

### Unit Contents

Lesson: Analysis Tools .....	34
Exercise 2: SQL Trace.....	81
Exercise 3: Runtime Analysis.....	85
Exercise 4: Code Inspector .....	91

# Lesson: Analysis Tools

## Lesson Overview

### Contents:



- Analyzing transaction steps
- SQL performance analysis
- ABAP runtime analysis
- Code Inspector
- System trace
- ABAP debugger



## Lesson Objectives

After completing this lesson, you will be able to:

- Determine whether a program causes poor performance due to a database or CPU problem
- Analyze database problems using the SQL trace
- Analyze performance problems with the Code Inspector
- Analyze performance problems with the system trace
- Identify and assess performance problems using runtime analysis

## Business Example

An international company that uses SAP software wants to enhance the SAP standard with programs of its own devising. Mr. Jones is charged with developing an ABAP application. He knows that the database accesses are an important factor in the runtime of a program. He uses the SQL trace to analyze his program's database accesses and identifies potential for optimization. This results in significantly improved runtime performance for his programs. He also wants to use runtime analysis to find out which internal tables in his program are most frequented and use the most memory.

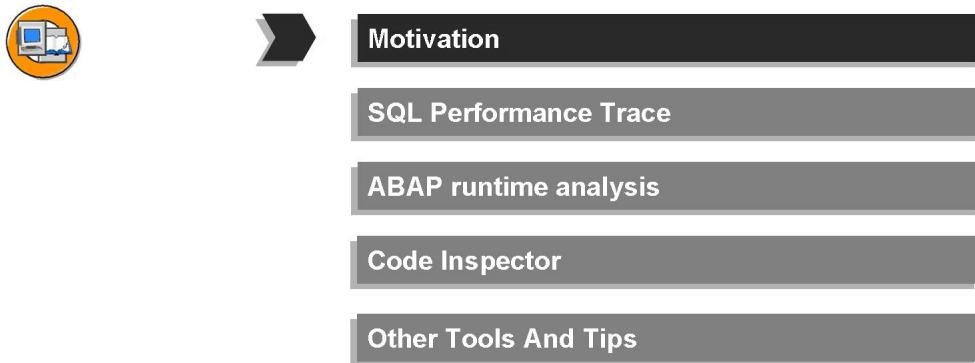


Figure 22: Analysis Tools (1)

### Explanation: Which analysis tool is used when?

In the following text, the term “single object” means a program (report), a transaction or a concatenation of reports or transactions. They implement a business process or parts of a business process.

Single-object analysis can be performed proactively in the quality assurance system for non-production objects, or reactively in the production system for production objects.

The analysis of individual objects may be performed by the developer of the object or by the performance analyst. The performance analyst must be familiar with ABAP and the SAP Basis.

The single object is known. The analysis of individual objects is performed proactively in quality assurance testing before the object is used in production, and reactively typically after an SAP system analysis.

A prerequisite for performing the analysis of individual objects is that there are no fundamental problems with the Basis. You can use the SAP standard services GoingLive and EarlyWatch to confirm the integrity of the Basis.

To analyze individual objects, representative business data must be available in the quality assurance system. In the production system, individual objects should be analyzed at times of representative user activity.

Contact the person responsible for the business process or the programmer if you have business-oriented or technical questions.



*A program shows anomalous behavior in the workload analysis.*

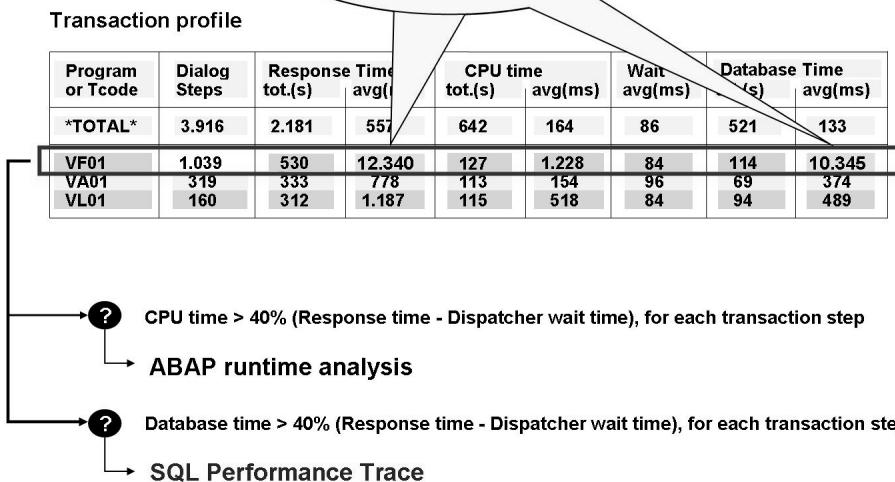


Figure 23: Analysis Tools - Which Tool for Which Purpose?

The graphic shows a single object in the workload with a poor database access time. Further analysis with the SQL Trace tool would be a good idea here. Ultimately it is always within the discretion of the analyst to choose the most appropriate tool for the present case.

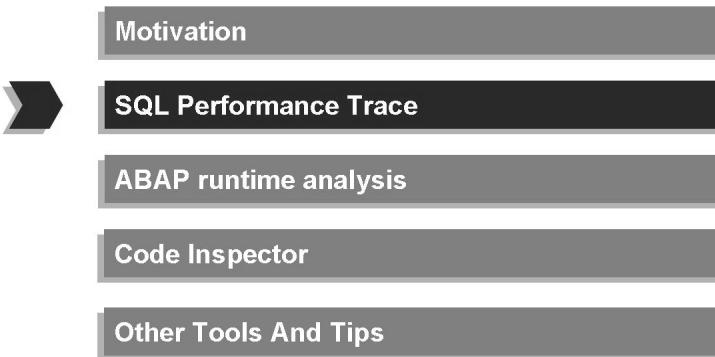
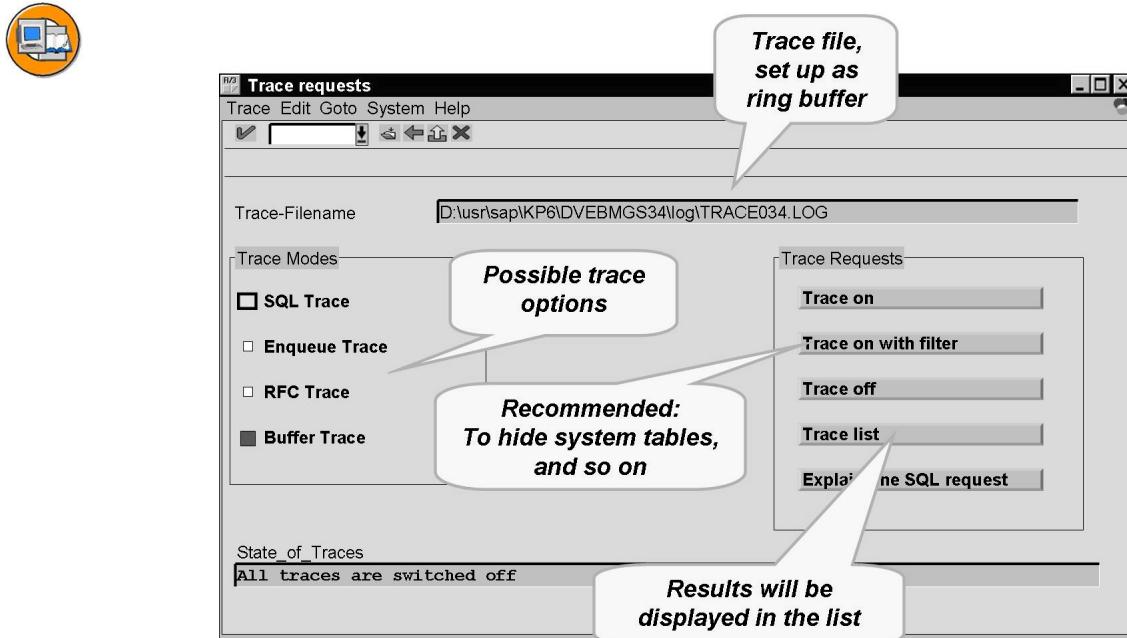


Figure 24: Analysis Tools (2)

## SQL Performance Trace

To start the SQL performance trace (or simply “SQL Trace”), enter transaction code ST05 or choose *System → Utilities (H) → Performance Trace*.



**Figure 25: SQL Trace - Initial Screen and Overview**

To obtain an overview of the behavior of the SAP system, you can record various processes during the execution of an individual object. To analyze performance problems with database accesses, activate the SQL trace and the buffer trace in transaction ST05.

Accesses to buffered tables are displayed in blue and those in the database in yellow. That makes it easy to distinguish accesses in the results list.

The SQL trace records SQL statements. There may be considerable differences between the SQL statement formed on the ABAP level and the SQL statement that is received by the database.

The buffer trace lets you trace SQL statements that access buffered tables. SQL statements that access buffered tables can cause buffer reloads, which are indicated in a buffer trace. An SQL statement accessing a buffered table can result in a completely different SQL statement if it cannot be satisfied by the SAP table buffer.

Additional tools include the enqueue trace, which records requests or releases of SAP locks, indicating the enqueue key and the objects involved; as well as the RFC trace, which records the RFC calls received or sent. We will go into this in detail at a later stage.

You can activate the trace in a production system as well, because it does not cause any errors or inconsistencies.

Before you activate the SQL trace for a program or transaction, you should execute that program or transaction once first, to avoid recording the refresh operations for the table buffer, program buffer, and so on.



**Note:** You can restrict the SQL performance trace to any user and any single object. You should make sure, however, that the user being recorded is not working in any other sessions and that no other processing forms (batch processing, update tasks) are running in parallel under the same user on the same application server. If there are, the SQL performance trace will be unreadable.



**Hint:** The tracing itself needs runtime and causes slight delays in the measuring operation.

Each application server only has one active trace file. As a result, the trace can only be activated for **one user on each application server**. The standard size of the trace file is 800K. We recommend increasing the size of the trace file to 16 MB (release 4.6: parameter rstr/max\_filesize\_MB = 16 384 000).

In higher releases than 4.6, the valid parameters are rstr/filename, rstr/max\_files and rstr/max\_filesize\_MB. Up to 99 trace files of up to 100 MB apiece can be generated there.

The SQL trace measures the time between the sending of the request to the database server and reception of the results by the database interface (DBSS). The data written to the trace file includes the database time, a time stamp, the number of transferred records, the database return code, the text of the SQL statement, and additional administrative information for each database operation.



**Basic SQL List - Sorted by PID**

Trace Edit Goto System Help

DDIC info Explain SQL ABAP display Extended

Transaction = PID = Client = DIA Client = 100 User

Duration	Objectname	Oper	Rec	RC	Statement
41	VBAP	REOPEN	0	0	SELECT WHERE "MANDT" = '100' AND "VBEL"
805	VBAP	FETCH	5	1403	1 data package (max. 32 KB) filled with 5 records
41	VBAP	REOPEN	0	0	SELECT WHERE "MANDT" = '100' AND "VBEL"
785	VBAP	FETCH	5	1403	SELECT WHERE "MANDT" = '100' AND "VBEL"
41	VBAP	REOPEN	0	0	SELECT WHERE "MANDT" = '100' AND "VBEL"
788	VVBAP	FETCH	5	1403	SELECT WHERE "MANDT" = '100' AND "VBEL"
40	VBAK	REOPEN	0	0	SELECT WHERE "MANDT" = '100' AND "VBEL"
1.061	VBAK	FETCH	10	1403	SELECT WHERE "MANDT" = :AO AND "VBELN"
1.327	VBAP	PREPARE	0	0	SELECT WHERE "MANDT" = :AO AND "VBELN"
85	VBAP	OPEN	0	0	SELECT WHERE "MANDT" = '100' AND "VBEL"
6.919	VBAP	FETCH	49	1403	SELECT WHERE "MANDT" = :AO AND "VBELN"
826	VBAK_VBAP	PREPARE	0	0	SELECT WHERE "MANDT" = :AO AND "VBELN"
41	VBAK_VBAP	OPEN	0	0	SELECT WHERE "MANDT" = '100' AND "VBEL"
5.924	VBAK_VBAP	FETCH	49	1403	SELECT WHERE ( T_01 . "MANDT" = :AO AND "VBELN"
800	VBAK	PREPARE	0	0	SELECT WHERE ( T_01 . "MANDT" = '100' AND "VBELN"
46	VBAK	OPEN	0	0	SELECT WHERE ( T_01 . "MANDT" = '100' AND "VBELN"
5.704	VBAK	FETCH	49	1403	SELECT WHERE ( T_01 . "MANDT" = :AO AND "VBELN"
217	VBAK	REOPEN	0	0	SELECT WHERE "MANDT" = '100' AND "VBEL"

Figure 26: SQL Trace - Trace List

There are slight differences in the trace list between the different database systems. Data operations that take longer than 100 ms are highlighted in red. Each section in the trace list contains the user ID, transaction, SAP work process number, and client as header information. The columns in the trace list have the following meanings:

#### Duration

Duration of the database operation in microseconds

#### Obj. name (objectname)

Name of the table/view from the SQL statement

#### Op. (operation)

Database operation to perform

#### Recs. (Rec)

Number of records returned by the database operation

#### RC (returncode)

Return code of the database system statement

#### Statement (statement)

Text of the SQL statement

When an SQL read statement is executed, the following DBSS operations take place:

The **PREPARE** operation parses the text of the SQL statement and translates it to a statement that the database can process. The access strategy is also determined. Bind variables (placeholders) are used for the variable values in the database statement. The **OPEN / REOPEN** operation replaces these bind variables in the database statement with specific values and the access is prepared in the database (a cursor is opened). The actual transport of records from the database server to the application server takes place during the **FETCH** operation. The number of records transported by each **FETCH** operation is (partially) dependent on a profile parameter.

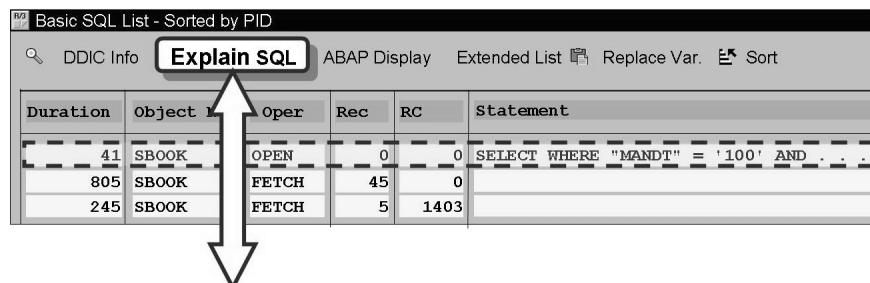
SQL write statements are processed similarly, with the exception that EXEC/REEXEC operations are performed instead of the OPEN/REOPEN and FETCH operations. The accesses (cursors) are buffered in the SAP cursor cache for each SAP work process, which makes it possible to skip the PREPARE operation when an SQL statement with an identical structure is executed.



**Hint:** The SQL statement in the ABAP code, which is normally formulated in OPEN SQL, may differ from the SQL statement processed by the database. The following list contains some ABAP statements that are converted by the SAP database interface:

- A WHERE clause is used for each SQL statement that accesses a client-dependent table (except for SELECT ... CLIENT SPECIFIED)
- *SELECT ... FOR ALL ENTRIES => multiple SELECT statements with corresponding OR clauses* (details later)
- *MODIFY → INSERT and/or UPDATE*
- Accessing pooled and cluster tables
- *EXPORT/IMPORT TO/FROM DATABASE → INSERT/SELECT on INDEX-like tables*
- *OPEN CURSOR/FETCH NEXT → No equivalent in the SQL trace / equivalent SELECT*

The *Explain SQL* function lets you analyze individual SQL statements database-specifically.



**"Explain SQL" returns the following information:**

- Display of the native SQL statement (*DB-specific*)
- Information about the optimizer strategy
- Information about the costs of the access (*DB-specific*)
- The index used by the statement
- Information about the table and its indexes
- Information about the optimizer statistics

**Figure 27: SQL Trace - EXPLAIN**

To reach the *Explain SQL* function, choose menu path *SQL Trace* → *Explain SQL* or press the *Explain SQL* button from within the trace list.

*Explain SQL* is performed for the SQL statement in the current cursor position in the trace list.



**Hint:** *Explain SQL* is only possible for the following database operations: PREPARE, OPEN, and REOPEN. If you positioned the cursor on a FETCH operation, the explain will not work.

Note that the use and functions of Explain SQL **vary widely between databases!**

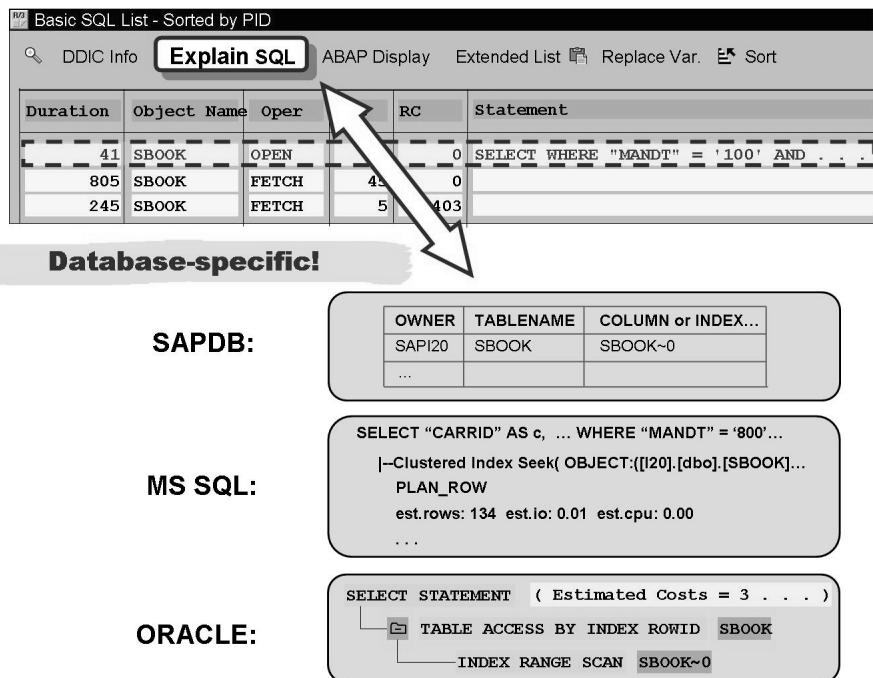
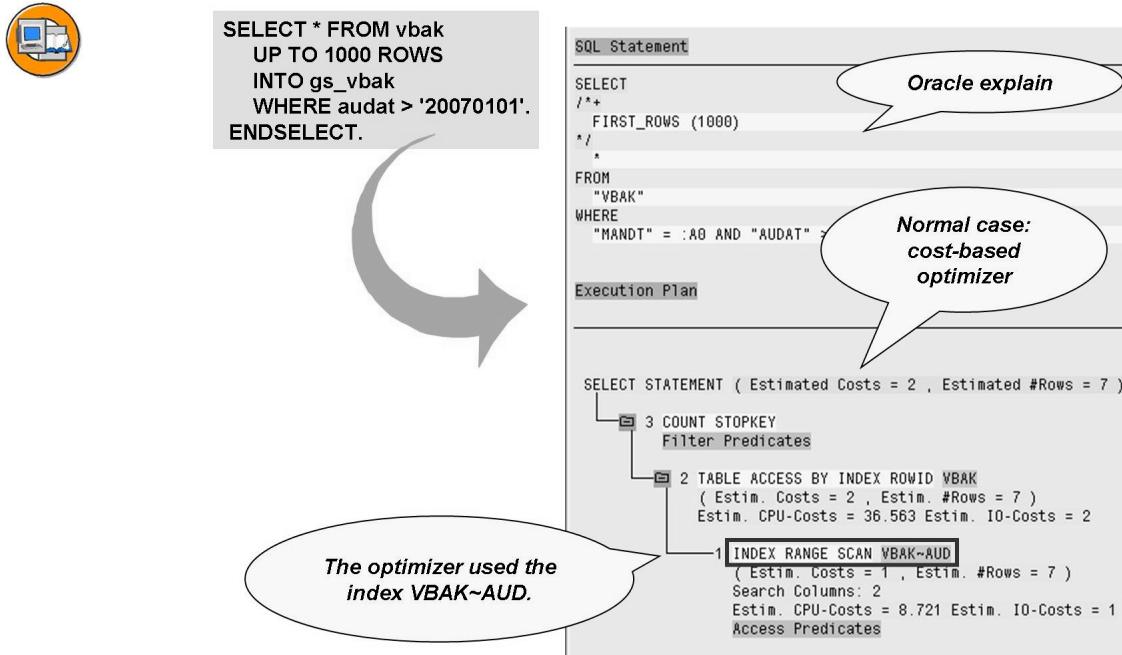


Figure 28: The Explain Is Database-Specific

Some database display the text of the SQL statement with placeholders (also called **bind variables**) in addition to the access strategy of the SQL statement. These variables are serve as placeholders in case the same command is executed again with different contents. The proper navigation (usually a double-click) will display the content of the bind variables.

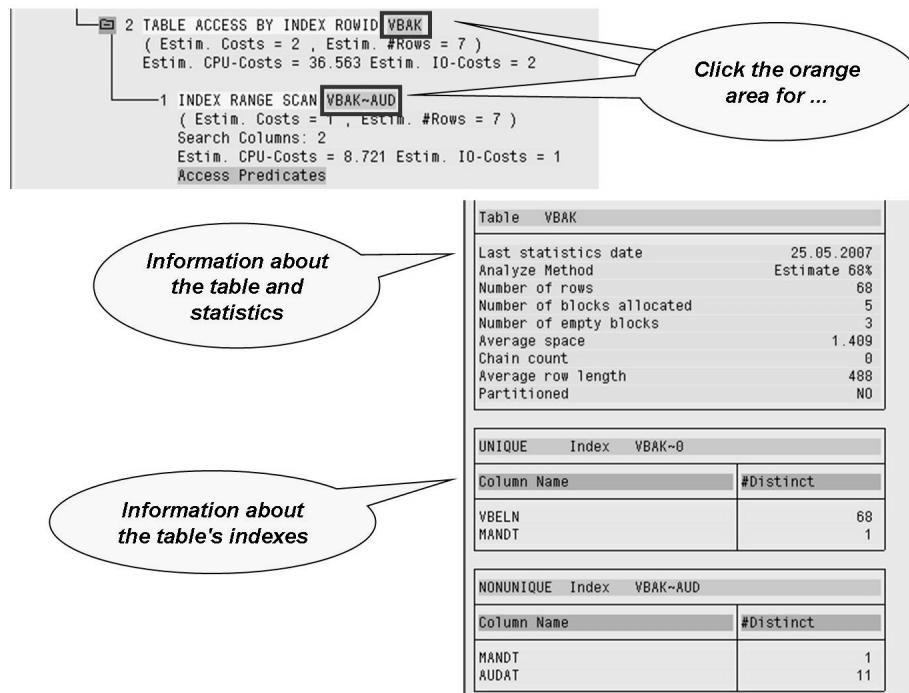
The most important function within the Explain is probably the display of the **access strategy** that the optimizer uses – that is, the utilized index. The diagram shows examples for three databases. When a suitable index is used, the search effort required to compile the result set for the SQL statement can be reduced significantly. Therefore, developers can verify whether the resulting command was processed by the database quickly.



**Figure 29: Example of an explain on the ORACLE database**

The upper part of the explain window displays the statement sent to the database, while the lower part displays the access strategy used by the optimizer with the calculated costs. The cost values are database-specific. The cost calculation includes the expected block read operations (the lower the costs, the lower the expense for the DB).

→ **Note:** The various database manufacturers offer courses covering the fundamental strategies of the optimizer and the basics of cost calculation. Detailed knowledge in this area can be helpful, but it is not necessary to write high-performance programs in the ABAP environment. Remember that there can be significant differences not only between the individual manufacturers, but also between different releases of the optimizer used.



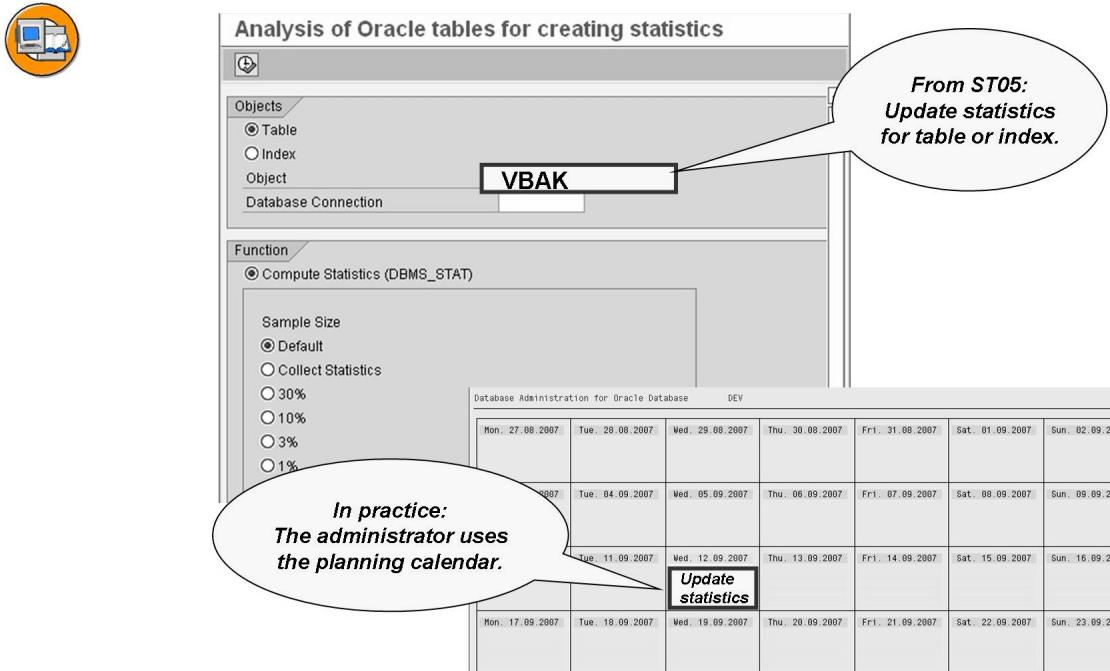
**Figure 30: Explain - Details on Tables and Indexes**

Details about the table and its indexes can be obtained by clicking the orange area. Note the date of the last statistics update in the upper part of the window. Statistics are essential to the correct and efficient functioning of any optimizer; old statistics can severely compromise performance.

In practice, batch jobs are scheduled (SM36/SM37) to run regularly and determine which statistics need to be updated. The administrator can call upon the functions of a scheduling calendar for backup measures, statistics updates, etc., in the prosecution of his daily duties (DB13).

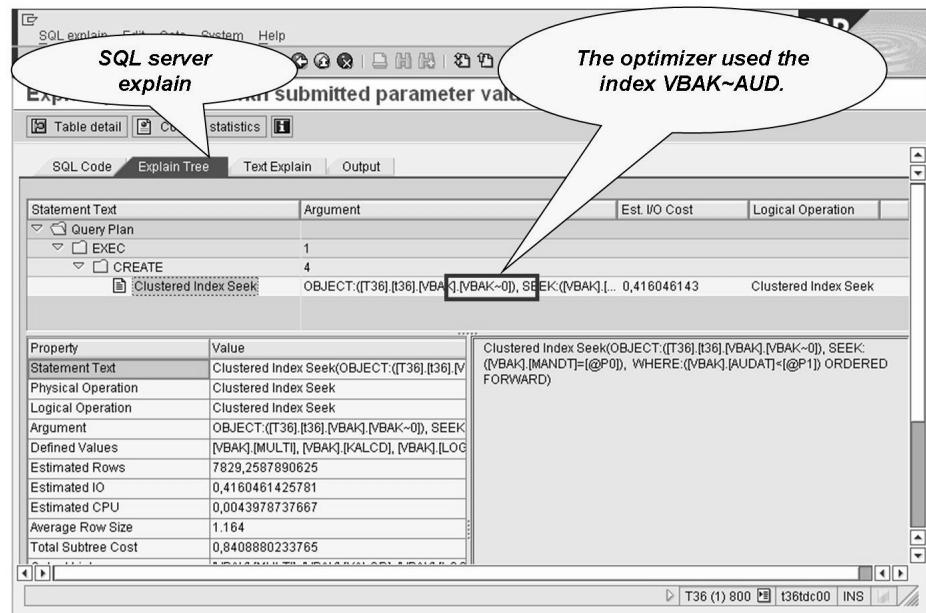
It is also possible to start a statistics update from the detail window of the explain. Press *Analyze* and, if in Oracle, branch to the RSANAORA program, from which the refresh can be started (note: you will find the button on the screen for the table and index details). As already noted, in practice the administration of statistics will not be done by the developer, nor will it be done from ST05. This is neither common nor a conventional practice, but rather an exceptional case. As a rule, the administrator will monitor and update the statistics with the aid of more professional tools.

Use the *Explain with hint* button to display the execution path of an SQL statement together with so-called "hints". Hints are used to force the database to choose a particular execution path. The available hints are explained later.



**Figure 31: Administration of statistics**

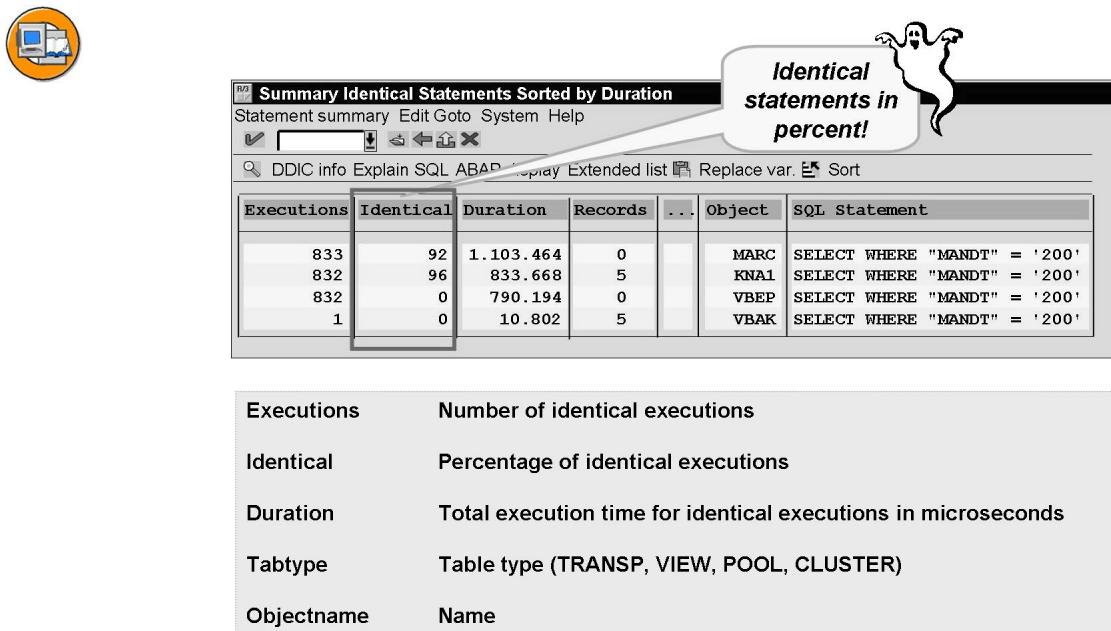
Don't forget that this subject is database-dependent. The program RSANAORA is available on Oracle. Other databases offer similar tools. The SAP Planning Calendar for the administrator is available for most SAP-supported databases. The course ADM100 imparts the basics of DB administration; additionally, specific, advanced administration courses are offered for each particular database.



**Figure 32: Example of an explain with the MS SQL server database**

Another example in the graphic shows an explain with a Microsoft SQL server database. This database also works with a cost-based optimizer. However, the cost factor is not at all comparable with the previous examples.

One major contrast with Oracle is the use of index-organized tables (IOTs). This means that data on SQL servers is sorted according to the primary index.



**Figure 33: SQL Trace - Sum of All Commands**

From within the trace list, choose menu path *Trace List → Summarize Trace by SQL Statement* to display an overview of how many **identically structured SQL statements** were performed for the various tables and/or views. “Identically structured” statements use the same access strategy, possibly with different values for the bind variables.

The list is sorted by total time required in microseconds.

It contains the following information for each SQL statement:

- How often was the statement executed?
- How many records were transported (total and average)?
- How much time was required for the access (total and average)?
- What percentage of executions were completely identical to a previous execution?

The share of identical accesses is particularly relevant for performance aspects, because identical accesses return identical results.

→ **Note:** Prior to SAP Web AS 6.10, select *Goto → Statement Summary* to reach this point.



### Avoid identical selects!



Executions	Object Name	Where Clause
4	VBAK	"MANDT" = '100' AND "VBELN" BETWEEN '000000000
2	VBAP	"MANDT" = '100' AND "VBELN" = '0000000001'
2	VBAP	"MANDT" = '100' AND "VBELN" = '0000000002'
2	VBAP	"MANDT" = '100' AND "VBELN" = '0000000003'
2	VBAP	"MANDT" = '100' AND "VBELN" = '0000000004'
2	VVBAP	"MANDT" = '100' AND "VBELN" = '0000000005'
2	VVBAP	"MANDT" = '100' AND "VBELN" = '0000000006'
2	VBAP	"MANDT" = '100' AND "VBELN" = '0000000007'
2	VBAP	"MANDT" = '100' AND "VBELN" = '0000000008'
2	VBAP	"MANDT" = '100' AND "VBELN" = '0000000009'
2	VBAP	"MANDT" = '100' AND "VBELN" = '0000000010'
2	VBAP	"MANDT" = '100' AND "VBELN" = '0000000001' OR

Figure 34: SQL Trace - Identical SELECTs

A more detailed analysis of **identical SQL statements** is available under menu path *Trace List → Display Identical Selects* from within the trace list. “Identical” statements use the same access strategy with the same values for the bind variables.

Because identical SQL statements return identical result sets, they can be avoided by buffering the results, which can significantly improve the performance of an application.

The better strategy is to avoid such identical selects.

The optimization potential for an SQL statement is derived from the number of identical executions. If you have four identical executions of an SQL statement accessing a database table, where the results are buffered in an internal table in the program, three of the executions are superfluous. The result is an optimization potential of 60-75% for this SQL statement.

→ **Note:** Prior to SAP Web AS 6.10, select *Goto → Identical Selects* to reach this point.

The basic list of the trace offers another useful function which returns a **simplified or aggregated display** of the records. You navigate to this function via the menu path *Trace List → Combined Table Accesses*. The list shows the SQL operation, the

number of affected data records, and the processing time in percent. The list is also useful for identifying loop constructs. It can be further aggregated. The *Aggregate* button returns useful, summary information on the most expensive table accesses.

 **Note:** Prior to *SAP Web AS 6.10*, select *Goto → Summary* to reach this point.

The following list contains rules of thumb and tips that can help you locate expensive or critical statements in the trace list. Mind that these categories do not necessarily imply a programming error; view them as a notice that it would be a good idea to have a closer look at the place in question.

- Take note of the following indicators in the trace list:
  - Database operation  $\geq 200\,000$  microseconds
  - SQL statement  $\geq 10$  fetches
  - Nested SQL statements of the same structure  $\geq 200\,000$  microseconds (for all nested SQL statements)
- Note the functions in the trace list of the aggregated display



You can see:

- Loop constructs
- Percentage of avoidable statements
- Benchmark for the most-accessed tables and views
- Identical selects



#### Motivation

#### SQL Performance Trace

#### ABAP runtime analysis

#### Code Inspector

#### Other Tools And Tips

**Figure 35: Analysis Tools (3)**

## ABAP Runtime Analysis

The ABAP runtime analysis enables you to analyze the elapsed run time for individual objects such as transactions, programs, or function modules, or subobjects such as modularization units or commands. An ABAP runtime analysis is especially useful for individual objects that are CPU-intensive (see Workload Analysis).

The SAP system parameter *abap/atrasizequota* specifies how much memory should be reserved in the file system to store the results data. The SAP system parameter *abap/atrappath* specifies where the results data are stored in the file system. In the field *Short Description*, enter a short text to identify your analysis.

To access the initial screen of the ABAP runtime analysis, in any SAP screen choose *System → Utilities (H) → Runtime Analysis → Execute*.



ABAP Runtime Analysis: Overview

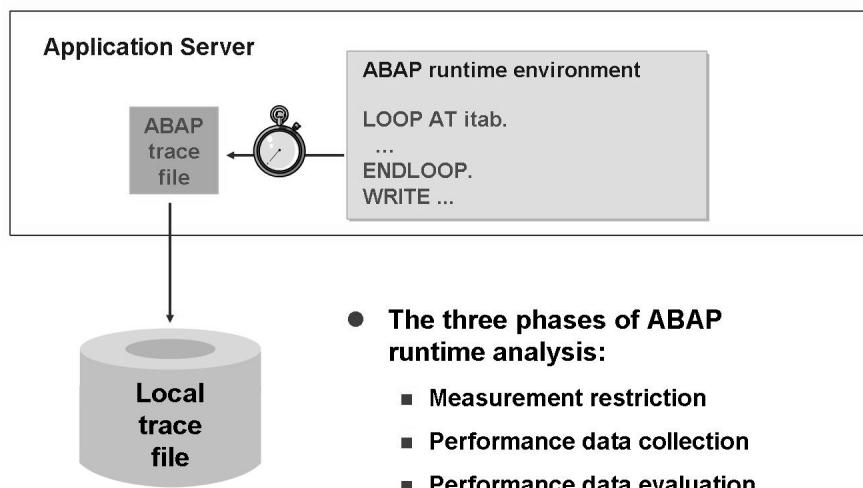
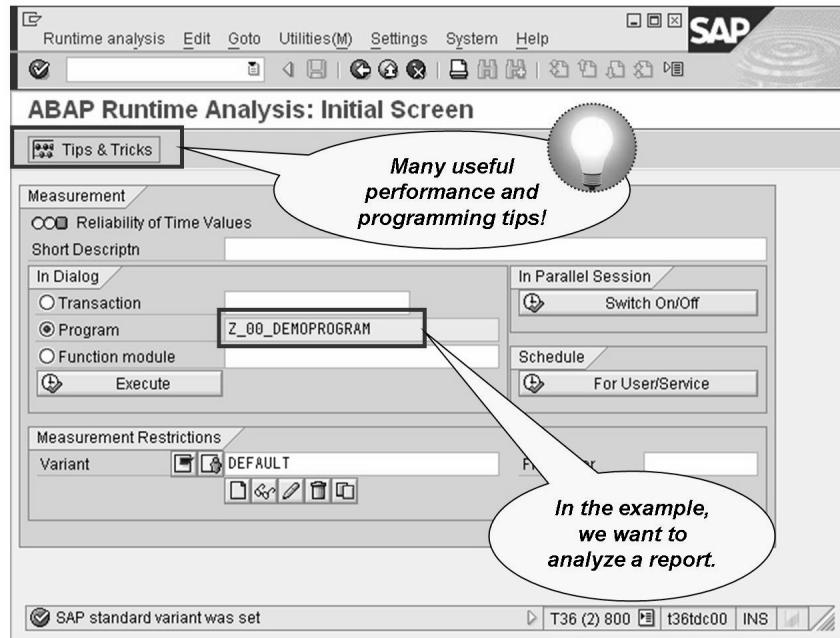


Figure 36: ABAP runtime analysis - overview

An ABAP runtime analysis has three phases:

- **Measurement restriction:** In this phase, you decide which object is to be analyzed. You also decide the type of aggregation that is used (full aggregation, aggregation per calling position, no aggregation) and whether to use filter options (for modularization units or commands).
- **Performance data collection:** You can execute performance data collection for single objects processed in the same internal mode as the ABAP runtime analysis. Or you can activate the ABAP runtime analysis for any given SAP work process and record the single object processed therein (possibly another external mode).
- **Performance data evaluation:** When using full aggregation, you analyze the data in the hit list, which provides a log record for each called modularization unit or instruction. For performance data evaluation in non-aggregated form or aggregated per calling position, there are further hit lists, such as table hit lists or a call hierarchy.



**Figure 37: Runtime analysis initial screen**

Under *In Dialog* in the ABAP runtime analysis initial screen, specify the object to be analyzed (a transaction, program, or function module). To execute the object, choose *Execute*. This causes the object to be run in the same user session as the runtime analysis. After the object has run, the runtime analysis results screen appears, and you can begin evaluating the performance data.

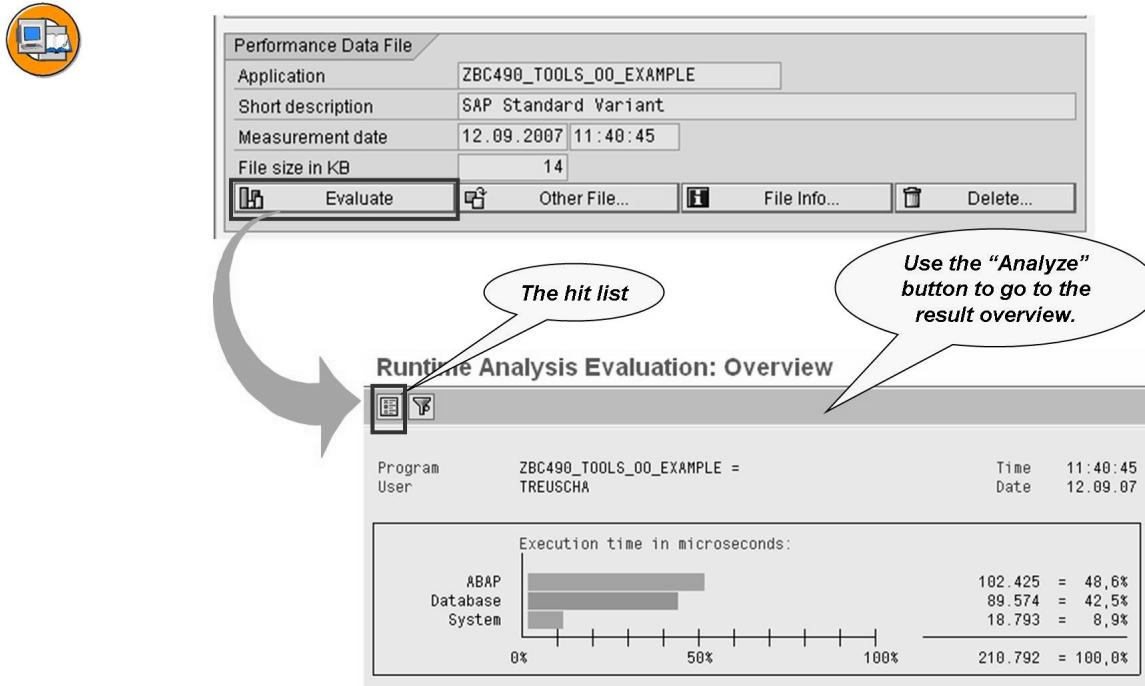
You can limit the data to be recorded in the *Measurement Restrictions* area. To do this you define or modify variants.

If you choose *Switch On/Off* under *In Parallel Session*, a screen appears listing all SAP work processes running on the current application server. This option, which can be accessed by pressing the *Start Measurement* button or via the menu path *Edit → Measurement → Switch on*, enables you to switch on runtime analysis for the object currently running in any SAP work process. To stop the runtime analysis, click the *End Measurement* button or choose *Edit → Measurement → Switch off*. This type of analysis is well suited to snapshot analyses. You can directly monitor long-running programs in order to identify critical program parts or statements.

The measurement restrictions that you set on the initial screen apply equally to runtime analyses runs in the current user session and to runtime analyses run in a parallel session. Possible aggregation techniques will be explained later.

The runtime-analysis results are either saved to an operating system file on the current application server for up to 8 days, or transferred to the frontend. The results file can be loaded into the ABAP runtime analysis from the frontend at any time.

The SAP system parameter *abap/atrasizequota* specifies how much memory should be reserved in the file system to store the results data. The SAP system parameter *abap/atrappath* specifies where the results data are stored in the file system. In the field *Short Description*, enter a short text to identify your analysis.



**Figure 38: SE30 - result, graphic and hit list**

From the ABAP runtime analysis initial screen, you can select the performance results file you want to access, and then choose *Evaluate* to display those results.

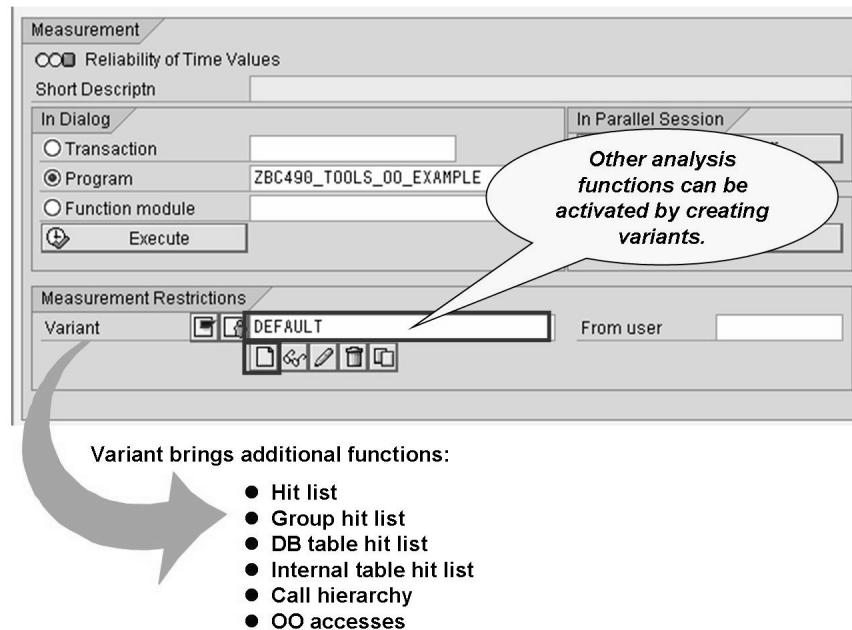
The Runtime Analysis Evaluation: Overview screen initially provides a bar chart comparing the distribution of the runtime between the ABAP processing, the database, and the SAP system. Beneath the bar chart is information on the conversions and the total number of programs that were loaded. Note that the time for the runtime analysis itself accrues to the system time. If delays occur during execution of the object to be tested (e.g. a transaction with screen change or selection screens), these delays show up in the system time!

From this overview screen you can access various other detailed data views. To access these views, choose from the icons in the toolbar or from the menu, choose *Goto → Hit list* or *Goto → Object-centered hit list*.

 **Note:** The results screen features other analysis functions in addition to the hit list. To activate these functions, you need a specially-configured variant of the measurement restrictions.

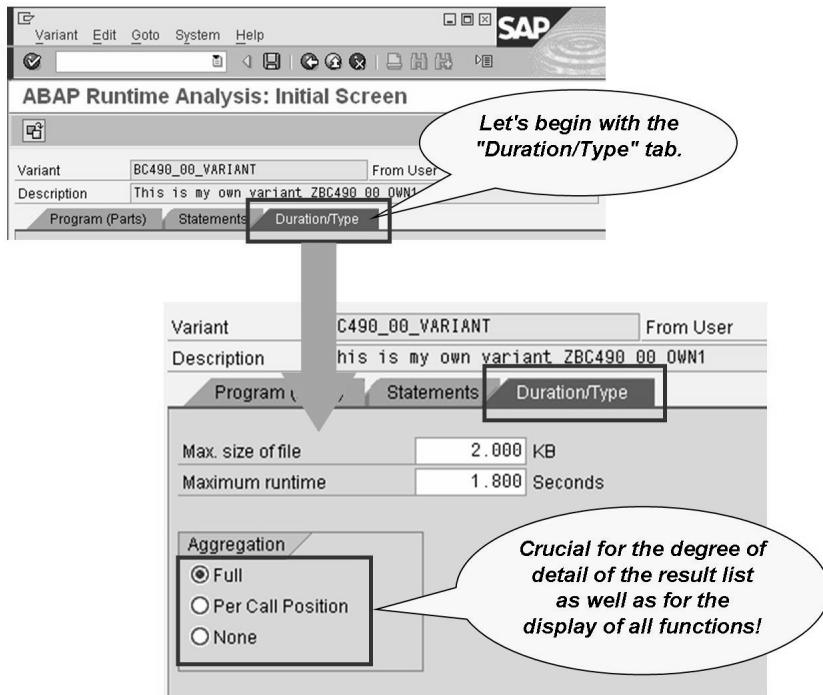
Use the *Create* button (available as of 6.10) to create a variant of the measurement restrictions.

For technical reasons, in release 4.6 it was necessary first to create a temporary variant (button for temporary variants) and then edit it in order to be able to save a user-defined variant. In 4.6, please proceed in just this manner.



**Figure 39: Details on Measurement Restrictions Using Variants**

Under *Measurement Restrictions*, you can restrict the amount of information to be logged and activate other useful functions of the tool. You can also create and change variants. You change variants on the tabs *Program (Parts)*, *Statements* and *Duration/Type*.



**Figure 40: SE30 Variant - Duration and Type, Aggregations**

It is advisable to start by maintaining the settings for “duration and type”. The degree of detail of the result as well as all other functions depends on selecting the appropriate aggregation. The next graphic illustrates how the three aggregation options work:



**The following ABAP is specified:**

```
REPORT Z_AGGREGATIONS.

PERFORM subprog1.
COMPUTE ...
DO 1000 TIMES.
  PERFORM subprog1.

ENDDO.
```

#### aggregation

- full
- per calling position
- no aggregation

**The result list contains x entries for subprog1:**

- [1] SE30 provides only the hit list as a function
- [2] SE30 provides only the hit list as a function
- [1001] SE30 provides all functions in addition to the hit list, e.g. the group hit list, table hit list, etc.

**Figure 41: SE30 Variant - Possible Aggregations**

If you opt for “None”, the only analysis function available to you later is the hit list; this, however, may be enough for an initial rough analysis. The results log, by contrast, is much clearer and easier to interpret.

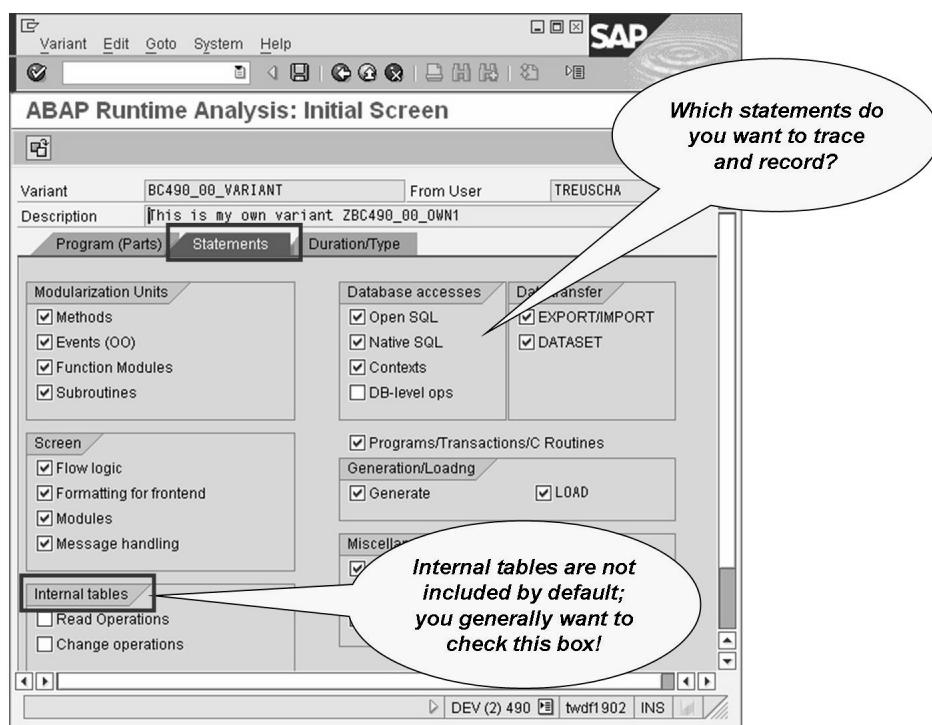
On the *Duration and Type* tab, you can specify the maximum size of the results file and the maximum recording duration. You select the aggregation level in the *Aggregation* area. If you select *Full* aggregation, one record **per program part or statement** is written to the results file. If you select *Per Call*, one record **per call** is written to the results file. If you select *None*, one record per measurement object (form routine, select statement, function module) is written to the results file.

The ABAP runtime analysis is normally used with a top-down procedure. First, record the run data for an individual object with full aggregation and identify critical program parts and statements. Then you restrict the recording to critical program parts and statements. Set the aggregation level to *Per Call* or *None* to receive more detailed information.

In sum: The advantage of full aggregation is a manageable results log. A second run without selecting an aggregation level would allow you to analyze all the details. So it is up to the developer operating SE30 to decide which is the best course in a given case.



**Caution:** Only if you select *None* are all analysis functions of the runtime analysis available to you (group, table, internal table, OO hit lists, etc.). If you select any other aggregation level, only the hit list icon is available in the toolbar. No other functions can be selected.



**Figure 42: SE30 Variant - Possible Statement Types**

On the *Statements* tab, you can limit the recording of data to specific statements or types of statements. It is almost always a good idea to select the internal tables too; in practice, nary an ABAP program can make do without internal tables.

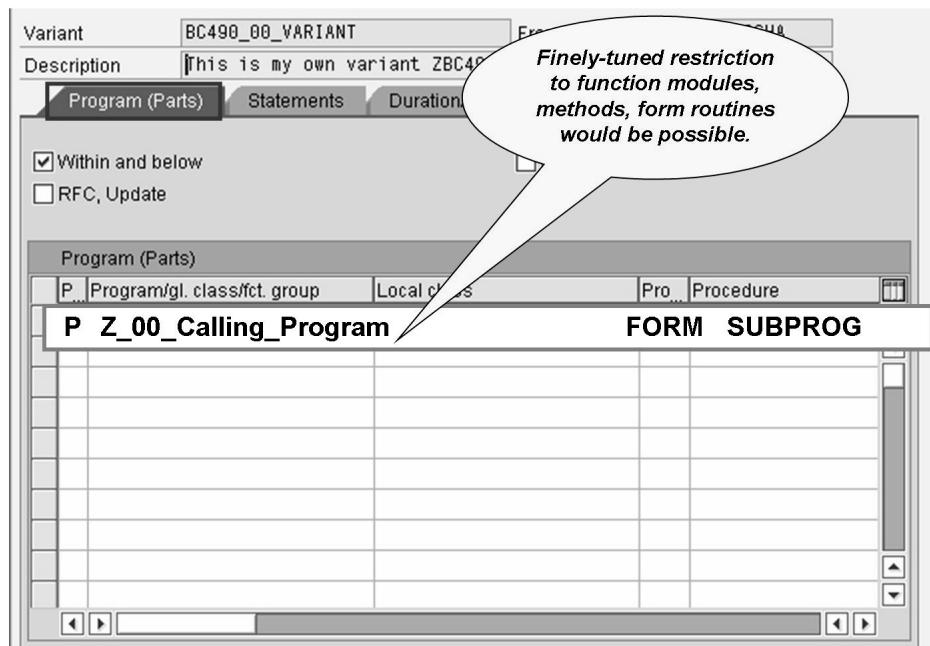


Figure 43: Restriction to Certain Program Parts

The *Program (Parts)* tab lets you restrict the recording of data to specific program parts. For this you must specify the program and its type (program, global class, or subprogram), and the procedure to be measured. By selecting *Within and below*, you extend the recording of data to include program parts and statements within the program parts you specified.

You can also restrict recording by selecting *Particular Units* and programming SET RUNTIME ANALYZER ON and SET RUNTIME ANALYZER OFF (to be explained in full later in the course).



- Hit list
- Group hit list
- Table hit list
- Hit list of internal tables
- Call hierarchy
- Class, instance, method hit lists
- Other lists

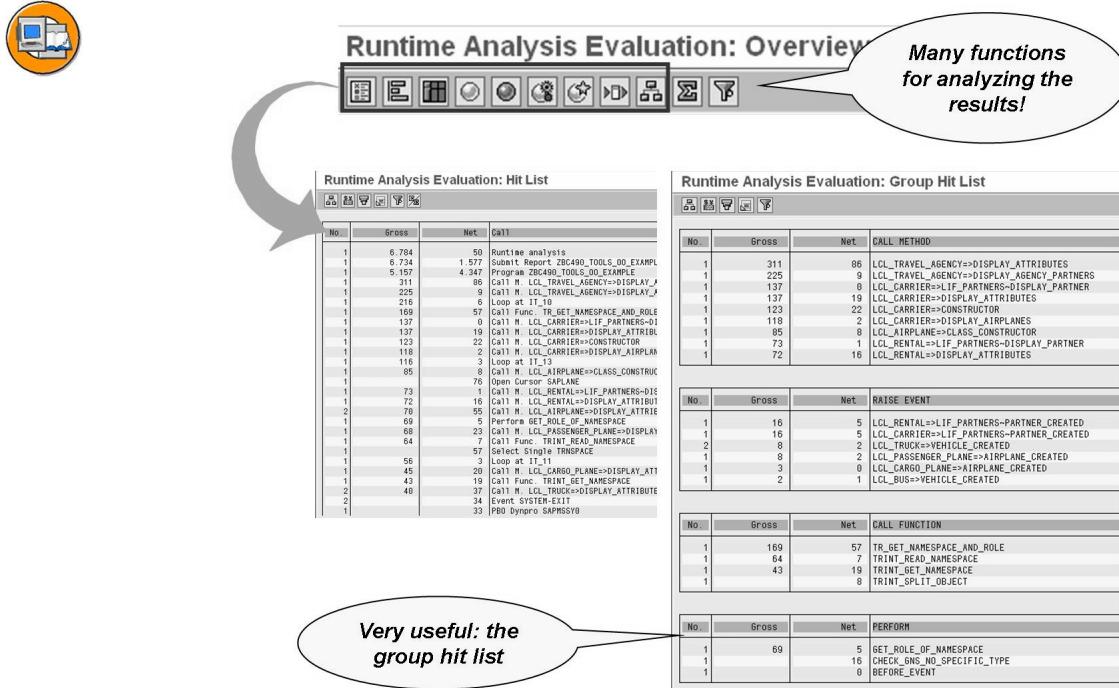


Figure 44: Runtime Analysis Evaluation Functions

Description of the runtime analysis evaluation functions:

**Hit list:** A list of all called program parts or statements, sorted according to gross time. It contains one row for each program part or statement. This row shows the name of the program part or statement, the gross time, the net time, and the number of times this program part or statement was called.

**Group hit list:** A list of individual calls of statements grouped according to statement categories (such as forms, function modules, methods), indicating the gross run time and the number of times the statement was called. The list contains one row for each call of an statement.

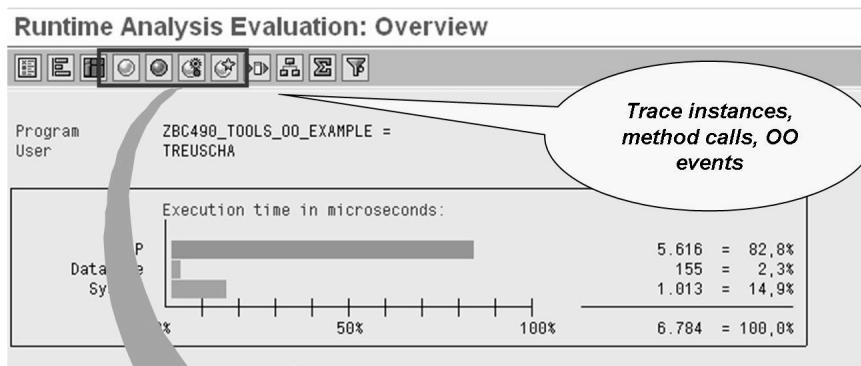
**Database table hit list:** A list showing accesses to database tables indicating the access name, the number of accesses, the table type, the buffering mode, and other information sorted according to access time.

**Internal table hit list:** A list of all internal table accesses that indicates the internal name, the number of accesses, the gross and net time taken (as an absolute value and as a percentage), the memory required, and other information. The list is sorted according to gross time.

**Call hierarchy:** A list showing the chronological sequence of analyzed objects, and indicating the gross and net time taken and the call level.

**OO hit lists (object-oriented parts):** A list of instantiated classes and method calls.

→ **Note:** The group hit list is particularly noteworthy as it provides a list sorted by “command group” and displays the most expensive commands for each group or command type (function modules, itabs, DB tables, forms, etc.).



*Trace instances,  
method calls, OO  
events*

- Class hit list
- Instance hit list
- Method hit list
- Event hit list

Figure 45: Evaluation of OO Components in Runtime Analysis

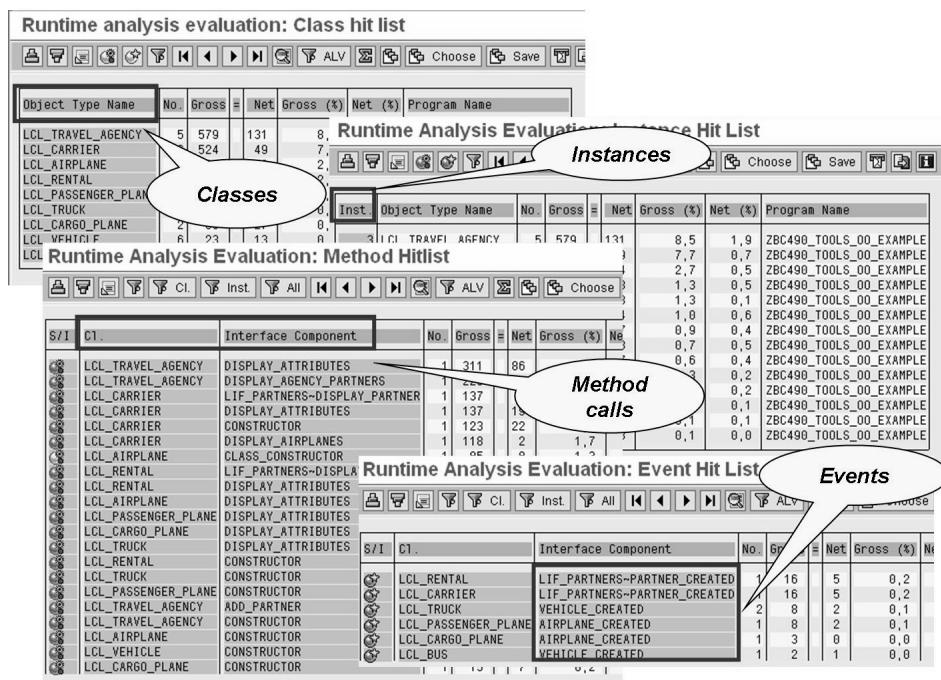


Figure 46: Analysis of Calls and Runtimes of Instances, Methods, Events

This analysis function tells you, for example, what are the most expensive method calls or from which classes the most instances were generated.

Now we shall turn our attention to gross and net time.

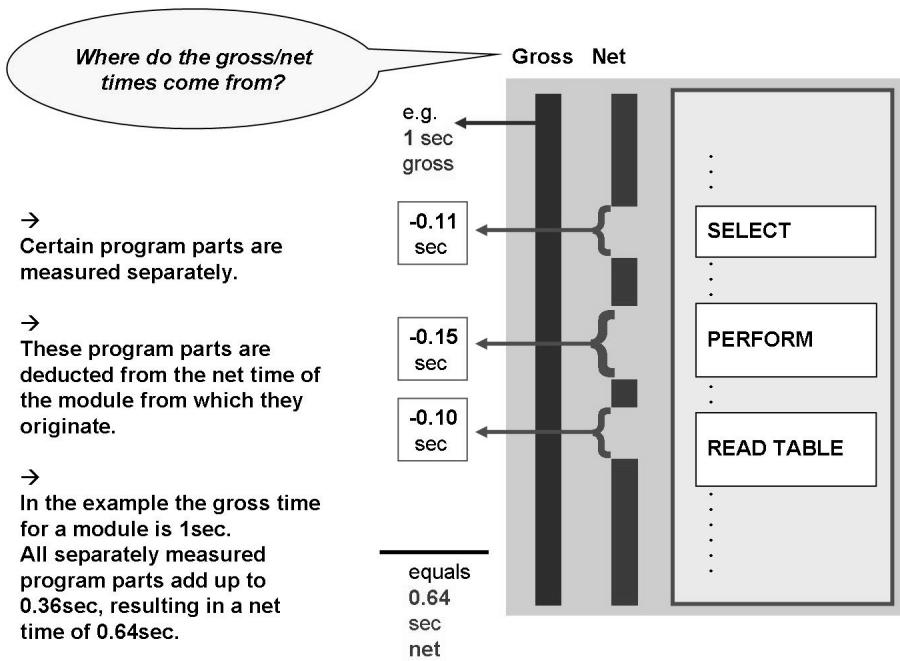


Figure 47: Runtime Analysis - Gross / Net

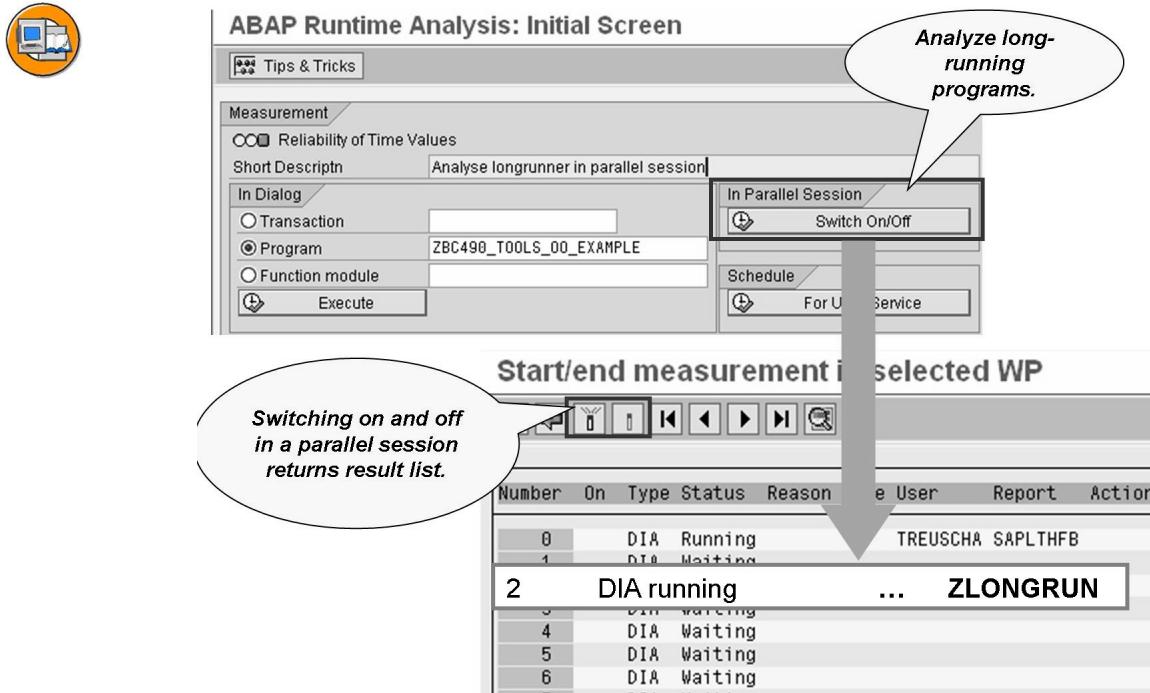
**Gross time** = total time required to execute the function

**Net time** = gross time minus the time taken by any modularization units and statements measured separately (definable on the initial screen using variants). Net times therefore represent the statements not slated for separate measurement.

If the gross time is the same as the net time, these times are not indicated separately.

You can display times either as a percentage or as absolute times in microseconds (*Absolute <-> %* button).

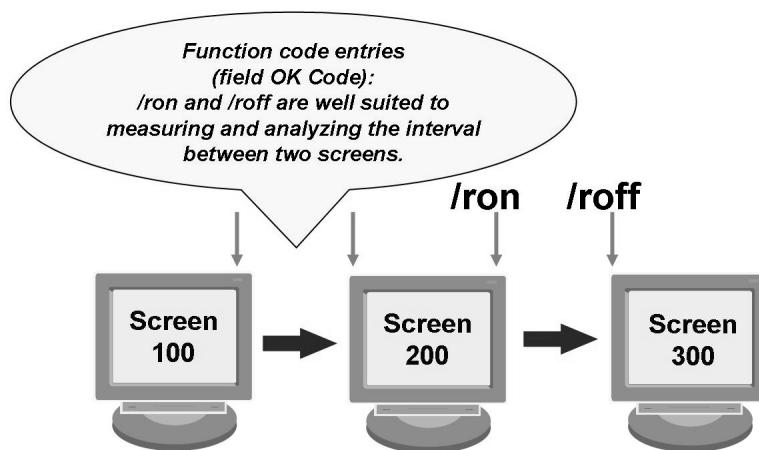
Another area of operation of runtime analysis is **parallel mode**.



**Figure 48: Runtime Analysis - Parallel Mode**

If you choose *Switch On/Off* under *In Parallel Session*, a screen appears listing all SAP work processes running on the current application server. You can analyze a single object running in any given SAP work process. You switch the function on/off using the two icons in the toolbar (two matches). This type of analysis is well suited to snapshot analyses. You can directly monitor long-running programs in order to identify critical program parts of statements.

Additional option: In the debugger, halt the application at the desired place, activate measurement in the parallel session of SE30 and proceed in the debugger!



Alternatively, the ABAP commands could be programmed:

- SET RUN TIME ANALYZER ON
- SET RUN TIME ANALYZER OFF

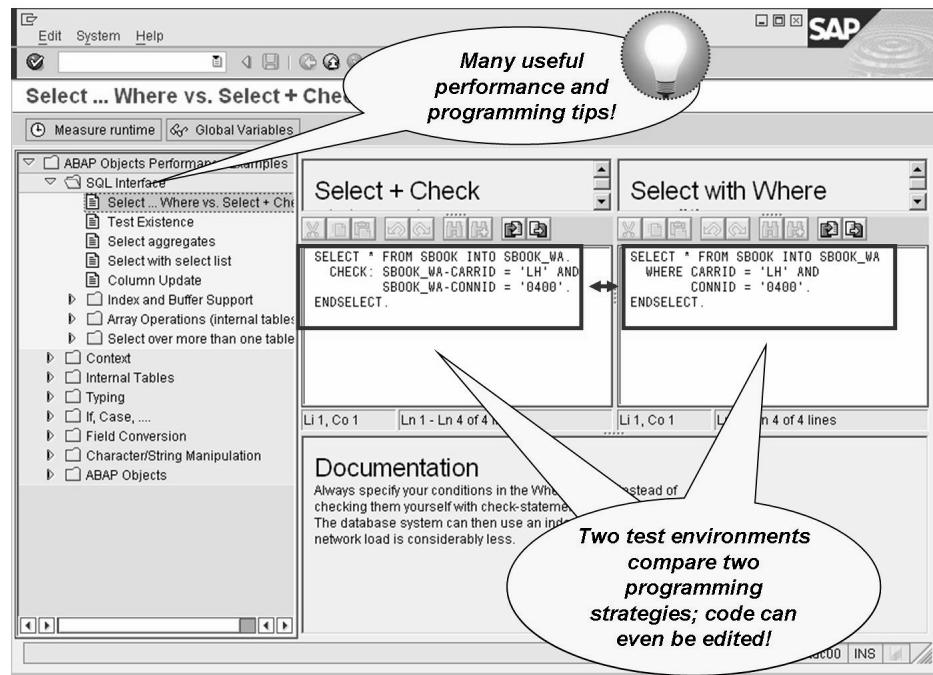
Prerequisite:

- In the performance data variant, the "Particular Units" indicator must be selected!

**Figure 49: Runtime Analyzer ON / OFF**

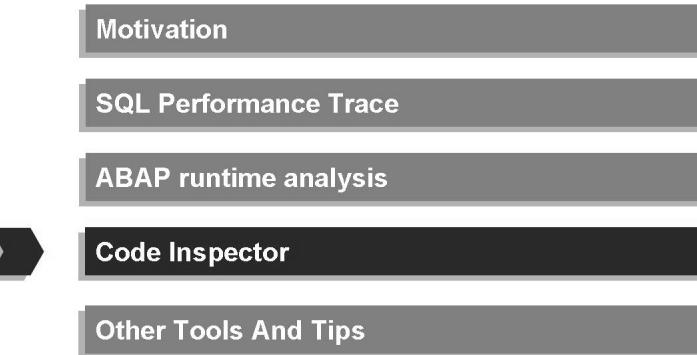
Selecting *Particular Units* in the measurement restrictions enables you to record data that falls between two measurement points. You have two options here: either by direct ABAP programming (SET RUN TIME ANALYZER ON / OFF commands) or by entry in the OK Code field (/ron and /roff).

This would allow you to trace the code between two screens (PAI/PBO), for example (see graphic).



**Figure 50: Runtime Analysis - Tips & Tricks**

The various programming examples that SAP provides as templates under “Tips & Tricks” are highly recommended. Many programming strategies are compared to each other there in measurement or test environments. The examples can even be extended in an editor. These examples are well worth a closer look!



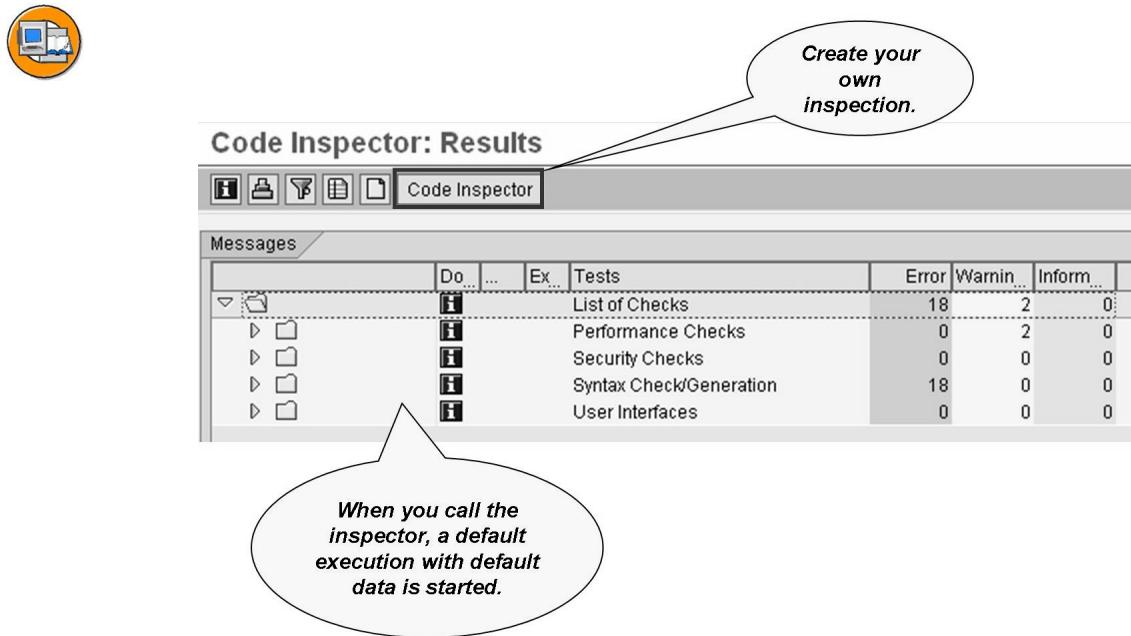
**Figure 51: Analysis Tools (4)**

## Code Inspector

One way of opening the Code Inspector from SE80 is via *Program → Check → Code Inspector*. Directly after the call, the inspector executes a default test with a default variant. The parameters of this test can be changed.

You get the same result when you start the *Code Inspector* from the context menu in the object list of SE80 (recommended procedure).

 **Note:** The Code Inspector is available as of release 6.10



**Figure 52: The Code Inspector (transaction SCI or SCII)**

As the result of an inspection you receive a list of error and warning messages. The information button that belongs to the message shows a detailed error description as well as suggestions for improvements. Double-clicking on the error text displays the corresponding program statement.

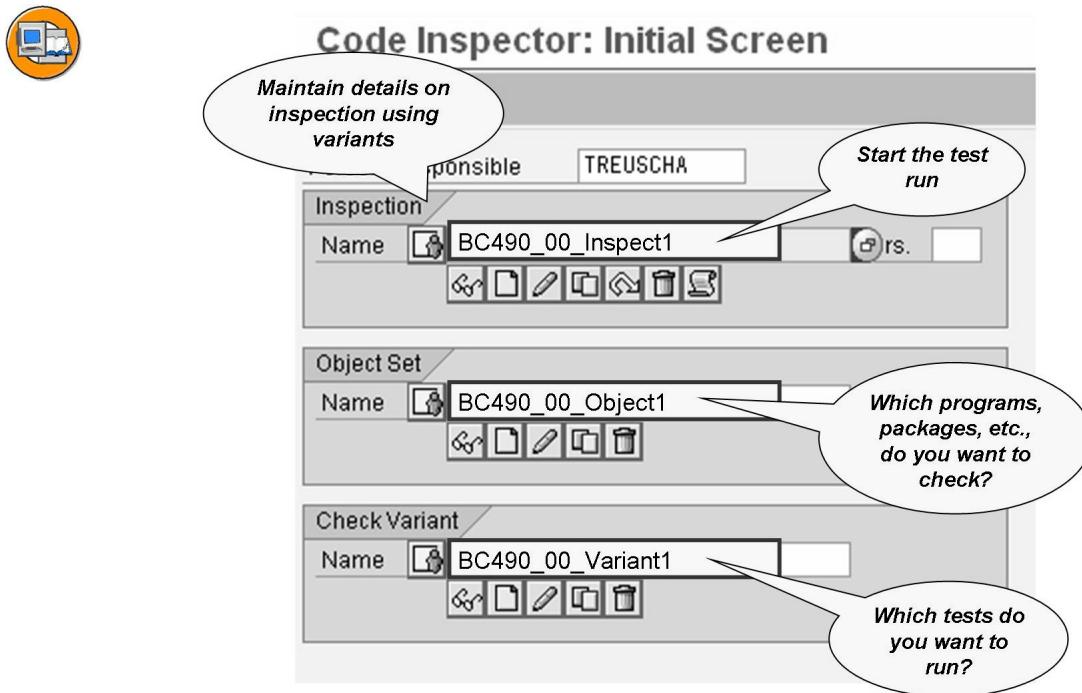
In this type of standard check, the *Code Inspector* uses a default check variant in which the checks to be performed are predefined. This default check variant contains:

- Extended syntax check
- Check of critical statements (such as C calls or Native SQL)
- Selected performance checks



You can overwrite this default check variant by creating a new check variant with the name **DEFAULT**. Note, however, that this new check variant will override the standard variant for your user (in all clients).

If you delete your default check variant, the standard variant is used automatically for future checks. For this reason, you should always define your own check variants instead of overwriting the default variant. We will examine this option in the next section.



**Figure 53: Maintenance of variants in the Code Inspector (SCI)**

Start maintenance of your variants using transaction SCI. The initial screen contains three areas:

- Inspection
- Object Set
- Check variant

#### Check variant

Defines which checks will be performed

#### Object set

Determines which (Repository) objects should be checked.

## Inspection

Name of the actual check. If you want to save the inspection (persistent), enter a name; if you do not enter a name for the inspection, the results are not saved.

Therefore, before you can start an inspection, you have to define an object set and a check variant.



**Hint:** You can define check variants, object sets, and inspections as either **public** or **private**. Use the pushbutton next to each input field to toggle between these two categories. Note that private objects are for your personal use only, whereas public objects are available to all users in the system.

Accordingly, the most important item when using the *Code Inspector* is the check variant. A check variant consists of one or more check categories, which in turn consist of one or more single checks (inspections). These single checks can be parameterized, for example, with a keyword or an indicator for a specific partial aspect of the check. In most cases, single checks only investigate a specific object type – such as the "Check of Table Attributes", which only examines DDIC tables.

The single checks are assigned to various check categories. The most important check categories are described below:

- General Checks contain data formatting, such as the list of table names from SELECT statements
- Performance Checks contain checks of performance and resource usage, such as
  - analysis of the WHERE condition for SELECT / UPDATE and DELETE
  - SELECT statements that bypass the table buffer
  - low-performance access to internal tables
- Security Checks contain checks of critical statements, cross-client queries, and insufficient authority checks
- Syntax Check/Generation contain the ABAP syntax check, an extended program check, and generation
- Programming Conventions contain checks of naming conventions
- Search Functions contains searches for tokens (words) and statements in ABAP coding

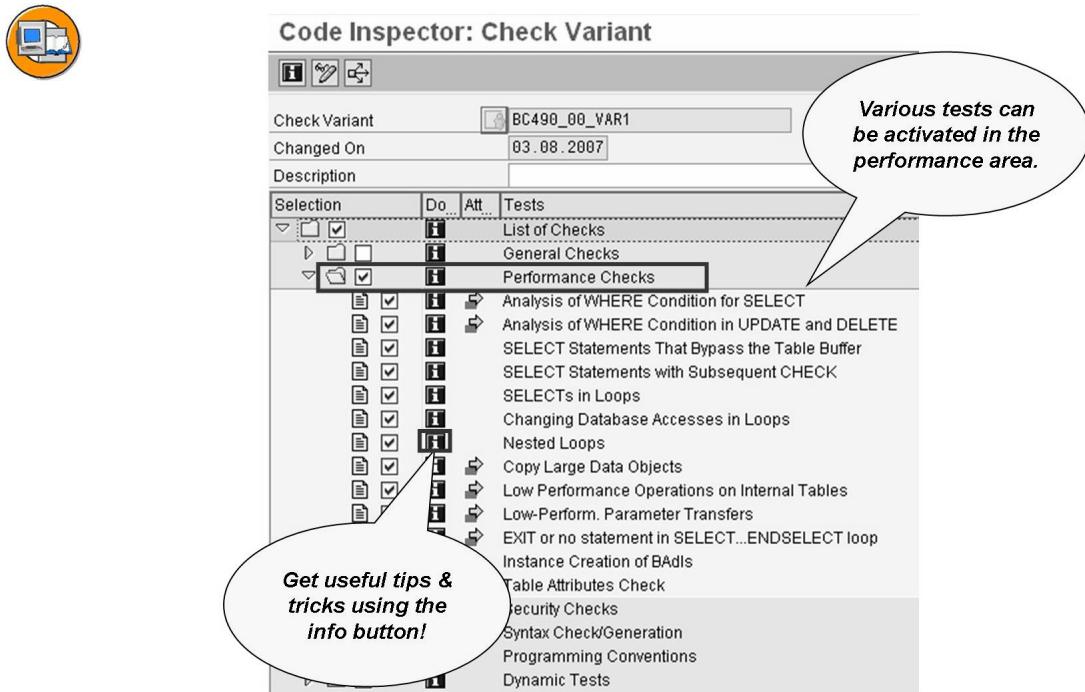


Figure 54: Selection of Performance Tests

The Information button is very useful. Here you will find tips & tricks on performance conventions!



## Performance tips & tricks can be called in the inspector

**Code Inspector: Check Categories**

The category **Performance** contains checks for performance and resource use.

- Analysis of WHERE Condition for SELECT
- Analysis of WHERE Condition for UPDATE and DELETE
- SELECT Statements that Bypass the Table Buffer
- SELECT Statements with Subsequent CHECK
- SELECTs in Loops
- Changing Database Accesses in Loops
- Nested Loops
- Performance-Intensive Operations on Internal Tables
- Performance-Intensive Parameter Passing
- EXIT Statement Within a SELECT/ENDSELECT Loop
- Table Attributes Check

**Code Inspector**

**Analysis of the WHERE condition for SELECT**

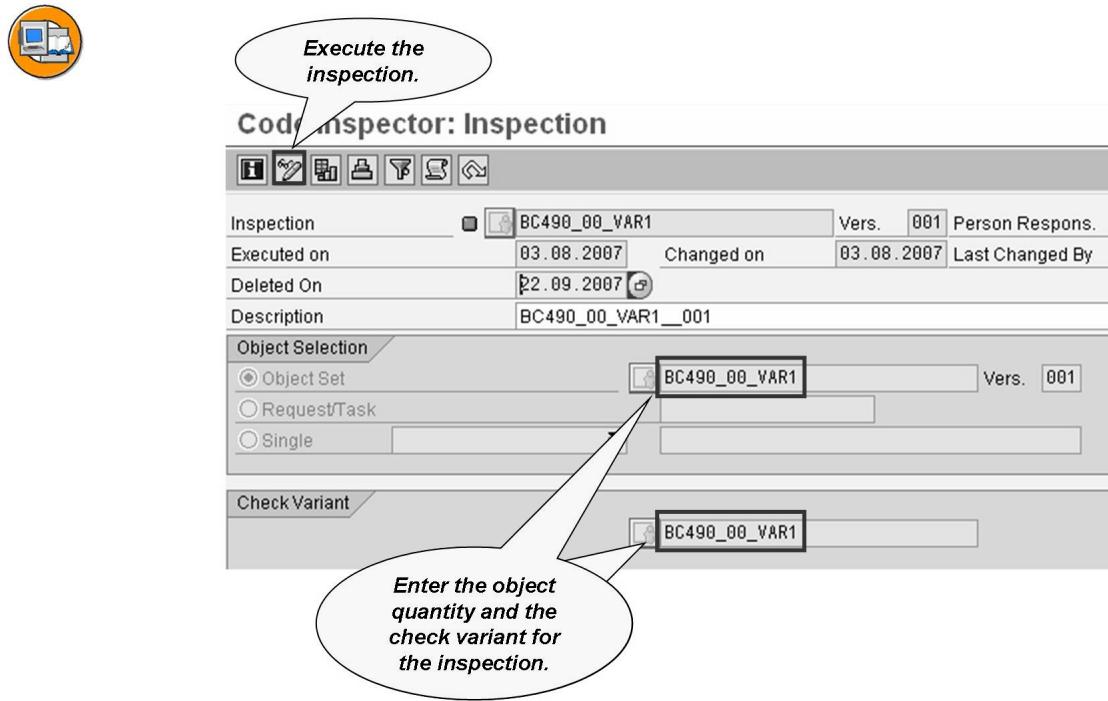
This check examines the WHERE condition of the SELECT statement for accesses to **non-buffered** tables according to the criteria below. JOINs are not processed; accesses that bypass the table buffer are not taken into consideration either.

- SELECT statement does not contain any WHERE condition.
- WHERE condition does not contain any field of a table index.
- WHERE condition does not contain any first field of a table index.
- Despite the addition 'CLIENT SPECIFIED', no client field in the WHERE condition.
- Table does not exist or has no nametab entry.

The priority of a message depends on the size category of the table that is accessed.

**Figure 55: Information about the Performance Tests**

To start the inspection, specify an inspection name, the previously-defined check variant and object set variant. Then you start it with F8 or the *Execute* button.



**Figure 56: Starting the Inspection**

The obtain the results log by clicking the *Results* icon in the toolbar.

You get another inspection with the same settings using the “Create New Version” button; each inspection run receives a version ID.



Person Responsible	TREUSCHA	Inspection	BC490_00_VAR1	Version
Messages	Do ...	Ex.	Tests	
↓	↓	↓	List of Checks	0 0 0
			Performance Checks	0 0 0
			Analysis of WHERE Condition for SELECT	0 0 0
			Warnings	0 1 0
			Message Code 0001	0 0 0
			Program SAPBC490_TOOL_SCL_1 Include SAPBC490_TOOLS_AIRPLANES_SC...	0 1 0
			Table SAPLANE: No WHERE Condition	0 1 0
			==> Table SAPLANE: No WHERE Condition	0 1 0
			Message Code 0002	0 1 0
			==> Table SFLIGHT: No Field of a Table Index in WHERE Condition	0 1 0
			Analysis of WHERE Condition in UPDATE and DELETE	0 0 0
			SELECT Statements That Bypass the Table Buffer	0 0 0
			SELECT Statements with Subsequent CHECK	0 0 0
			SELECTS in Loops	0 0 1
			Changing Database Accesses in Loops	0 0 0
			Nested Loops	0 0 0
			Copy Large Data Objects	0 0 0
			Low Performance Operations on Internal Tables	0 0 1
			Low-Perform. Parameter Transfers	0 0 0
			EXIT or no statement in SELECT...ENDSELECT loop	0 0 0
			Instance Creation of BAdIs	0 0 0
			Table Attributes Check	0 0 0
			Security Checks	1 0 0
			Critical Statements	0 0 0
			Dynamic and Client-Specific Accesses in SELECT	0 0 0
			Dynamic and Client-Specific Accesses with INSERT, UPDATE, MODIFY, DELETE	0 0 0
			Check of SY-SUBRC Handling	1 0 0
			Errors	1 0 0
			Message Code 0003	1 0 0
			Program SAPBC490_TOOL_SCL_1 Include SAPBC490_TOOLS_AIRPLANES_SC...	1 0 0
			No Handling of SY-SUBRC after AUTHORITY-CHECK	0 0 0
			==> No Handling of SY-SUBRC After AUTHORITY-CHECK	0 0 0
			Client-Specific Shared Objects Methods	0 0 0
			Syntax Check Generation	0 0 0
			Internal Performance Tests	0 0 0

The result of the inspection shows one error, some warnings and informations

Figure 57: Inspection Results

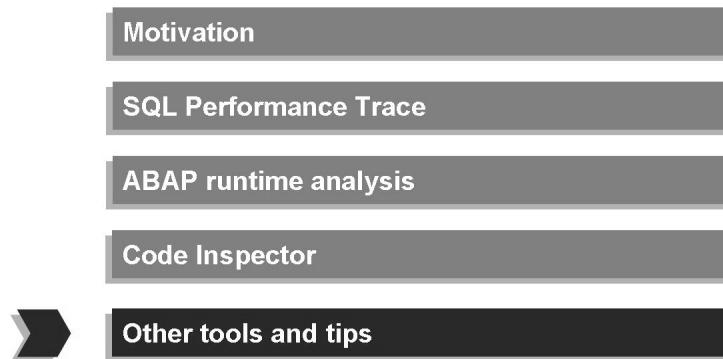
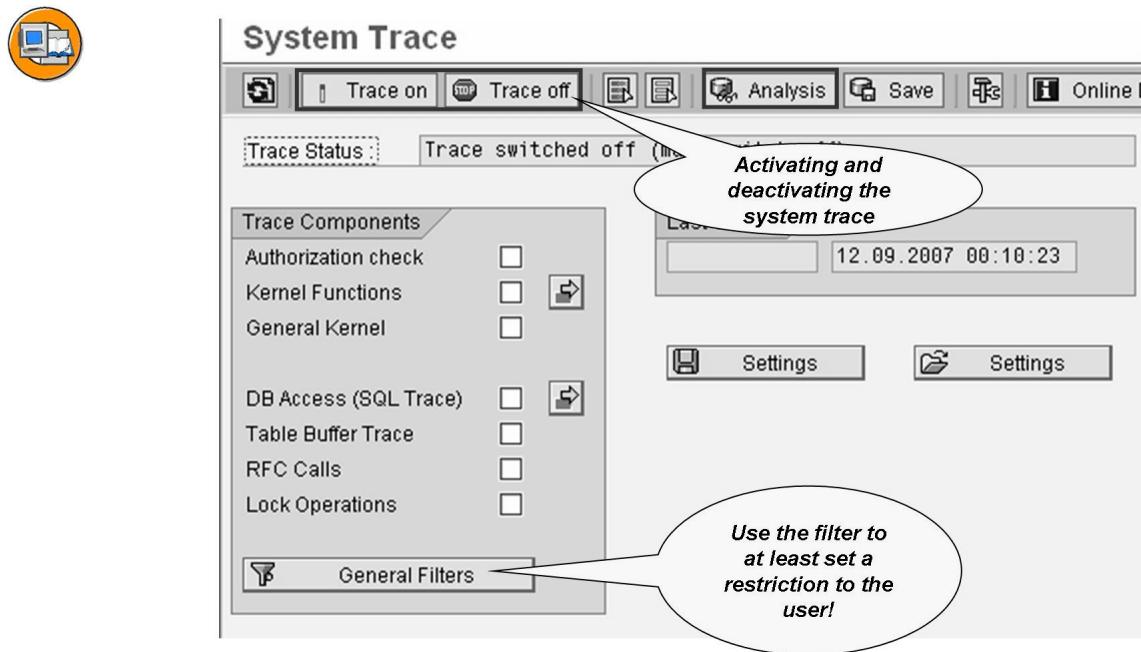


Figure 58: Analysis Tools (5)

## System Trace



**Figure 59: System Trace**

Before starting the trace, the user and the program or transaction should be specified in the filter settings. Otherwise the system would record everything that takes place in a given timeframe, which is generally not desirable. You can save your filter settings by clicking the *Save Settings* button. A constantly running trace is a drag on performance, so be sure to turn it off as soon as you no longer need to have it on.

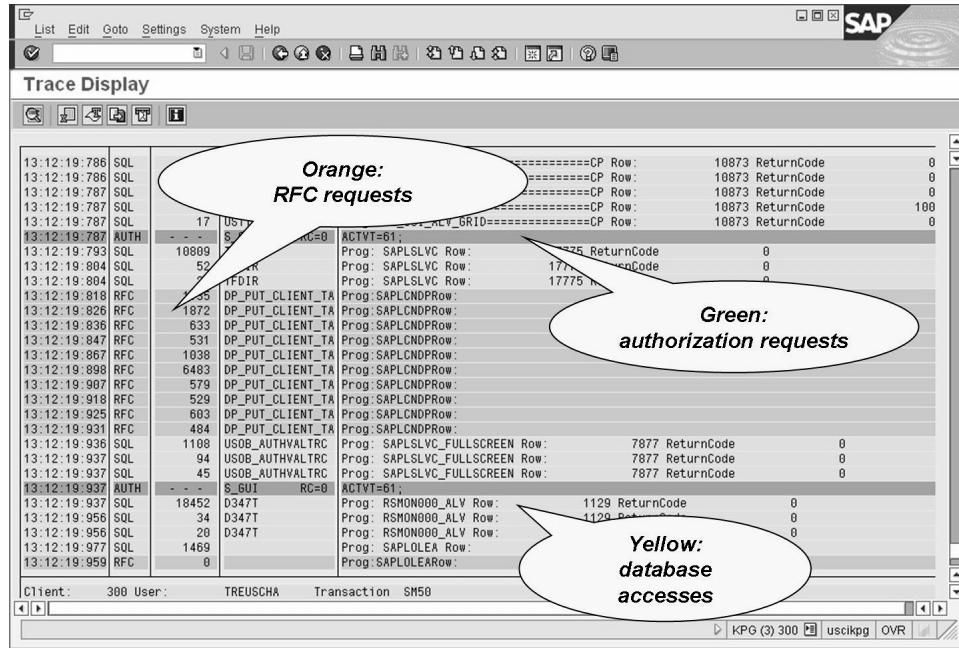


Figure 60: System Trace Results Log

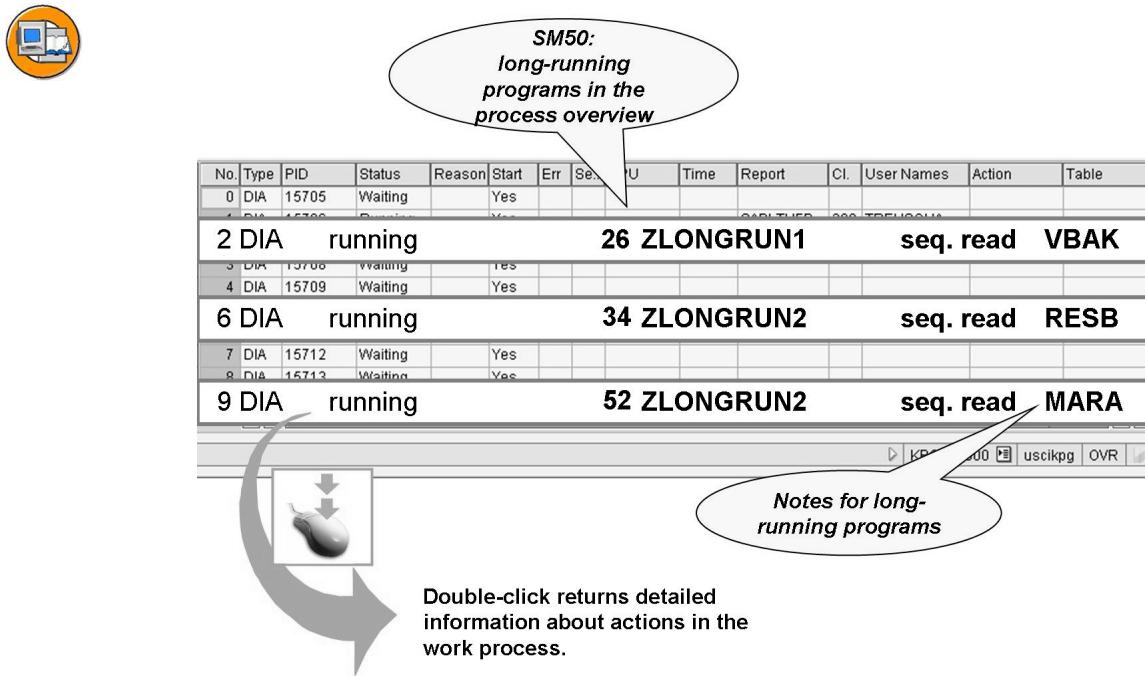
When you select the results log, be sure to choose the correct time interval; if you get an empty list, it may be due to selection of the wrong time interval, or incorrect filter settings.

→ **Note:** See also OSS notes 516398, 35720, 855974, 66056, 412978 on the system trace

## Snapshot Analysis

The goal of snapshot analysis is to provide you with an overview of the processes currently running in your system. You are looking for long-running programs. These are usually caused by long-running SQL statements or database operations.

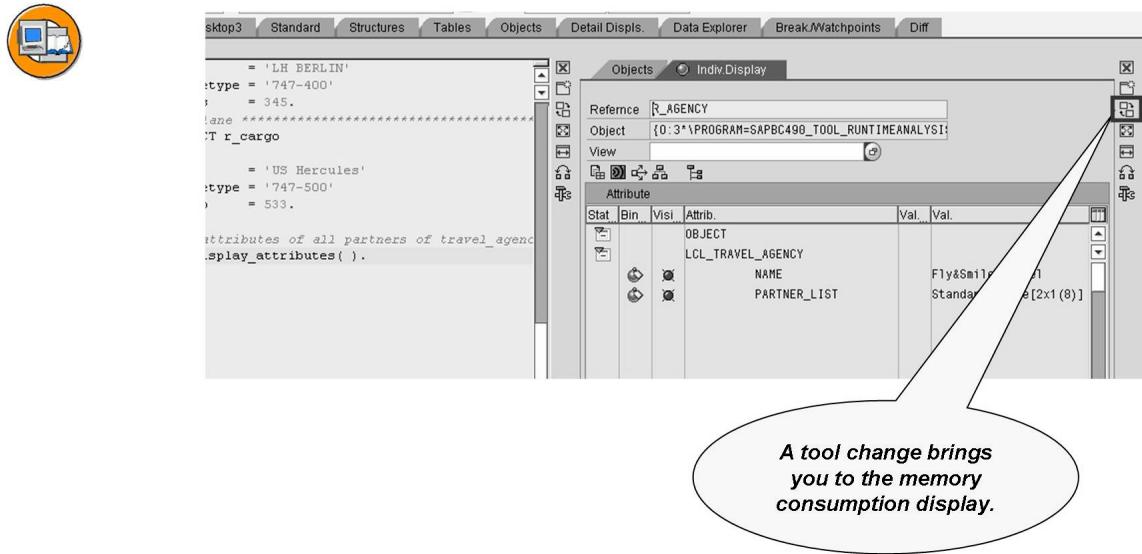
In the case of a currently running program, it is possible to create a “snapshot” of currently-running actions via the process overview (SM50).



## Debugger Functions

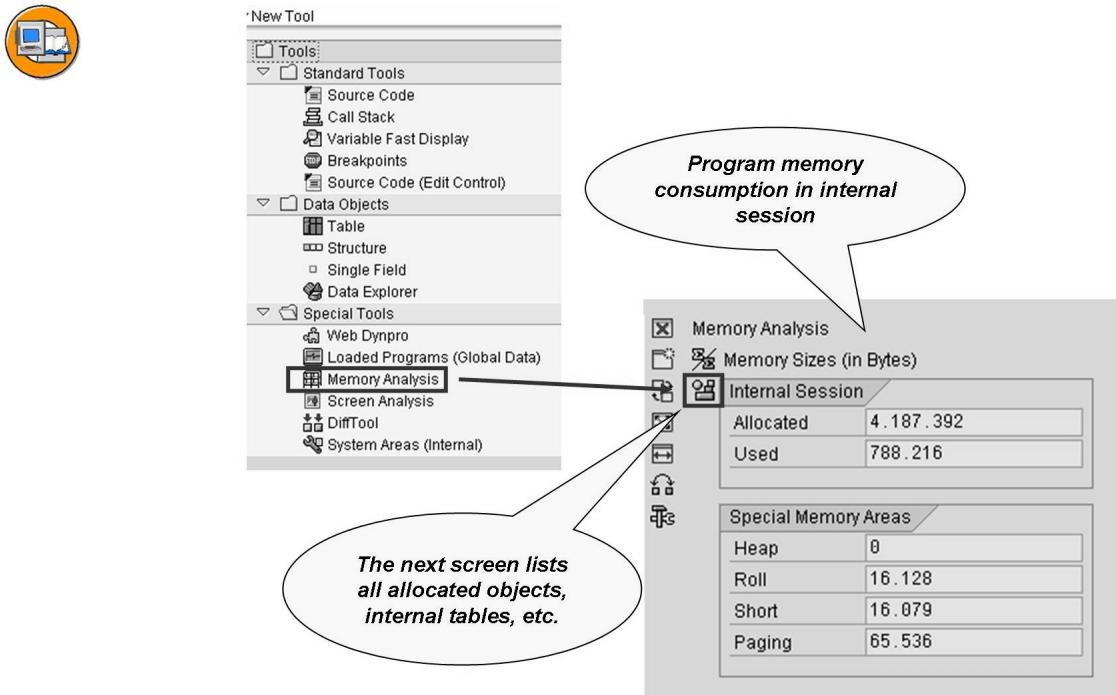
The ABAP debugger enables you to run through an object step by step and is normally used to check functionality. Specific functions of the debugger are useful in the context of analyzing performance.

From a performance perspective, large internal tables are often problematic, since statements that affect them (such as LOOP AT ... ENDLOOP, READ TABLE ..., or SORT ) often require much CPU time and effectively increase object runtime and slow down the system in general. Thus it is useful to check the memory use of internal tables during the running of a program.



**Figure 62: Special Tools in the New Debugger (from SAP NW AS 7.0)**

Special tools in the debugger enable the display of objects, internal tables and memory data. Observe the corresponding button. The example above shows an OO program with the instantiated class LCL\_TRAVEL\_AGENCY.



**Figure 63: Tool Change in the Debugger - Memory Analysis**

You can trace the memory occupied by objects, strings or internal tables. On the initial screen you can see the total memory consumption for a session.

The screenshot shows the SAP debugger's 'Memory Objects' view. It lists various memory objects, including ABAP, memory objects, and internal tables like LCL\_TRAVEL\_AGENCY, LCL\_CARRIER, and LCL\_TRUCK. A callout bubble points to the 'Memory Objects' section, labeled 'Objects, internal tables with memory consumption'.

	Rank	Bound	Occupancy ...	Referenced	Occupancy Refer...
ABAP: 72.864 Bytes					
Memory Objects					
[26x18(120)]	1	5.952	55%	5.952	55%
[1x15(240)]	2	1.400	30%	1.400	30%
(0.5)LCL_RENTAL	3	680	75%	1.032	78%
(0.3)LCL_TRAVEL_AGENCY	4	504	70%	2.392	76%
(0.11)LCL_CARRIER	5	504	70%	856	77%
[3x1(8)]	6	416	62%	768	73%
[2x1(8)]	7	240	43%	2.128	74%
[2x1(8)]	8	240	43%	592	70%
(0.7)LCL_TRUCK	9	176	84%	176	84%
(0.14)LCL_CARGO_PLANE	10	176	88%	176	88%
(0.13)LCL_PASSENGER_PLANE	11	176	87%	176	87%
(0.6)LCL_BUS	12	176	86%	176	86%
(0.9)LCL_TRUCK	13	176	85%	176	85%
(S:3)	14	88	82%	88	82%
(S:5)	15	88	67%	88	67%
(S:4)	16	88	82%	88	82%
(S:8)	17	88	82%	88	82%
(S:10)	18	88	76%	88	76%
(S:9)	19	88	74%	88	74%
(S:6)	20	88	73%	88	73%
(S:7)	21	88	69%	88	69%

**Figure 64: Internal Tables and Objects in the Memory Display**

To create a memory extract for the Memory Inspector, you have to select the root of the displayed tree control and then work with the right mouse button. Position the cursor on “memory objects”.

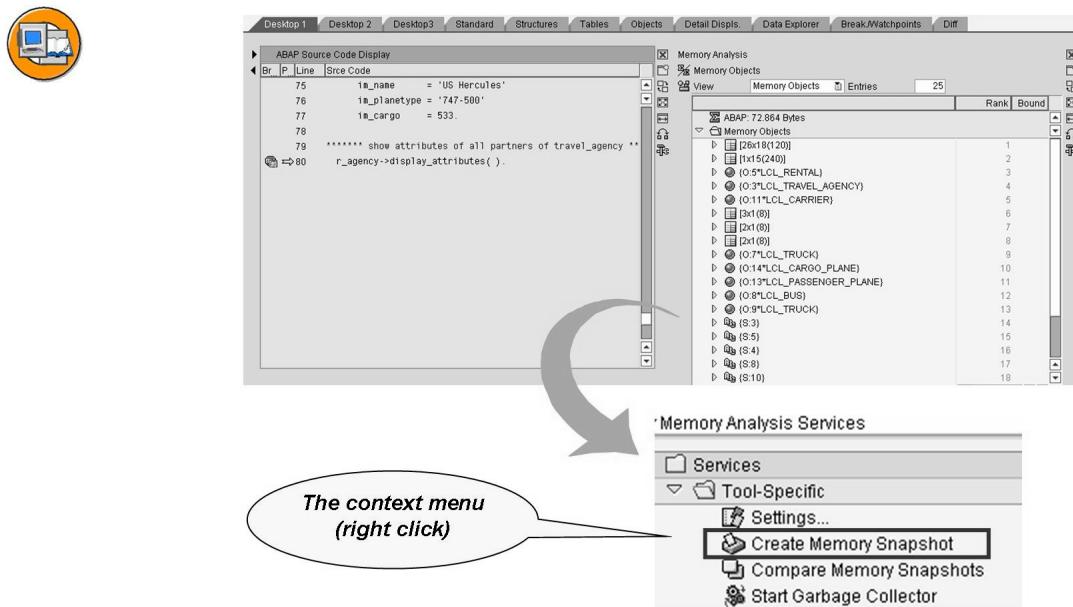
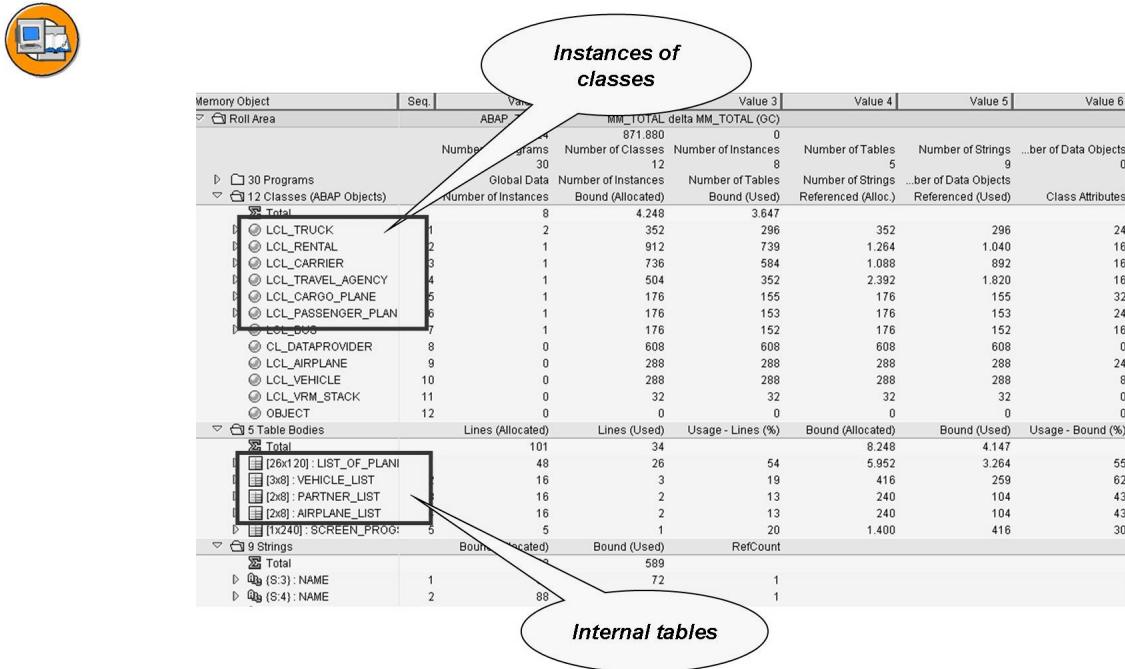


Figure 65: Memory Inspector - Creating a Memory Extract

In the debugger you can create a memory extract for further analysis with the Memory Inspector (transaction: S\_MEMORY\_INSPECTOR).



**Figure 66: The Memory Inspector**

Use transaction S\_MEMORY\_INSPECTOR to go to the Memory Inspector.



## Exercise 2: SQL Trace

### Exercise Objectives

After completing this exercise, you will be able to:

- Analyze SAP OPEN SQL accesses to the database

### Business Example

In the context of a customer development, a company wants to analyze the accesses to the database in order to identify performance-relevant program parts.

### Task: The SQL Trace Tool

Workload analysis has detected a performance-critical single object with an excessive database access time. This object is now to be analyzed.

- In the workload analysis, try to localize the performance-critical object that the instructor started beforehand.
- Perform an SQL trace with this object and make a note of the execution times. What is particularly interesting is the number of executions of each statement, the identical selects, etc.
- Copy program SAPBC490\_TEMP\_JOIN3 to Z\_##\_JOINS3 in order to complete additional subroutines there.

Then optimize your own program by implementing the coding in second subroutine **NESTED\_SELECT\_BETTER** as an alternative to the coding in the first subroutine **NESTED\_SELECT\_VERYBAD**. Comment out the call of the first subroutine.

First of all, only the reading order of the tables is to be inverted in a first step. Use VBAK as the outer driving table and VBAP as the inner driven table. Of course, the same data as before should be read after this rearrangement; otherwise there would be no fair comparison. Then restart the SQL trace and compare all relevant key figures.

What did you notice regarding the runtime, number of executions of the statements, identical selects and so on?

## Solution 2: SQL Trace

### Task: The SQL Trace Tool

Workload analysis has detected a performance-critical single object with an excessive database access time. This object is now to be analyzed.

1. In the workload analysis, try to localize the performance-critical object that the instructor started beforehand.
  - a) Navigate into ST03; there you will find the program that the instructor started beforehand under *Detailed Analysis -> Last Minute Load*. At the top left, under Workload you will only find programs that ran up until an hour ago; so unfortunately, the programs that just ran (interval < 1 hour)
2. Perform an SQL trace with this object and make a note of the execution times. What is particularly interesting is the number of executions of each statement, the identical selects, etc.
  - a) In ST05, start the trace, ideally with the option Activate Trace With Filter to trace the relevant tables (ideally, exclude the system tables; they start with D\*, P\*, W\*, ...)

You start the program to be measured and stop the trace once it ends. Monitor the result in the results list of the trace.

You can reach various interesting functions from this trace list. To do so, navigate to *Trace List -> Summarize Trace by SQL Statement / Combined Table Accesses/ Display Identical Selects*.

3. Copy program SAPBC490\_TEMP\_JOIN3 to Z\_##\_JOINS3 in order to complete additional subroutines there.

Then optimize your own program by implementing the coding in second **subroutine NESTED\_SELECT\_BETTER** as an alternative to the coding in the first subroutine NESTED\_SELECT\_VERYBAD. Comment out the call of the first subroutine.

First of all, only the reading order of the tables is to be inverted in a first step. Use VBAK as the outer driving table and VBAP as the inner driven table. Of course, the same data as before should be read after this rearrangement; otherwise there would be no fair comparison. Then restart the SQL trace and compare all relevant key figures.

What did you notice regarding the runtime, number of executions of the statements, identical selects and so on?

*Continued on next page*

- a) Implement the coding as shown in the solution.

Following that, trace the solution program as usual and observe the filter settings in the trace. Only DB accesses to VBAK, VBAP and KNA1 are of interest.

## Result

```

*
* Part of Solution: SAPBC490_SQL2_JOIN3
*
TYPES: BEGIN OF ty_rec,
        vbeln TYPE vbap-vbeln,
        posnr TYPE vbap-posnr,
        matnr TYPE vbap-matnr,
        erdat TYPE vbak-erdat,
        kunnr TYPE vbak-kunnr,
        name1 TYPE kna1-name1,
      END OF ty_rec.
DATA: rec TYPE ty_rec.

SELECT-OPTIONS: vbeln FOR rec-vbeln
                 DEFAULT '1000000000' TO '1000021000'.

DATA: t1 TYPE f, t2 TYPE f.
*
* PERFORM nested_select_better.
*
*-----*
*      FORM nested_select_better          better !!!   *
*-----*
FORM nested_select_better.
  GET RUN TIME FIELD t1.

*** trace form with ST05 --> less Identical selects, but
*** other prblems stay the same

SELECT vbeln erdat kunnr FROM vbak
      INTO (rec-vbeln, rec-erdat, rec-kunnr)
      WHERE vbeln IN vbeln.
SELECT SINGLE name1 FROM kna1

```

*Continued on next page*

```
INTO rec-name1
WHERE kunnr = rec-kunnr.

SELECT posnr matnr FROM vbap
      INTO (rec-posnr, rec-matnr)
      WHERE vbeln = rec-vbeln.

*      WRITE: / rec-vbeln COLOR COL_HEADING,
*              rec-posnr COLOR COL_HEADING,
*              rec-erdat COLOR COL_POSITIVE,
*              rec-kunnr,
*              rec-name1.

      ENDSELECT.

ENDSELECT.

GET RUN TIME FIELD t2.

t2 = t2 - t1.  WRITE: / 'runtime = ', t2.

ENDFORM.          "nested_select_vbak_vbap
```

## Exercise 3: Runtime Analysis

### Exercise Objectives

After completing this exercise, you will be able to:

- Identify performance-critical parts of an ABAP program using runtime analysis

### Business Example

Within the context of a customer development, a company wants to identify performance-critical parts of ABAP programs.

### Task 1: Runtime Analysis - Analyzing a Simple Program

A specified program is to be inspected using runtime analysis.

Program **SAPBC490\_TOOL\_RUNTIMEANALYSIS** is specified. Measure this program using SE30.

Keep in mind that this first exercise for SE30 is intended as an introduction to this tool and you should learn how to use and handle this tool. The object-oriented program to be analyzed therefore does not have too complex a structure or any really performance-critical parts. You will be confronted with a more complex case in the next exercise.

1. First of all, perform a rough analysis of the program. In the first check run, use the DEFAULT variant and observe the resulting hit list in the result. (So you are simply not making any changes to the variant selection.)
2. In the second measurement run, use your own variant BC490\_##\_VAR1. Do not use an aggregation here.

In this run, focus your analysis on subroutines, function modules, OO calls, internal tables and Open SQL calls.

In the result you will see not only the normal hit list but also the group hit list and various analysis functions for object-oriented programming.

Determine the most expensive calls in the individual group categories. (See group hit list)

Which methods of which classes were called up and which method calls required the most time?

*Continued on next page*

## Task 2: Runtime Analysis - Analyzing a Complex Program

You should now use a more complex program **SAPBC490\_TOOL\_CASE\_STUDY** and inspect it using runtime analysis. Identify the main reasons for the good or bad performance of this program.

1. Measure this program using SE30.

We recommend starting with a rough analysis using the default variant followed by a detailed analysis with your own variants.

In which program part is the most runtime caused and where do you think is the most potential for optimization? What are the most expensive table accesses, internal table accesses, subroutines, etc.? (expensive= longest runtime and largest number of executions)

How could you then monitor the database accesses in more detail?

## Solution 3: Runtime Analysis

### Task 1: Runtime Analysis - Analyzing a Simple Program

A specified program is to be inspected using runtime analysis.

Program **SAPBC490\_TOOL\_RUNTIMEANALYSIS** is specified. Measure this program using SE30.

Keep in mind that this first exercise for SE30 is intended as an introduction to this tool and you should learn how to use and handle this tool. The object-oriented program to be analyzed therefore does not have too complex a structure or any really performance-critical parts. You will be confronted with a more complex case in the next exercise.

1. First of all, perform a rough analysis of the program. In the first check run, use the DEFAULT variant and observe the resulting hit list in the result. (So you are simply not making any changes to the variant selection.)
  - a) The program to be checked must be active! In SE30, enter this program in the Program field and start the analysis. You do not have to select anything for Measurement Restriction for the time being.
2. In the second measurement run, use your own variant BC490\_##\_VAR1. Do not use an aggregation here.

In this run, focus your analysis on subroutines, function modules, OO calls, internal tables and Open SQL calls.

In the result you will see not only the normal hit list but also the group hit list and various analysis functions for object-oriented programming.

Determine the most expensive calls in the individual group categories. (See group hit list)

*Continued on next page*

Which methods of which classes were called up and which method calls required the most time?

- a) Under Measurement Restriction, create a new variant using the Create button. You will then find three tab pages, which you should ideally process from right to left. The aggregation level is defined on the far right. Only the NONE selection returns all results functions later on. If you wanted to analyze the group hit list, for example, you would have to set this to NONE. Use the tab page in the middle to select all program parts that are supposed to be measured and should subsequently be available in the log. You decide, for example, if extra methods are supposed to be traced and measured by pressing the corresponding button.

On the results screen, navigate to the group hit list (second button at the top left) and into the other result functions. The selection buttons in the middle describe the measured method calls, instantiations and so on.



**Note:** In the analysis of the internal tables, runtime analysis uses technical, symbolic names (IT\_XX) for the actual internal tables in the program. If you want to know which itabs are behind the symbolic designations, you have to use the *Display Source Code* button (icon). This takes you directly to the source code of the program line with the internal table that you are looking for.

## Task 2: Runtime Analysis - Analyzing a Complex Program

You should now use a more complex program **SAPBC490\_TOOL\_CASE\_STUDY** and inspect it using runtime analysis. Identify the main reasons for the good or bad performance of this program.

1. Measure this program using SE30.

We recommend starting with a rough analysis using the default variant followed by a detailed analysis with your own variants.

In which program part is the most runtime caused and where do you think is the most potential for optimization? What are the most expensive table accesses, internal table accesses, subroutines, etc.? (expensive= longest runtime and largest number of executions)

*Continued on next page*

How could you then monitor the database accesses in more detail?

- a) Even the first rough analysis with the evaluation of the hit list returns the main cause of bad performance, namely subroutine GET\_ELIGIBLE\_SALES\_ITEMS

In the group hit list, KNA1 and MARC stick out as the most expensive tables.

When you look at the coding of the routine you will notice that nested selects are used. This put a lot of strain on the system. It causes a large number of fetches on the database, the data packages are not utilized well. To analyze these SQL statements in the expensive subroutine in more detail, you could undertake further analysis with the SQL trace.



## Exercise 4: Code Inspector

### Exercise Objectives

After completing this exercise, you will be able to:

- Identify program parts critical to performance with the Code Inspector tool.

### Business Example

A company operates customer developments and would like to check their own programs for performance-critical areas.

### Task: Working with Code Inspector

The SAPBC490\_TOOL\_SCI\_1 program is specified. It is to be analyzed with the Code Inspector. All checks which relate to performance are relevant.

1. Inspect the SAPBC490\_TOOL\_SCI\_1 program with the Code Inspector. Activate all analysis options under *Performance Checks* under your check variant BC490\_##. The SY-SUBRC handling checks under *Security Checks* should also be activated.

## Solution 4: Code Inspector

### Task: Working with Code Inspector

The SAPBC490\_TOOL\_SCI\_1 program is specified. It is to be analyzed with the Code Inspector. All checks which relate to performance are relevant.

1. Inspect the SAPBC490\_TOOL\_SCI\_1 program with the Code Inspector.  
Activate all analysis options under *Performance Checks* under your check variant BC490\_#. The SY-SUBRC handling checks under *Security Checks* should also be activated.
  - a) Proceed as shown on the slides in the unit. As a result, the Code Inspector should return at least objections to nested loops.  
In the check variants you must activate the performance aspect checks and SY-SUBRC handling under security checks. In this way, you decide which program parts are to be checked in the Inspector.



## Lesson Summary

You should now be able to:

- Determine whether a program causes poor performance due to a database or CPU problem
- Analyze database problems using the SQL trace
- Analyze performance problems with the Code Inspector
- Analyze performance problems with the system trace
- Identify and assess performance problems using runtime analysis



## Unit Summary

You should now be able to:

- Determine whether a program causes poor performance due to a database or CPU problem
- Analyze database problems using the SQL trace
- Analyze performance problems with the Code Inspector
- Analyze performance problems with the system trace
- Identify and assess performance problems using runtime analysis

# Unit 3

## Basics for Indexes

### Unit Overview

#### Contents:



- Using indexes
- Designing indexes
- Looking at the administration of indexes



### Unit Objectives

After completing this unit, you will be able to:

- Optimize SQL statements using indexes
- Explain and implement rules on selectivity and design of indexes

### Unit Contents

Lesson: Basics for Indexes .....	96
Exercise 5: Creating and Using Indexes .....	129

# Lesson: Basics for Indexes

## Lesson Overview

### Contents:



- Using indexes
- Designing indexes
- Looking at the administration of indexes



## Lesson Objectives

After completing this lesson, you will be able to:

- Optimize SQL statements using indexes
- Explain and implement rules on selectivity and design of indexes

## Business Example

A company defines its own database tables in the ABAP Dictionary as part of a customer development. To optimize accesses, the secondary indexes are to be created in this table.



### Motivation: Index Usage

### Architecture and Administration of Indexes

### Using Indexes

### Rules for Designing Indexes

### Hints: Forcing Index Usages

Figure 67: Basics for Indexes (1)

## Motivation: Using Indexes

When formulating a select command, there are two essential issues to be addressed:

- What data needs to be read?
- What search effort is needed for this?

The developer primarily focuses on the first issue. The effort and resulting costs for the database and therefore the entire system are often not taken into consideration when formulating the request. The factor in question is the quantity of data that the database must search through to find the required data.

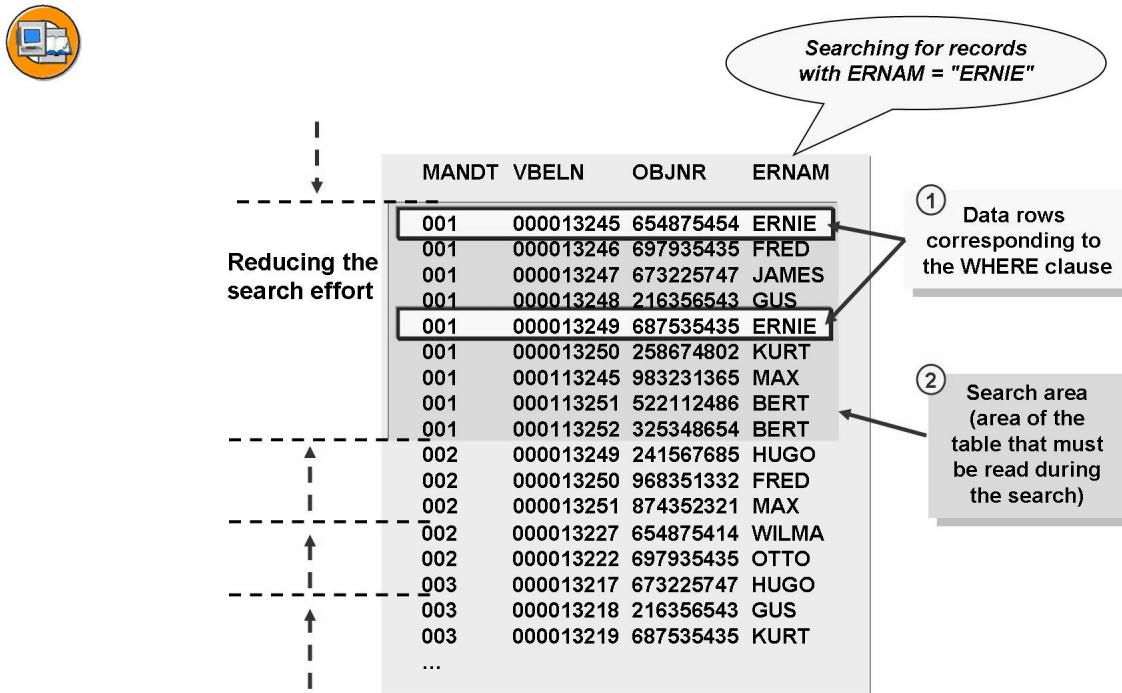
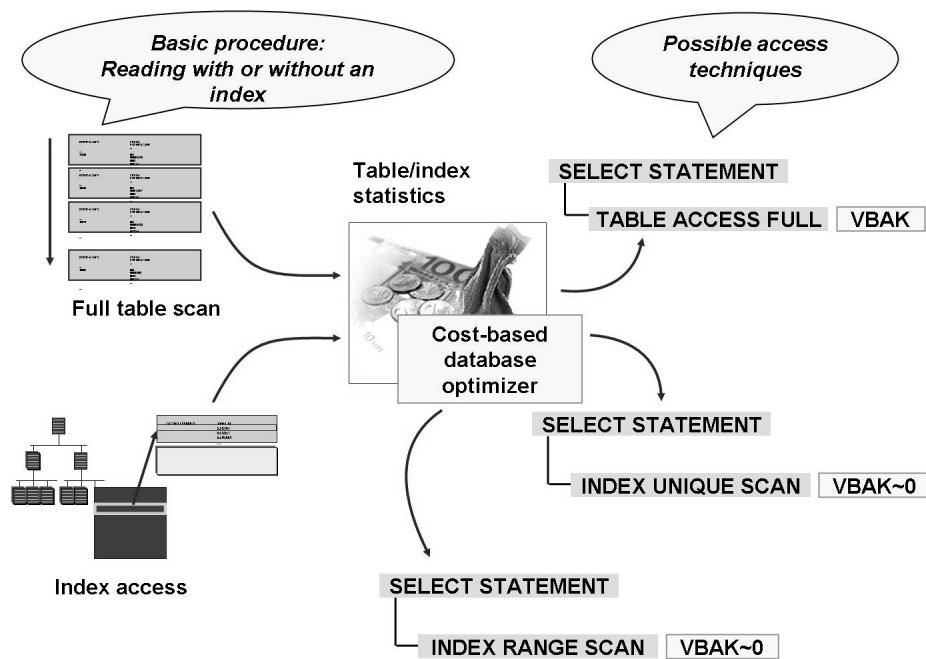


Figure 68: Using Indexes

The search range is the set of data records in a table that has to be checked in order to satisfy an SQL statement. The search range is calculated dynamically when the SQL statement is processed in the database - from the WHERE clause, the database index used, and the selected access strategy.



**Figure 69: The Where Clause and the Database Search Area**

You can reduce the effort required for searching for the database with the following measures:

- Changing the ABAP coding
- Changing the index design
- Using database hints
- Avoiding and solving technical problems

These themes will be discussed later in the course.

For each SQL statement, the database optimizer determines the strategy for accessing data records. The access can take place with (index access) or without (full table scan) the support of database indexes.

The cost-based database optimizer determines the access strategy on the basis of:

- Conditions in the WHERE clause of the SQL statement
- Database indexes of the table(s) affected
- Selectivity of the table fields contained in the database indexes
- Size of the table(s) affected

The table and index statistics supply information about the selectivity of table fields, the selectivity of combinations of table fields, and table size.

Before a database access is performed, the database optimizer cannot calculate the exact cost of a database access. It uses the information described above to estimate the cost of the database access.

The optimization calculation is the amount by which the data blocks to be read (logical read accesses) can be reduced. Data blocks show the level of detail in which data is written to the hard disk or read from the hard disk.



## Motivation

## Architecture and Administration of Indexes

## Using Indexes

## Rules for Designing Indexes

## Hints: Forcing Index Usages

Figure 70: Basics for Indexes (2)

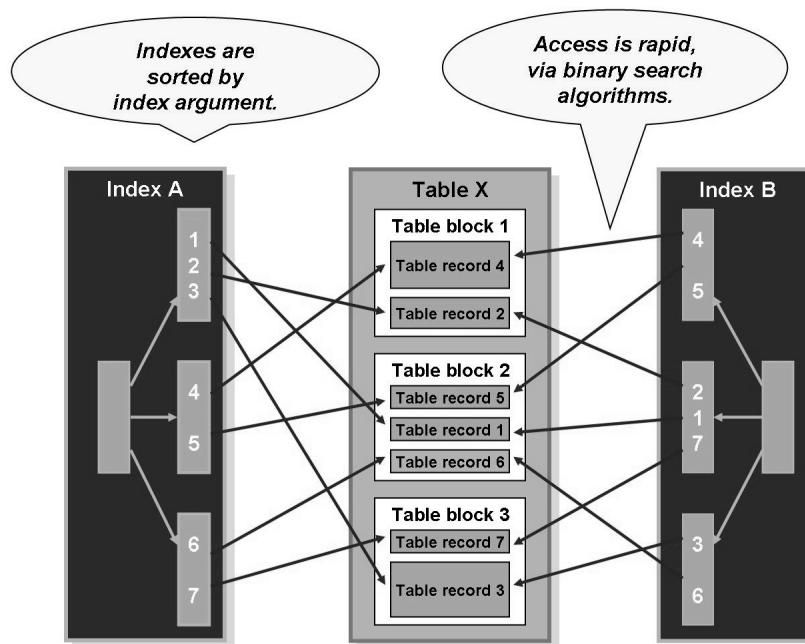
## Architecture of Indexes

### Different architecture of indexes

When you create a database table in the ABAP Dictionary, you must specify the combination of fields that enable an entry within the table to be clearly identified. Position these fields at the top of the table field list, and define them as key fields.

When the table is activated, an index formed from all key fields is created on the database, in addition to the table itself. This index is called a primary index. The primary index is unique by definition.

As well as the primary index, you can define one or more secondary indexes for a table in the ABAP Dictionary, and create them on the database. Secondary indexes can be either unique or non-unique.



**Figure 71: Example of Index Design in Oracle**

When you create a database table in the ABAP Dictionary, you must specify the combination of fields that enable an entry within the table to be clearly identified. Position these fields at the top of the table field list, and define them as key fields.

When the table is activated, an index formed from all key fields is created on the database, next to the table itself. This index is called a primary index. The primary index is unique by definition.

As well as the primary index, you can define one or more secondary indexes for a table in the ABAP Dictionary, and create them on the database. Secondary indexes can be either unique or non-unique.

The example above shows two unique indexes (Index A and Index B) pointing to table X. Both index records and table records are organized in data blocks.

If you dispatch an SQL statement from an ABAP program to the database, the program searches for the data records requested either in the database table itself (full table scan) or by using an index (index unique scan or index range scan). If all the requested fields are contained in the index, the table itself does not have to be accessed.

A data block shows the level of detail in which data is written to the hard disk or read from the hard disk. Data blocks can contain several data records; conversely, one record can also stretch over several data blocks.

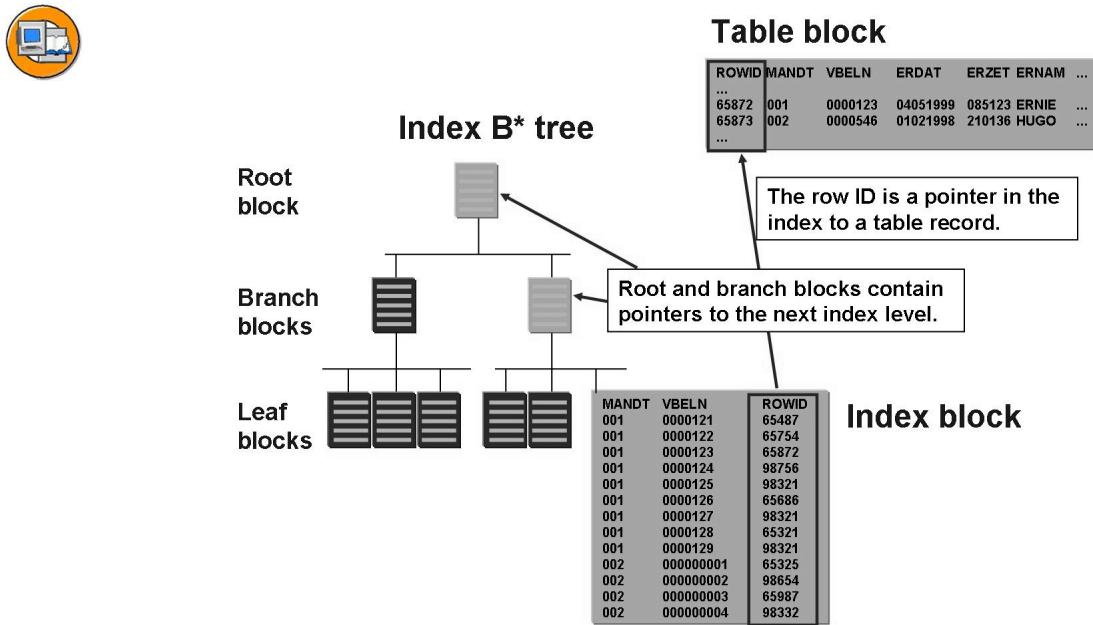


Figure 72: Example of an Oracle Index Access: Unique Scan

Data blocks can be index blocks or table blocks. The database organizes the index blocks as a multilevel B\* tree. The root level contains a single index block that contains pointers to the index blocks at branch level. The branch blocks either contain part of the index fields and pointers to the index blocks at leaf level, or all the index fields and a pointer to the table records organized in the table blocks. The index blocks at leaf level contain all index fields and pointers to the table records from the table blocks.

The pointer, which identifies a table record, is called *ROWID* in Oracle databases. The *ROWID* consists of the number of the database file, the number of the table block, and the row number within the table block.

The index records are saved in the index tree sorted by index field. This is what makes index-based access faster in the first place. The table records in the table blocks are not sorted.

An index should not contain too many fields. Using only a few, carefully selected fields will promote reusability and reduce the possibility that the database optimizer will select a processing-intensive access path.

## Access Strategies for Tables



- Index Unique Scan
  - Unique index is fully specified using AND after the 'equals' sign
- Index Range Scan
  - Index is not fully specified or not unique
- Full table scan
  - No index is used
  - All table blocks are read
- Concatenation
  - Indexes are used more than once
  - Used with OR or IN operation

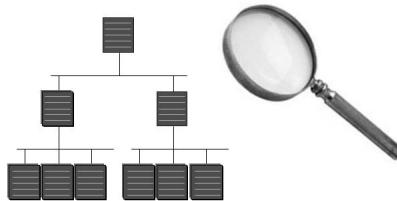
**Index unique scan:** The index selected is unique (primary index or unique secondary index) and fully specified. One or no table record is returned. This type of access is very effective, because a maximum of four data blocks needs to be read.

**Index range scan:** The index selected is unique or non-unique. For a non-unique index, this means that not all index fields are specified in the WHERE clause. A range of the index is read and checked. An *index range scan* may not be as effective as a *full table scan*. The table records returned can range from none to all.

**Full Table Scan:** The whole table is read sequentially. Each table block is read once. Since no index is used, no index blocks are read. The table records returned can range from none to all.

**Concatenation:** An index is used more than once. Various areas of the index are read and checked. To ensure that the application receives each table record only once, the search results are concatenated to eliminate duplicate entries. The table records returned can range from none to all.

## Administration of Indexes



- The administrator's tasks also include monitoring indexes.
- Possible technical problems with indexes:
  - SAP and database system parameterized wrongly
  - rsdb/max\_blocking\_factor as an example
  - Communication problems
  - Remedyng and preventing fragmenting of indexes
  - SAP provides GoingLive and EarlyWatch services for this purpose.

Figure 73: Technical Monitoring of Indexes

Optimizing SQL statements only makes sense if the SAP system and database system are correctly configured. This is the responsibility of the SAP system administrator and the database administrator respectively. Technical problems may have an effect on the whole system (for example, wrongly set parameters, communication problems, old table and index statistics), or may only affect SQL statements on certain tables (for example, missing or fragmented indexes).

To avoid technical problems: Follow the SAP installation instructions and refer to corresponding Notes or implement the recommendations given by the SAP standard services GoingLive and EarlyWatch. Communication problems may occur between the application server and database server (for example, TCP No Delay problem with Oracle databases, see Note 72638). Parameters on the database interface may be wrongly set, and thus also cause problems (for example, incorrect value for rsdb/max\_blocking\_factor, see the slides on SELECT FOR ALL ENTRIES).

On some databases (for example, *Oracle* and *Informix*), *fragmented indexes* may occur. The term fragmented indexes means that index blocks are less than 50% full. Fragmented indexes can occur, for example, if a large number of table records are deleted and then a large number of table records are inserted. If you discover that an index is suitable for an SQL statement, but that executing the SQL statement causes heavy database system load, your database administrator should check whether a fragmented index (database-specific) is causing the problem.

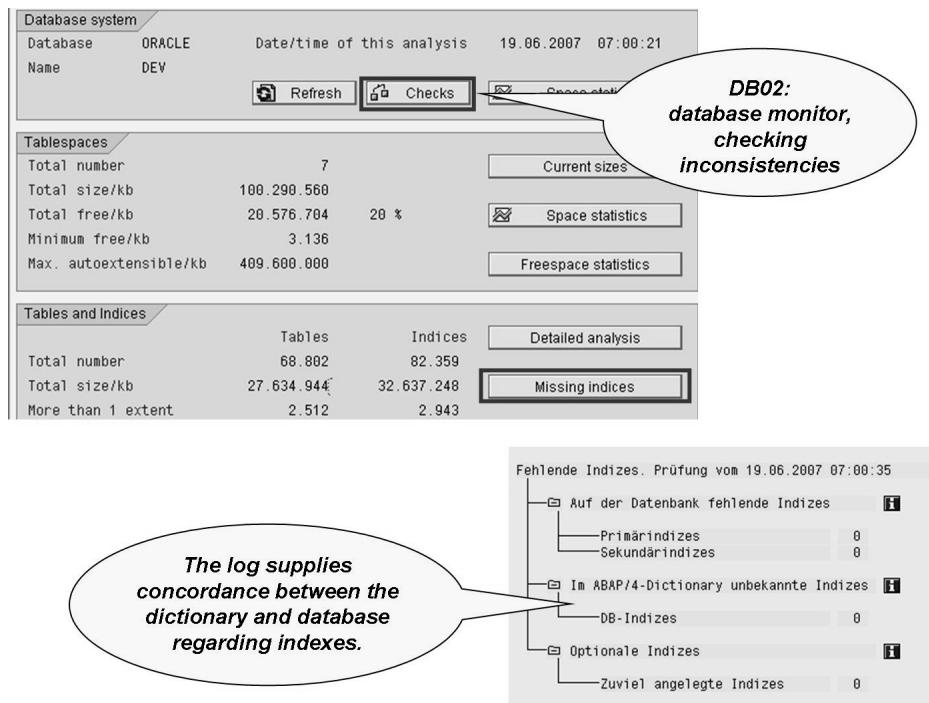


Figure 74: Administration - Monitoring Tables and Indexes

Database reorganization or deliberately deleting database objects can cause inconsistencies between the indexes in the ABAP Dictionary and the indexes on the database.

You can find these inconsistencies by using the database monitor. You reach the corresponding screen using transaction code *DB02* (DB02OLD as of SAP NW 7.0 SP12). For an overview of indexes that are missing in the database or not found in the ABAP Dictionary, choose *Missing indices*.

If you find indexes that are missing on the database, inform your database administrator. Database indexes can be activated from this screen or from the ABAP Dictionary, and thus created on the database. Database indexes that exist on the database but not in the ABAP Dictionary can be created in the ABAP Dictionary. They are transferred to the database when they are activated. The index that already exists on the database is then overwritten. After you have changed the index design for a table, you must create new index statistics - at least for the changed indexes.



**Hint:** Only make changes to database indexes at times of low system activity, because the database locks the database table while the indexes are being created.

To ensure that the cost-based database optimizer functions properly, the table and index statistics must be updated regularly. The database optimizer uses these statistics as a basis for selecting the access strategy used to execute a particular SQL statement.

The database administrator must update the table and index statistics regularly. To find out when the last update was performed, choose *Tools -> CCMS -> DB Administration -> Planning Calendar* or transaction code *DB13*. You must ensure that the last update was not performed more than a week ago, and that it was performed successfully (refer to the log).

When changing the index design, the following procedure should be implemented: The ABAP programmer designs the database indexes to support his/her SQL statements. In a production system or test system, only the database administrator should be responsible for creating database indexes. The database administrator must verify whether the programmer's index design is correct.

When database indexes are created, the table and index statistics must be updated for the relevant tables. If the index design is correct, you can transport the database index to the production system at a time of low system activity, and then update the table and index statistics in the production system.



#### Motivation

#### Architecture and Administration of Indexes

#### Using Indexes

#### Rules for Designing Indexes

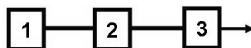
#### Hints: Forcing Index Usages

Figure 75: Basics for Indexes (3)

## Using Indexes



**Index build**



field1	field2	field3	field4	field5	field6	field7	...
...							
710	Ab	12	abc...	km	a	100	
710	Ab	13	def...	mi	b	200	
710	Ab	14	ijk...	m	a	110	
710	Bb	12	text...	mi	c	080	
711	Ab	11	yxc...	km	c	130	
711	Ab	12	ghj...	m	a	100	
712	Bc	12	ert...	km	b	170	
...							

```
SELECT ... FROM dbtable
  INTO TABLE itab
 WHERE field1 = '710'
   AND field2 = 'Ab'
   AND field6 = 'a'
```

*Qualifying index fields  
from left to right  
benefits the use  
of an index.*

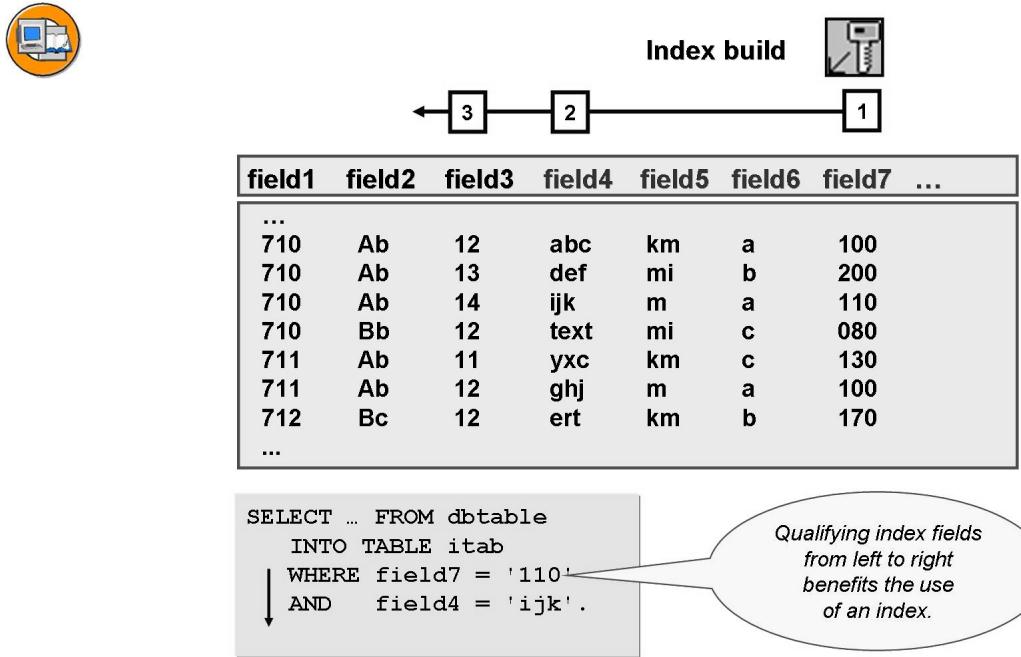
Figure 76: Index Usage Graphic – Example 1

The way in which the database optimizer selects and uses database indexes depends partly on the WHERE conditions specified in the SQL statement. The order of the WHERE conditions in the SQL statement is not important (note that this is not the case with the order of the fields in the database index).

The index search string is based on the fields that are specified without gaps and with '=' in the WHERE condition. The index blocks to be checked are selected using the **index search string**.



**Note:** The sequence of fields in the WHERE clause does not affect index usage, however to ensure that the software can be easily read and maintained, it makes sense to program the fields in the sequence that they are defined in the table.



**Figure 77: Index Usage Graphic – Example 1**

When qualifying index fields it is essential that the first fields in the index are qualified. Gaps later on in the index search string are not as critical as gaps towards the front.



**Caution:** Always observe the sequence of the fields in the index.

The index fields are always defined from top to bottom in the Data Dictionary (SE11). The top fields correspond to the fields further towards the front and the bottom fields to those towards the back of the index.



**Note:** The following lesson will give a few rules of thumb. These statements should be understood as guidelines; they are intended as a rough orientation for the developer. You should always consult user manuals or the specifications of the database manufacturer for exact, reliable information.



```
SELECT * FROM VBAK WHERE VBELN = '0000123'. . . ENDSELECT.
```

Unique index (primary or secondary index)

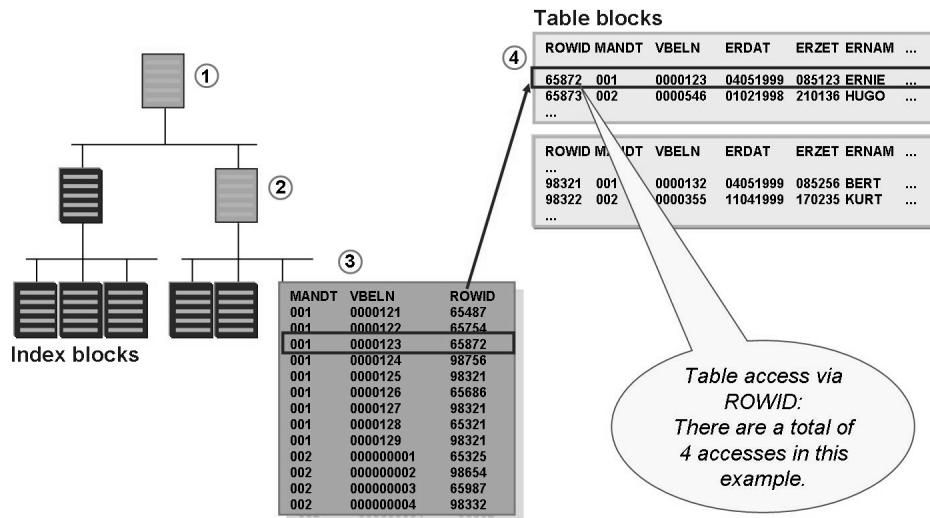


Figure 78: Introduction to DB Indexes: Index Unique Scan

If, for all fields in a unique index (primary index or unique secondary index), WHERE conditions are specified with '=' in the WHERE clause, the database optimizer selects the access strategy *index unique scan*.

For the *index unique scan* access strategy, the database usually needs to read a maximum of four data blocks (three index blocks and one table block) to access the table record.

In the SELECT statement shown above, the table VBAK is accessed. The fields MANDT and VBELN form the primary key, and are specified with '=' in the WHERE clause. The database optimizer therefore selects the *Index Unique Scan* access strategy, and only needs to read four data blocks to find the table record requested.

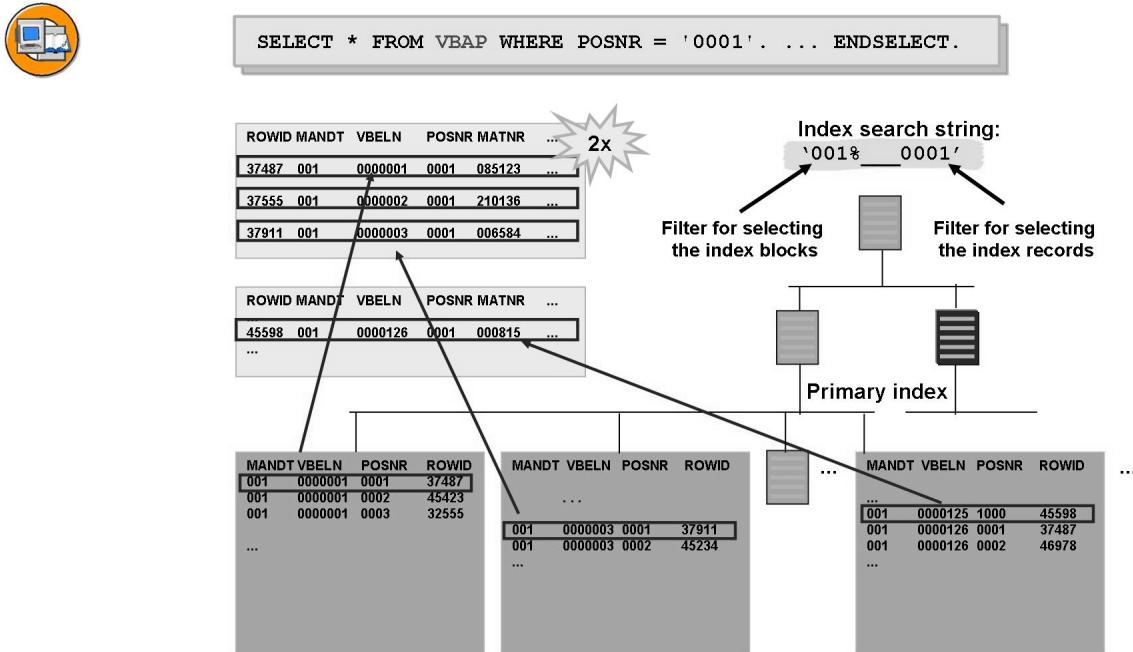


Figure 79: Introduction to DB Indexes – Index Range Scan II

In this example, an *index range scan* is performed for table VVBAP, using the primary index. Because no WHERE condition is specified for field VBELN in the SQL statement, the corresponding places for VBELN are set to "\_" in the index search string.

Therefore, the fully specified area that is used to select the index blocks only consists of the MANDT field. Because many index records fulfill the condition MANDT = "001", many index blocks are read and their index records are checked. From the set of index records, all those that fulfill the condition POSNR="0001" are filtered out. The relevant index records point to the table records.

If the SQL statement contains an additional WHERE condition for a field that is not contained in the index, this is not evaluated until after the table have been read from the table blocks.



```
SELECT * FROM VBAP
  WHERE VBELN IN ('0000123', '0000133', '000143').
ENDSELECT.
```

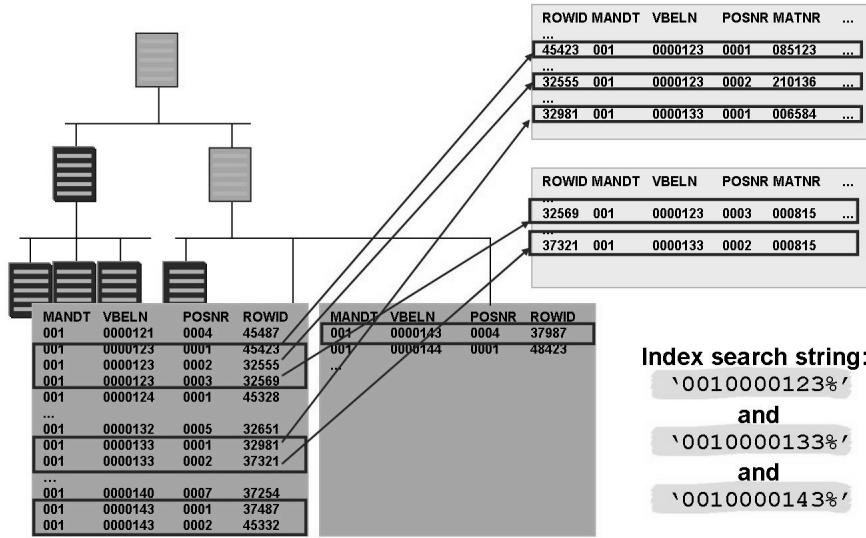


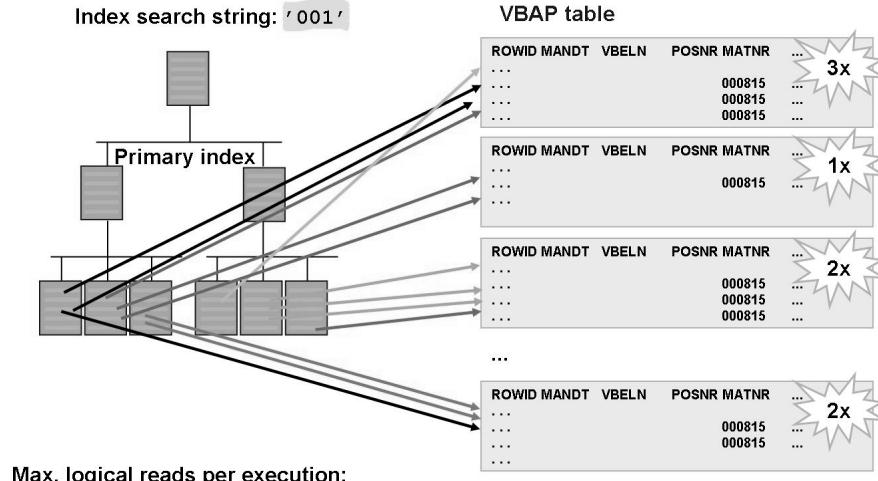
Figure 80: Introduction to Database Indexes – Concatenation

In the *concatenation* access strategy, one index is reused repeatedly. Therefore, various index search strings also exist. An *index unique scan* or an *index range scan* can be performed for the various index search strings. Duplicate entries in the results set are filtered out when *Concatenation* is performed for the search results.

In the SQL statement above, a WHERE condition with an IN operation is specified over field VBELN. The fields MANID and VBELN are shown on the left of the primary index. Various index search strings are created, and an *index range scan* is performed over the primary index for each index search string. Finally, *concatenation* is performed for the result.



```
SELECT * FROM VBAP WHERE MATNR = '000815'. . . ENDSELECT.
```



**Figure 81: Introduction to DB Indexes – Index Range Scan III**

Due to old update statistics, database optimizer errors, missing indexes, or inappropriate ABAP coding, the database optimizer may select a completely unsuitable index, and then perform an unselective *index range scan*.

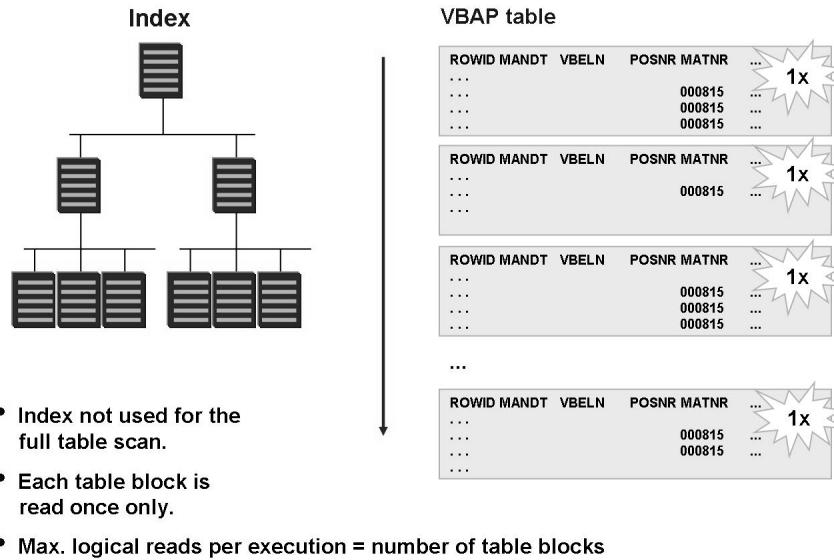
In the example above, WHERE conditions were only specified for MANDT and MATNR. In spite of this, the database optimizer chooses to perform an *index range scan* over the primary index (MANDT, VBELN, POSNR). Since there is only one unselective WHERE condition (MANDT = '001') to select the index blocks (due to the WHERE clause), a large number of index blocks are read. There is no further WHERE condition to filter the index records. The table records corresponding to all suitable index records are therefore read.

More data blocks are read than if the table is read sequentially, if an unselective *index range scan* is performed. This is because a table block is read more than once if index records in different index blocks point to the same table record. The maximum number of data blocks to be read per SQL statement execution is thus calculated from the product of the number of table blocks and the number of index blocks.

All index blocks and table blocks read during an index range scan are stored in the data buffer at the top of a LRU (least recently used) list. This can lead to many other data blocks being forced out of the data buffer. Consequently, more physical read accesses become necessary when other SQL statements are executed.



```
SELECT * FROM VBAP WHERE MATNR = '000815'. . . ENDSELECT.
```



**Figure 82: Full Table Scan**

If the database optimizer selects the *full table scan* access strategy, the table is read sequentially. Index blocks do not need to be read.

For a *full table scan*, the read table blocks are added to the end of an LRU list. Therefore, no data blocks are forced out of the data buffer. As a result, in order to process a *full table scan*, comparatively little memory space is required within the data buffer.

The *full table scan* access strategy becomes ever more effective the more data there is to be read in the table (for example, more than 5% of all table records). In the example above, a *full table scan* is more efficient than access using the primary index.

## Refining Index Accesses



```
SELECT bukrs belnr gjahr
FROM bkpf
INTO TABLE gt_bkpf
WHERE belnr = '0000000100'.
```

Spec. index fields: MANDT, BELNR  
=> gap in index search string  
Index search string: '001 \_\_\_\_0000000100'



### Ineffective use of index

```
SELECT bukrs belnr gjahr
FROM bkpf
INTO TABLE gt_bkpfp
WHERE bukrs = '0001'
AND belnr = '0000000100'.
```

Spec. index fields: MANDT, BUKRS, BELNR  
=> index fields specified fully  
Index search string: '00100010000000100'

### Effective use of index

**Figure 83: Missing WHERE Conditions (1)**

The way in which the database optimizer selects and uses database indexes depends partly on the WHERE conditions specified in the SQL statement. The order of the WHERE conditions in the SQL statement is not important (note that this is not the case with the order of the fields in the database index).

The index search string is based on the fields that are specified without gaps and with '=' in the WHERE condition. The index blocks to be checked are selected using the index search string.

In the top example, a WHERE condition is not specified for field BUKRS (second field in primary index). The index search string over the primary index, which selects index blocks to be checked, is therefore not selective.

In the second example, all fields in the index are specified without gaps and with WHERE conditions with '='. Therefore, only a few index blocks need to be checked. The index search string is selective.



```
SELECT vbeln erdat FROM vbak
  CLIENT SPECIFIED
  INTO TABLE gt_vbak
 WHERE vbeln = '0000000100'.
```

Spec. index fields: VBELN  
=> index fields not specified  
Index search string: '\_\_0000000100'



## Ineffective use of index

```
SELECT vbeln erdat FROM vbak
  INTO TABLE gt_vbak
 WHERE vbeln = '0000000100'.
```

```
SELECT vbeln erdat FROM vbak
  CLIENT SPECIFIED
  INTO TABLE gt_vbak
 WHERE mandt = '490'
 AND vbeln = '0000000100'.
```

Spec. index fields: MANDT, VBELN  
=> index fields specified fully  
Index search string: '0010000000100'

## Effective use of index

**Figure 84: Missing WHERE Conditions II**

Incorrect use of the addition CLIENT SPECIFIED can lead to ineffective use of a database index.

In the top example, the addition CLIENT SPECIFIED blocks the effect of the WHERE condition at the database interface, and thus no WHERE condition for field MANDT is stated in the SQL statement.

As a rule, a full table scan of the database is performed as the index cannot be used. With IOTs (SQL Server) the primary index appears in the execution plan nevertheless.

In the second example, the WHERE conditions for field MANDT are incorporated in the SQL statement. The use of an index is therefore effective



```
SELECT vbeln erdat kunnr FROM vbak
INTO TABLE gt_vbak
WHERE kunnr in so_kunnr
AND NOT auart IN ('TA', 'K
                   'ZBV', 'I')
```



**Spec. index fields: MANDT, KUNNR and  
AUART => ineffective use of indexes  
due to negative formulation**

## Negative formulation

```
SELECT vbeln erdat kunnr FROM vbak
INTO TABLE gt_vbak
WHERE kunnr in so_kunnr
AND auart IN ('BV', 'WV', 'I
               'MV', 'KM', 'I')
```

**Spec. index fields: MANDT, KUNNR, and  
AUART => effective use of indexes  
due to positive formulation**

## Positive formulation

**Figure 85: Critical Operators (NOT and <> ) (1)**

If fields in the WHERE clause are specified with operators NOT or <>, these WHERE conditions cannot be used for a binary search over a database index. You should therefore formulate SQL statements positively wherever possible (applies to most DBs).

If a positive formulation cannot be used, for example because the IN list would be too long, you should nevertheless specify the WHERE condition with NOT, to reduce the amount of data to be transferred.

In both of the examples above, an index range scan is performed over the index consisting of the fields MANDT, KUNNR and AUART.

For the search over the index in the first example, only the fields MANDT and KUNNR can be used (the index search string is: '0010000000050\_\_'). In Explain SQL, you therefore see an index range scan.

However, in the second example, the fields MANDT, KUNNR and AUART can be used for the search over the index (index search string: '0010000000050BV' & '0010000000050WV', and so on). You therefore see a concatenation of index range scans in Explain SQL.



```
SELECT * FROM vbak INTO TABLE gt_vbak
WHERE vbeln BETWEEN '0000005001' AND '0000005005'.
```



## BETWEEN operator

```
SELECT * FROM vbak INTO TABLE gt_vbak
WHERE vbeln IN ('0000005001', '0000005002', '0000005003',
'0000005004', '0000005005').
```

## IN operator

WHERE <f> LIKE <value>  
replace with  
WHERE <f> = <value>

WHERE <f> LIKE '%'  
leave out  
entirely

**Figure 86: Critical Operators (BETWEEN, LIKE, > and <) III**

On ORACLE databases, a WHERE condition with BETWEEN is evaluated with the costs for reading 5 % of all index blocks and table blocks, regardless of the size of the BETWEEN interval. If the BETWEEN interval is sufficiently small, then you can replace WHERE <field> BETWEEN <value 1> AND <value 2> by WHERE <field> IN (<value 1>, ..., <value n>), or alternatively by WHERE <field> IN <range\_tab>.

In the first example of the WHERE condition with BETWEEN, the index must be searched sequentially. In the second example, with IN in the WHERE condition, the index can be used more efficiently (this is known as concatenation in Oracle).

A WHERE condition with LIKE, > (GT) or < (LT) is rated by ORACLE (often regardless of the Release) with 10% of the costs for a full table scan. Therefore, use LIKE sparingly and only where necessary. Replace SQL statements of the type SELECT ... WHERE <field> LIKE '999' (field is fully specified) by SELECT ... WHERE <field> = '999'. You can omit WHERE conditions of the type <field> LIKE '%'.

A field specified with LIKE can narrow the selection of database indexes and be used for a search over a database index only if it does not begin with the wildcard character '\_' or '%'. In ABAP, the wildcard '+' is used for any character, and '\*' is used for any character string. For database accesses, however, the characters '\_' and '%' are used.



```
SELECT vbeln erdat kunnr FROM vbak
INTO TABLE gt_vbak
WHERE vbeln LIKE '000000511%' OR vbeln LIKE '000000512%'
OR vbeln LIKE '000000513%' OR vbeln LIKE '000000514%'
```



## Full table scan?

1 SELECT vbeln erdat kunnr FROM vbak  
INTO TABLE gt\_vbak  
WHERE vbeln LIKE '000000511%' OR vbeln LIKE '000000512%'

---

2 SELECT vbeln erdat kunnr FROM vbak  
APPENDING TABLE gt\_vbak  
WHERE vbeln LIKE '000000513%' OR vbeln LIKE '000000514%'



## Index range scan?



- Idea for causing use of index
- Needs to be tested thoroughly
- Will depend heavily on the optimizer
- Comment on this kind of special procedure!

**Figure 87: Critical Operators (BETWEEN, LIKE, > and <) II**

The example above illustrates the possible optimizer strategies and provides food for thought. Whether these results would be replicated exactly in practice is not important here.

WHERE conditions with critical operators that are linked to the WHERE clause with OR can be particularly problematic. Since BETWEEN is calculated as costing 5% and LIKE is calculated as costing 10% (of course, this depends on the DB and release in use), the cost-based database optimizer opts for a *full table scan* after a certain number of corresponding WHERE conditions, instead of using an index.

In the first example, WHERE conditions are linked with LIKE by OR. The database optimizer calculates the cost of each of these WHERE conditions as being 10% of the cost of a *full table scan*. Because the sum of the costs for the WHERE conditions is large, the database optimizer opts for a *full table scan*.

In the second example, the SQL statement is split into several statements, and an index range scan is performed for each. The results are grouped together by APPENDING TABLE. If you use this procedure, you must also ensure that any duplicate entries in the internal table are eliminated.

## Designing Indexes – Selectivity



Motivation

Architecture and Administration of Indexes

Using Indexes



Rules for Designing Indexes

Hints: Forcing Index Usages

Figure 88: Basics for Indexes (4)

### Rules for Designing Indexes



- General Rules
  - Use disjunct indexes
  - As few indexes as possible per table (as few as possible, but as many as necessary, recommendation: 5-7)
- Selection of index fields
  - As few fields as possible in index
  - Fields that are as selective as possible
  - Selective fields as near to the beginning as possible
- Do not change SAP standard indexes
  - Unless recommended by SAP

A rule of thumb states that an index only makes sense if SQL statements that use the index return less than 5% of the table records. The index fields must therefore significantly reduce the resulting sets. Otherwise, the database optimizer would perform a *full table scan* anyway.

Indexes should differ from one another in terms of the sequence and structure of fields, they should be disjunct.

To keep additional work for the database to a minimum, create as few indexes as possible for each table (approximately 5 indexes per table).

As a general rule, an index should consist of a maximum of 4 fields. If too many fields are specified in an index, additional work is created every time a database operation is performed, and the memory space required grows accordingly. Consequently, the index becomes less effective and the probability of it being selected is reduced.

The selective fields in an index should be as near to the beginning of the index as possible (see *index range scan* access strategy). Selective fields are, for example, document number, material number, and customer number. Unselective fields are, for example, client, company code, header account, and plant.

Do not change SAP standard indexes unless SAP explicitly recommends that you do so.



### Selectivity Analysis: Semantic Typing

Type of field	Example	Selectivity
Customer number identifier	KNA1-KUNNR	++
Organizational unit sales organization	VBAK-VKORG	--
Status delivery status	VBUK-LFSTK	- (+)
Classifier order type	VBAK-AUART	--
Date and time entry date	KNA1-ERDAT	+
Text field name	KNA1-NAME1	+ (-)

Before you perform a technical selectivity analysis, you must be sure about what the various index fields in question mean. You should therefore type index fields according to their meaning.

**Identifiers:** These are particularly selective table fields, as they are usually characterized by consecutive numbers (document number, object number, and so on), for example, because they are taken from a number range object.

**Organizational units:** These are fields such as sales organization, company code, or distribution channel. They are often very unselective, and should only be used in secondary indexes in exceptional circumstances.

**Status fields:** (+) These can be very selective if, in an SQL statement, values are selected where only a few corresponding data records exist in the table (for example, open orders, if most of them are complete). Conversely, they can also be very unselective (-).

**Classifiers:** These are fields where typing is performed (for example, sales order, planned order, and production order). In general, classifiers are not selective, as few different versions exist, and they are usually distributed relatively evenly.

**Date and time:** These are often selective, and can be used in a secondary index.

**Text fields:** These are generally selective (+), but they are also very long. They should therefore not be used in a secondary index, because it would become too wide (-) and therefore take up too much memory space.

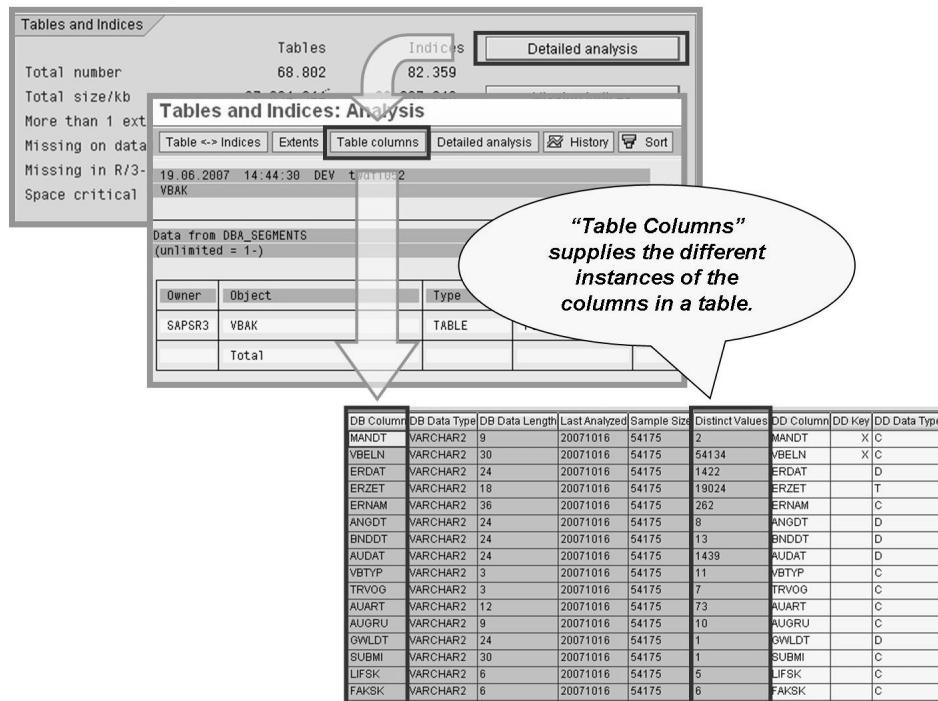


Figure 89: Many Differing Field Contents – High Selectivity (DB02)

When you are sure about what the index fields mean (that is, their semantics), check how many distinct values there are for each index field. Distinct values are the number different values per field in a particular database table.

The relationship between the total number of data records in a table and the distinct values per field indicates how selective the index field is.

When the table and index statistics are updated, the distinct values are re-calculated. For index fields, the number of distinct values can be determined using Explain SQL (DB SQL cache or SQL trace).

For an overview of the distinct values for all fields of a database table, select transaction code *DB02*. Then choose *Detailed analysis*. In the dialog box that appears, enter the table name under *Object name*, and confirm. In the next screen, choose *Table columns* (the path depends on the release).



### Histograms for Combinations of Index Fields

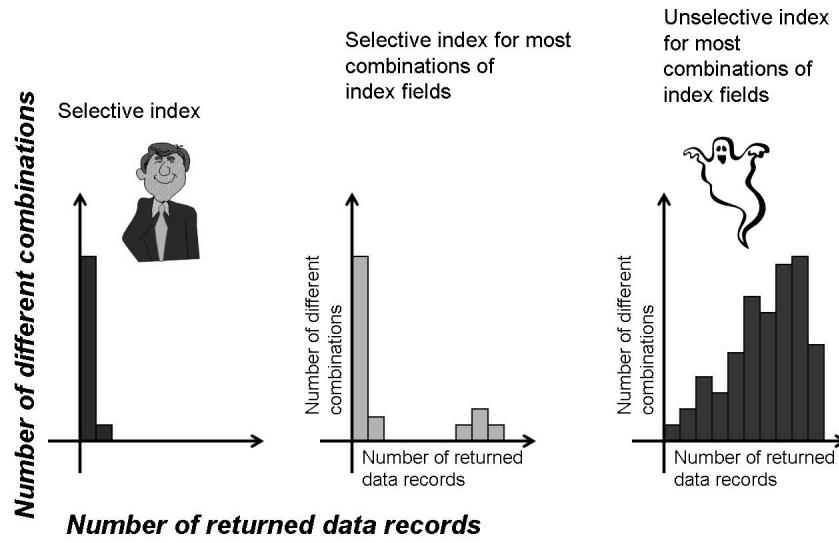


Figure 90: Selectivity Analysis: Histograms (DB05)

The number of data records that are returned for each combination of index fields shows how selective an index is. This means that, if only a few data records are returned for a large number of index field combinations, an index is selective.

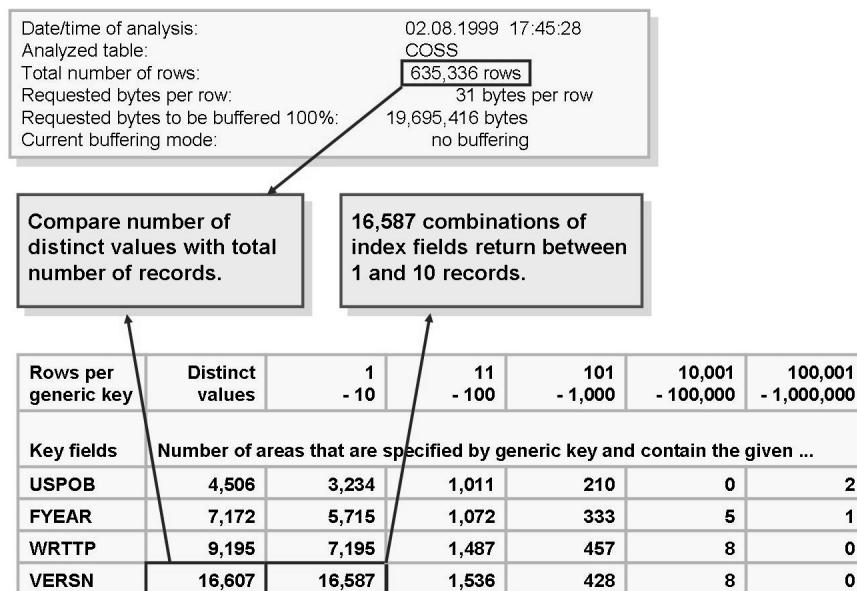
For an overview of how selective an index is over all index field combinations, a histogram can be calculated. Calculating a histogram is very expensive, because the number of data records returned must be checked for all index field combinations over the whole table.

The histogram on the left shows a selective index for all index field combinations. **This index can be created without any problems.**

The middle histogram shows that few data records are returned for most index field combinations. However, there are also some combinations for which a large number of data records are returned. This index can lead to ineffective accesses.

The histogram on the right shows an unselective index. A large number of data records are returned for almost all index field combinations. **This index should not be created.**

→ **Note:** The selectivity analysis (DB05) is a tool for constructing indexes; if the tool provides an apparently useful index, it remains at the discretion of the developer whether the index is actually created. The tool is to be regarded as an auxiliary tool. Before actually creating the index, it is recommended in all cases to contact other developers and the administrator to discuss the intended process. Just like a table, an index also requires memory space and must be maintained.



**Figure 91: DB05 - Example of a Selectivity Analysis I**

To perform a selectivity analysis, use Transaction *DB05*. In the initial screen of DB05, enter a table name, select an analysis for primary key or for specified fields, specify the index fields if applicable, and submit the analysis in background or dialog mode.

In the example above, there are 4,506 different values for the field USPOB, 7,172 values for the combination of USPOB and GJAHR, 9,195 different values for the combination of USPOB, GJAHR and WRTTP, and 16,607 values for all index field combinations.

If quotients are calculated from the total number of table records (635,336) and the number of distinct values for the index fields (16,607), the results show that an average of 30 data records are returned when the index is used.

If the number of distinct values does not increase from one index field to the next index field, this index field is not selective. It should therefore be omitted.

In the other columns in the table above, you can see how many index field combinations fall into other categories. For example, between 1 and 10 data records are returned for 16,587 index field combinations. For a selective index, this number is similar to the number of distinct values.

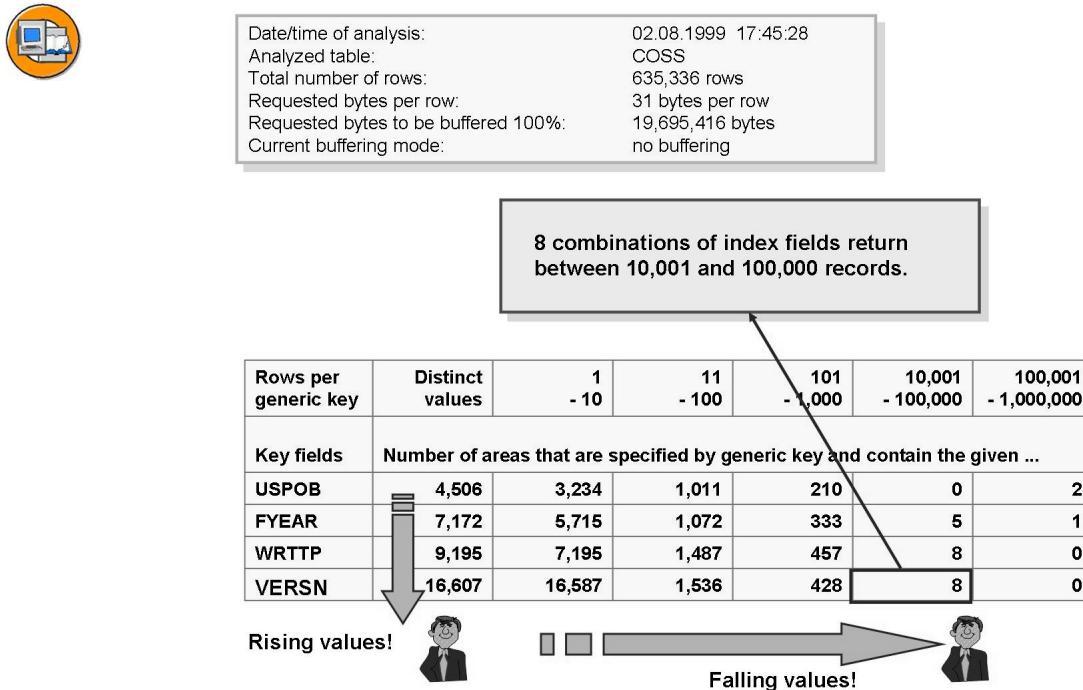


Figure 92: DB05 - Example of a Selectivity Analysis II

In the example above, we can also see that between 10,001 and 100,000 data records are returned for 8 index field combinations. If an SQL statement with one of these combinations is dispatched, the result is an **unselective index range scan**.

You can decide whether to create the checked secondary index or not, only if you follow the logic behind the table.

Remember that a selectivity analysis is an expensive operation, which should only be performed at times of low system activity.

## Summary: General Rules for Designing Indexes



### Changing the index design?

- SAP programs
  - Searching for notes; contact SAP
  - No changes to SAP tables (especially Basis tables)
  
- Customer-defined programs
  - Avoid indexes on transaction data where possible
  - Change existing indexes where possible
  - Possibly delete indexes



### Rules for index design

- Use disjunct indexes where possible
- As few indexes per table as possible
- As few fields in the index as possible, short fields
- Selective fields in the index, selective fields first

**Figure 93: Rules for Designing and Using Indexes**

If you determine that an expensive SQL statement originates from an SAP program, you should first search for corresponding notes. Do not change an SAP index without due consideration! If you cannot find an applicable SAP Note, create a problem message in SAPNet.

In your own programs, if you find expensive SQL statements on SAP standard tables with master data or transaction data, avoid creating secondary indexes. Transaction data tables usually grow linearly with their indexes. Therefore, a search using a secondary index becomes less effective as time goes on. You should look for alternative methods of access, for example, using index tables or auxiliary tables. Before you create a secondary index to optimize an SQL statement, check whether you can adapt a secondary index that already exists, so that it can be used by the SQL statement. Before changing an index, a selectivity analysis can be performed. In some cases, deleting a secondary index makes sense. For example, if a secondary index

is not used, or if a secondary index is a subset of another index, updating it causes unnecessary work for the database. As a rule, avoid secondary indexes over SAP basis tables (for example, tables NAST, D010, D020, or tables beginning with DD\*).

→ **Note:** A rule of thumb states that an index only makes sense if SQL statements that use the index return less than 5% of the table records.

The index fields must therefore significantly reduce the resulting sets.

Depending on the database setting, the database optimizer will often conduct a full table scan in other cases. Indexes must not be contained in other indexes (that is, they must be disjunct), because the cost-based database optimizer can also select the index with the greater quantity. In addition, do not create indexes that can be selected accidentally. To keep additional work for the database to a minimum, create as few indexes as possible for each table (approximately 5 indexes per table). As a general rule, an index should consist of a maximum of 4 fields. If too many fields are specified in an index, additional work is created every time a database operation is performed, and the memory space required grows accordingly. Consequently, the index becomes less effective and the probability of it being selected is reduced. The selective fields in an index should be as near to the beginning of the index as possible (see Index Range Scan Access Strategy).

Selective fields are, for example, document number, material number, customer number and so on.

Unselective fields are client, company code, main account, plant and so on. Modifications to the SAP standard index should only be made at the explicit recommendation of SAP. (See also OSS Notes / Notes in the Service portal).



#### Motivation

#### Architecture and Administration of Indexes

#### Using Indexes

#### Rules for Designing Indexes

#### Hints: Forcing Index Usages

Figure 94: Basics for Indexes (5)

## Using Database Hints



```
SELECT * FROM vbap INTO TABLE gt_vbap  
WHERE ABGRU IN ('01','02','03','04','05')  
%_HINTS ORACLE 'index(VBAP "VBAP~A")'.
```

### Database hints in OPEN SQL

- Examples of database hints (ORACLE):
  - FULL(Table): Full table scan over one table
  - INDEX(Table): Index range scans with lowest costs
  - INDEX(Table “Index”): Index range scan over one index
  - FIRST\_ROWS: Use the access path that returns the first records as fast as possible.
  - ALL\_ROWS: Use the access path that returns all records as fast as possible.

**Figure 95: Using Database Hints**

If you want to force the database optimizer to choose a particular execution path, use database hints. In SAP Basis releases before R/3-4.5A, database hints can only be used in the respective database-specific SQL dialects (native SQL in the parentheses EXEC SQL. ... ENDEXEC.).

As of R/3 Release 4.5A, you can use database hints directly from OPEN SQL. To do this, add the addition %\_HINTS DBMS ‘DBHINT’ to the end of the SQL statement (see above). Under DBMS, enter the applicable database system (for example, ORACLE, INFORMIX). Under DBHINT, enter the relevant database-specific hint.

If the hint cannot be interpreted by the database system, it is treated as a comment, and does not affect the execution of database statements.

## Database Hints II



- **Disadvantages**
  - Database hints are database-specific
  - Database hints force the database optimizer to make a decision, regardless of:
    - The SQL statement
    - The table and index statistics
- **Supported database systems:**
  - Oracle (see SAP Note 130480, 129385)
  - MS SQL Server (see SAP Note 133381)
  - DB2/6000 - UDB (see SAP Note 150037)
  - Informix (see SAP Note 152913)
  - DB2/390 (see SAP Note 162034)

A database hint should be incorporated into an SQL statement only as a last option (regardless of whether it is formulated in native SQL or OPEN SQL). Database hints are database-specific. If you replace the DBMS, you must change all the SQL statements that have database hints (and document your changes).

Database hints ignore the database optimizer's strategy of dynamically making selections according to table and index growth and the selectivity of fields. Therefore, an index that is good today, could be bad tomorrow.

To find out in which database systems you can formulate database hints from OPEN SQL, refer to Note 140825.



# Exercise 5: Creating and Using Indexes

## Exercise Objectives

After completing this exercise, you will be able to:

- Create indexes
- Verify index accesses
- Describe and apply general rules for index design

## Business Example

Indexes can be used to reduce the search effort in the database.

### Task:

Create a program whose SELECT statement accesses a table using an index.

1. Make a copy of table BC490KNA1 and call it Z##KNA1 (as usual, ## stands for your group number). Activate the target table.
2. Copy the data from the original table to your own table, using program SAPBC490\_INDX\_COPY\_KNA1.
3. Write a new program called Z\_##\_INDEX that accesses the table you just created.

This program should have a selection screen that lets the user enter an area for postcodes (PSTLZ). (Use the SELECT-OPTIONS statement).

Program the SELECT statement to access all the records in table Z##KNA1 that have this postcode as an attribute.

4. Perform an SQL trace for your program and analyze the database access. Write down the times required for the database accesses (statement summary and so on in the trace) and check which index the database uses.
5. If no suitable index exists for your table, create one now in the *ABAP Dictionary*. The index should be selective. Verify that the index improves performance. Compare the access times.

## Solution 5: Creating and Using Indexes

### Task:

Create a program whose SELECT statement accesses a table using an index.

1. Make a copy of table BC490KNA1 and call it Z##KNA1 (as usual, ## stands for your group number). Activate the target table.
  - a) Copy the Repository object as you usually would.
2. Copy the data from the original table to your own table, using program SAPBC490\_INDX\_COPY\_KNA1.
  - a) Open the specified program in the editor and run it directly.
  - b) Specify the name of the transparent table you created in the last step and start the program.
3. Write a new program called Z\_##\_INDEX that accesses the table you just created.

This program should have a selection screen that lets the user enter an area for postcodes (PSTLZ). (Use the SELECT-OPTIONS statement).

Program the SELECT statement to access all the records in table Z##KNA1 that have this postcode as an attribute.

- a) See source code of model solution SAPBC490\_INDX\_SOLUTION
4. Perform an SQL trace for your program and analyze the database access. Write down the times required for the database accesses (statement summary and so on in the trace) and check which index the database uses.
  - a) As usual, use transaction ST05 to perform the trace. Choose *Explain SQL* from within the trace list to display the access details for your table.
5. If no suitable index exists for your table, create one now in the *ABAP Dictionary*. The index should be selective. Verify that the index improves performance. Compare the access times.
  - a) Create a secondary index with the *MANDT* and *PSTLZ* fields.

Don't forget the *MANDT* field, because the client is usually restricted to the logon client in client-specific database tables.

*Continued on next page*

- b) You should detect a performance improvement in the SQL trace (transaction ST05), because creating the selective index enables the database optimizer to determine the hit list from the highly restricted search set.



**Hint:** After you copy the table and create the new indexes, the statistics are not created automatically in all databases. With some databases, you have to trigger this manually from within the Explain function. The exact procedure depends on which database is used. Please ask your instructor if you encounter any problems.



**Hint:** If the user does not evaluate the Select option, the PSTLZ field is not transferred to the DB in the Where clause. The trace documents this! To test the success of your index, you should make an entry for the PSTLZ in your selection screen.

## Result

### Source text of the model solution:

```
*&-----*
*& Report  SAPBC490_INDX_SOLUTION *
*&-----*
*& Read data using secondary index ( if index exists ! )      *
*& Index should be created on : MANDT / PSTLZ                *
*&-----*
REPORT SAPBC490_INDX_SOLUTION
* Data Declaration
DATA: gs TYPE knal,
      gt_knal TYPE STANDARD TABLE OF knal.
SELECT-OPTIONS: selopt FOR gs-pstlz default '0000060000' to '0000069483'.

START-OF-SELECTION.
#####
SELECT kunnr name1 ort01 stras telf1
  FROM knal
  INTO CORRESPONDING FIELDS OF TABLE gt_knal
 WHERE pstlz in selopt.

LOOP AT gt_knal INTO gs_knal.
  WRITE: / gs_knal-kunnr, gs_knal-name1, gs_knal-ort01,
         gs_knal-stras, gs_knal-telf1.
ENDLOOP.
```



## Lesson Summary

You should now be able to:

- Optimize SQL statements using indexes
- Explain and implement rules on selectivity and design of indexes



## Unit Summary

You should now be able to:

- Optimize SQL statements using indexes
- Explain and implement rules on selectivity and design of indexes



# Unit 4

## Accessing Single Tables

### Unit Overview

#### Contents:



- Optimizing the SELECT command
- Using field lists
- Using UP TO N ROWS
- Using aggregate functions
- Using SELECT-OPTIONS / RANGES tables
- Using the ARRAY FETCH technique
- Accessing pooled and cluster tables



### Unit Objectives

After completing this unit, you will be able to:

- Program efficient SELECT commands to access individual tables
- Explain the fundamentals of pooled and cluster tables
- Initiate efficient change accesses

### Unit Contents

Lesson: Accessing Single Tables .....	136
Exercise 6: (Optional) Ranges Tables / Select Options .....	157
Exercise 7: Aggregate Functions.....	161

# Lesson: Accessing Single Tables

## Lesson Overview

### Contents:



- Optimizing the SELECT command
- Using field lists
- Using UP TO N ROWS
- Using aggregate functions
- Using SELECT-OPTIONS / RANGES tables
- Using the ARRAY FETCH technique
- Accessing pooled and cluster tables



## Lesson Objectives

After completing this lesson, you will be able to:

- Program efficient SELECT commands to access individual tables
- Explain the fundamentals of pooled and cluster tables
- Initiate efficient change accesses

## Business Example

An enterprise uses customer developments and wants to initiate high-performance selects from the tables.



**Read Accessing Single Tables**

**Change Accesses to Tables**

**Pooled and Cluster Tables**

**Figure 96: Accessing Single Tables (1)**

## Read Accessing Single Tables

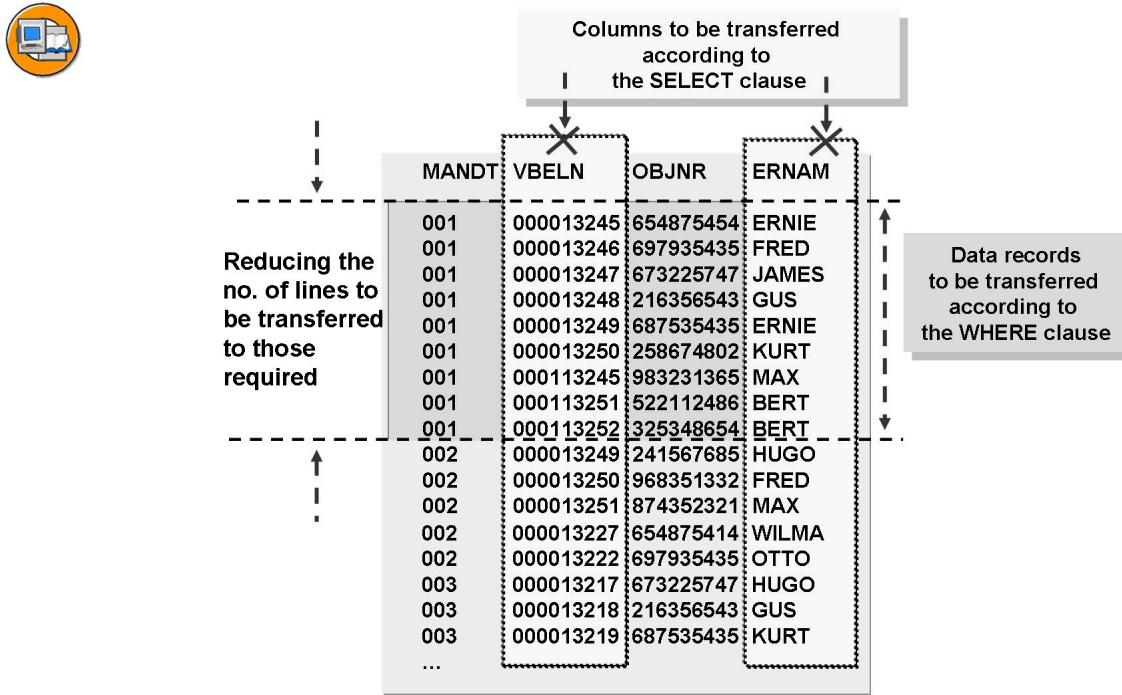


Figure 97: Reducing the Dataset - Field List

If the single object analysis establishes that the SQL statement analyzed has a suitable access path, you can optimize it by:

- Reducing the rows to be transferred
- Reducing the columns to be transferred
- Reducing the data transfer load
- Avoiding unnecessary SQL statements

You can reduce the number of rows to be transferred, for example, by optimizing the WHERE clause or by using aggregate functions.

You can reduce the number of **columns to be transferred**, for example, by formulating a **suitable field list** or, in the case of database changes, by using UPDATE .. SET (change accesses will be discussed later).



```
SELECT ... FROM resb
  INTO TABLE gt_resb.
```



## Missing WHERE clause

```
SELECT ... FROM resb
  INTO TABLE gt_resb
  WHERE matnr IN so_matnr.
```



## Unstable WHERE clause

```
SELECT ... FROM resb
  INTO TABLE gt_resb
  WHERE matnr IN so_matnr
  AND kzeare IS INITIAL.
```



## Stable WHERE clause

**Figure 98: Stable WHERE clause**

To reduce the number of data records that have to be transferred, you have to specify a sufficiently selective WHERE clause in each SQL statement. A SELECT statement without a WHERE condition is an indication of a design error in the program. Especially when selecting transaction tables (such as BKPF, BSEG, COBK, COEP, LIPK, MKPF, VBAK, and VBAP), you have to make sure that the number of selected records remains constant over time.

The first example shows an SQL statement involving the rapidly growing table RESB (RESB can grow to several hundred megabytes in size) without a WHERE condition. Statements like this can have dramatic effects on the overall performance of a system. In general, SQL statements involving master data and transaction data should never be formulated without a selective WHERE condition.

In the second example, a WHERE condition is specified for field MATNR. This field is selective, and thus suitable for restricting the quantity of data selected. Because material numbers are reserved continuously, the number of selected data records will increase successively over time, diminishing the performance of the SQL statement. As such, the WHERE clause will be unstable over time.

In the last example, another WHERE condition is specified for field KZEAR. This field contains the final status of the reservation. Accordingly, this WHERE condition would only select those records from RESB that are relevant for the corresponding material. This WHERE clause will be stable over time.



```

SELECT vbeln erdat kunnr FROM vbak
  INTO gs_vbak.
    CHECK gs_vbak-vbeln GE gv_low
      AND gs_vbak-vbeln LE gv_high.

  APPEND gs_vbak TO gt_vbak.
ENDSELECT.

```



## CHECK condition

```

SELECT vbeln erdat kunnr FROM vbak
  INTO gs_vbak.
    WHERE vbeln BETWEEN gv_low AND gv_high.
  APPEND gs_vbak TO gt_vbak.
ENDSELECT.

```



## WHERE condition

**Figure 99: CHECK - WHERE**

To reduce the amount of data to be transferred, do not formulate conditions for selecting data using a CHECK statement within a SELECT ... ENDSELECT. Instead, formulate conditions as part of a WHERE clause.

In the top example, the records are not checked for validity until they are checked by the program using a CHECK statement. The system has to read the entire contents of table VBAK, transfer them over the network to the database interface, and transfer them into ABAP. In the second example, the DBMS only reads the data that is really required.

Always specify all known conditions in the WHERE clause. Without a WHERE clause, the DBMS cannot use an index to optimize a statement.

To make reusing SQL statements possible, and thus to optimally utilize the DB SQL cache, always adhere to the order of fields specified in the ABAP Dictionary when formulating SELECT clauses and WHERE clauses.



```
SELECT * FROM vbak
  INTO TABLE gt_vbak_all
 WHERE vbeln IN so_vbeln.
```

## SELECT \*



```
SELECT vbeln erdat kunnr FROM vbak
  INTO CORRESPONDING FIELDS
    OF TABLE gt_vbak_fieldlist
 WHERE vbeln IN so_vbeln.
```

## SELECT field list INTO ...



```
SELECT vbeln erdat kunnr FROM vbak
  INTO TABLE gt_vbak_fieldlist
 WHERE vbeln IN so_vbeln.
```

## SELECT field list INTO TABLE



**Figure 100: Reducing the Columns to be Transferred**

When you use `SELECT *`, more data than is necessary is often transferred from the database server to the application server. If you only need the contents of a few columns, you should always specify the table fields individually in a field list.

In the first example, you read all the columns in the table and pass them on to the application server, although only three columns are needed. Accordingly, the transfer effort (number of fetches) increases along with the amount of data transferred.

The `INTO CORRESPONDING FIELDS ...` addition in the second example copies the field contents to target fields with the same names. This copy operation is only slightly more processing-intensive than left-aligned copying into a suitable internal table using `INTO TABLE` (see third example) or specifying the target fields specifically. The same applies to `APPENDING CORRESPONDING FIELDS` and `APPENDING TABLE`.

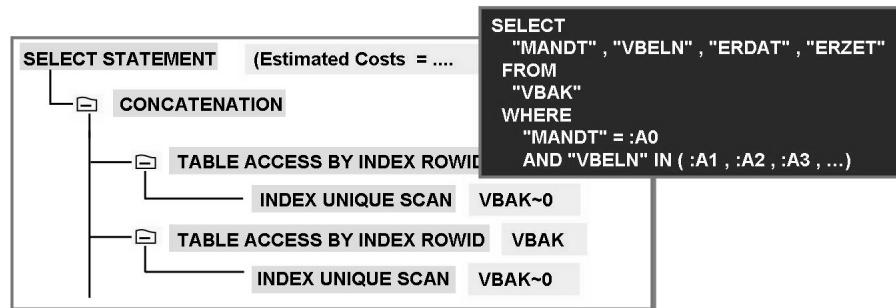
If you need different projections of a database table at several places within a program, it makes sense to read all the required fields from the database at once, buffer them in an internal table, and distribute the data in the program internally.



```
SELECT mandt vbeln erdat erzet
FROM vbak
INTO TABLE gt_vbak
WHERE vbeln IN gt_vbeln.
```



## SELECT with RANGES



**Figure 101: RANGES Tables**

ABAP lets you work with RANGES tables. To declare RANGES, use the ABAP statements SELECT-OPTIONS or RANGES. These two statements implicitly declare an internal table with the following fields SIGN, OPTION, LOW, and HIGH (see documentation). RANGES tables that are declared with SELECT-OPTION are usually filled dynamically on screen by the end user, while RANGES tables that are declared with RANGES are filled by the program.

The database interface fills the individual rows in a RANGES table in a format that the DBMS can interpret and links the rows with OR. The created SQL statement is then passed on to the database.

The RANGES table can contain rows with complex expressions (BT => BETWEEN, CP => LIKE, and so on). Accordingly, the SQL statement that results from a RANGES table can be extremely complex and difficult for the DBMS to process.

If you have a program fill the RANGES tables, you have to make sure that the number of items is limited. If the RANGES table is too large, the analysis of the OR or IN list will result in a complex database statement that will be extremely time-consuming for the database to process. In the extreme case, an ABAP dump can occur if the database-specific maximum statement length (64KB in the 32-bit Oracle version, for example) is exceeded.

→ **Note:** See also Note 13607



```
SELECT vbeln zmeng matnr
FROM vbap
INTO TABLE gt_vbap
WHERE vbeln IN so_vbeln.
```



```
LOOP AT gt_vbap.
* Compute maximum of zmeng and minimum of matnr ...
ENDLOOP.
```

## Aggregate functions in ABAP

```
SELECT vbeln max( zmeng ) min ( matnr )
FROM vbap
INTO TABLE gt_vbap
WHERE vbeln IN so_vbeln
GROUP BY vbeln.
```



## Aggregate functions at DB level

**Figure 102: Aggregate Functions**

You can use aggregate functions (COUNT, SUM, MAX, MIN, AVG) to perform calculations using the DBMS. This is only useful, however, if it significantly reduces the number of data records to transfer.

In the top example, an aggregation is performed in ABAP using control level processing. A data record must be transferred for each document item that fulfills the condition in the WHERE clause. However, in the second example, only one data record needs to be transferred for each document header.

Aggregate functions are usually used together with a GROUP BY clause. SQL statements that contain aggregate functions bypass SAP table buffers. (Note that this can be a disadvantage if tables are buffered.)

Field values are processed differently, depending on whether they are in ABAP or at database level. The database uses a different method than ABAP for rounding data (for example, when converting F to I). The database recognizes the value NULL. In contrast, ABAP only recognizes initial values.

When you use aggregation function AVG, you should use data type F (floating point number) for the target field. When you use aggregation function SUM, you should use a data type for the target field that is longer than the data type of the source field, to deal with overflows.



```

SELECT MAX( matnr ) vbeln AVG( zmeng )
  FROM vbap INTO gs_vbap
 WHERE vbeln IN so_vbeln
 GROUP BY matnr.
    CHECK (      gs_vbap-zmeng GE gv_min
              AND gs_vbap-zmeng LE gv_max ).
 APPEND gs_vbap TO gt_vbap.
ENDSELECT.

```

Object	Oper	Rec
VBAK	REOPEN	0
VBAK	FETCH	100

## SELECT with CHECK

```

SELECT MAX( matnr ) vbeln AVG( zmeng )
  FROM vbap INTO TABLE gt_vbap
 WHERE vbeln IN so_vbeln
 GROUP BY matnr.
 HAVING (      AVG( zmeng ) GE gv_min
              AND AVG( zmeng ) LE gv_max ).

```

Object	Oper	Rec
VBAK	REOPEN	0
VBAK	FETCH	2

## SELECT with HAVING

Figure 103: HAVING Clause

HAVING is used with a SELECT statement to apply a logical condition to the groups defined in the GROUP BY clause. The database determines the resulting quantity, that is, the data quantity to be transferred depends on the selectivity of the HAVING clause.

Using HAVING clauses is similar to using aggregate functions. HAVING clauses create an extra load on the database. Therefore, only use HAVING clauses if they significantly reduce the quantity of data records to be transferred. Before you use a HAVING clause, check whether the condition can be specified in the SELECT clause instead.

In the examples above, the goal is to return the largest document number and the average target quantity per material number. The average target quantity should be within a certain range.

In the top example, the target quantity range is read within the SELECT ... ENDSELECT. Therefore, all data records that fulfill the conditions in the WHERE and GROUP BY clauses are transferred.

In the second example, the target quantity is queried using the HAVING clause. Only the data records that fulfill the conditions in the WHERE, GROUP BY, and HAVING clauses are transferred.



```
SELECT  kunnr  adrnr
      FROM kna1
      INTO gs_kna1.
      APPEND gs_kna1 TO gt_kna1.
ENDSELECT.
```



## Conventional SELECT

Object	Oper	Rec
KNA1	REOPEN	0
KNA1	FETCH	403
KNA1	FETCH	403
KNA1	FETCH	194

Data quantity and transfer for SELECT ... ENDSELECT  
IDENTICAL TO  
data quantity and transfer for ARRAY FETCH

```
SELECT  kunnr  adrnr
      FROM kna1
      INTO TABLE gt_kna1.
```



## ARRAY FETCH

Figure 104: ARRAY FETCH

For SELECT .. ENDSELECT, and ARRAY FETCH, the same number of data records must be transferred from the database server to the application server. Because the transfer is always made in 32K blocks, the same transfer effort (number of fetches) is required for both variants.

However, an ARRAY FETCH is preferable to a SELECT ... ENDSELECT loop, because the records are transferred **record by record** from the **database interface** to the ABAP program in the SELECT .. ENDSELECT version, while the records are passed on to the ABAP program in a block with ARRAY FETCH.



```

SELECT vbeln erdat kunnr
FROM vbak
INTO gs_vbak
WHERE vbeln IN so_vbeln.
IF sy-dbcnt GT gv_limit.
EXIT.
ENDIF.
APPEND gs_vbak TO gt_vbak.
ENDSELECT.

```



## SELECT ... ENDSELECT

```

SELECT vbeln erdat kunnr
FROM vbak
INTO TABLE gt_vbak
UP TO gv_limit ROWS
WHERE vbeln IN so_vbeln.

```



## SELECT ... Up TO n ROWS

**Figure 105: UP TO n ROWS**

There are two ways to read a fixed number of records: You can transfer the records to the application server and use SELECT..ENDSELECT to discard the ones you do not need. The better method, however, is to use SELECT UP TO n ROWS and only transfer the desired number of data records from the database to the application.

In the first version, system field SY-DBCNT is queried within SELECT ... ENDSELECT, and the select loop is cancelled when a certain number of data records has been read. As a result, the last transferred 32KB block may unnecessarily be transferred completely.

In the second version, the UP TO n ROWS addition ensures that only the necessary dataset is read and transferred from the database to the database interface.



**Read Accessing Single Tables**



**Change Accesses to Tables**

**Pooled and Cluster Tables**

**Figure 106: Accessing Single Tables (2)**



```

SELECT * FROM vbap
INTO gs_vbap
WHERE vbeln IN so_vbeln.
gs_vbap-zmeng = gs_vbap-zmeng + 5.
UPDATE vbap FROM gs_vbap.
ENDSELECT.

```



## Row-by-row access

Reducing the number of records to be transferred  
+  
Reducing the no. of columns to be transferred

```

UPDATE vbap
SET zmeng = zmeng + 5
WHERE vbeln IN so_vbeln.

```



## Mass access

**Figure 107: UPDATE ... SET**

In an UPDATE, only the field contents that have actually been changed should be updated to the database. These fields are specified with UPDATE ... SET <f>= .... You can use the WHERE condition to restrict the table lines that are changed.

In the examples above, the goal is to increase field ZMENG by a constant amount for a range of document numbers.

In the top example, the data records are read from the database and transferred to the application server. On the application server, the data records are changed and then transferred back to the database.

In the second example, there is no data transfer. Only the UPDATE request needs to be transferred, since a database field is specified to the right of the equals sign and not a program-internal field, and the calculation is performed by the database.

You can use the UPDATE ... SET statement for multiple fields of your database table.

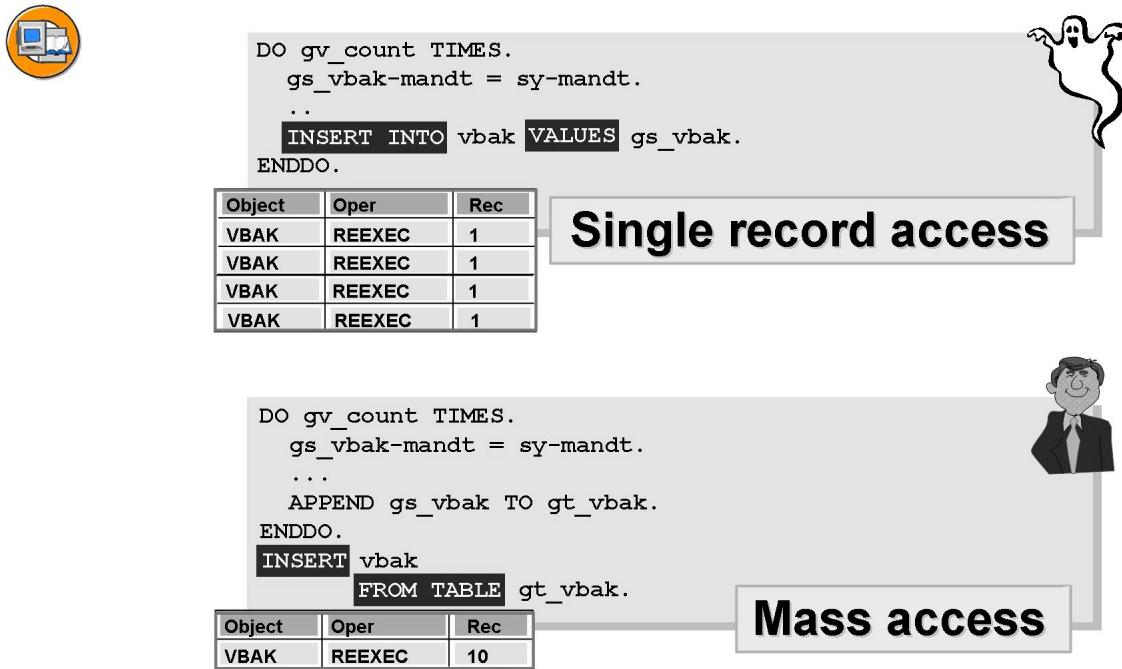


Figure 108: Creating, Changing, and Deleting Mass Data

You should always use mass data for insert operations in database tables. If you want to prevent runtime errors that could be caused by inserting duplicate entries, you can use the ACCEPTING DUPLICATE KEYS addition (existing data records are not changed in this case; SY-SUBRC is set accordingly).

In the above example, a number of entries is to be inserted into a database table. This is done record by record in the first example and using mass data in the second example.

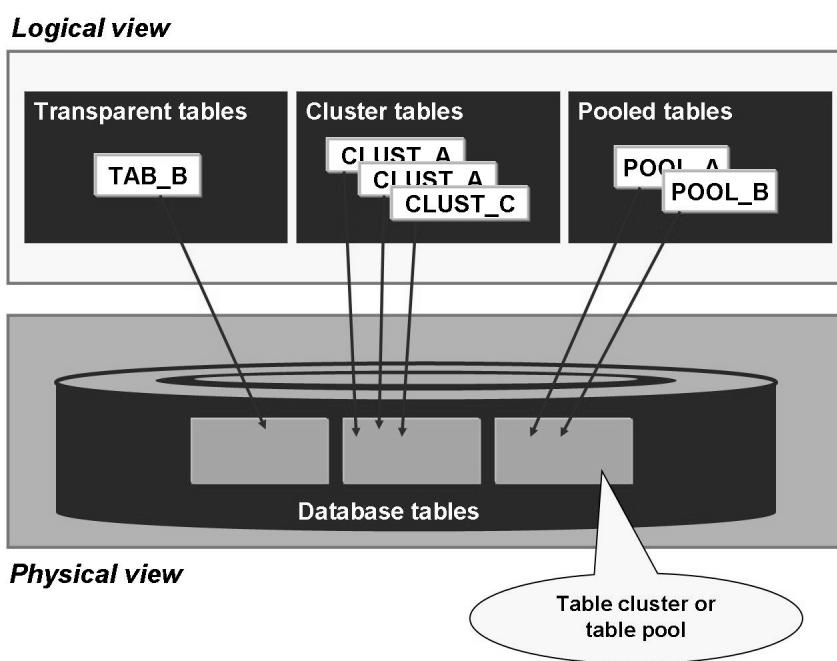
For operations to change (UPDATE, MODIFY) or delete (DELETE) data, you should also use mass data where possible (UPDATE dbtab FROM TABLE itab, MODIFY dbtab FROM TABLE itab, DELETE FROM TABLE itab). For MODIFY .. FROM TABLE itab, there is no performance improvement (since transfer is record by record).

The variant DELETE FROM dbtab WHERE .. is also available to delete multiple entries from database tables. However, you can only specify a WHERE clause together with UPDATE dbtab SET and DELETE.

Note on updating: Especially with regard to asynchronous updates, you should not use CALL FUNCTION ... IN UPDATE TASK within loops. The records to be changed should be collected and updated in an internal table. Otherwise, not only the application tables will require additional effort; the update tables (insert and delete) will as well. (Update concepts are covered in the BC414 course)



**Figure 109: Accessing Single Tables (3)**



**Figure 110: Pooled and Cluster Tables: Overview**

In addition to transparent tables, where the definition in the ABAP Dictionary and in the database are identical, there are pooled and cluster tables.

Pooled and cluster tables are characterized by the fact that several tables logically defined in the ABAP Dictionary are combined in a physical database table (table pool or table cluster).



### Pooled and Cluster Tables: Pros and Cons

Advantages	Disadvantages
<p>Fewer tables and table fields</p> <ul style="list-style-type: none"> <li>• Simpler administration</li> </ul> <p>Data compression</p> <ul style="list-style-type: none"> <li>• Less memory space</li> <li>• Less network load</li> </ul> <p>For cluster tables</p> <ul style="list-style-type: none"> <li>• Fewer database accesses</li> </ul>	<p>Limitations on database functions</p> <ul style="list-style-type: none"> <li>• No native SQL</li> <li>• No views or ABAP JOINs</li> <li>• No secondary indexes</li> <li>• No GROUP BY, ORDER BY, ...</li> </ul> <p>No table appends</p> <p>For cluster tables</p> <ul style="list-style-type: none"> <li>• Limited selection on clusterkey fields</li> </ul> <p>For pooled tables</p> <ul style="list-style-type: none"> <li>• Longer keys than necessary</li> </ul>

The decisive advantage of pooled and cluster tables is that the data can be stored in compressed form in the database. This reduces the memory space required as well as the network load.

Combining tables into table pools or table clusters results in fewer tables and compressing the data results in fewer fields in the database. The result is that fewer different SQL statements are carried out.

Pooled and cluster tables are not stored as separate tables in the database. In this way, the administration becomes simpler. With cluster tables, functionally dependent data is read together, which results in fewer database accesses.

The decisive disadvantage is the restricted database functionality. It is not possible to create an index for non-key fields. There are neither primary indexes nor indexes on a subset of the key fields. It is not possible to use database views or ABAP JOINs either, nor table appends. You can access the data in pooled or cluster tables only via OPEN SQL (no native SQL).

For pooled tables, only the WHERE conditions for key fields are transferred to the database. For cluster tables, only the WHERE conditions for the fields of the cluster key (subset of the key fields) are transferred to the database. ORDER BY (or GROUP BY) clauses are not transferred for non-key fields. You need longer keys than semantically necessary for pooled tables.

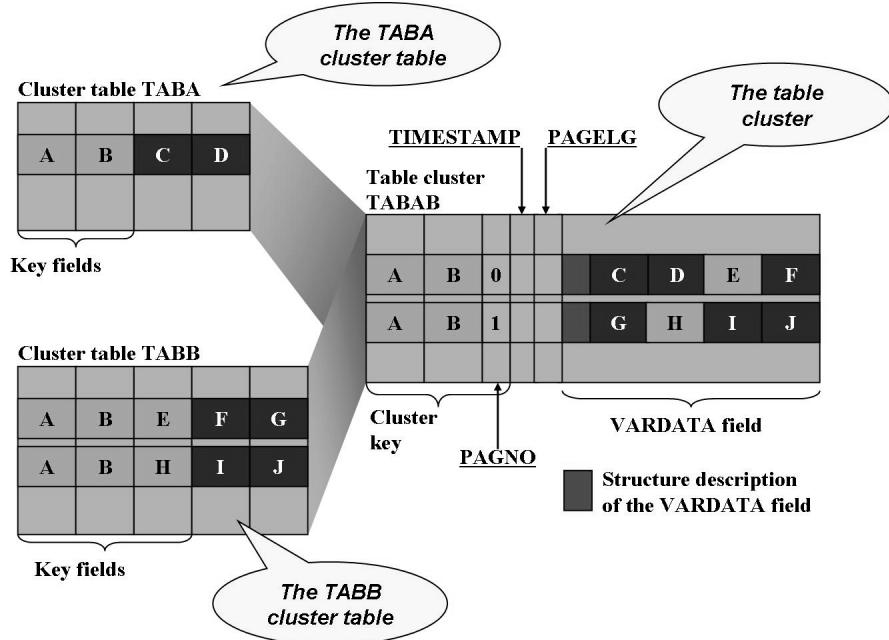


Figure 111: Cluster Tables

The idea of cluster tables is that you group together functionally dependent data that is divided among different tables and store it in one database table. Accordingly, the intersection of the key fields in the cluster tables forms the key for the table cluster (cluster key).

The data dependent on one cluster key is stored in the VARDATA field of the table cluster. If the VARDATA field does not have the capacity to take on all dependent data, the database interface creates an overflow record. The PAGNO field ensures uniqueness within the table cluster.

The content of the VARDATA field is compressed by the database interface. Accordingly, the VARDATA field contains a description for decompressing its data. The TIMESTAMP and PAGELG fields contain administrative information.

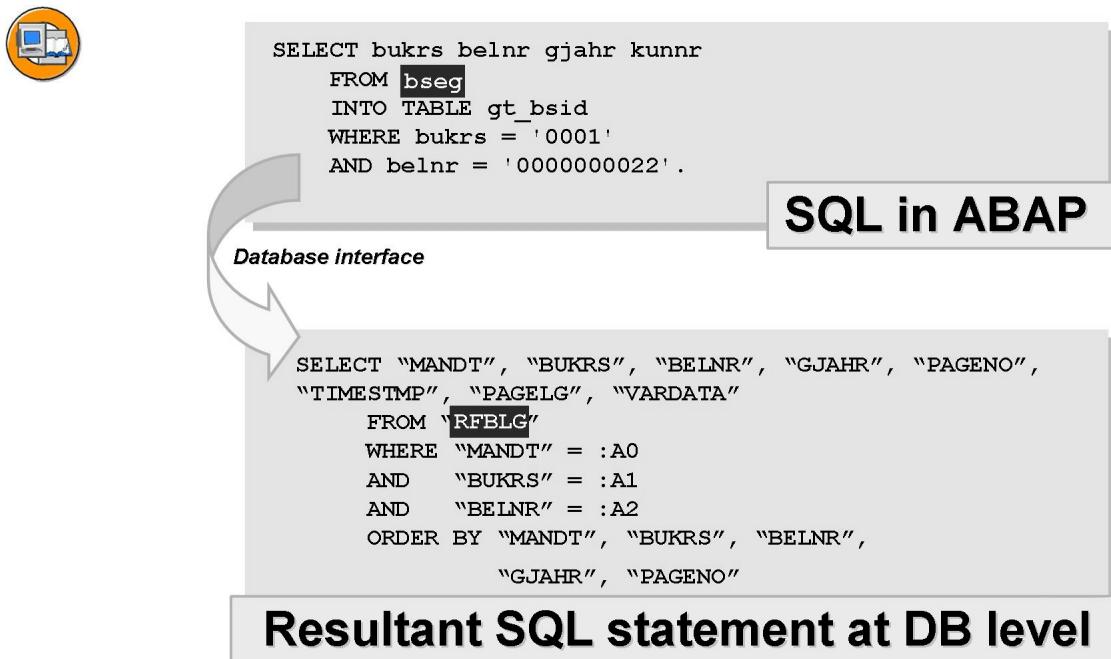


Figure 112: Selective Access to Cluster Tables

In the example above, data is read from cluster table BSEG. The SQL statement is transferred to the database interface and converted into an SQL statement for table cluster RFBLG. The SQL statement returns all corresponding data records (for example, for BUKRS = '0001' and BELNR = '0000000022') from cluster RFBLG, regardless of whether the data records are logically included in BSEG or in one of the other tables from RFBLG.

Because some of the fields in the cluster key are specified in the WHERE clause, the corresponding WHERE conditions are transferred to the database. The database interface also incorporates the WHERE clause for the field MANDT and the ORDER BY clause for the fields in the cluster key.

When accessing cluster tables, because only the primary index can be used, ensure that WHERE clauses for the key fields are specified starting at the left and continuing without any gaps to a selective field.

In the example above, if the WHERE condition for the field BUKRS was omitted, the index search string would have the form '001\_\_\_\_\_0000000022'. In this case, when the SQL statement is processed, many index blocks would be read (see Unsuitable Access Path => Unselective Index Range Scan).



```
SELECT bukrs belnr ...
FROM bseg
INTO TABLE gt_bsid
WHERE kunnr = '0000000100'.
```

```
SELECT "MANDT", "BUKRS", ...
FROM "RFBLG"
WHERE "MANDT" = :A0
ORDER BY "MANDT", "BUKRS",
...
```

## Unselective access

```
SELECT bukrs belnr ...
FROM bsid
INTO TABLE gt_bsid
WHERE kunnr = '0000000100'.
```

```
SELECT "BUKRS", "MANDT", ...
FROM BSID
WHERE "MANDT" = :A0
AND "KUNNR" = :A1.
```



## Selective alternative access

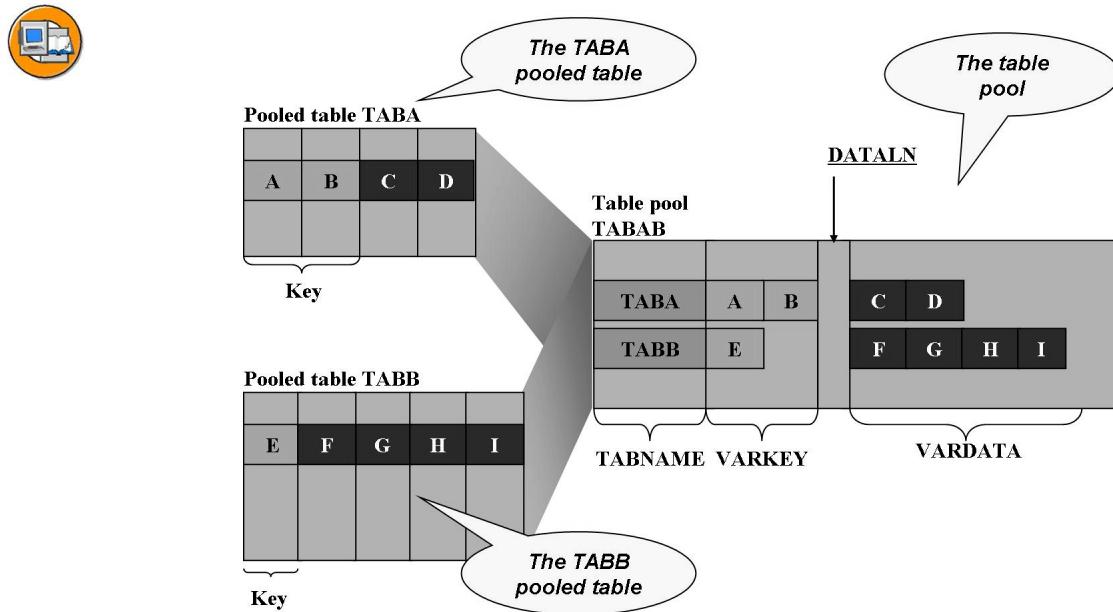
**Figure 113: Unselective Access to Cluster Tables**

In the top example, the cluster table BSEG is accessed with a WHERE condition for the field KUNNR, which is located in the compressed VARDATA field. The database interface does not transfer the WHERE condition for the field KUNNR to the database.

Correspondingly, all data records from the logon client are read from the table cluster RFBLG (client range scan). This means that data records that are not logically included in BSEG are also read. The database interface evaluates the WHERE clause for the field KUNNR.

Alternatively, you can access the data by using secondary indexes. Secondary indexes are transparent tables, where data can be redundantly stored for easier access. In the example, the required information can be obtained from the transparent table BSID (secondary index for accounts receivable).

If SAP does not provide an alternative access path, you can consider removing the table from the table cluster. However, you must consult SAP (SAP Note or support) before doing this.



**Figure 114: Pooled Tables**

The basic idea of a table pool, as opposed to table clusters, is to store data records from tables defined in the ABAP Dictionary that are not dependent on one another. You would like to combine small SAP tables to create one database table.

In the example above, you can recognize that the intersection of the key fields in TABA and TABB is empty. Despite this, the data records from TABA and TABB are stored in the TABAB table pool.

The key for a data record of the TABAB table pool consists of both the fields TABNAME and VARKEY. The TABNAME field assumes the name of the pooled table. The VARKEY field consists of the concatenation of the key fields of the pooled table. As a result of this, the key fields of a pooled table must be of the type C.

In the VARDATA field, the non-key fields of the pooled tables are stored in an unstructured, compressed form by the database interface. The DATALEN field contains the length of the VARDATA field.

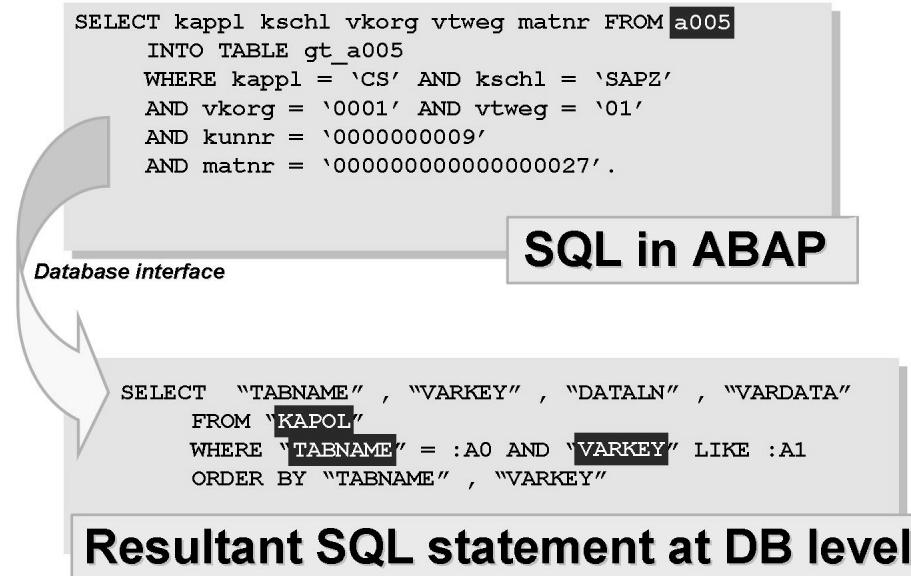


Figure 115: Selective Access to Pooled Tables

In the example above, data is read from pooled table A005. The SQL statement is transferred to the database interface and converted into an SQL statement for table pool KAPOL.

The WHERE conditions specified in the WHERE clause of the SQL statement refer to key fields in table A005 and are therefore transferred in their entirety to the database in field VARKEY. The database interface incorporates the ORDER BY clause for fields TABNAME and VARKEY (key fields in the table pool).

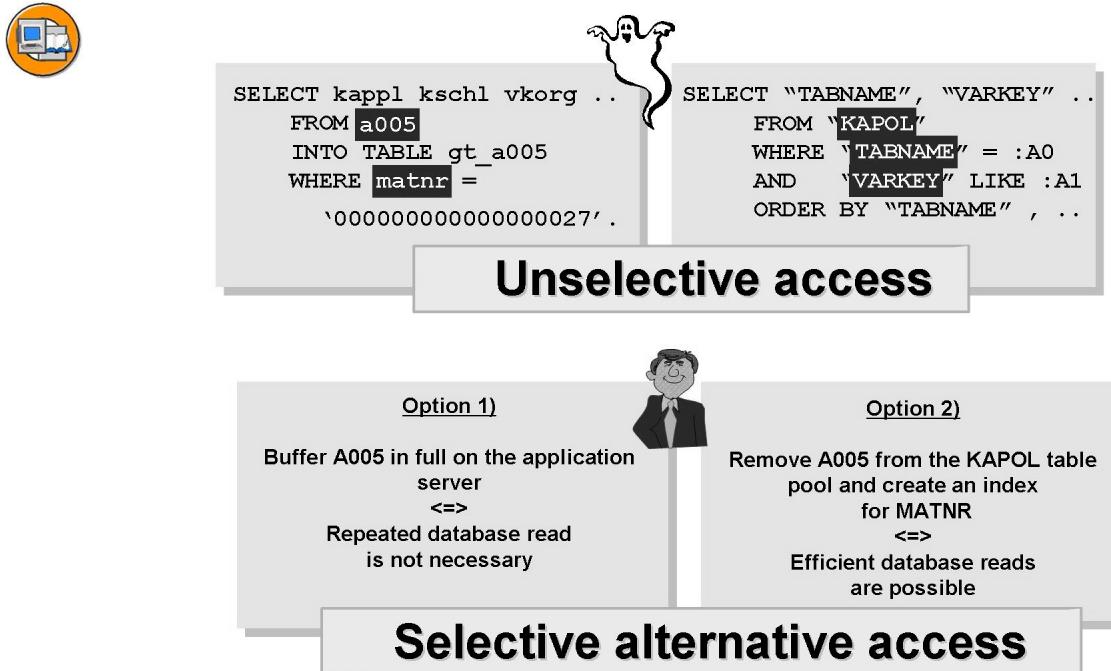


Figure 116: Unselective Access to Pooled Tables

SQL statements for pooled tables must have WHERE conditions specifying the key fields with an equals sign. The key fields must be specified starting with the key field that is left-most in the table. In addition, any key fields specified must be strictly contiguous, that is, no 'gaps' should occur in the sequence of specified fields. In the example above, a WHERE condition is not specified for field KUNNR. The database therefore uses the index search string 'A005CSSAPZ000101\_\_\_\_\_010000000000000000027'.

The database uses the partial string 'A005CSSAPZ000101' to select data blocks. This means that too many data blocks are read (see unit Unsuitable Access Path => Unselective Index Range Scan).

To solve this problem, first check whether the missing WHERE conditions can be specified. If this is not possible, check whether any changes can be made to SAP table buffering (see slides at end of this unit).

If you cannot change SAP table buffering, you can remove the table from the table pool and convert it to a transparent table. Then create a secondary index for the specified fields to enable efficient access from the database.



# Exercise 6: (Optional) Ranges Tables / Select Options

## Exercise Objectives

After completing this exercise, you will be able to:

- Explain performance-related aspects of programming with select options
- Take action to optimize programming with select options

## Business Example

Mr. Jones, a software developer at a major corporation that uses SAP, is supposed to develop a new business application to analyze mass data. He knows that reading data is a runtime-critical operation. In particular, the database – by accessing the selected set – has a critical effect on runtime behavior. Mr. Jones wants to integrate a selection screen with select options into his reports and to optimize access to this ranges table.

### Task:

Copy the template program SAPBC490\_TEMP\_RANGES to your own program Z##\_RANGES. The program contains a select options statement via the CARRID field in table SBOOK. The aim is to optimize access to this ranges table.

1. For the program you copied, create a variant with a large number of selections, positive arguments and negative (excluding) arguments. Save this variant.
2. Carry out an SQL trace for the program with the variant and analyze the SQL statement sent to the database. Also make a note of the access times.
3. Improve access to the database by optimizing the content of the select option using a separate RANGES table.

Construct a separate RANGES table with

```
DATA: it_range TYPE RANGE OF s_carr_id.
```

Tip: The check table is the SCARR table. First use the SCARR table to read all the airlines found and insert them in a separate ranges table (you will also need a work area for separate it\_ranges). Then access this optimized ranges table in the SBOOK loop.

4. Trace your optimized program again. Pay particular attention to the new access times and the SQL statement sent to the database.

## Solution 6: (Optional) Ranges Tables / Select Options

### Task:

Copy the template program SAPBC490\_TEMP\_RANGES to your own program Z\_##\_RANGES. The program contains a select options statement via the CARRID field in table SBOOK. The aim is to optimize access to this ranges table.

1. For the program you copied, create a variant with a large number of selections, positive arguments and negative (excluding) arguments. Save this variant.
  - a) Create the variant as usual.
2. Carry out an SQL trace for the program with the variant and analyze the SQL statement sent to the database. Also make a note of the access times.
  - a) Proceed as you have done previously with ST05. You will identify a select statement with a relatively complex WHERE clause. Make a note of the costs of the optimizer.
3. Improve access to the database by optimizing the content of the select option using a separate RANGES table.

Construct a separate RANGES table with

```
DATA: it_range TYPE RANGE OF s_carr_id.
```

Tip: The check table is the SCARR table. First use the SCARR table to read all the airlines found and insert them in a separate ranges table (you will also need a work area for separate it\_ranges). Then access this optimized ranges table in the SBOOK loop.

- a) See source text of the model solution
4. Trace your optimized program again. Pay particular attention to the new access times and the SQL statement sent to the database.
  - a) You should see that a select statement with an IN list was sent to the database. In most databases, this is better than the statement (in the WHERE clause) generated by the original settings for the old variants (this type of statement includes BETWEENs, NOTs and so on).

### Result

#### Source text of the model solution:

*Continued on next page*

```

*&-----*
*& Report SAPBC490_SQL1_RANGES_SOLUTION *
*&-----*
*& Improve access to SBOOK table by creating own RANGES table   *
*& and acces the much smaller SCARR proving table
*&-----*
REPORT SAPBC490_SQL1_RANGES_SOLUTION.

DATA: gs_sbook TYPE sbook.

SELECT-OPTIONS selopt FOR gs_sbook-carrid.

DATA: gt_range TYPE RANGE OF s_carr_id.
DATA: gs_range LIKE LINE OF gt_range.

START-OF-SELECTION.
*#####
gs_range-sign = 'I'.
gs_range-option = 'EQ'.

*** just build up a new better RANGE table
*** we access the proving table scarr

SELECT carrid FROM scarr INTO gs_range-low
WHERE carrid IN selopt.
APPEND gs_range TO gt_range.
ENDSELECT.

*** now the access to sbook is much better!

SELECT * FROM sbook
INTO gs_sbook
WHERE carrid IN gt_range.
ENDSELECT.

```



# Exercise 7: Aggregate Functions

## Exercise Objectives

After completing this exercise, you will be able to:

- Explain performance-related aspects of programming with aggregates

## Business Example

Mr. Jones, a software developer at a major corporation that uses SAP, is supposed to develop a new business application to analyze mass data. He plans to use aggregate functions and would like to compare them with alternatives.

### Task:

Copy the template program SAPBC490\_TEMP\_AGGREGATE to your own program Z\_##\_AGGREGATE.

In the template, the largest FORCURAM amount is supplied for each CARRID airline. The set of results therefore includes as many records as there are airlines in the SBOOK table.

1. Develop an alternative to the specified aggregate technique for the first form routine, AGGREGATE\_ON\_APPLICATIONSERVER, which was already implemented.

Implement the second form routine AGGREGATE\_ON\_DB. The aggregate here should be built on the database.

Compare the runtimes for the two options and carry out an SQL trace.

2. In which cases would which of the two techniques be best or recommendable? What would the request to the application look like?

## Solution 7: Aggregate Functions

### Task:

Copy the template program SAPBC490\_TEMP\_AGGREGATE to your own program Z\_##\_AGGREGATE.

In the template, the largest FORCURAM amount is supplied for each CARRID airline. The set of results therefore includes as many records as there are airlines in the SBOOK table.

1. Develop an alternative to the specified aggregate technique for the first form routine, AGGREGATE\_ON\_APPLICATIONSERVER, which was already implemented.  
Implement the second form routine AGGREGATE\_ON\_DB. The aggregate here should be built on the database.  
Compare the runtimes for the two options and carry out an SQL trace.
  - a) Trace in the normal way and pay attention to the excerpt from the source code.
2. In which cases would which of the two techniques be best or recommendable? What would the request to the application look like?
  - a) When aggregating via the application server, a large amount of data has to be transported via the network to the application server, where the aggregate is built. If there is a large amount of data, that is precisely the disadvantage of this method. Therefore, if aggregating on the database would avoid a large data transport, the database aggregate would be beneficial. A GROUP BY clause that causes very few result records also has an effect on this data transfer. Furthermore, a decisive factor is which other actions are also executed in this program using the returned data. If the data records have to be utilized in any case, aggregating on the application server makes sense.

If you would aggregate via CARRID CONNID FLDAT, the database aggregate would also have to transport more records than the result, and the first technique would tend to be marginally better.

### Result

#### Source text of the model solution:

```
*&-----*  
*& Report SAPBC402_SQL1_AGGREGATE_SOLUTION *  
*&-----*
```

*Continued on next page*

```

*& Aggregate on application server and aggregate on database      *
*& What might be the difference?                                *
*&-----*                                                       *
REPORT SAPBC402_SQL1_AGGREGATE_SOLUTION.

TYPES: BEGIN OF ty_rec,
        carrid TYPE sbook-carrid,
        forcuram TYPE sbook-forcuram,
        END OF ty_rec.

TYPES: ty_itab TYPE SORTED TABLE OF ty_rec
      WITH NON-UNIQUE KEY carrid.

DATA: itab TYPE ty_itab,
      wa TYPE ty_rec.

DATA: t1 TYPE f, t2 LIKE t1.

START-OF-SELECTION.
*****#
PERFORM aggregate_on_applicationserver.
PERFORM aggregate_on_database.

*&-----*                                                       *
*&      Form  aggregate_on_applicationserver
*&-----*                                                       *
FORM aggregate_on_applicationserver .

FIELD-SYMBOLS: <fs>  TYPE ty_rec.
DATA: old_carrid TYPE sbook-carrid,
      forcuram TYPE sbook-forcuram,
      counter TYPE i.

GET RUN TIME FIELD t1.

SELECT carrid forcuram
  FROM sbook
  INTO TABLE itab.

*** compute maximum of zmeng for each VBELN in VBAP table
LOOP AT itab ASSIGNING <fs>.

```

*Continued on next page*

```

        IF <fs>-carrid <> old_carrid. "does the carrid change ?
        IF old_carrid IS INITIAL.    "just once at the beginning
          old_carrid = <fs>-carrid.   " initialize algorithm
          forcuram = <fs>-forcuram.
        ELSE.
          WRITE: / old_carrid, forcuram. "output the result for each carrid
          counter = counter + 1.
          CLEAR forcuram.
          old_carrid = <fs>-carrid. "save new carrid in old_carrid
        ENDIF.
      ELSE.
        IF forcuram < <fs>-forcuram.
          forcuram = <fs>-forcuram.
        ENDIF.
      ENDIF.
    ENDLOOP.
    WRITE: / old_carrid, forcuram.    "output the last carrid.
    counter = counter + 1.

    GET RUN TIME FIELD t2.
    t2 = t2 - t1.
    WRITE: / 'Runtime = ', t2, ' coutner = ', counter.
    ULINE.

ENDFORM.           " aggregate_on_applicationserver
*&-----*
*&      Form aggregate_on_database
*&-----*
FORM aggregate_on_database .

FIELD-SYMBOLS: <fs> TYPE ty_rec.
DATA: counter TYPE i.

GET RUN TIME FIELD t1.

SELECT carrid MAX( forcuram )
  FROM sbook
  INTO TABLE itab
  GROUP BY carrid.

*** Output all the resulting carrids with their highest forcuram
LOOP AT itab ASSIGNING <fs>.

```

*Continued on next page*

```
        WRITE: / <fs>-carrid, <fs>-forcuram.  
        counter = sy-tabix.  
      ENDLOOP.  
  
      GET RUN TIME FIELD t2.  
      t2 = t2 - t1.  
      WRITE: / 'Runtime = ', t2, ' coutner = ', counter.  
      ULINE.  
  
    ENDFORM.          " aggregate_on_database
```



## Lesson Summary

You should now be able to:

- Program efficient SELECT commands to access individual tables
- Explain the fundamentals of pooled and cluster tables
- Initiate efficient change accesses



## Unit Summary

You should now be able to:

- Program efficient SELECT commands to access individual tables
- Explain the fundamentals of pooled and cluster tables
- Initiate efficient change accesses



# Unit 5

## Accessing Multiple Tables

### Unit Overview

#### Contents:



- Nested SELECTS
- SELECT with FOR ALL ENTRIES
- Joins and views
- Subquery
- Logical databases



### Unit Objectives

After completing this unit, you will be able to:

- Explain the advantages of join/ views compared to nested SELECTS and FOR ALL ENTRIES

### Unit Contents

Lesson: Accessing Multiple Tables .....	170
Exercise 8: Joins Compared to Nested Selects .....	195
Exercise 9: (Optional) Access Using FOR ALL ENTRIES .....	199

# Lesson: Accessing Multiple Tables

## Lesson Overview

### Contents:



- Nested SELECTS
- SELECT with FOR ALL ENTRIES
- Joins and views
- Subquery
- Logical databases



## Lesson Objectives

After completing this lesson, you will be able to:

- Explain the advantages of join/ views compared to nested SELECTS and FOR ALL ENTRIES

## Business Example

A company wants to read data efficiently from several tables in the context of a customer development.

## Join Techniques

In the SAP system, users often need to access information stored in different tables. To read data distributed over various tables, you must create a join between functionally dependent tables. The corresponding logical database operator is called a JOIN. In ABAP, there are various ways to implement the JOIN operator (nested SELECTS, SELECT FOR ALL ENTRIES, DB views, ABAP JOINS, subqueries, explicit cursor). If you want the database system to determine the resulting set of the JOIN operator, you can use DB views, ABAP JOINS or subqueries. DB views are defined in the ABAP dictionary and activated to create them in the database. DB views created by other developers can also be used. ABAP joins are defined in an ABAP program.

The following graphic illustrates all options for realizing a join that are available in ABAP.

### Possible ways of accessing several tables



INNER JOIN	LEFT OUTER JOIN
<ul style="list-style-type: none"> <li>• Nested SELECTs</li> <li>• Logical databases</li> <li>• FOR ALL ENTRIES (processing logic)</li> <li>• <b>Database view</b></li> <li>• <b>ABAP JOIN</b></li> <li>• (Subquery)</li> </ul>	<ul style="list-style-type: none"> <li>• Nested SELECTs</li> <li>• Logical databases</li> <li>• FOR ALL ENTRIES (Processing logic)</li> <li>• <b>ABAP JOIN</b></li> <li>• (Subquery)</li> </ul>



## Join Basics

### Nested Selects

### Views in the Dictionary

### ABAP Joins

### Subquery

### Additional Special Topics

**Figure 117: Accessing Multiple Tables (1)**

JOINS can be implemented in different ways. One method is to read the data records from the outer table in a SELECT loop. Within the loop, a further SELECT statement is nested, in which the information from the inner table is read. Depending on how the data records read are subsequently processed, you can implement an INNER JOIN or an OUTER JOIN.

Reading from several, semantically dependent tables is also possible using nested selects. Since nested SELECT statements have some disadvantages, you can replace them by reading the data records to an internal table, and then using a SELECT FOR ALL ENTRIES. This method enables you to partly remove loops from nested SELECT statements. However, this solution is only suitable for small quantities of data, and can cause problems. An INNER JOIN or a LEFT OUTER JOIN can also be implemented.

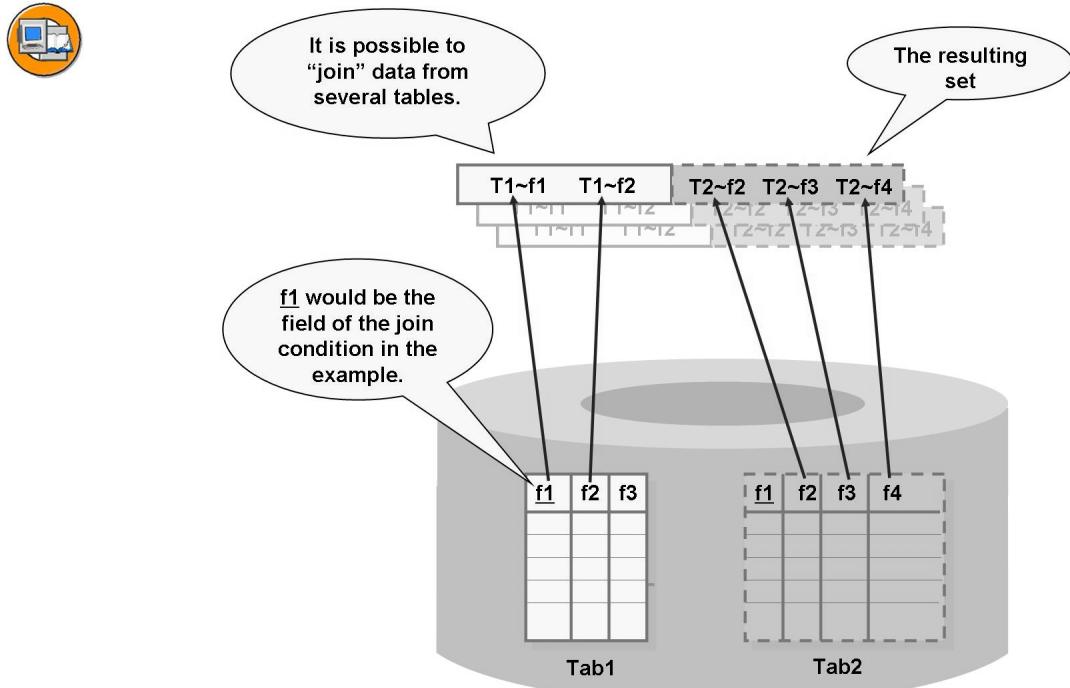
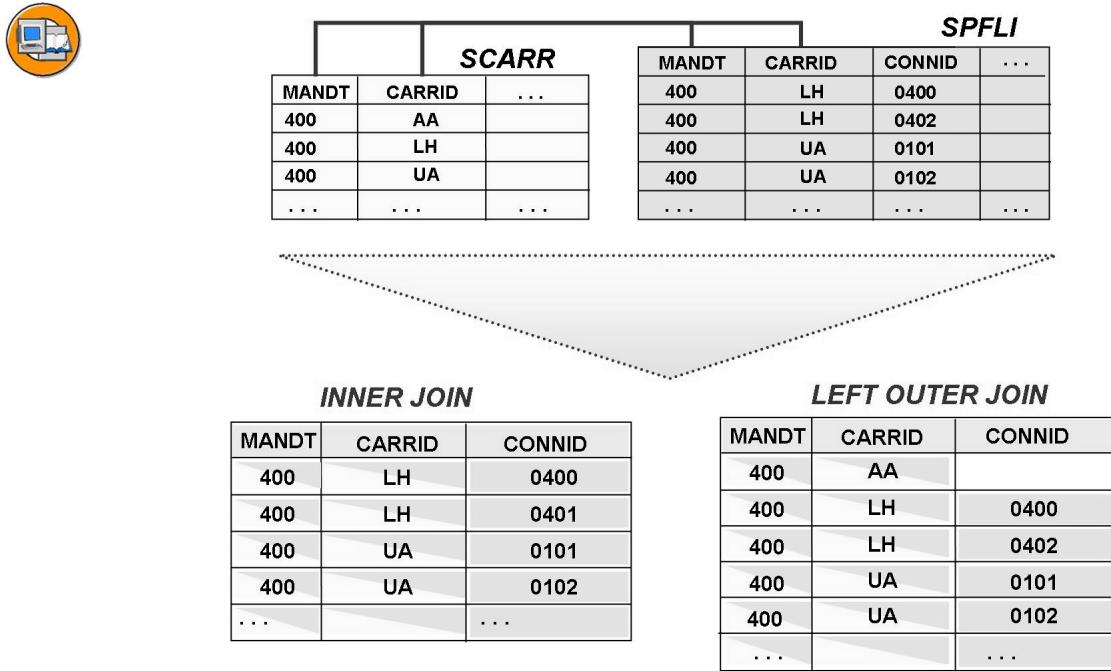


Figure 118: Accessing Multiple Tables - Overview

The most flexible and often the most effective way to perform JOINS is at database level. In OPEN SQL, you can use subqueries, database views, and ABAP JOINs. Only an INNER JOIN implements a database view. ABAP JOINs are used to perform both INNER JOINS and LEFT OUTER JOINS. These techniques are discussed in the following section.



**Figure 119: Inner / Outer Joins**

INNER JOINS and LEFT OUTER JOINS can be derived from the resulting set.

An INNER JOIN produces a results set containing only the records of the outer table whose records in the inner table match the JOIN conditions (shown in the example above left).

The terms driving and driven table are also used.

A LEFT OUTER JOIN produces a resulting set containing all data records from the outer table, regardless of whether matching records exist in the inner table. If there are no matching data records in the inner table, NULL values are returned for the fields in the inner table (see example above right). The Null values are replaced with the initial value of the correct type in the results set. In ABAP, IS INITIAL is therefore queried.

The tables involved in a JOIN are called base tables. Using the results of a JOIN, you can perform a projection (column selection) or a selection (row selection).

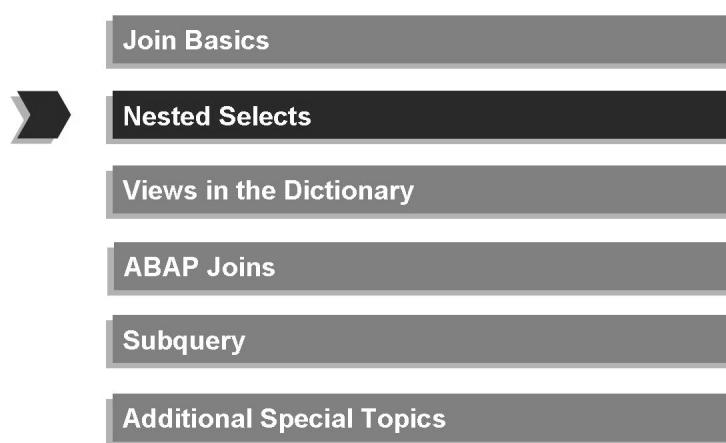
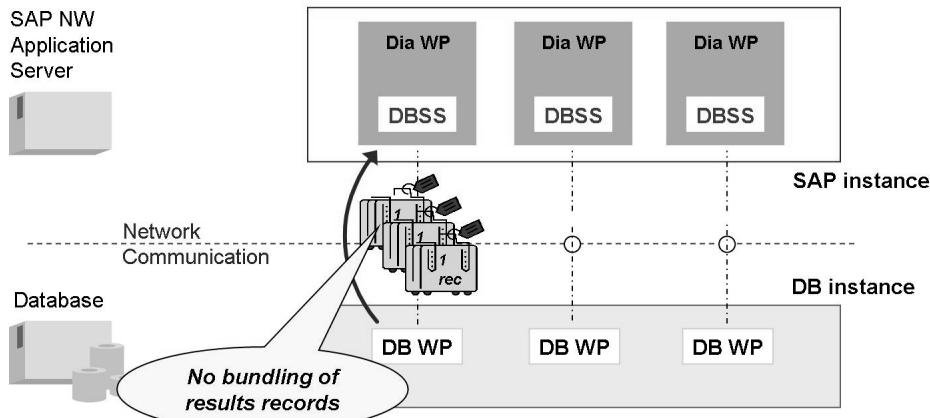


Figure 120: Accessing Multiple Tables (2)



9	VBAK	REOPEN	41	1403	8	SELECT WHERE "MANDT" = '900' AND "VBELN" BETWEEN '0000006000' AND '0000008000'
332	VBAK	FETCH			0	SELECT WHERE "MANDT" = '900' AND "KUNNR" = 'SDD00LST'
5	KNA1	REOPEN			0	SELECT WHERE "MANDT" = '900' AND "KUNNR" = 'SDD00LST'
143	KNA1	FETCH	1		0	SELECT WHERE "MANDT" = '900' AND "KUNNR" = 'SDD00LST'
4	KNA1	REOPEN			0	SELECT WHERE "MANDT" = '900' AND "KUNNR" = 'SDD00LST'
128	KNA1	FETCH	1		0	SELECT WHERE "MANDT" = '900' AND "KUNNR" = 'SDD00LST'
6	KNA1	REOPEN			0	SELECT WHERE "MANDT" = '900' AND "KUNNR" = 'SDD00LST'
124	KNA1	FETCH	1		0	SELECT WHERE "MANDT" = '900' AND "KUNNR" = 'SDD00LST'
5	KNA1	REOPEN			0	SELECT WHERE "MANDT" = '900' AND "KUNNR" = 'SDD00LST'
124	KNA1	FETCH	1		0	SELECT WHERE "MANDT" = '900' AND "KUNNR" = 'SDD00LST'
5	KNA1	REOPEN			0	SELECT WHERE "MANDT" = '900' AND "KUNNR" = 'SDD00LST'
123	KNA1	FETCH	1		0	SELECT WHERE "MANDT" = '900' AND "KUNNR" = 'SDD00LST'
4	KNA1	REOPEN			0	SELECT WHERE "MANDT" = '900' AND "KUNNR" = 'SDD00LST'

Figure 121: Disadvantages of Nested Selects Visible in Trace

## Problems with nested SELECTs



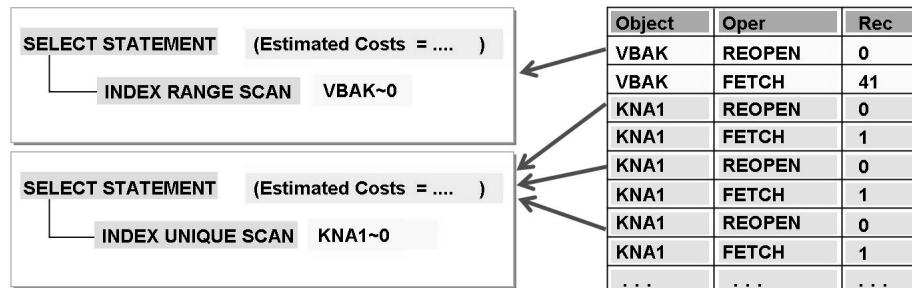
- **Inefficient transfer**
  - Data records of the inner table are transferred row-by-row to the driving table
  - Lots of small fetches, instead of fewer and more compact fetches
- **SELECTs on (identical) values**
  - The data records of the inner table are read multiply (the WHERE clause working between the inner table and the driving table uses a key field and a non-key field)
- **Defined access sequence**
  - Whether a table is an inner or outer table is defined in coding



```

SELECT vbeln_kunrn FROM vbak
  INTO (gs_vbakkna1-vbeln, gs_vbakkna1-kunrn)
 WHERE vbeln IN so_vbeln.
   SELECT SINGLE adrnr FROM kna1
     INTO gs_vbakkna1-adrnr
     WHERE kunrn EQ gs_vbakkna1-kunrn.
     IF sy-subrc EQ 0.
       APPEND gs_vbakkna1 TO gt_vbak_kna1.
     ENDIF.
   ENDSELECT.

```



**Figure 122: Nested SELECTs**

In the example above, nested SELECTs are used to perform an INNER JOIN. From *Explain SQL*, we can see that one data record from the inner table KNA1 is read for each data record in the outer table VBAK. Access to VBAK and to KNA1 is performed using a primary index.

ABAP distinguishes between INNER JOIN and LEFT OUTER JOIN by performing the APPEND to the target table in the inner or outer SELECT .. ENDSELECT.

**The major disadvantage** of nested SELECTs is that they are expensive to transfer. As shown in the example, for every data record of the outer table VBAK, one data record is read from the inner table KNA1. Therefore, the database operation REOPEN must be performed separately for each SELECT on table KNA1, and only one data record is transferred from KNA1 each time. This causes an unnecessarily high number of small fetches (that is, 32 KB blocks that are not completely filled).

**A further disadvantage** of nested SELECTs is that identical SQL statements are executed more than once. When WHERE clauses associate a non-key field of the outer table with a key field of the inner table, this generally results in value-identical SELECTs.

 **Note:** Example: One example is when customer data is read for a sales document. As a general rule, one customer has several sales and distribution documents. Identical customer data is read multiple times from the inner table.

**The third disadvantage** of nested SELECTs is that the order of access is fixed in coding. That is, a table is defined as either an inner or an outer table. The effects are particularly negative if you use dynamic WHERE conditions in a SELECT statement (for example, SELECT-OPTIONS, PARAMETERS) that is supplied with user entries at runtime. The user entry (and therefore the WHERE condition) often determines which table the JOIN uses for access. If the join is implemented using database views or ABAP JOINS, the database optimizer determines the order of access.

 **Note:** Nested SELECTs and SELECT FOR ALL ENTRIES can be satisfied by the table buffer, but joins processed at database level always bypass SAP table buffering. This can make JOINS using nested SELECTs or SELECT .. FOR ALL ENTRIES the better alternative.



**Join Basics**

**Nested Selects**

**Views in the Dictionary**

**ABAP Joins**

**Subquery**

**Additional Special Topics**

**Figure 123: Accessing Multiple Tables (3)**

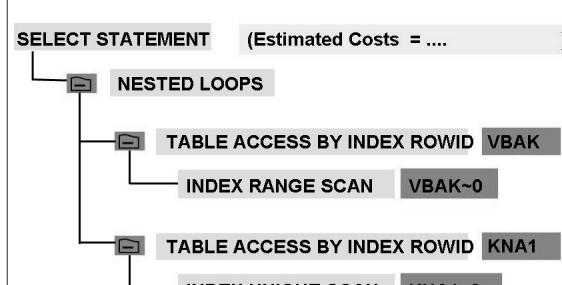
## Views in the Dictionary



*View defined  
in dictionary*

```
SELECT vbeln kunnr adrnr FROM vbak_kna1
INTO TABLE gt_vbak_kna1
WHERE vbeln IN so_vbeln.
```

Create  
vbak\_kna1  
as DB view  
in dictionary



**Figure 124: Database view**

A database view is created in the ABAP Dictionary as a Dictionary object. The view is generated in the database after being activated in the ABAP Dictionary. However, the database view data is stored in the tables used to define the database view. Therefore,

defining a database view does not cause extra load, for example, due to redundant data retention. When you define a database view, you can perform a column selection (projection) and a row selection (selection), as well as a JOIN condition.

You cannot define an index via a database view. In processing an SQL statement for a database view, the indexes of the involved tables are usually used. An SQL statement using a database view is always processed as an INNER JOIN.

Unlike an ABAP JOIN, a database view can easily be reused. If various SQL statements use one database view, the DB SQL cache performs more efficiently.

An SQL statement on a database view always bypasses SAP table buffering. However, as of Release 4.0, contents of database views can be buffered on the application server.

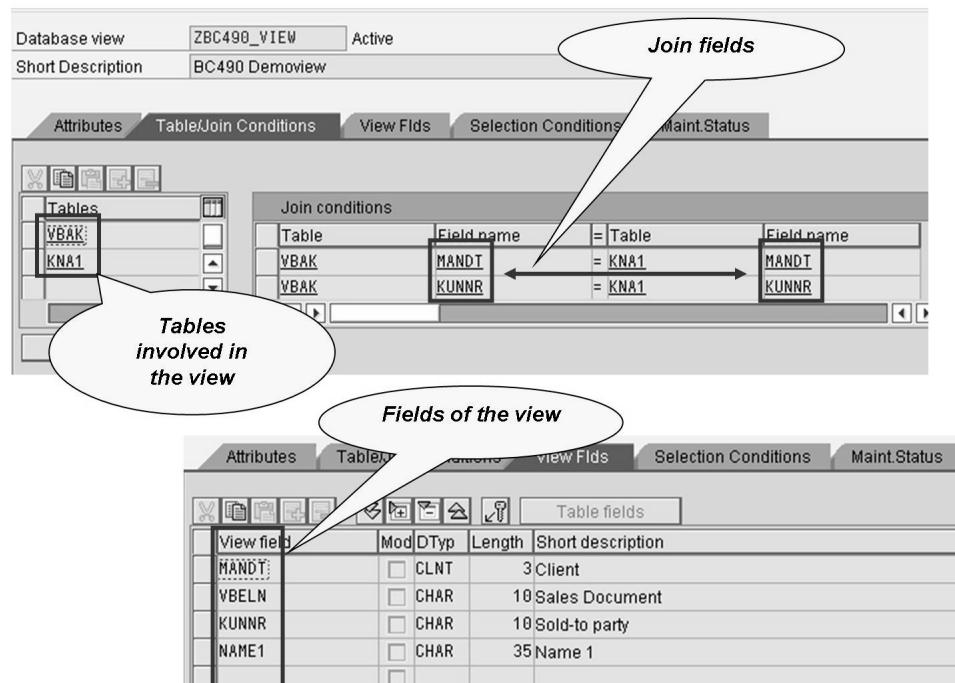


Figure 125: Example of a View in the Dictionary

In creating views, make sure that you include the client both in the view fields and in the join conditions (of course, this only applies to client-specific tables). For an ABAP join, this would be copied by the DBSS.



**Hint:** When you design a view, use the MANDT field from the table from which the optimizer is supposed to start reading (the outer driving table). Think about which table restricts the result set and the search quantity the most. It might be beneficial and it could make the optimizer's work easier if you "promoted" this table. The optimizer usually starts reading in the table that reduces the search quantity the most.

In the selection of the fields of the join of the view and in the conditions, use as many fields as possible from this table, if the same fields occur in several tables. This might also have a positive effect on the optimizer response.



```
SELECT * FROM ZBC490_VIEW
INTO gs_view
WHERE vbeln BETWEEN ...
ENDSELECT.
```

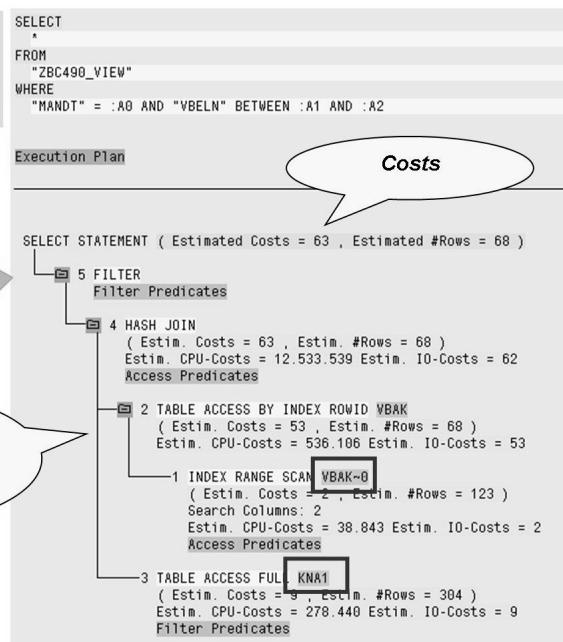


Figure 126: The Explain for the View Example

The explain with the access strategy is later contrasted to a comparable ABAP join. In anticipation: There are no relevant processing differences on the database-side and therefore no significant performance differences!



Additional details regarding the processing a view by the DB correspond to the descriptions of the ABAP join. So please read the descriptions in the following category for ABAP joins.

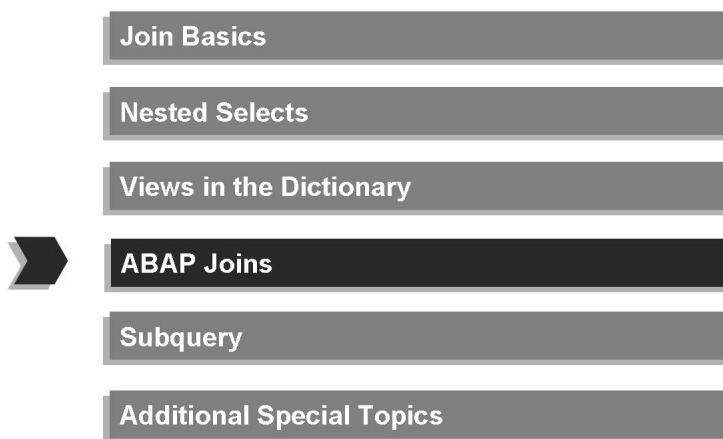


Figure 127: Accessing Multiple Tables (4)

## ABAP Joins

Unlike with nested SELECTs, SELECT FOR ALL ENTRIES, and subqueries, the database optimizer dynamically determines the order of access for ABAP joins and views from the database optimizer. The criterion for the order of access is to have as few data blocks to be read as possible. There is no way to control this from the application program (apart from using database hints).

To make access to the outer table as selective as possible, formulate an appropriate WHERE clause. Then, fewer data records need to be read from the outer table, and consequently fewer data records also need to be read from the inner table.

The fields used for creating a join (JOIN condition) should also be as selective as possible. The goal is to have as few data records as possible returned from the inner table per data record from the outer table (if possible, just one using index unique scan).

There should be a database index at least on the inner table for the fields in the JOIN conditions.

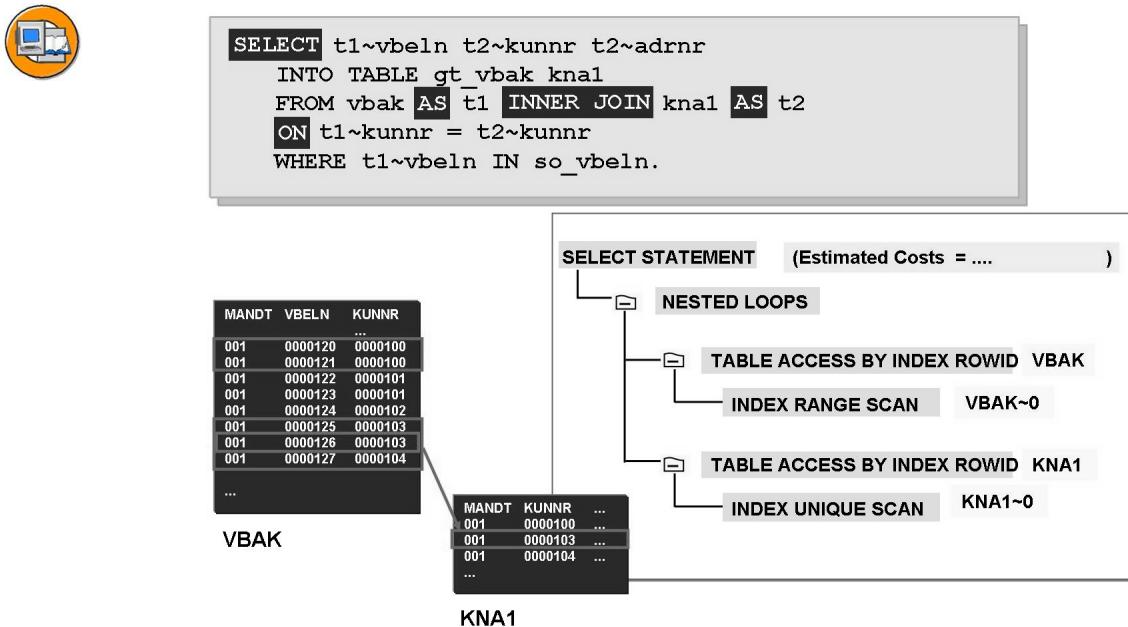


Figure 128: ABAP INNER JOIN

In an ABAP JOIN, database tables are joined using the ON clause. You can use aliases for the relevant tables, as shown in the example above. You can also specify field lists (projection) and WHERE clauses (selection) for ABAP JOINS.

Ensure that you clearly allocate the field name in the SELECT clause, in the ON clause, and in the WHERE clause to a table or to an alias for a table. If the database does not support Standard SQL, the ON conditions are converted to WHERE conditions.

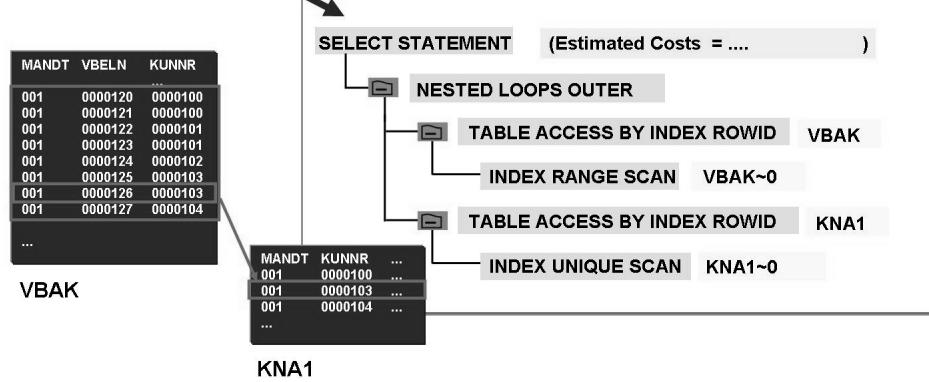
ABAP JOINS also bypass the SAP table buffering. To implement a JOIN on a buffered table, it may be best to define a buffered database view on the table.

If you use parentheses, an ABAP JOIN can link more than two tables. Do not join too many tables, because this increases the complexity of processing the SQL statement on the database.

In the example above, an INNER JOIN is implemented. Therefore, the resulting set (with a blue frame) contains only the data records with entries for field KUNNR in both VBAK and KNA1.



```
SELECT t1~vbeln t2~kunnnr t2~adrnr
INTO TABLE gt_vbak kna1
FROM vbak AS t1 LEFT OUTER JOIN kna1 AS t2
ON t1~kunnnr = t2~kunnnr
WHERE t1~vbeln IN so_vbeln.
```



**Figure 129: ABAP OUTER JOIN**

A JOIN with LEFT OUTER logic can only be created using an ABAP JOIN (database views always implement an INNER JOIN).

Since some DBMS approved by SAP do not support the standard syntax for ON conditions, syntax restrictions were made to allow only those joins that produce the same results on all database systems.

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

In the example above, a LEFT OUTER JOIN is implemented. Therefore, the resulting set (with a blue frame) also contains the data records from VBAK that do not have entries for field KUNNR in KNA1.

- An example of an ABAP join follows
- The join maps the same selection as the view discussed before
- Both the access times and the result in the explain match



```

4 REPORT z_00_join_access.
5
6 DATA: BEGIN OF wa,
7   vbeln TYPE vbak-vbeln,
8   kunnr TYPE vbak-kunnr,
9   name1 TYPE kna1-name1,
10  END OF wa.
11
12 START-OF-SELECTION.
13 *#####
14
15  SELECT vbak~vbeln vbak~kunnr kna1~name1 INTO wa
16    FROM vbak INNER JOIN kna1
17      ON vbak~kunnr = kna1~kunnr
18      WHERE vbeln BETWEEN '0000006000' AND '0000008000'.
19      WRITE: / wa~vbeln, wa~kunnr, wa~name1.
20  ENDSELECT.
21

```

The same example now as an ABAP JOIN

Figure 130: Comparison: View Example With ABAP Join First

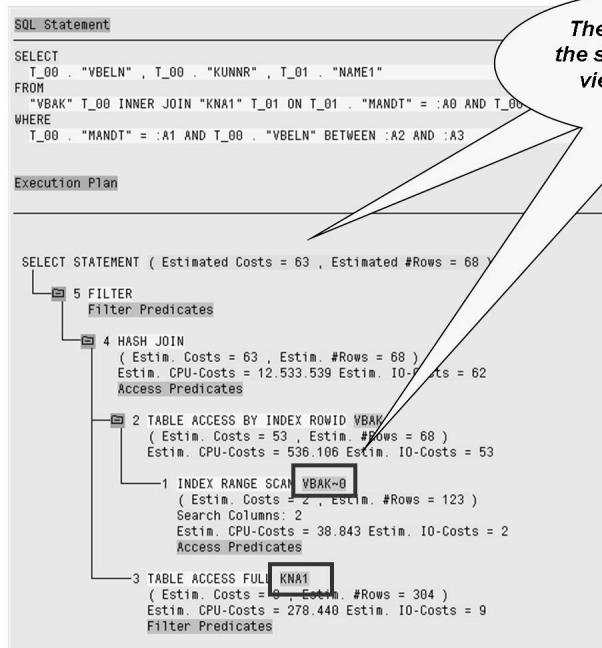


Figure 131: Comparison: The Explain of the ABAP Join

In conclusion it could be said that the database proceeds in the same way for processing the view and the ABAP join. It tries to mix the data records for the results set in the available memory. If the available memory does not suffice, it temporarily moves the data to the drive and tries to form the results set there.

From a performance aspect you can use ABAP joins or views. The great advantage of views is that they can be reused. In working with views, where-used lists and search systems such as the repository info system (SE84) can also be used. Which technique is rated as more readable depends on the perspective of the observer. At least the access to a view is easier to code!

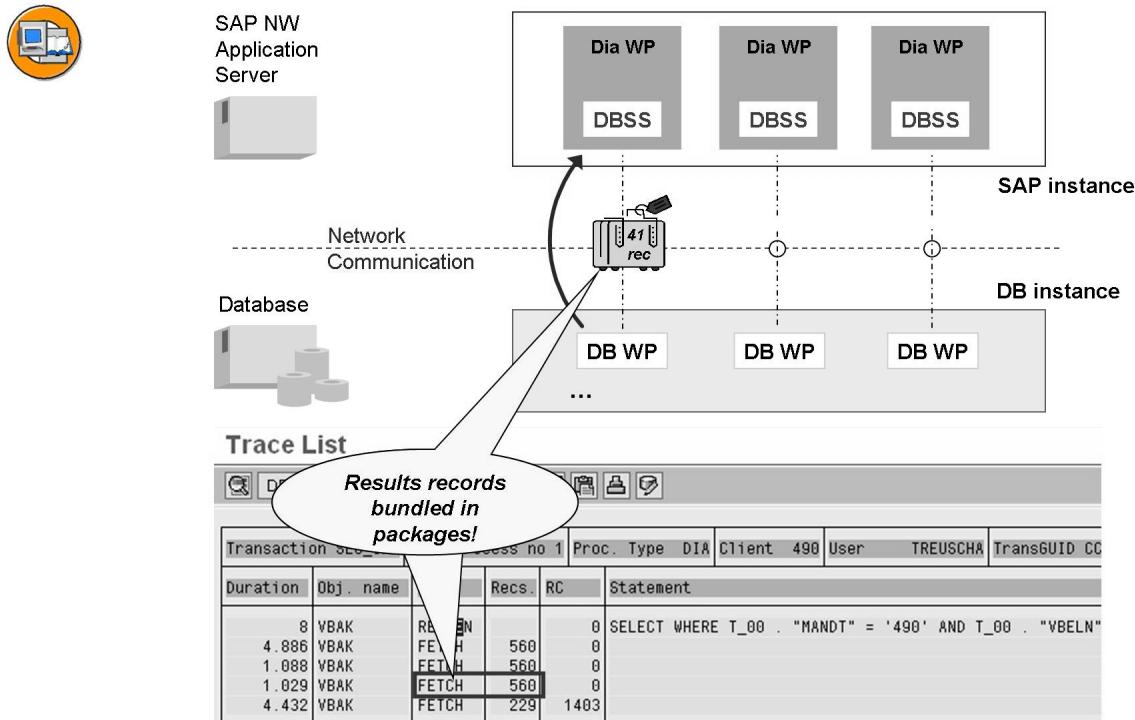


Figure 132: Performance Advantage for Views and Joins

- **Note:** The advantage of the join technology is in the bundling of the resulting data, which is returned from the database to the application server via the network.

For nested selects, all dependent records of the inner table (the driven table) would be read for each record and then transferred in a separate fetch operation. The SQL trace illustrates this perfectly. In the trace, contrast a join and a nested select. The number of fetch operation usually varies greatly between the two cases (irrespective of runtime differences!).

 **Note:** Both joins and views have the disadvantage that some data of the higher level table occurs redundantly in the results set in case of a 1:n relationship between the outer and inner table. This can considerably increase the amount of data transferred to the application. For this reason, you should use a field list with fields that you really need when you form the join.



**Hint:** As already described, you are thinking about which table reduces the results set and search quantity the most and are trying to “sell” this table to the optimizer. The optimizer usually starts reading in the table that reduces the search quantity the most.

In the selection of the fields in the ON and WHERE conditions, use as many fields as possible from this table, if the same fields occur in several tables. This might also have a positive effect on the optimizer response.

#### We will now present some **background information on the internal processing of joins at Oracle:**

For processing ABAP joins, views and subqueries, the Oracle database optimizer can make use of the following important access strategies, amongst others:

- *Nested Loop*
- *Sort Merge Join*
- *Hashed Join*



```
SELECT vbak~vbeln vbuks~vboobj INTO ...
  FROM vbak INNER JOIN vbuks
  ON vbak~vbeln = vbuks~vbeln
 WHERE vbak~objnr = '0000777'.
ENDSELECT.
```

Nested loop

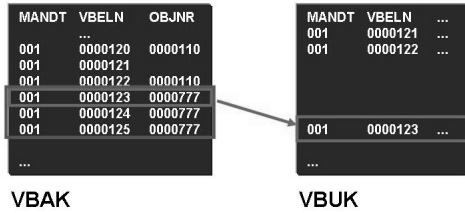


Figure 133: Optimizer Strategy: Nested Loop

When a join is processed using the *Nested Loop* strategy, two steps are performed:

- Specification of access sequence (driving and driven table)
- Data is selected from the tables

The database optimizer first determines the order in which the tables in the join are to be accessed. To do this, it uses the WHERE clause to estimate the number of data records or blocks to be returned per table in the join. For the outer driving table, it selects the table with the lowest number of returned table records or data blocks, because it assumes that in total the fewest data blocks need to be read in this table. Therefore, the goal once more is to minimize the number of data blocks to be read (indexes and table blocks). If there are more than two tables in the join, the inner table is selected in the same way.

In the second step, the table records from the outer table are first selected. For these table records, the table records from the next inner driven table are read according to the join condition, and so on.

For the SQL statement above, the optimizer selects the table VBAK as the outer table, because a selective WHERE condition exists for the field VBAK-OBJNR. For each table record that fulfills the WHERE condition, table records from VBUK are selected according to the JOIN condition.



```
SELECT vbap~vbeln vbak~vobj INTO ...
  FROM vbap INNER JOIN vbak
  ON vbap~vbeln = vbak~vbeln
 WHERE vbap~cuobj = '12345'
   AND vbak~objnr = '0000777'.
ENDSELECT.
```

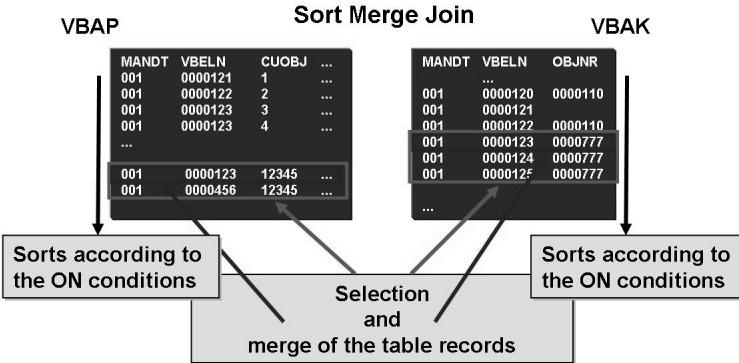


Figure 134: Optimizer Strategy: Sort Merge Join

When a join is processed using the *Sort Merge Join* access strategy, the following steps are performed:

- The table records that correspond to the WHERE clause are selected
- The tables in the join are sorted according to the JOIN condition
- the table records are merged

For the SQL statement above, the selective WHERE condition for field CUOBJ, table VBAP, and the selective WHERE condition for field OBJNR, table VBAK are evaluated. The resulting sets are sorted according to VBELN. The results are then merged.

If selective WHERE conditions exist for the relevant tables, a *Sort Merge Join* is very effective. If the JOIN conditions are not selective for any of the relevant tables, the *Sort Merge Join* access strategy is more effective than a *Nested Loop*. If there are more than two tables in the join, you can combine the *Nested Loop* and *Sort Merge Join* access strategies.

#### Summary of access strategies for DB joins at Oracle:

**Nested Loop:** This strategy is relevant for database views and ABAP JOINS. First, the WHERE clause is used as a basis for selecting the (outer) table to be used for access. Next, starting from the outer table, the table records for the inner tables are selected according to the JOIN condition.

**Sort Merge Join:** First, the WHERE clause is evaluated for all tables in the join, and a resulting set is produced for each table. Each resulting set is sorted according to the JOIN conditions and then merged, also according to the JOIN conditions.

**Hashed Join:** Information about Hashed Join is available in OSS notes 124361, 830576.

**Important information and additional tips for join formation:**

- Database optimizer determines order of access
- Access to outer table should be as selective as possible (selective WHERE conditions)
- For each data record in the outer table, as few records as possible should exist in the inner table (selective JOIN conditions)



**Join Basics**

**Nested Selects**

**Views in the Dictionary**

**ABAP Joins**

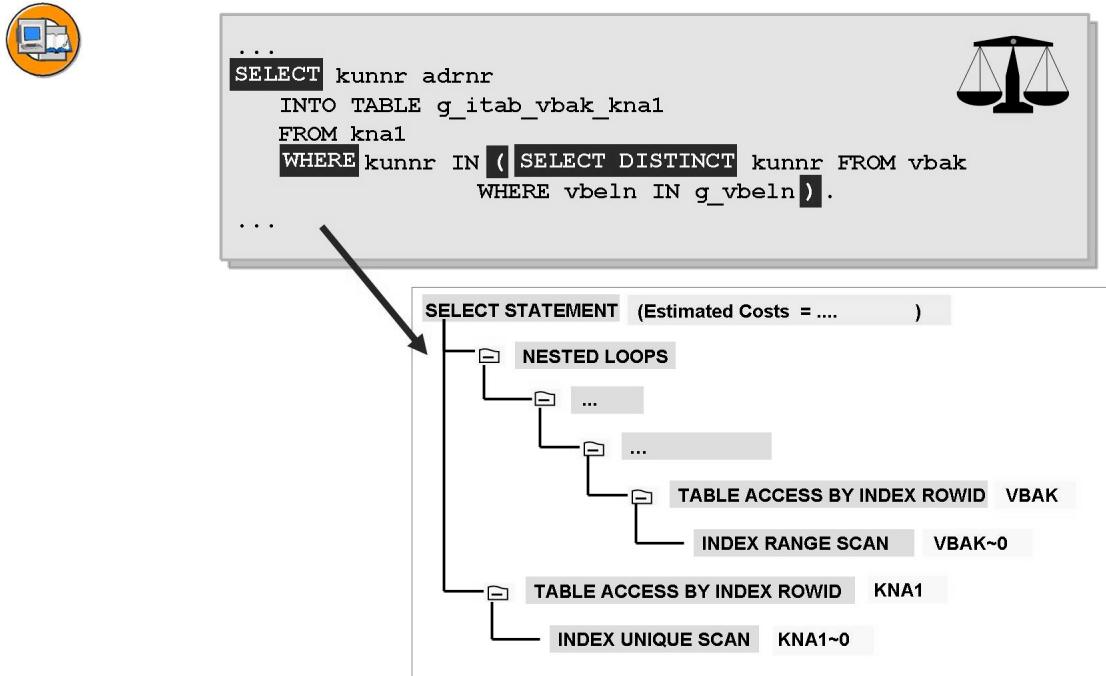


**Subquery**

**Additional Special Topics**

Figure 135: Accessing Multiple Tables (5)

## The Subquery



**Figure 136: Subquery**

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

A SELECT statement with a subquery has a more restricted syntax than a SELECT statement without a subquery: `SELECT ... FROM dbtab [WHERE ...] [GROUP BY ...] [HAVING ...]`. If the subquery returns one value, standard relational operators can be used, with the exception of LIKE and BETWEEN.

In the example above, the goal is for the subquery to return several rows with one value each. If you want to compare all values returned, use IN.

Subqueries in which fields from the main query are used in the WHERE clause are called **correlated subqueries**. If subqueries are nested, each subquery can use all the fields from the higher-level subqueries in the hierarchy.

You should also formulate positive subqueries whenever possible. Negative formulations can cause inefficient accesses to the database if there is no suitable index.



Nested SELECT Statements

Join Basics

Views in the Dictionary

ABAP Joins

Subquery



Additional Special Topics

Figure 137: Accessing Multiple Tables (6)



```
LOOP AT gt_vbak INTO gs_vbak.
  SELECT kunnr adrnr FROM kna1
    APPENDING TABLE gt_kna1
    WHERE kunnr = gs_vbak-kunnr.
ENDLOOP.
```

## SELECT inside LOOP

```
DESCRIBE TABLE gt_vbak LINES gv_lin.
IF gv_lin EQ 0.
  MESSAGE 'itab empty' TYPE 'I'.
  EXIT.
ENDIF.
```

Check whether driver table is empty

```
SORT gt_vbak BY kunnr.
DELETE ADJACENT DUPLICATES
  FROM gt_vbak COMPARING kunnr.
```

Eliminate duplicates

```
SELECT kunnr adrnr FROM kna1
  INTO TABLE gt_kna1
  FOR ALL ENTRIES IN gt_vbak
  WHERE kunnr = gt_vbak-kunnr.
```



## SELECT with FOR ALL ENTRIES

Figure 138: SELECT FOR ALL ENTRIES !

The SELECT ... FOR ALL ENTRIES addition ensures that SQL statements are divided into several SELECT statements by the database interface. The individual SELECT statements are processed on the database and the results are buffered in the database interface. The database interface eliminates duplicate data records from the results set and transfers the results set to the ABAP program.

This variant enables you to create a JOIN between an internal table and a database table. In the example above, only the records from table KNA1 are read that have customer numbers in the internal table gt\_vbak. For the customer number (the join criterion between itab and DB table) there should be an index on the database table to be read! This is essential for achieving good performance, especially with For All Entries.

**In connection with SELECT ... FOR ALL ENTRIES the following problems occur:**

- **You cannot have an empty driver table.** If the driver table is empty, selection is not limited. In particular, WHERE clause conditions are **not** evaluated if they do not refer to the internal table.
- **Duplicate table entries** must be **deleted** from the internal driver table before the SELECT FOR ALL ENTRIES is executed. If they are not deleted, identical data is read unnecessarily from the database.
- The parameter **rsdb/max\_blocking\_factor** must be implemented according to SAP's database-specific recommendations.

If you want to replace nested SELECTs with SELECT FOR ALL ENTRIES, this only eliminates the nesting and its disadvantages partially. In general, you bundle the inner SELECT select statements into packages, for example, of 5 statements each (depending on the database and its release). This reduces the transfer effort required and avoids identical SQL statements. The access sequence is defined in the ABAP coding, like for nested SELECTs.

When you use SELECT FOR ALL ENTRIES, you have to define whether an INNER or OUTER JOIN is used in ABAP. If you want to read large data volumes, you should only use SELECT FOR ALL ENTRIES in exceptional cases.

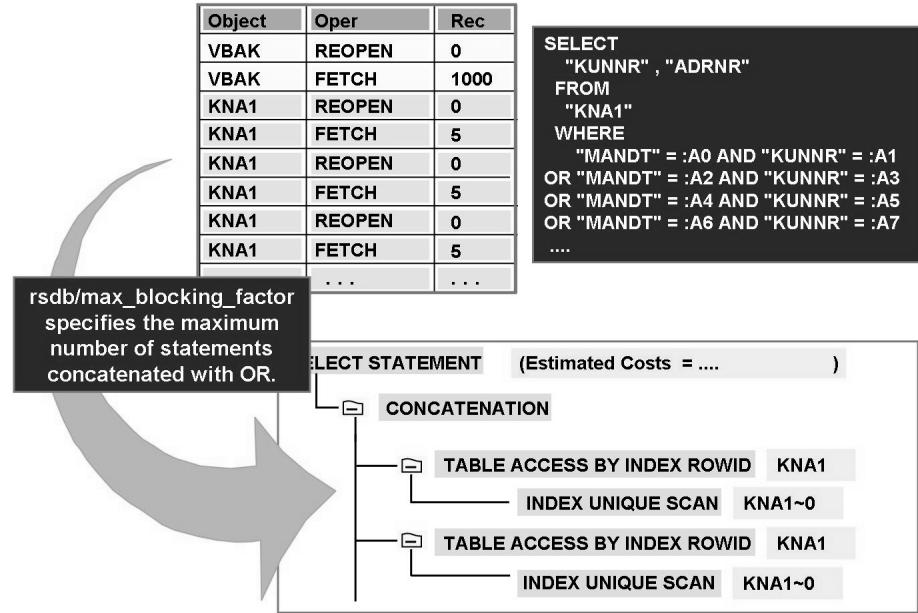


Figure 139: SELECT FOR ALL ENTRIES II

Each SELECT statement processed on the database is executed as a concatenation of individual accesses (for example, index range scans or index unique scans). Concatenation is performed over the entries in the internal driver table. Because SELECT ... FOR ALL ENTRIES can become complex, avoid combining it with RANGES tables.

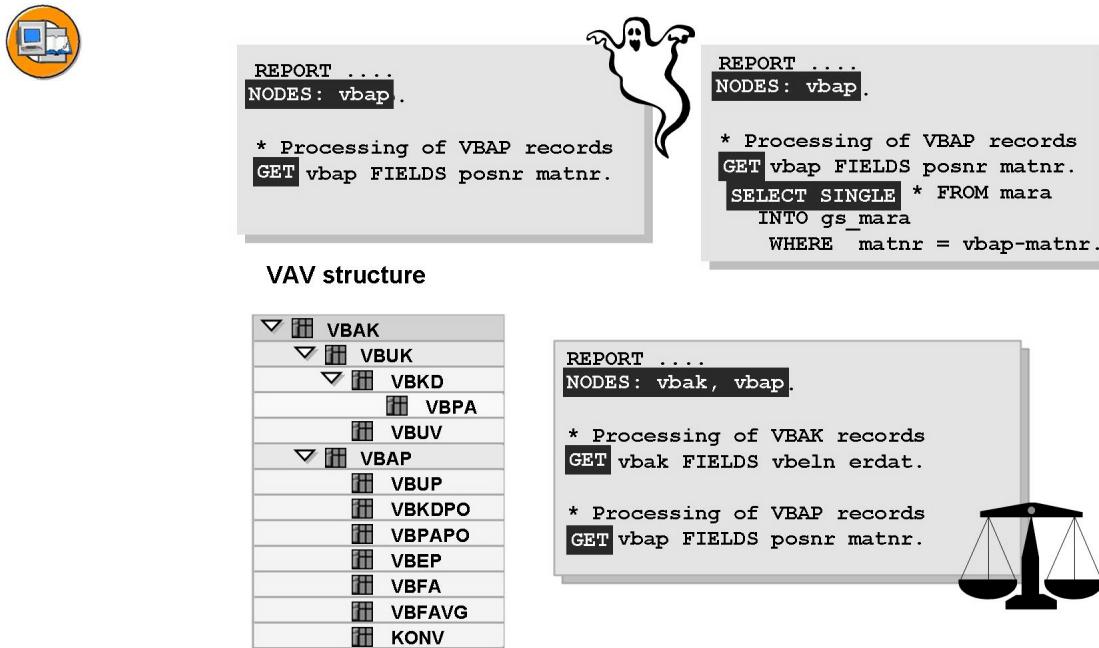
The implementation of the WHERE clause for individual SELECT statements depends on the database and is realized by the database interface.

The profile parameter **rsdb/max\_blocking\_factor** controls the maximum number of individual WHERE conditions that can be concatenated. SAP preconfigures this parameter according to the database. For Oracle it is between 5 - 15 depending on the release; for MS SQL-Server it is 50.

→ **Note:** Also see notes 170989 and 634263.

#### Historical background:

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



**Figure 140: Using Logical Databases**

Logical databases are encapsulated, reusable ABAP read routines. Besides the read functionality, they also offer users selection screens and automatic authorization checks, which every developer would have to program again otherwise.

The structure of a logical database defines the hierarchy, and thus the read sequence of the nodes (usually tables). The depth of data read depends on a report program's GET events. The logical database reads down to the deepest GET event of all nodes that lie in the direct access path.

GET events are processed several times in the program, so a GET statement is like loop processing. Therefore, you should not program a SELECT statement within a GET event, as this ultimately corresponds to a nested SELECT.

Because the order of reads is predetermined, only program a GET event that refers to a node lower down in the hierarchy if the data above it is also required. The logical database reads the key fields of these nodes as well (VBAK, in the example above). In this situation, it is better to program the SELECT statements yourself than to use the logical database.

If you do not require all fields in your program, and particularly if the fields in your table are wide, always use a FIELDS addition when you formulate a GET statement. The statement `GET <node> FIELDS <f1> <f2> ..` corresponds to a `SELECT <f1> >f2> ...` and should be evaluated similarly.



## Exercise 8: Joins Compared to Nested Selects

### Exercise Objectives

After completing this exercise, you will be able to:

- Estimate advantages of joins in reading across several tables
- Describe disadvantages of nested selects

### Business Example

Mr. Jones, a software developer at a major corporation that uses SAP, is supposed to develop a new business application to analyze mass data. He knows that the reading of data via nested joins puts a lot of strain on the system. He wants to use joins as an alternative.

#### Task:

Continue working with your old program Z\_##\_JOIN3 or use template SAPBC490\_TEMP\_JOIN3

1. The subroutine for the triple join is to be implemented. The result is to be compared to the already programmed triple nested select.
2. Once you have completed the trip join, compare the runtime to that of the nested select. In the trace, pay attention to the runtime and, in particular, to the number of fetches.
3. Optional sub-task: Implement the triple join using a view in the dictionary and compare all relevant tasks again. Note the use of the client in the view fields and ON conditions.

## Solution 8: Joins Compared to Nested Selects

### Task:

Continue working with your old program Z\_##\_JOIN3 or use template SAPBC490\_TEMP\_JOIN3

1. The subroutine for the triple join is to be implemented. The result is to be compared to the already programmed triple nested select.
  - a) See excerpt from source code in solution. Also note the F1 help for the joins (under Select - FROM clause)
2. Once you have completed the trip join, compare the runtime to that of the nested select. In the trace, pay attention to the runtime and, in particular, to the number of fetches.
  - a) You should detect a significantly lower number of packages (fetches) and the runtime should also be significantly lower!
3. Optional sub-task: Implement the triple join using a view in the dictionary and compare all relevant tasks again. Note the use of the client in the view fields and ON conditions.
  - a) See solution view BC490\_VIEW

### Result

#### Source text of the model solution:

```
*&-----*
*& Report  SAPBC402_SQL2_JOIN3          *
*&-----*
*& show a join on three tables
*&-----*
REPORT ZBC490_SQL2_JOIN3.

* ... calling different form-routines
  perform join3.

*-----*
*      FORM join3                      *
*                                         *
* to have a fair comparison we select exactly the same   *
* data as in the other exercises before                   *
```

*Continued on next page*

```
*-----*
FORM join3.
    GET RUN TIME FIELD t1.

***  
*** Trace with ST05 and look at the result, advantage ?  
***  
  
SELECT vbak~vbeln vbap~posnr vbap~matnr
      vbak~erdat vbak~kunnr knal~name1
      INTO (rec~vbeln, rec~posnr, rec~matnr, rec~erdat,
            rec~kunnr, rec~name1)
      FROM ( vbak INNER JOIN vbap
              ON vbak~vbeln = vbap~vbeln )
      INNER JOIN knal
              ON      knal~kunnr = vbak~kunnr
      WHERE vbak~vbeln IN vbeln.  
  
*     WRITE: / rec~vbeln COLOR COL_HEADING,
*             rec~posnr COLOR COL_HEADING,
*             rec~erdat COLOR COL_POSITIVE,
*             rec~kunnr,
*             rec~name1.  
ENDSELECT.
GET RUN TIME FIELD t2.
t2 = t2 - t1.  WRITE: / 'runtime = ', t2.
ENDFORM.
```



## Exercise 9: (Optional) Access Using FOR ALL ENTRIES

### Exercise Objectives

After completing this exercise, you will be able to:

- Estimate advantages and disadvantages of FOR ALL ENTRIES
- Program a FOR ALL ENTRIES statement

### Business Example

Mr. Jones, a software developer at a major corporation that uses SAP, is supposed to develop a new business application to analyze mass data. He knows that the reading of data from a database table via a driving internal table is possible using FOR ALL ENTRIES.

#### Task:

Short description of the exercise:

The exercise consists of the implementation of the functional instance method GET\_FLIGHT\_BOOKINGS, which is supposed to return the suitable booking data for the flights internally using FOR ALL ENTRIES. The import parameter of the method shall be the airline, and the returning parameter is supposed to be the internal table with the booking data (SBOOK interface). So there is a SELECT on the SBOOK table according to the data of the inner driving table LIST\_OF\_FLIGHT (this itab is filled in the constructor of the class).

1. Template program SAPBC490\_TEMP\_FAE is specified. Copy this program to your own Z\_##\_FAE.

This time, select the INCLUDES because you have to copy include SAPBC490\_TEMP\_AIRPLANES\_FAE to your own include Z\_##\_TEMP\_AIRPLANES\_FAE. Do not forget to select the includes to be copied!

After copying, activate the entire program first!

2. Take a look at the last lines of the main program. There, you will already find the call of method GET\_FLIGHT\_BOOKINGS for the US Hercules plane. The method is supposed to determine internally all flight bookings for Lufthansa flights in the internal table list\_of\_flights using For All Entries.

*Continued on next page*

The main program for access is already ready. The context of the method is already predefined, the visualization of bookings in the main program is implemented. What is missing is the coding of the actual method.

3. Rate the design and the solution of this requirement.

Is the use of For All Entries optimal in this case?

Is it optimal to already read all flights for the plane type in the constructor of the planes (LCL\_AIRPLANE) and then use FAE to access the bookings in SBOOK?  
Develop alternatives.

## Solution 9: (Optional) Access Using FOR ALL ENTRIES

### Task:

Short description of the exercise:

The exercise consists of the implementation of the functional instance method GET\_FLIGHT\_BOOKINGS, which is supposed to return the suitable booking data for the flights internally using FOR ALL ENTRIES. The import parameter of the method shall be the airline, and the returning parameter is supposed to be the internal table with the booking data (SBOOK interface). So there is a SELECT on the SBOOK table according to the data of the inner driving table LIST\_OF\_FLIGHT (this itab is filled in the constructor of the class).

1. Template program SAPBC490\_TEMP\_FAE is specified. Copy this program to your own Z\_##\_FAE.

This time, select the INCLUDES because you have to copy include SAPBC490\_TEMP\_AIRPLANES\_FAE to your own include Z\_##\_TEMP\_AIRPLANES\_FAE. Do not forget to select the includes to be copied!

After copying, activate the entire program first!

- a) Proceed as usual for copying programs.
2. Take a look at the last lines of the main program. There, you will already find the call of method GET\_FLIGHT\_BOOKINGS for the US Hercules plane. The method is supposed to determine internally all flight bookings for Lufthansa flights in the internal table list\_of\_flights using For All Entries.

The main program for access is already ready. The context of the method is already predefined, the visualization of bookings in the main program is implemented. What is missing is the coding of the actual method.

- a) Note the following solution program with the For All Entries in method GET\_FLIGHT\_BOOKINGS..
3. Rate the design and the solution of this requirement.

Is the use of For All Entries optimal in this case?

Is it optimal to already read all flights for the plane type in the constructor of the planes (LCL\_AIRPLANE) and then use FAE to access the bookings in SBOOK? Develop alternatives.

*Continued on next page*

- a) Get to know the data model BC\_TRAVEL. You will notice that there is a hierarchical relationship between SFLIGHT and SBOOK and a relational relationship between SFLIGHT and SAPLANE. So the flights refer to a plane type from SAPLANE. The flight bookings in SBOOK depend on the existence of the flights in SFLIGHT.

Many planes are based on the same plane type (A319 in the example). If planes in this application are created using CREATE OBJECT, all flights for the plane type are loaded into the instance-specific internal table LIST\_OF\_FLIGHTS in the constructor. So this happens redundantly many times because there are hundreds of flights for a plane type (A319). This is a disadvantage of this application.

In the same way, the FAE access with the reading of data from SBOOK would also occur way too many times. Many objects would have the same data loaded in relatively large internal tables.

It would therefore be better to define this internal table statistically and only fill it when necessary, when a plane type is required.

Another option would be to do without the internal table LIST\_OF\_FLIGHTS and to try to solve the task using a join between SFLIGHT and SBOOK.

## Result

### Source code of the model solution:

```

*
*  MAIN Programm SAPBC490_SQL2_FAE_SOLUTION
*
*...
data: it_bookings TYPE STANDARD TABLE OF sbook.
*...
***** The following plane is used to show the solution of the exercise!
***** The method get_flight_bookings( ) processes a FOR ALL ENTRIES
***** cargo Plane *****
CREATE OBJECT r_cargo
EXPORTING
  im_name      = 'US Hercules'
  im_planetype = 'A319'
  im_cargo     = 533.

WRITE: / 'All Bookings for the plane US HERCULES A319: '.

```

*Continued on next page*

ULINE.

```

it_bookings = r_cargo->get_flight_bookings( im_carrid = 'LH' ).

***** it_bookings is the result of your exercise!!!
      SORT it_bookings BY carrid connid fldate bookid.

      LOOP AT it_bookings INTO wa_bookings.
        WRITE: / wa_bookings-carrid, wa_bookings-connid, wa_bookings-fldate,
               wa_bookings-bookid, wa_bookings-customid.
      ENDLOOP.
*****
*** INCLUDE-Program with all the airplane classes
*** Demo for the FAE inside the method get_flight_bookings *****
*
*   LCL_AIRPLANE CLASS .....
*
METHOD get_flight_bookings.

*** look if driving table is empty
  IF NOT list_of_flights IS INITIAL.

*** sort the driving table
  SORT list_of_flights BY carrid connid fldate.
*** delete duplicates is not neccesarry beacause there can not be duplicates

  SELECT * FROM sbook
    INTO TABLE list_of_bookings
    FOR ALL ENTRIES IN list_of_flights
      WHERE carrid = im_carrid
      AND connid = list_of_flights-connid
      AND fldate = list_of_flights-fldate.

  re_bookings = list_of_bookings.
ENDIF.
***** the else-part is left out; also raising exception is not shown here...
***** for didactical reasons; usually exception should be raised here
ENDMETHOD.           "get_flight_bookings_for_planetype

```



## Lesson Summary

You should now be able to:

- Explain the advantages of join/ views compared to nested SELECTS and FOR ALL ENTRIES



## Unit Summary

You should now be able to:

- Explain the advantages of join/ views compared to nested SELECTS and FOR ALL ENTRIES



# Unit 6

## Buffering Strategies

### Unit Overview

#### Contents:



- Table buffering
- Import/ export
- Set / Get parameters
- Shared objects
- High-performance programming with internal tables



### Unit Objectives

After completing this unit, you will be able to:

- Explain the different buffering strategies
- Use buffering strategies to avoid database accesses

### Unit Contents

Lesson: Buffering Strategies .....	208
Exercise 10: Table Buffering .....	259
Exercise 11: Internal Tables.....	265

# Lesson: Buffering Strategies

## Lesson Overview

### Contents:



- Table buffering
- Import/ export
- Set/ get parameters
- Shared objects
- High-performance programming with internal tables



## Lesson Objectives

After completing this lesson, you will be able to:

- Explain the different buffering strategies
- Use buffering strategies to avoid database accesses

## Business Example

In the context of a customer development, a company wants to buffer data in order to speed up access to the data and reduce the database load. Different buffering approaches are to be analyzed.



### Buffer Strategy Overview

#### Table Buffering

#### Shared Objects

#### SAP / ABAP Memory

#### Internal Tables

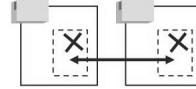
Figure 141: Buffering Strategies (1)

## Overview of Buffering Strategies



### Buffering Across Application Server Boundaries

- Buffering database tables on application servers



### Buffering on an instance of the application server

- Export/import via ABAP or shared memory
- SET / GET parameter via SAP memory
- Shared objects



### Buffering within programs

- Internal tables

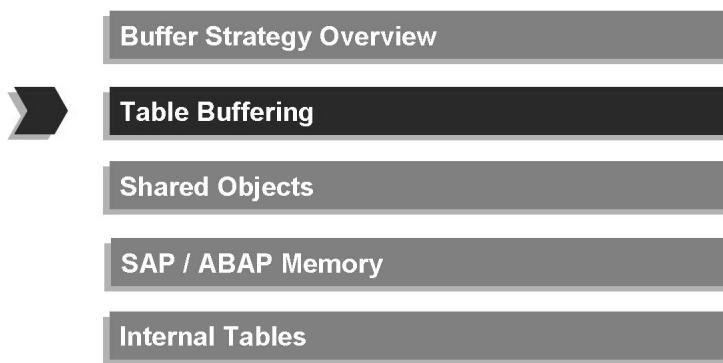


Figure 142: Buffering Strategies

For buffering strategies it must be distinguished which objects are supposed to be buffered on the one hand, and whether buffering is supposed to extend across programs or application servers on the other hand. Local buffering within a program is also conceivable, for example, via internal tables. The visibility and usability of the buffer as well as the mechanisms for renewing and managing the buffer are decisive.

## Buffering Database Tables

The aim of SAP table buffering is to buffer database table contents on the **instances of the application servers** to reduce accesses of the database, thus conserving the database resources (CPU load, main memory).



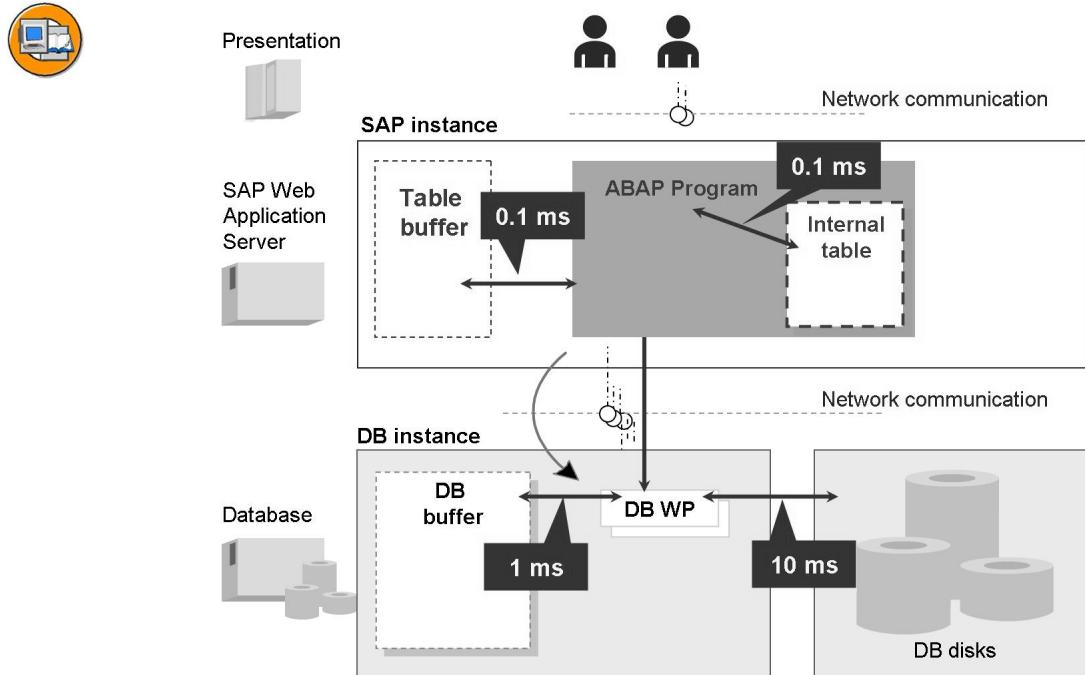
**Figure 143: Buffering Strategies (2)**

The SAP system is scalable at application server level. This means that you can add more application servers to an SAP system. The database, as the central data repository, is not scalable. That is, there can be only one database for each SAP system.

Table contents, repository objects, ABAP Dictionary objects, and so on, are data records on the database. Every time these objects are called, they must be read from the database and transferred to the application server. To avoid this, the objects are stored temporarily in the shared memory of the application servers in various buffer areas, for example:

- Table buffer (single-record buffer, generic buffer)
- Repository buffer
- Dictionary buffer
- Number range buffer
- Roll and paging buffers

## The SAP Table Buffer



**Figure 144: Table Buffering: Overview**

Records that are buffered on the application server can be accessed up to 100 times faster than data in the database. As a result, the work processes on the application server encounter shorter wait times, which means table buffering reduces the load on the entire SAP system.

Records in the table buffer are saved sorted by the primary index.

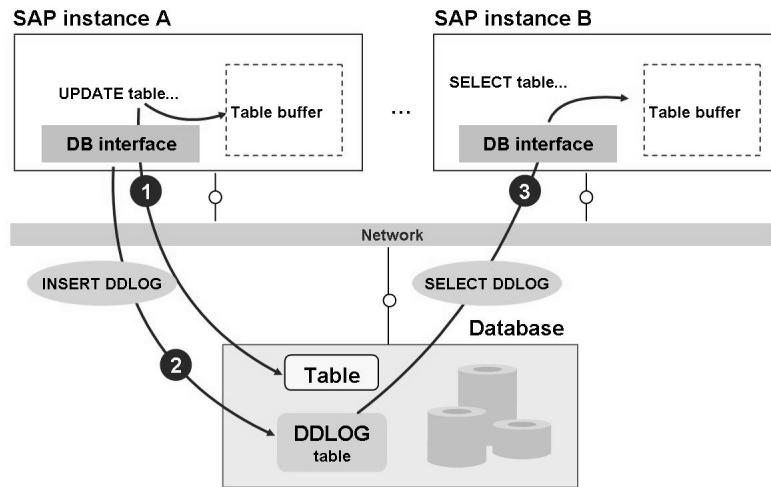


**Caution:** You cannot use secondary indexes to access the buffered tables.

## Buffer Synchronization



**Buffer synchronization of the SAP application servers:**  
**rdisp/bufreftime = every 1-2 minutes**



**Figure 145: The DDLOG Mechanism**

As the following diagram illustrates, buffer synchronization takes place in three steps:

1. In the first step of a database write operation to a buffered table, the corresponding records in the database are changed, and also changed in the table buffer of the local SAP instance (here: application server A). The table buffers of all other SAP instances (here: application server B) are not changed.
2. In the second step, the local database interface (here: application server A) propagates the changes to the other application servers by writing an appropriate entry to database table DDLOG. The database buffers of the other SAP instances are still not current.
3. In the third step, the database instances of the SAP instances start buffer synchronization every 1-2 minutes. Table DDLOG is queried to do this. If the returned records in table DDLOG indicate that change access has been made to buffered tables, the data in the table buffers of the non-local SAP instances are invalidated in accordance with the buffering type. As a result, SQL statements involving invalidated data are served by the database. The table buffers in the non-local SAP instances are loaded again.



**Caution:** Temporary data inconsistencies can occur during SAP table buffering, but are eliminated again by buffer synchronization. This synchronization is performed at fixed intervals, which you can configure using a system profile parameter. Therefore, you should only buffer data if the reading of obsolete data (within this limit) can be tolerated.



**Hint:** The system parameter for the synchronization interval is *rdisp/bufreftime*, which specifies the length of the interval in seconds. The value must be between 60 and 3600. A value between 60 and 240 is recommended.

## Buffering Types



### Full buffering (100%)

key1	key2	key3	data

### Generic buffering with one key field

### Generic buffering with 2 key fields

key1	key2	key3	data
001	A		
001	A		
001	B		
001	B		
002	A		
002	A		
002	B		
002	B		
002	B		
002	C		
002	C		
002	D		
003	A		
003	A		
003	A		
003	B		
003	B		
003	C		
003	C		
003	D		
003	D		
003	D		

## Single record buffering

key1	key2	key3	data
001	A	2	
001	A	4	
001	B	1	
001	B	3	
001	B	5	
002	A	1	
002	A	3	
002	A	6	
002	A	8	
002	B	1	
002	B	2	
002	B	3	
002	C	0	
002	C	3	
002	D	5	
003	A	2	
003	A	3	
003	A	6	
003	B	2	
003	B	4	
003	C	2	
003	C	3	
003	C	5	
003	C	8	
003	D	1	
003	D	2	
003	D	3	
003	D	4	

**Figure 146: Buffering Types**

We differentiate between the single record buffer, the generic table buffer and the buffer for complete tables (full table buffering). The single record buffer contains the records from the single record-buffered tables. Physically, the generic table buffer contains the records of the generically and fully buffered tables (technically, both of these buffering types use the same memory area on the application server).

### Full buffering

The first time the table is accessed, the entire table is loaded into the table buffer (generic table buffer). All subsequent accesses of the table can be served from the table buffer.

### Generic buffering

If you want to buffer a table generically, you first have to define the generic area. The generic area consists of the first n key fields of the table. If an SQL statement is executed with a specific instantiation of the generic area (such as `SELECT * FROM TAB1 WHERE KEY1 = '002'` or `SELECT * FROM TAB2 WHERE KEY1 = '002' AND KEY2 = 'A'`) for a generically buffered table, the corresponding records are loaded into the table buffer. All subsequent accesses with the same instantiation can be served from the table buffer.

### Single record buffering

Only single records are read from the database and loaded into the table buffer (single record buffer).

Note that `SELECT SINGLE` has to be used, however; a `SELECT` loop access would bypass the buffer.

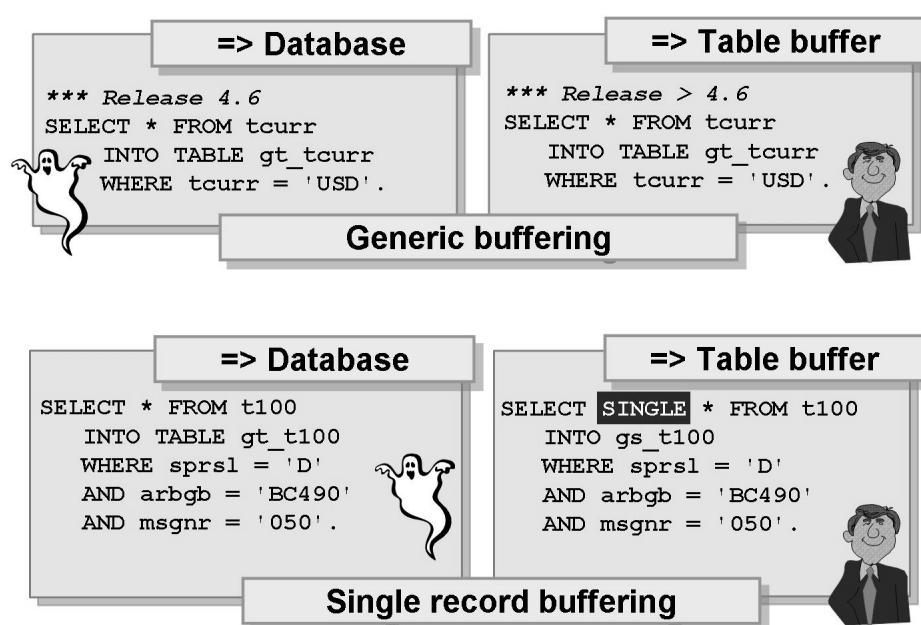


Figure 147: Examples of Buffer Access

The graphic shows examples of single record buffering and generic buffering. Please note that the generic buffering in TCURR tables differs depending on the SAP release. In Release 4.6 the generic key consisted of four fields, from Release 6.10 onwards it consists of only one field. This is, in higher releases, significantly larger parts of the TCURR table are buffered. For a production system with one client, for example, this would be the entire TCURR table. The graphic shows the same access in two different releases.

Each buffering type can be considered to be a variant of generic buffering with  $n$  key fields:

- Full buffering:  $n = 0$
- Generic buffering:  $0 < n <$  number of key fields
- Single record buffering:  $n =$  number of key fields



### Accesses past the buffer:



- Native SQL
- Subqueries, ABAP Joins
- **SELECT ... BYPASSING BUFFER**
- **SELECT FOR UPDATE**
- Aggregate functions (COUNT, MIN, MAX, SUM, AVG)
- **SELECT DISTINCT ...**
- **WHERE condition with "IS NULL"**
- **ORDER BY, GROUP BY (HAVING)**
- **For tables buffered by individual record:**
  - All SQL statements other than **SELECT SINGLE ...**
- **For generically buffered tables:**
  - All SQL statements other than **SELECT \* .. with WHERE cond. of the form field = value for all fields of the generic area**

**Figure 148: Statements Bypassing the Table Buffer**



**Note:** Please note the instructions in the above graphic. These accesses never address the table buffer. So even if the database table was buffered, they would bypass the buffer.

Note that the ORDER BY PRIMARY KEY is an exception. It accesses the buffer!

The following diagram should help you decide whether to use table buffering or not.



### When can you buffer tables?



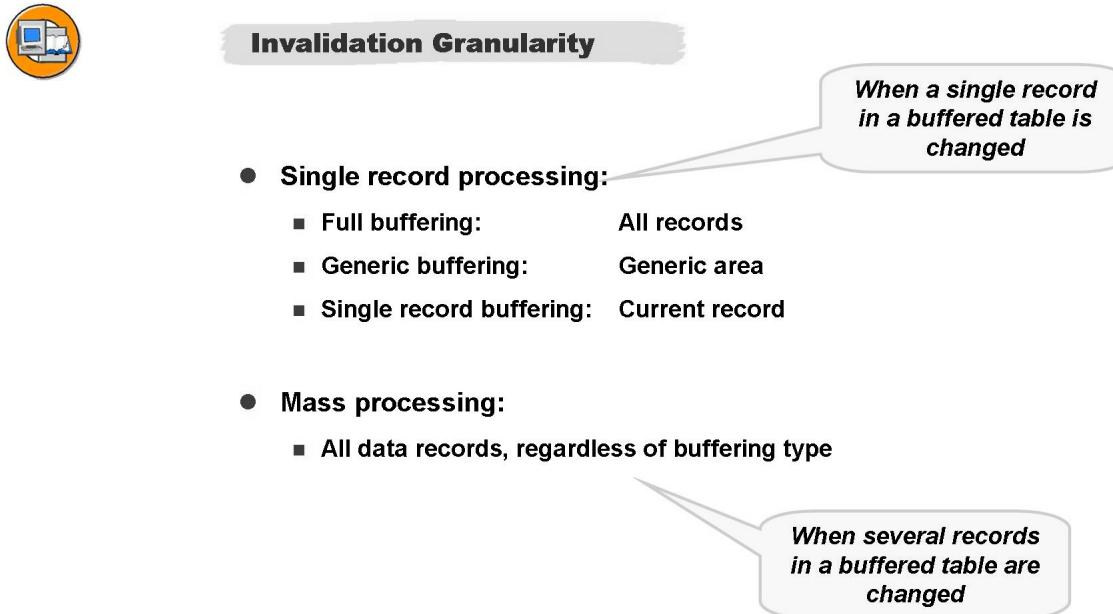
- **Criteria for buffering (rules of thumb):**
  - Small tables, < 10 MB
  - Frequently read, few changes
  - Temporary inconsistency (dirty read) tolerable
  - Accessed primarily using key fields
  - Access using secondary indexes not possible
  - Check memory before buffering additional tables
- **Never buffer transaction data**
  - Too large, too many changes
- **In general, do not buffer master data**
  - Too large, too many access paths
- **In general, buffer Customizing data**
  - Small, few changes

**Figure 149: Criteria for Table Buffering**

The records of the buffered tables are held redundantly in the shared memory of the participating application servers. As such, you should only buffer tables that are relatively small. Tables that are larger than 10 MB should only be buffered in exceptional cases. To keep the number of invalidations and subsequent buffer load operations as small as possible, you should only buffer tables that are accessed primarily for read access (change accesses < 1% of read accesses). Immediately after a change access to a buffered table, the table buffers of the non-local SAP instances are no longer current (see Buffer Synchronization). Inconsistent data may be read during this period. This must be taken into account when using buffering and should be tolerable. The records of buffered tables are saved sorted by the primary key. Accordingly, you should use key fields to access them. Secondary indexes cannot be used.

Before you decide to buffer tables, you have to consult your system administrator to make sure that sufficient space is available in the main memory (in Release 4.6: single record buffer approx. 40 MB minimum; generic table buffer approx. 80 MB minimum). Otherwise the table buffers could overwrite each other, which would be counterproductive. Before you change the buffer settings for standard SAP tables, you should search for appropriate Notes in the SAP Notes system. You will need a modification key to perform the changes.

## Invalidation Granularity



**Figure 150: Invalidating Buffered Records**

When database table contents are changed, the SAP buffer contents have to be invalidated and refreshed. Again, the buffering type is relevant here: Work area mode means the database is changed through single-record access, using the ABAP statements UPDATE/INSERT/MODIFY/DELETE dbtab (FROM wa) or INSERT dbtab VALUES wa. In contrast, you can also change a database table in set mode (mass processing). Database update access as mass processing is formulated in ABAP with the UPDATE/INSERT/MODIFY/DELETE dbtab FROM itab, UPDATE dbtab SET <field> = <value> WHERE <field> = <condition> or DELETE dbtab WHERE <field> = <condition> statements.

Database update accesses generally invalidate all the records in **fully buffered tables**.

For **generically buffered tables** in work area mode, the records whose generic area has the same expression as the work area fields in the SQL update statement are invalidated. If a generically buffered table is accessed in set mode, all the records are invalidated.

If a **single-record-buffered record** is accessed in work area mode, only the changed single record is invalidated. If an update access is performed in set mode, the entire single-record-buffered table is invalidated.

→ Note: Therefore, the granularity of the invalidation corresponds to the granularity used to fill the table buffers.

### Example of a Access to the Generic Table Buffer



**Caution:** As the following example of an access to a generically buffered table, a copy is created of the VBAP table. The copy, table ZVBAP\_GEN is set to *Generic Buffering* with 2 key fields (MANDT / VBELN). Note that this would be fatal in real live. In praxis, there are also change accesses to table VBAP with a high interval rate. It is not a candidate for buffering in a production SAP system.

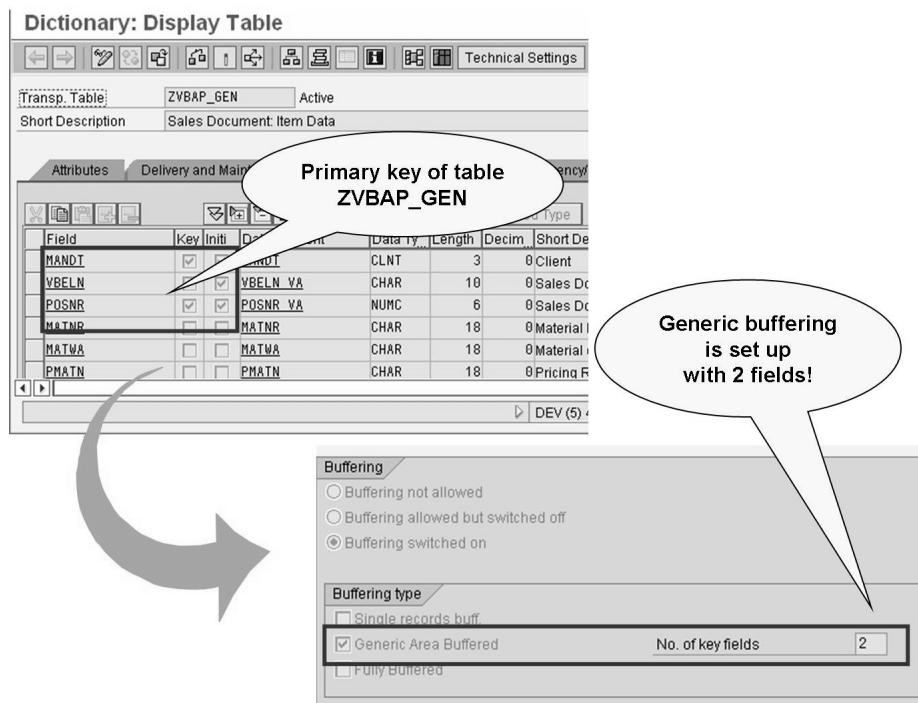


Figure 151: Experiment: VBAP with Generic Buffering via MANDT/VBELN

Important: Of course, you would not buffer the large VBAP table in practice.

In experiment ZVBAP\_GEN, the table has a triple primary key: MANDT / VBELN / POSNR. The first two fields are defined as the generic key.

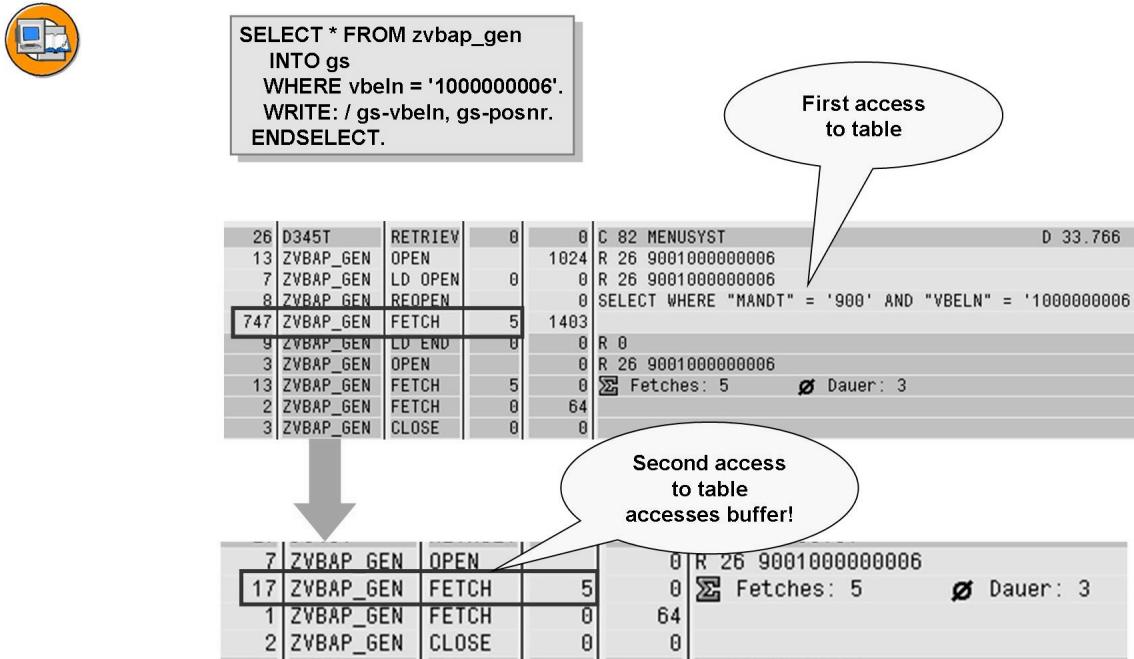


Figure 152: The Second Access Uses the Buffer

→ **Note:** Buffer accesses are always displayed in blue color in the SQL trace. Database accesses in yellow color.

In the first access to the generic area (qualified VBELN suffices) the data is loaded from the database to the buffer; in the second access, access takes place via the buffer of the application server. Of course, buffer access performs much better.

The system in the example is a central system with application server and database on one piece of hardware. If, as usual in practice, you were using two different computers and data exchange would therefore take place via the network, the difference in performance would be even greater.

**Hint:** If at least the generic key is qualified with EQ (or =), the access takes place via the generic buffer.

### Example of a Suggested Solution for a Fully Buffered Table

→ **Note:** In the following example, SCARR is a fully buffered table. In practice this would make sense because SCARR contains a manageable number of airlines. There must not be too many changes to this table either.

In the example, read and change accesses to the table take place via two application servers.

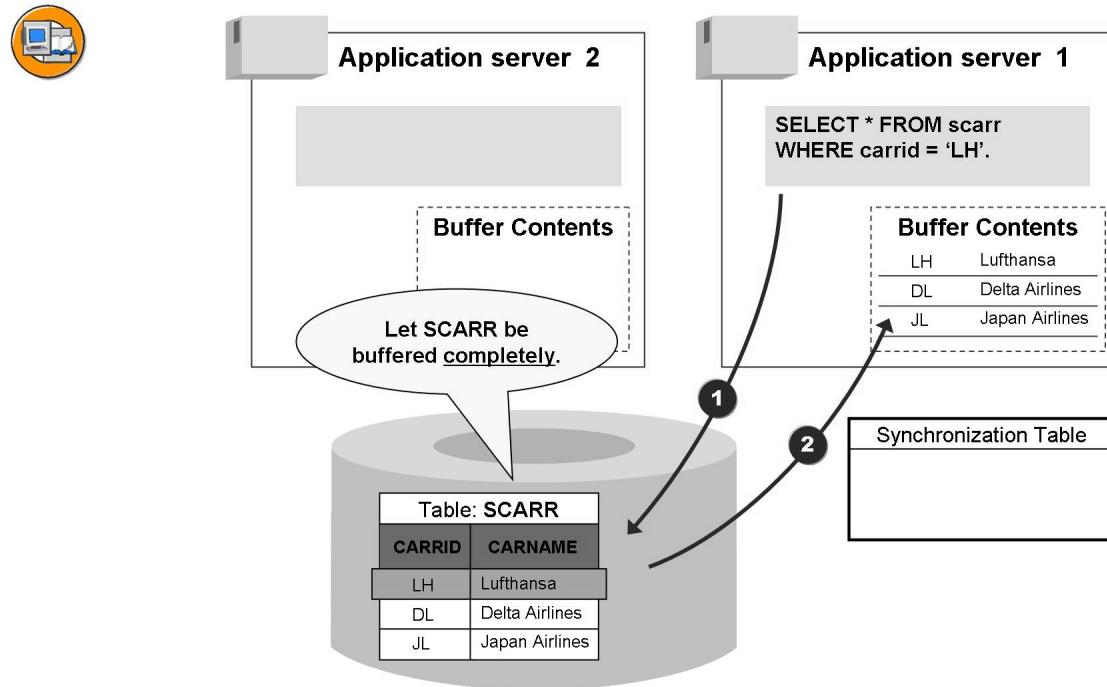
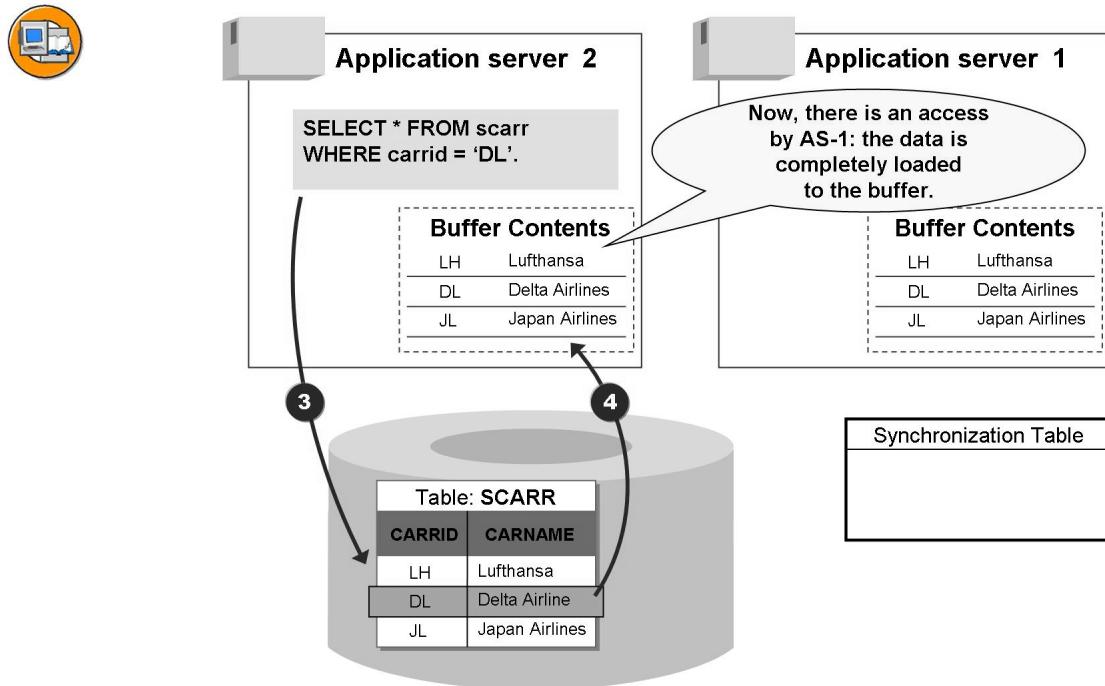


Figure 153: Filling the Buffer in AS-2

AS-1 reads (1) a record of the table. The entire table is loaded into the buffer of AS-1 (2).



**Figure 154: AS-2 Accesses the Table**

AS-2 also reads (3) from the table. There, the entire table is also loaded to the buffer (4).

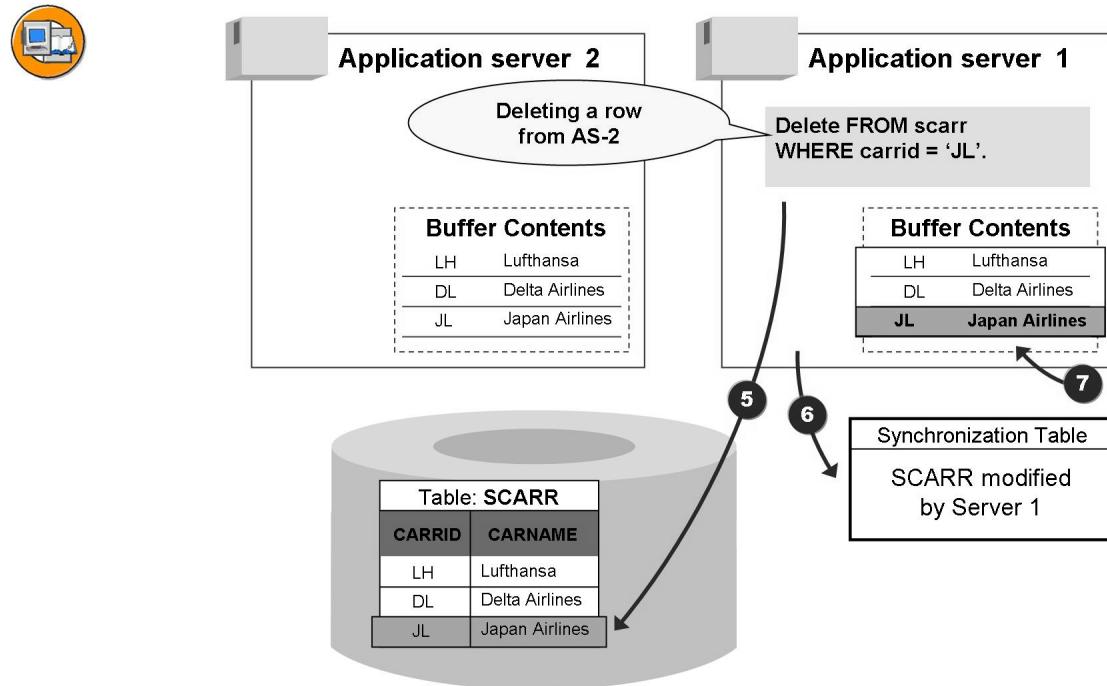
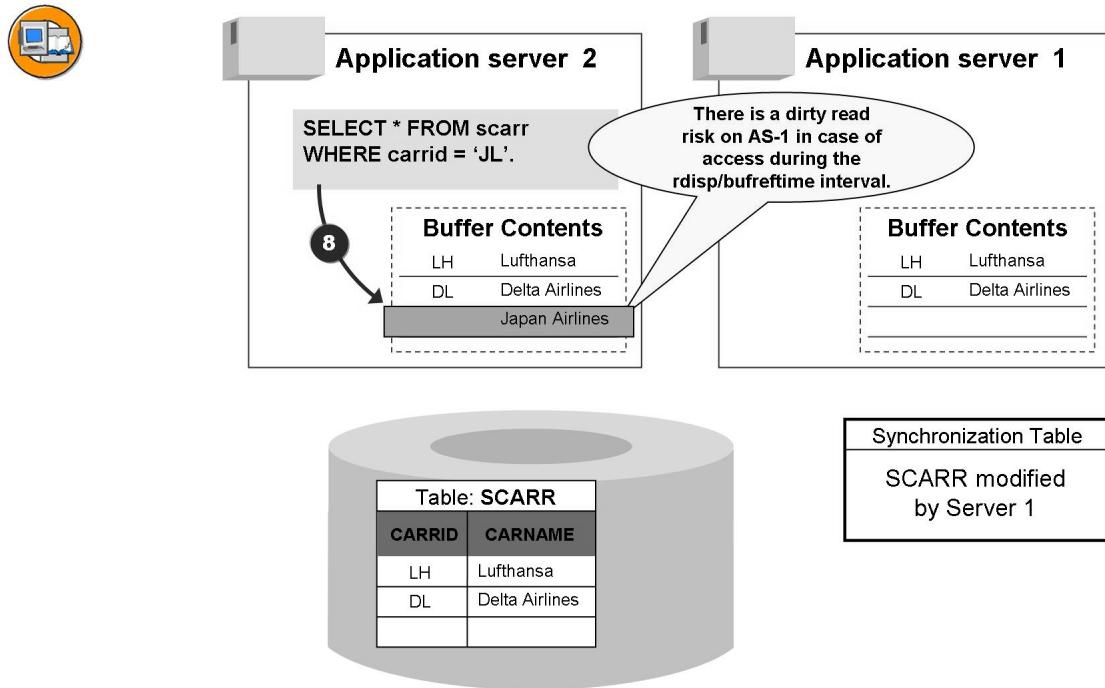


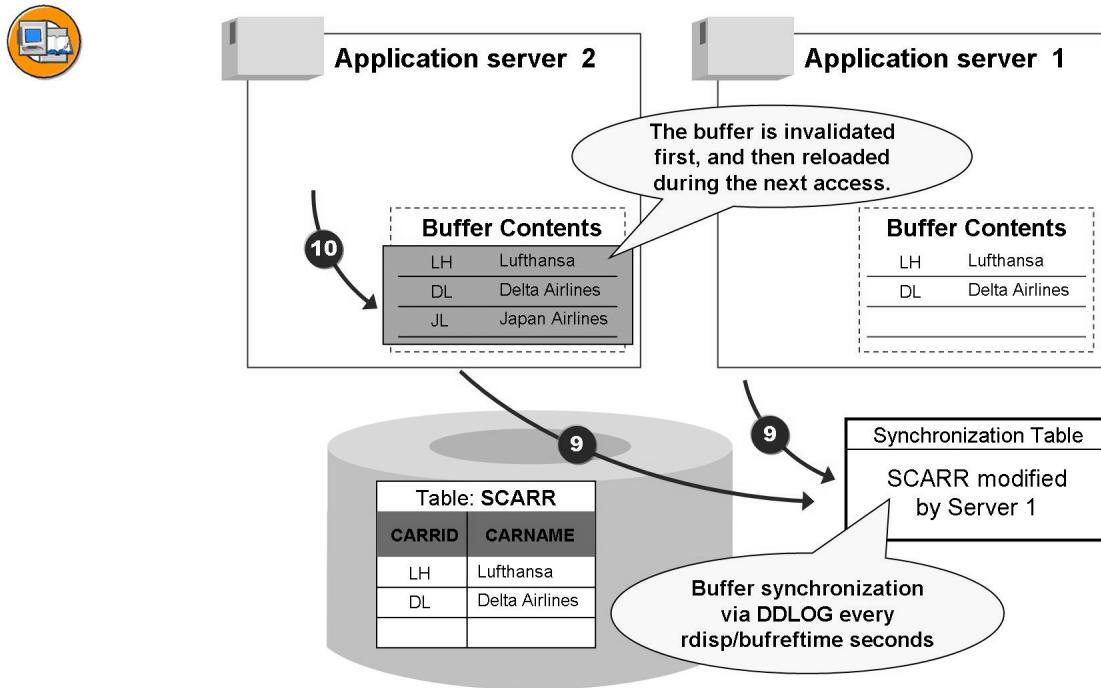
Figure 155: AS-1 Deletes a Record from the Table

AS-1 deletes a record from table SCARR. The data is physically changed in the database (5). An entry is written in synchronization table DDLOG (6). The one deleted record of the local buffer on AS-1 is also deleted (7).



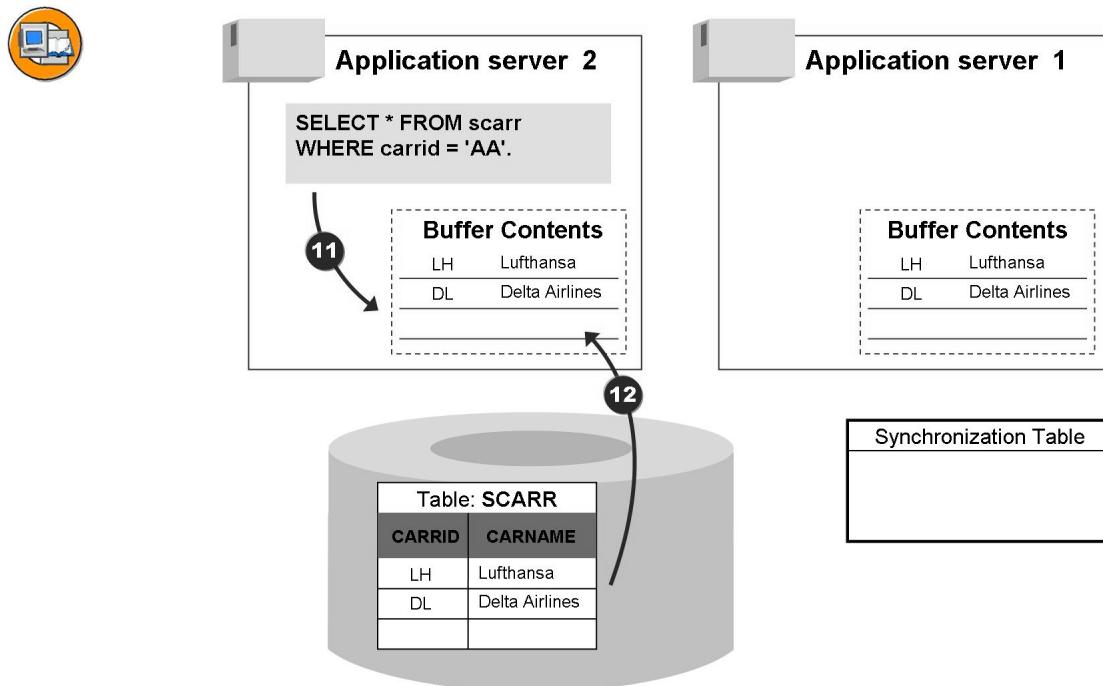
**Figure 156: “Dirty Read” Risk**

If an access of AS-1 was to take place in the synchronization interval, specified by rdisp/bufreftime, the current data would not be read.



**Figure 157: Buffer Synchronization**

The check access to the DDLOG synchronization table (9) takes place at regular intervals. If this includes a relevant entry for one of the application servers (10), the buffers or parts of the buffers are invalidated on the application servers (in this example the entire buffer because of the full buffering).



**Figure 158: Reloading Data into the Buffer**

In the next request (11) on SCARR, the buffer on AS-2 would be loaded again (12).

→ **Note:** That is, not all changes are immediately reloaded when the buffers are synchronized; this takes place only as required.

## Shared Objects

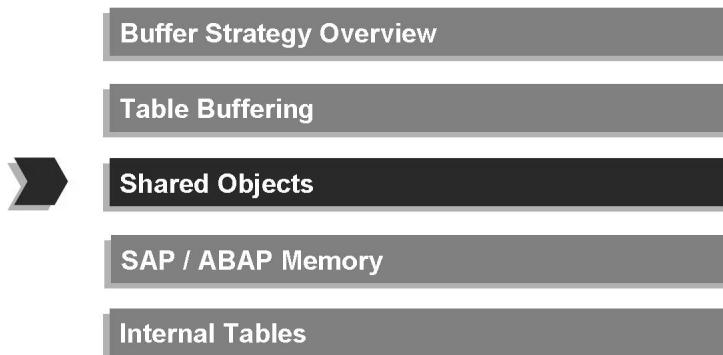


Figure 159: Buffering Strategies (3)

→ **Note:** A detailed technical view with programming examples for shared object is provided in course BC402. At this point, the subject is to be discussed under performance aspects.

Starting in *SAP Web AS 6.40*, you can save data as shared objects in shared memory, across different programs and even different user sessions. This enables you to create applications in which users write data to this memory area. Other users can then read this data later.



- Storing a **catalog**  
An “author” writes the catalog to the shared objects area – many users can then access this catalog at the same time.
- Saving a **shopping cart**  
The buyer fills the shopping cart and the seller reads it later.

## Shared Memory

Shared memory is a memory area on an application server that can be accessed by all the ABAP programs running on that server.

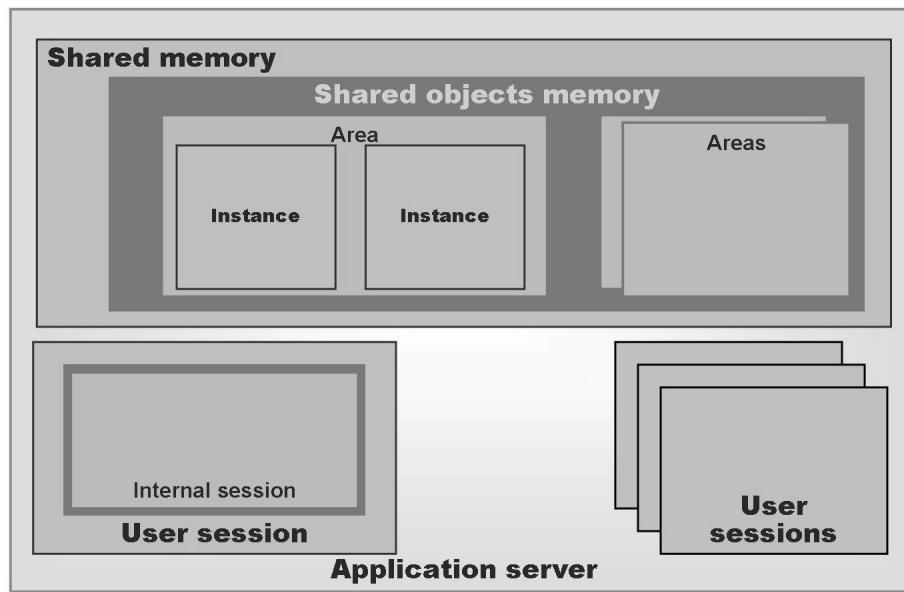
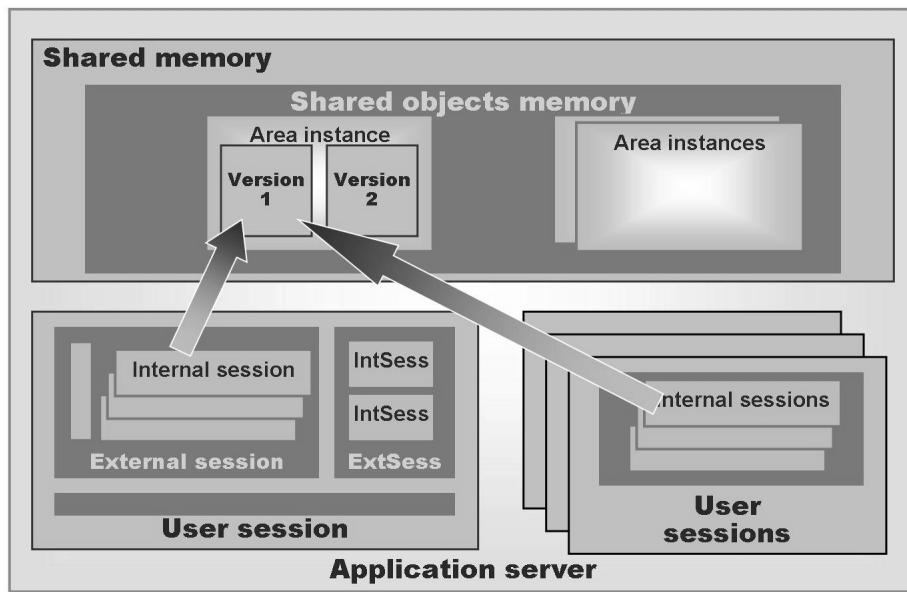


Figure 160: Memory Model of an Application Server

Before shared objects were introduced, ABAP statements had to use the EXPORT and IMPORT statements with the SHARED BUFFER additions to access this memory area. Instances of classes "lived" exclusively in the internal session of an ABAP program. With the introduction of shared objects in *SAP Web AS 6.40*, shared memory has been enhanced with **Shared Objects Memory**, where the shared objects can be saved. These shared objects are saved in areas of shared memory.

- **Note:** At the present time, instances of classes can be saved. It is not (yet) possible to save any data objects as shared objects. However, data objects can be stored as attributes of classes.



**Figure 161: Access to Shared Objects (SHMA, SHMM)**



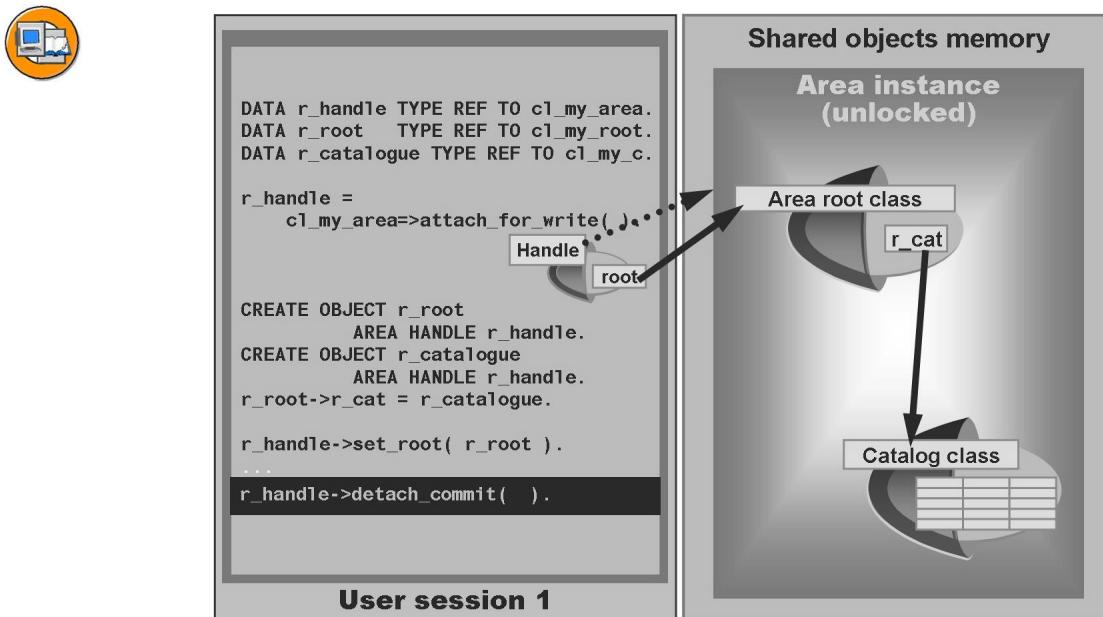
- Cross-program buffering of data that is often read, but rarely written, (Recommendation: Once per day to once per hour)
- Simultaneous read accesses are supported.
- Access is regulated by a lock mechanism.
- Data is stored as object attributes.
- Memory bottlenecks result in runtime errors and have to be caught.

Write accesses are seldom because writing data to the shared objects area is performance-intensive. It is runtime that you want to optimize here, though, which would be lost if write access were more frequent.

→ **Note:** SAP also uses shared objects in *SAP Web AS 6.40*. For example, this technique is used to navigate in the *ABAP Workbench*. In addition to saving memory (around 3 MB per user logon), navigation during the first access is faster by up to a factor of 100.

A prerequisite for saving an object in shared memory is that the class of that object is defined with the SHARED MEMORY ENABLED addition of the CLASS statement, or that the *Shared Memory Enabled* attribute is selected in the *Class Builder*.

 **Note:** All other technical details for the concept and programming are introduced in course BC402. Information is also available in the online document under [help.sap.com](http://help.sap.com) → SAP NetWeaver 7.1 → SAP NW Library → Function-Oriented View → Application Server ABAP → Application Development on AS ABAP → ABAP Programming Techniques → Shared Objects (under SAP NW7.0 use the path via the SAP NW Library...)



**Figure 162: Complete Program for Creating an Area**

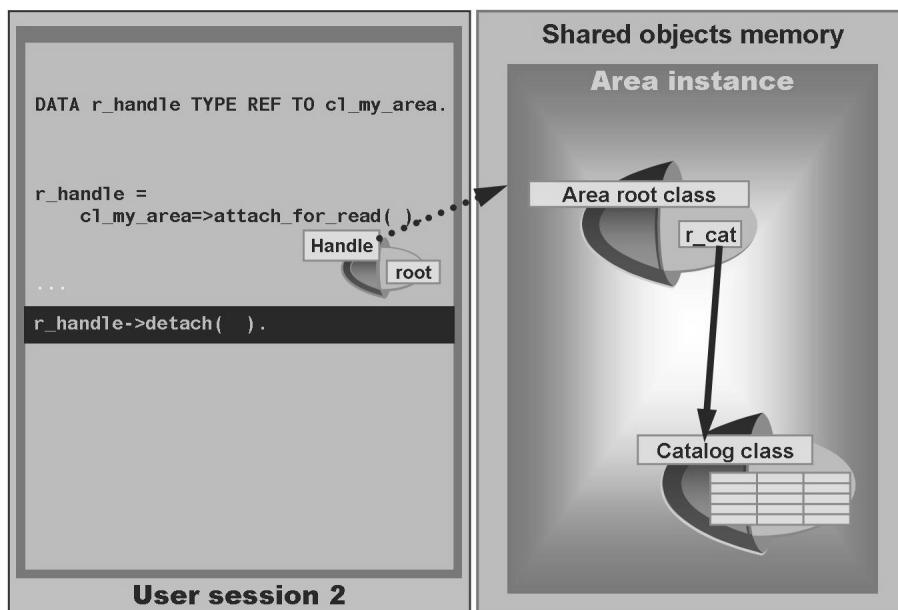
The graphic shows a program that creates an area without filling the area with data. This could take place for the first time and automatically in the class structure of the catalog class or typically using accesses to the methods of the class (for example: `r_cat->fill_catalogue( EXPORTING ... )`). But this is not depicted in the graphic.

Read access to an area instance is only possible when the write lock has been canceled. To do so, use the DETACH\_COMMIT method, which inherits the area class from class CL\_SHM\_AREA.

Once an area instance has been set up, any other users and applications can access it. The read programs have to implement the following steps:

The read program first needs a reference variable that is typed with the area class. This reference variable serves as a handle for the area instance that is being accessed. The program also has to get the handle of the area instance. This is performed using method ATTACH\_FOR\_READ, which is provided by the class CL\_SHM\_AREA. This sets a read lock that prevents the area instance from being erased during the access.

The objects in this area instance can now be accessed. They are always accessed using the area handle.



**Figure 163: Accessing the Area and Removing the Read Lock**

Once the read activity has been completed, the application releases the read lock. You can use the DETACH method for the area handle for this purpose. The read lock is also released automatically when the internal session is closed.

- It is also possible to use the commands **EXPORT/IMPORT ... TO SHARED BUFFER** to write directly to the shared memory. The object-oriented approach with the comfort of the SHMA monitor (area monitor), versioning of area instances and encapsulated accesses do offer distinct advantages for shared object though.
- We would also like to point out that the programming of shared objects replaces the concept of contexts (SE33) with SAP Web AS 6.40. Contexts were developed for Release 4.0 to buffer data across programs. SAP uses this concept internally.

## SAP / ABAP Memory



Buffer Strategy Overview

Table Buffering

Shared Objects



SAP / ABAP Memory

Internal Tables

Figure 164: Buffering Strategies (4)

The way in which main memory is organized from the program's point of view can be represented in the following simple model:

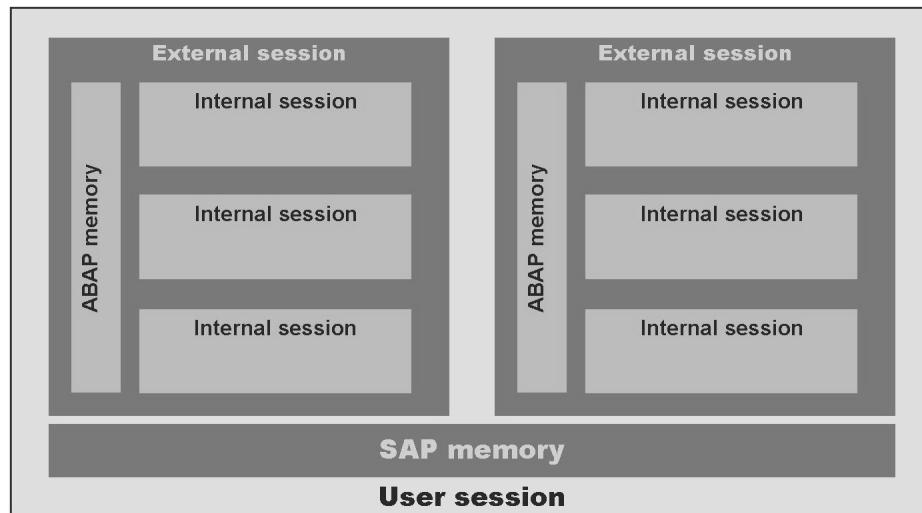


Figure 165: Logical Memory Model

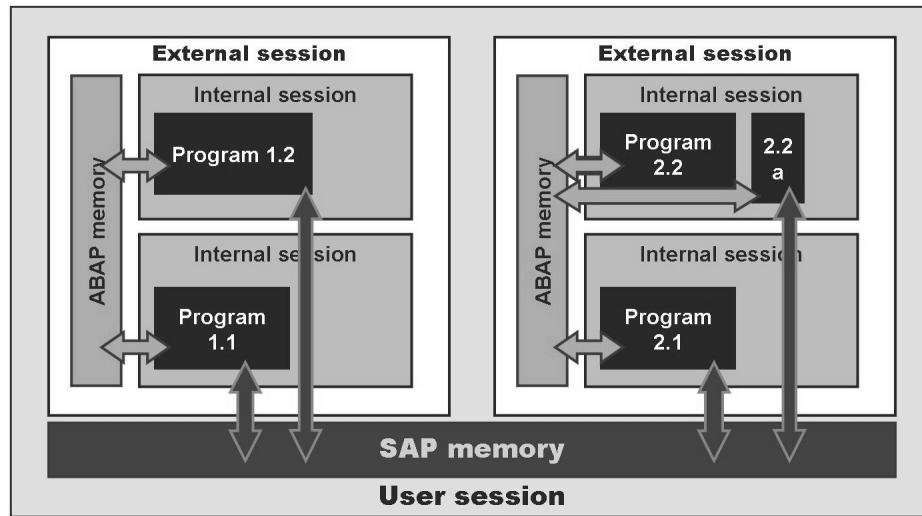
There is a distinction between internal and external sessions:

- Generally, an external session corresponds to an SAPGUI window. You can create a new session with *System -> Create Session* or by calling **/O<TCODE>**. Typically, a maximum of **six external sessions** can be created per session (release-dependent; see parameter rdisp/max\_alt\_modes).
- External sessions are subdivided into internal sessions (placed in a stack). Each program that is started occupies its own internal session. Each external session can contain up to **nine internal sessions**.

Data for a program is visible only within an internal session. In general, data visibility is restricted to the relevant program.

### ABAP Memory and SAP Memory

It is often necessary to transfer data between two different programs. However, it is not always possible to specific this data as an addition in the program call. In such cases, you can use the SAP memory and the ABAP memory to transfer data between programs:

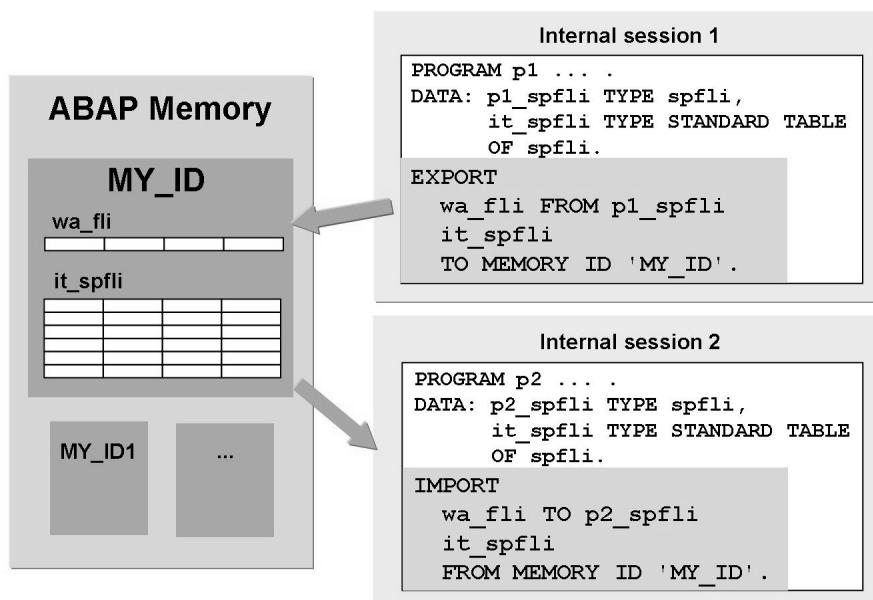


**Figure 166: Range of ABAP Memory and SAP Memory**

- SAP memory is a **user-specific** memory area for storing **field values**. It is therefore not that useful for passing data between internal sessions.  
Values in SAP memory are retained for the duration of the user's terminal session. SAP memory can be used between sessions in the same terminal session. **All external sessions** of a user session can access SAP memory.  
You can use the contents of SAP memory as default values for screen fields. (SET-GET parameter)
- ABAP memory** is also user-specific. There is a local ABAP memory for each external session. You can use it to exchange any ABAP data objects (fields, structures, internal tables, complex objects) between the internal sessions in any one external session.  
When the user exits an external session (/i in the command field), the corresponding ABAP memory is automatically initialized or released.

### ABAP Memory

The EXPORT . . . TO MEMORY statement copies any number of ABAP data objects with their current values (data cluster) to the ABAP memory. The **ID** addition (maximum 60 characters long) enables you to identify different clusters.



**Figure 167: Passing Data Using the ABAP Memory**

If you use a new EXPORT TO MEMORY statement for an existing data cluster, the new one overwrites the old one. The IMPORT ... FROM MEMORY ID ... statement allows you to copy data from ABAP memory into the corresponding fields of your ABAP program.

You can also restrict the selection to a part of the data cluster in the IMPORT statement. The variables into which you want to read data from the cluster in ABAP memory must have the same types in both the exporting and the importing programs.

You use the FREE MEMORY ID ... statement to release a data cluster explicitly.

When you call programs using transaction codes, you can only use the ABAP memory to pass data to the transaction in the insert case (CALL TRANSACTION).

→ **Note:** Programs that were started independently of each other and can even run independently of each other cannot exchange data via the ABAP memory. But this would be possible by using shared objects or table buffering.

## SAP Memory

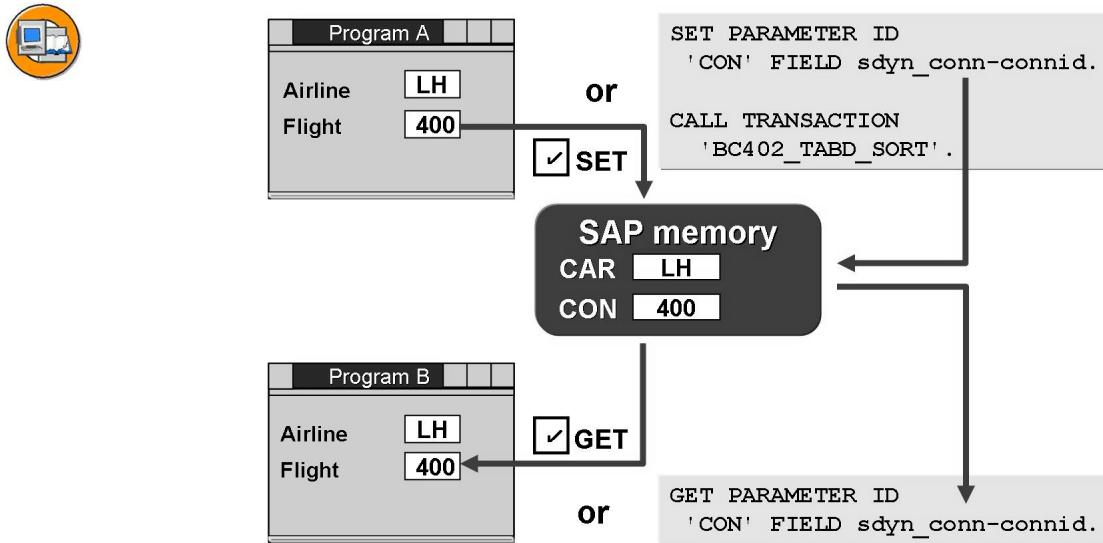


Figure 168: Passing Parameters Using SAP Memory

You can define memory areas (parameters) in SAP memory in various ways:

- By creating input/output fields with reference to the *ABAP Dictionary*. These take the parameter names of the data elements to which they refer.

Alternatively, you can enter a name in the attributes of the input/output fields. Then, you can also choose whether the entries from the field should be transferred to the parameter (SET), or whether the input field should be filled with the value from the parameter (GET).

To find out about the names of the parameters assigned to input fields, display the field help for the field with the (F1) key and choose Technical info.

- You can also fill the memory areas directly with the `SET PARAMETER ID 'PAR_ID' FIELD var.` statement and read from them with the `GET PARAMETER ID 'PAR_ID' FIELD var.` statement.
- Finally, you can define parameters in the *Object Navigator* and let the user fill them with values.

→ **Note:** That is, the SAP memory cannot be used to exchange internal tables or structured data objects. The data exchange is restricted to the logon, that is, the session of the user. If this is supposed to be implemented across sessions and if internal tables or structured data are supposed to be exchanged, we recommend using shared object or table buffering.

## Internal Tables

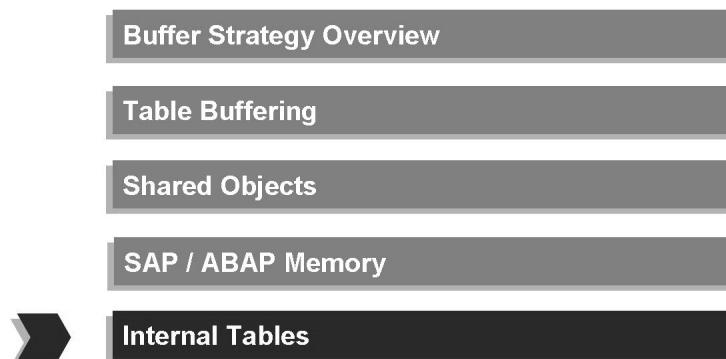


Figure 169: Buffering Strategies (5)

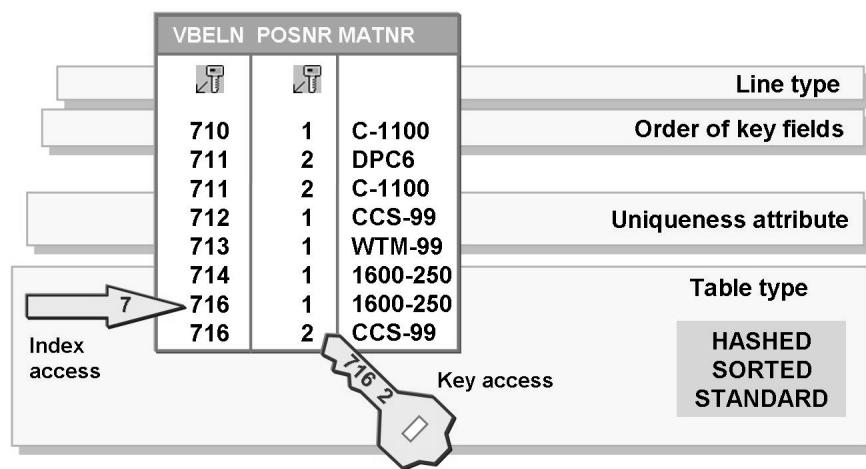


Figure 170: Internal Table Attributes

An internal table is fully specified by the following information:

- **Line type**

The line type of an internal table can be any ABAP data type. It is usually a structured data type (line type).

- **Order of key fields**

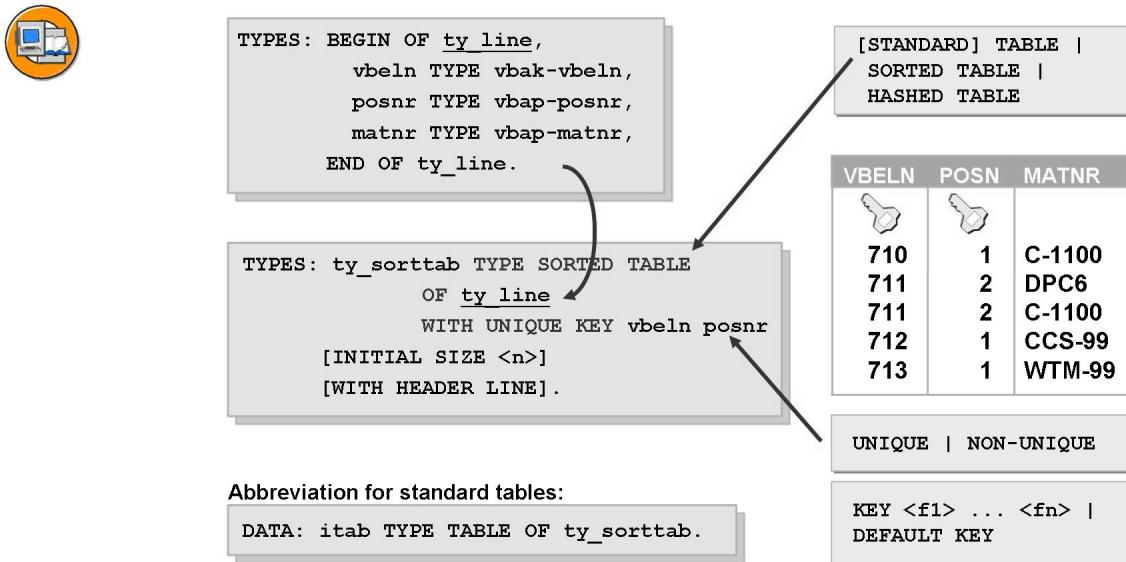
The key fields and their order determine the criteria by which the system identifies table lines.

- **Uniqueness attribute**

You can define the key of an internal table as either UNIQUE or NON-UNIQUE. If the key is unique, there can be no duplicate entries in the internal table.

- **Table type**

The table type defines how ABAP access individual table lines. This can be either by **table index** (access by row ID) or by the **table key** (access by key fields).



**Figure 171: Example: Defining Internal Tables Locally**

The statement

```
DATA <itab> TYPE <tabkind> OF <linetype>
[WITH [UNIQUE | NON-UNIQUE] <keydef> [INITIAL SIZE <n>]]
```

is used to create an internal table of table type <tabkind> via the line type <linetype>.

Normally, you also specify a user-defined key in the form <f1> ... <fn>. All of the fields in the key must have a flat structure. The addition DEFAULT KEY allows you to set the standard key (all non-numerical fields) as the table key, as required by the definition of internal tables before Release 4.0.

You can use the UNIQUE or NON-UNIQUE addition to specify whether the table should be allowed to contain entries with duplicate keys.

If you know that your internal table will be smaller than 8KB, you can use the INITIAL SIZE <n> addition to reserve its initial memory space.

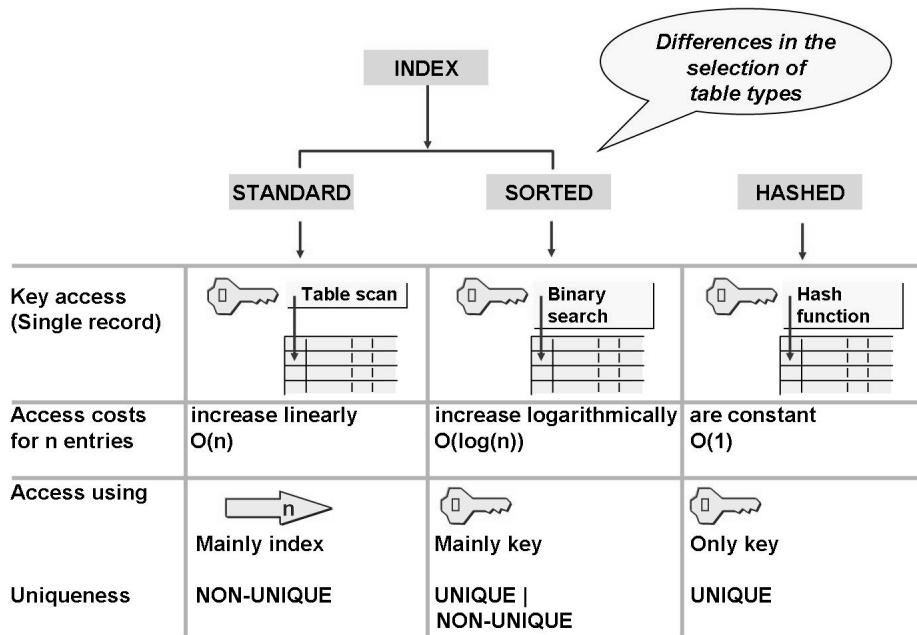


Figure 172: Table Types

The table type defines how ABAP access individual table lines. There are three types of internal tables:

- In a **standard table**, you can access data using either the table index or the key. If access proceeds via the primary key, the response time is linearly related to the number of table entries. The key of a standard table is always non-unique.
- **Sorted tables** are always stored sorted in ascending order according to their key. You can access them using either the table index or the key. If you use the key, the response time is in logarithmic relation to the number of table entries, since the system uses a binary search to access the table. The key of a sorted table can be either unique or non-unique.
- Standard tables and sorted tables are generically known as **index tables**, since their values can be accessed via an index.
- **Hashed tables** can only be accessed via the primary key. The response time is constant, regardless of the number of table entries, since access proceeds via a hash algorithm. The hash algorithm is determined internally. The key of a hashed table must be unique. You can neither implicitly nor explicitly access hash tables through indexes.

At runtime, you can find out the current type of table using the statement DESCRIBE TABLE <itab> KIND <k>.



*Index  
accesses*

	n	Single Record Processing	Mass Processing
Read		<pre>READ TABLE &lt;itab&gt; INDEX &lt;n&gt; [INTO &lt;wa&gt;   ASSIGNING &lt;fs&gt;]</pre>	<pre>LOOP AT &lt;itab&gt; [FROM &lt;n1&gt;] [TO &lt;n2&gt;] [INTO &lt;wa&gt;   ASSIGNING &lt;fs&gt;] ENDLOOP</pre>
Append		<pre>APPEND [&lt;wa&gt; TO] &lt;itab&gt;</pre>	<pre>APPEND LINES OF &lt;itab1&gt; [FROM &lt;n1&gt;] [TO n2] TO &lt;itab2&gt;</pre>
Insert		<pre>INSERT [&lt;wa INTO] &lt;itab&gt; [INDEX &lt;n&gt;]</pre>	<pre>INSERT LINES OF &lt;itab1&gt; [FROM &lt;n1&gt;] [TO &lt;n2&gt;] INTO &lt;itab2&gt; [INDEX &lt;n3&gt;]</pre>
Change		<pre>MODIFY &lt;itab&gt; [INDEX &lt;n&gt;] [FROM &lt;wa&gt;]</pre>	
Delete		<pre>DELETE &lt;itab&gt; [INDEX &lt;n&gt;]</pre>	<pre>DELETE &lt;itab&gt; FROM &lt;n1&gt; TO &lt;n2&gt;</pre>

Figure 173: Overview: Index Operations on Internal Tables

Internal table operations can be classified as follows:

- By their access method, as either **index operations** or **key operations**.
- By the scope of their access, as either **single record processing** or **mass processing**.

Single record processing accesses one line in each operation, while mass processing accesses n lines.

- By the operation type (Read, Append, Insert, Change, or Delete).

Index operations cannot be used for hashed tables. For index tables, index operations are faster than key operations.



		<i>Key accesses</i>	
		Single Record Processing	Mass Processing
		<i>Implicit from &lt;wa&gt;</i>	<i>Explicit</i>
Read		READ TABLE <itab> FROM <wa> [INTO <wa>   ASSIGNING <FS>].	READ TABLE <itab> [INTO <wa>   ASSIGNING <FS>] WITH [TABLE] KEY <k> = <v> ...  LOOP AT <itab> [INTO <wa>   ASSIGNING <FS>] WHERE <condition>.
	Insert	INSERT [<wa> INTO] TABLE <itab>.	INSERT LINES OF <itab1> INTO TABLE <itab2>.
Cumulative insert		COLLECT [<wa> INTO] <itab>.	
Change		MODIFY TABLE <itab> [FROM <wa>].	MODIFY <itab> [FROM <wa>] WHERE <condition>.
Delete		DELETE TABLE <itab> [FROM <wa>].	DELETE <itab> WHERE <condition>.
		DELETE TABLE <itab> WITH TABLE KEY <k> = <v>.	

Figure 174: Key Operations on Internal Tables

Rows can be identified either via the work area from which the key values are taken (in an implicit key access), or via an explicit key specification using WITH [TABLE] KEY. Key accesses are possible for all table types, but they show different levels of performance for each table type.

Hashed tables are useful for reading a single data record via the table key, since the resulting access time is independent of the number of table entries. The location of the data record to be read is determined using the hash function. The time required is independent of the number of table entries.

An INSERT for a standard table or hashed table using the key has the same effect as an APPEND statement. For hashed tables, however, the address of the data row to be entered must also be specified. For inserts in sorted tables, the sort sequence must be strictly observed, which means the access costs grow with increasing table size.

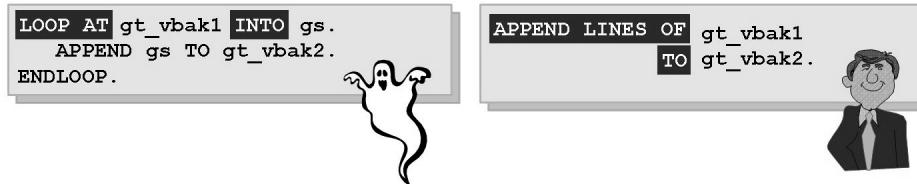
With a COLLECT for standard tables, a hash function is created internally. Using this hash function results in good performance. The hash function for standard tables is, however, non-persistent, since it is destroyed by each INSERT, DELETE, and so on; after which performance is reduced. A COLLECT is only allowed when all non-key fields are numerical.

With the key operations MODIFY and DELETE, the location of the table entry must be determined.



### Use the commands for mass processing

```
<itab2> = <itab1>.          (without header row)
APPEND LINES OF <itab1> [FROM <n1> TO <n2>] TO <itab2>.
INSERT LINES OF <itab2> [FROM <n1> TO <n2>] INTO <itab1>
[INDEX <n3>].
```



**Figure 175: Inserting Mass Data in Internal Tables**

If you are augmenting an internal table from an internal table, it is a good idea to use the indicated mass processing instructions. A prerequisite of this is, however, that the internal tables have fields that are left-aligned convertible. These fields must not necessarily have identical names. The MOVE-CORRESPONDING statement (requiring identical names for fields) can only be used to work on work areas or headers.

APPEND/INSERT LINES can be used to extend an existing internal table <itab2>. You can copy an interval of lines from the source table <itab1> by using a FROM .. TO addition.



### Totaling values with COLLECT

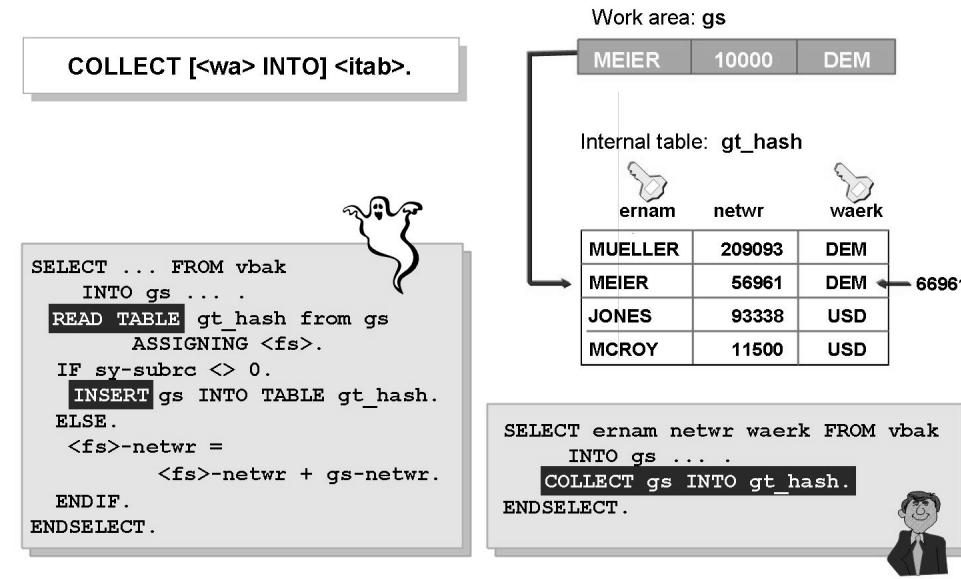


Figure 176: Filling an Internal Table with Cumulative Values

With the COLLECT statement, the content of the work area is added to an entry with the same key or added to the table as a new entry. This allows you to create aggregated internal tables.

A COLLECT searches the internal table for the entry according to the table type and defined key. If it finds an entry in the table, it adds all numeric fields of the work area or header that are not part of the key to the corresponding fields of the entry. If no entry is found, the contents of the work area or header row are added as a new table entry.

You can only use the COLLECT statement with internal tables whose non-key fields are all numeric (type I, P, or F). If not, you get a syntax error. From SAP Basis release 4.0A, the table key may contain numeric fields.

The COLLECT statement is internally optimized for standard tables so that the response time is constant. This optimization applies as long as the tables are not modified with DELETE, INSERT, or SORT (non-persistent hash function). Using the latter instructions loses the rapid internal access, and leads to linearly increasing access costs for the COLLECT.

With hashed tables and sorted tables, the cost of the COLLECT statement is constant, even if you do use DELETE, INSERT or SORT.



Only access the table lines you really need

```
LOOP AT <itab> ... WHERE ...
MODIFY <itab> ... WHERE ...
DELETE <itab> ... WHERE ...
```

```
LOOP AT <itab> ... FROM ... TO
DELETE <itab> ... FROM ... TO
```

```
SELECT-OPTIONS:
  s_vbeln FOR vbak-vbeln.

LOOP AT it_vbap INTO wa.
  IF wa-vbeln IN s_vbeln.
    ...
  ENDIF.
ENDLOOP.
```



```
SELECT-OPTIONS:
  s_vbeln FOR vbak-vbeln.

LOOP AT it_vbap INTO wa
  WHERE mandt = sy-mandt
    AND vbeln IN s_vbeln.
ENDLOOP.
```



Figure 177: Conditional Mass Accesses for Itabs

Just as a SELECT ... WHERE is preferable to a SELECT used in conjunction with CHECK, you should use a WHERE clause to process internal tables.

To improve the performance of a LOOP ... WHERE statement, the fields compared should be of the same type. It is best to define the comparison fields as follows:  
DATA <comparison field> TYPE <table field>.

**LOOP ... WHERE:** Since you cannot access part-fields of tables that do not have a structured line type (for example, tables via type I), use a TABLE\_LINE notation to access the entire table line within the WHERE clause.



### Only copy required fields

```
MODIFY ... TRANSPORTING <f1> ... <fn>.
READ ... TRANSPORTING [<f1> ... <fn> | NO FIELDS].
LOOP ... TRANSPORTING NO FIELDS WHERE ....
```

**gt\_vbap**

		1	DPC6	...
mandt	vbeln	posnr	matnr	
400	844	1	WTM-99	
400	844	2	DPC6	
400	900	1	DPC6	
400	900	2	CCS-99	
400	910	1	WTM-99	



READ TABLE gt\_vbap INTO gs\_vbap  
WITH KEY mandt = 400  
vbeln = '0000000900'.

READ TABLE gt\_vbap INTO gs\_vbap  
TRANSPORTING posnr matnr  
WITH KEY mandt = 400  
vbeln = '0000000900'.

**Figure 178: Selective Field Transport for Itabs**

LOOP, READ, and MODIFY normally place all entries of a line into the table work area; or from the work area into the table body.

You can use TRANSPORTING to specify the fields you are interested in. Not the whole line, but only the contents of the relevant fields are copied. This is especially important if the table has a deep structure (with nested internal tables), since here copying costs can be reduced considerably. Since SAP Basis release 4.5A, a technique is available that allows direct access to the table body, thus causing zero copying costs. This technique uses field symbols (LOOP ... ASSIGNING).

You can use TRANSPORTING NO FIELDS to specify a number of row indexes (an index quantity) or table rows that satisfy a particular condition.

LOOP ... TRANSPORTING can only be used with a WHERE condition.



**FIELD-SYMBOLS <fs> [TYPE type | LIKE f].**

**ASSIGN <field> TO <fs>.**

```
FIELD-SYMBOLS <fs> TYPE i.  
DATA gv_i1 TYPE i VALUE 4,  
      gv_i2 TYPE i.
```

```
ASSIGN gv_i1 TO <fs>.
```

Field symbol  
<fs>

Data object  
gv\_i1



The value in  
variable gv\_i2 is  
assigned to gv\_i1.

...  
gv\_i2 = 5.

<fs> = gv\_i2.

Field symbols have a  
“value semantic”.

Field symbol  
<fs>

Data object  
gv\_i1



**Figure 179: Using Field Symbols**

Field symbols in ABAP are similar to pointers in other programming languages.

**Field symbols.** However, pointers (as used in PASCAL or C) differ from ABAP field symbols in their reference syntax.

You declare field symbols using the FIELD-SYMBOLS statement. They can be qualified using LIKE or TYPE. AS of R/3 Release 4.6, a field symbol should always be qualified. For a qualification that is as general as possible, use TYPE ANY.

The statement ASSIGN f TO <fs> assigns the field f to field symbol <fs>. The field symbol <fs> then points to the contents of field f at runtime. This means that all changes to the contents of f are visible in <fs> and vice versa (value semantic).

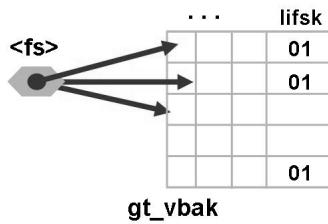
Before using MOVE for a field symbol, you must use ASSIGN to point the field symbol to a data object. To find out whether the field symbol is assigned to a data object, use IF <fs> IS ASSIGNED.

If you declare a non-qualified field symbol, or a field symbol of type ANY, it automatically receives the type of the field it points to. If you declare a qualified field symbol, you can only let it point to data objects with a compatible type.



### Avoid all copy costs

```
LOOP AT <itab> ... ASSIGNING <fs> [WHERE ...].
READ TABLE <itab> ... ASSIGNING <fs> ....
```



```
DATA: gt_vbak TYPE TABLE of vbak.
FIELD-SYMBOLS: <fs> TYPE vbak.
```

```
...
LOOP AT gt_vbak ASSIGNING <fs>.
  WHERE lifsk = '01'.
  <fs>-lifsk = SPACE.
ENDLOOP.
```

*Direct access to itab data!*

**Figure 180: Direct Access to Tables without Field Transfer**

As of SAP Basis release 4.5A, table rows can be accessed directly in the table body. This means that a work area or header row is not required. Instead, a table row is accessed using `LOOP AT itab ... ASSIGNING <fs>` and field symbols.

In general, the broader the line type of the internal type, the more you save with the pointer technique because the copy costs caused by the work area become more important then.

`ASSIGNING` is available for `READ` and `LOOP` statements.



**Caution:** Note that `ASSIGNING` always works directly on the table body. You cannot change the key components of a sorted or hashed internal table. Within a `LOOP... ASSIGNING` you must not let the field symbol point to another data object with an `ASSIGN`.

After the loop, the field symbol is positioned on the last line.

If you want to access individual components using the field symbol within a `LOOP`, use the addition `TYPE/LIKE` with the instruction `FIELD-SYMBOLS`.

Changes of field contents are performed by a value assignment and must not be performed using `MODIFY`.



<b>Evaluation by Type</b>				
<b><i>READ Variants</i></b>	<b><i>Remarks</i></b>	<i>standard</i>	<i>sorted</i>	<i>hashed</i>
READ TABLE ... INDEX <n>	Costs independent of number of table entries			
READ TABLE ... FROM <wa>	Key is implicit.			
READ TABLE ... WITH KEY <k1> = <v1> ...	Standard table: BINARY SEARCH improves access.			
READ TABLE ... WITH TABLE KEY <k1> = <v1> ...	Key must be defined completely.			

**Figure 181: Reading Single Records**

The graphic illustrates the runtime behavior of READ key accesses. Keep in mind that index accesses usually offer better performance. Index accesses are not possible with hashed tables. Accesses on single data records in hash tables are always realized using a unique key.

The READ variants FROM <wa> and WITH TABLE KEY..., new in Release 4.0, are defined for all table types. The internal access is optimized according to the table type. The cost of the READ statement is linear for standard tables, logarithmic for sorted tables, and constant for hashed tables.

If you frequently wish to access a standard table via the key, you can sort the table according to the relevant fields and then use the addition BINARY SEARCH. With the addition BINARY SEARCH, the access costs increase only logarithmically compared to the linearly increasing access expense without this addition. The required SORT has a cost of  $O(n \log n)$ , where  $n$  is the number of rows in the internal table.

If you can specify all keys, no internal optimization is possible if you only use WITH KEY, so that a full table scan is required instead of a binary search or a hash algorithm.

The addition WITH TABLE KEY is internally translated into a FROM <wa>.

## Application Examples for Internal Tables

### Sensible Use of the Table Type



Single record access using a fully specified table key

Used for read  
modules such as  
FMs, methods, etc.



```
FUNCTION VBAP_SINGLE_READ...
...
READ TABLE gt_hashvbap INTO ls WITH TABLE KEY
      mandt = fp-mandt
      vbeln = fp-vbeln
      posnr = fp-posnr

IF sy-subrc <> 0.
  SELECT SINGLE * FROM vbap ... INTO ls.
  INSERT ls INTO TABLE gt_hashvbap.
ENDIF.
...
ENDFUNCTION.
```

**Figure 182: Hashed Table**

Use a **hashed table** whenever **many single record accesses** are required for an internal table via the **fully specified key**.

We particularly recommend using it for very large tables with more than 10000 entries.

Hashed tables are useful for read modules. These function modules basically buffer the read data in an internal table. Before sending the SELECT to the database, the buffer is checked to see if the data record is already there. If so, the SELECT is not used; if the SELECT is performed, the new data record is added to the buffer. In this read module, more application logic is required. For example, more than one data record is read by the SELECT, controlling of the the buffer size is performed, and so on. You can find such function modules in, for example, the function group V45I.

Another use for hashed tables is, for example, when accessing the intersection of two groups of data. Finding out whether an entry is present in an internal table requires an effort of O(1) if you use a hashed table. Therefore, accessing the intersection of two groups of data is possible with an effort of O(n) if you use a hashed table with n entries for the (probably) larger table.

The operating system limits the maximum number of entries in a hash table.



### Mass processing using left-aligned specification of the key

→ Try to evaluate as many key fields as possible from the left without gaps in the WHERE condition.

LOOP .. WHERE    DELETE ... WHERE    MODIFY ... WHERE

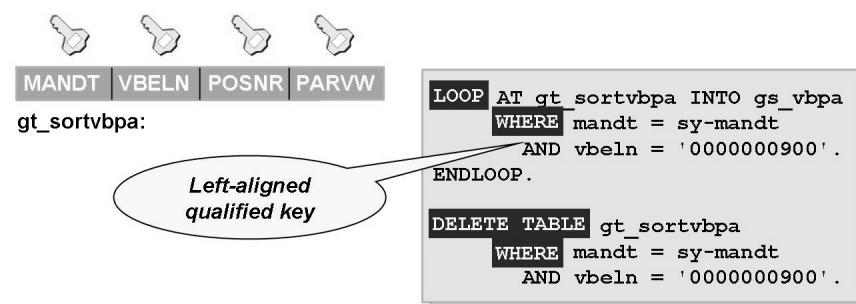


Figure 183: Sorted Table

In **partial sequential processing** for **sorted tables**, you process **optimized loops** (that is, only the lines that fulfill the WHERE condition are processed), provided the WHERE condition has the following form: WHERE k1 = v1 AND k2 = v2 AND ... AND kn = vn, and the components k1 to kn exactly reflect the beginning of the table key.

If gaps are in the sequence of key fields when specifying the WHERE condition, the access cannot be optimized.

If the internal table has n entries and the WHERE clause is satisfied by m entries, an optimized access requires an effort of O(m+log n).

For tables of type STANDARD or HASHED TABLE, processing a WHERE clause involves processing all table entries in a full table scan. Therefore, these tables require a time of O(n).

The MODIFY and DELETE commands are optimized by a suitable WHERE clause. This optimization does not occur when a standard table is sorted and subsequently processed with a WHERE condition.

**Flexible access**

for example, using various keys

```

PARAMETERS: p_ernam TYPE vbak-ernam,
             p_vdatu TYPE vbak-vdatu.

SORT gt_vbak BY ernam.
READ TABLE gt_vbak WITH KEY ernam = p_ernam INTO gs
          BINARY SEARCH.
IF sy-subrc <> 0.    ... ENDIF.

SORT gt_vbak BY erdat vdatu.
LOOP AT gt_vbak INTO gs WHERE erdat = sy-datum
...
ENDLOOP.
```

**Figure 184: Standard Table**

It is appropriate to use a **standard table** whenever the **requirements** on the table in different processing steps **varies**.

This is the case, for example, when one key operation accesses one group of fields, while another key operation accesses a different group of fields. This is particularly so if some of the specified fields are not key fields. In this case, the other table types also require a full table scan. A standard table has the advantage of not being able to be sorted by any field, and subsequently processed by, for example, a **READ... BINARY SEARCH** (logarithmic cost).

You can also use the stable SORT (by adding STABLE to the SORT statement). This preserves the relative order of data records that have the same sort key.

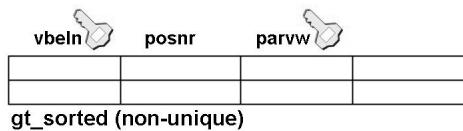
**If you are working mainly with index accesses, a standard table is a good choice,** since it can be augmented faster than the other two table types: For a standard table, INSERT acts like an APPEND. During an INSERT, with a sorted table, the system must observe the sort order, while with a hashed table, the system must repeatedly reapply the hash algorithm.

## Additional Performance Tips for Internal Tables



( SORT <itab> f1 f2 ... )

**DELETE ADJACENT DUPLICATES FROM <itab>**  
**[COMPARING f1 f2 ... ].**



```

SELECT * FROM vbpa INTO gs_vbpa
...
READ TABLE gt_sorted INTO ...
FROM gs_vbpa.
IF sy-subrc <> 0.
INSERT gs_vbpa INTO TABLE
      gt_sorted.
ENDIF.
ENDSELECT.
```



```

SELECT * FROM vbpa INTO TABLE
      gt_sorted . . .
DELETE ADJACENT DUPLICATES
      FROM gt_sorted.
* the key is significant!
```



Figure 185: Composing Unique Internal Tables

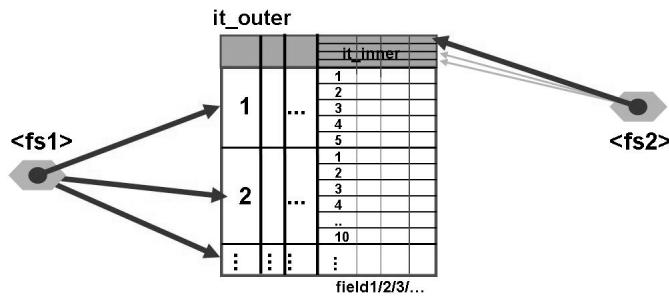
It is sometimes a good idea to build an internal table using a SELECT ... INTO ... , so that the internal table has fewer key fields than the database table. In this case duplicates may occur in the internal table. If you decide to set up a hashed table, for example, it must be unique. In this case it may be a good idea to use a non-unique sorted or standard table, delete the duplicates, and subsequently use a MOVE on the hashed table.

Faster than building a unique table with SELECT ... ENDSELECT and checking whether the current entry already exists in the internal table, is building a non-unique table with ARRAY FETCH and then deleting any duplicates. If you want all key fields of the database table to appear in the internal table, but not in the key, you cannot use a SELECT DISTINCT.

This elimination of duplicates can be optimally realized using the **DELETE ADJACENT DUPLICATES FROM <itab>** command. This checks whether adjacent rows have the same key. The internal table should be sorted.

**COMPARING** lets you define duplicates in terms of other keys.

You can use DELETE ADJACENT DUPLICATES FROM <itab> to subsequently aggregate internal tables.



```
LOOP AT it_outer INTO wa_1.
  LOOP AT wa_1-it_inner
    INTO wa_2.
    MODIFY ... .
  ENDLOOP.
ENDLOOP.
```



```
LOOP AT it_outer ASSIGNING <fs1>.
  LOOP AT <fs1>-it_inner
    ASSIGNING <fs2>.
    <fs2>-field3 = <v1>.
  ENDLOOP.
ENDLOOP.
```



**Figure 186: Processing Nested Tables**

This causes high copying costs when working with a work area, especially with multidimensional (nested) internal tables. The ASSIGNING variant is quicker with these internal tables because there are no associated copying costs.

You can change the field contents by value assignments. Thus, you do not require a MODIFY.

With multidimensional (nested) tables, be aware of memory consumption. You can display the memory consumption in the debugger.

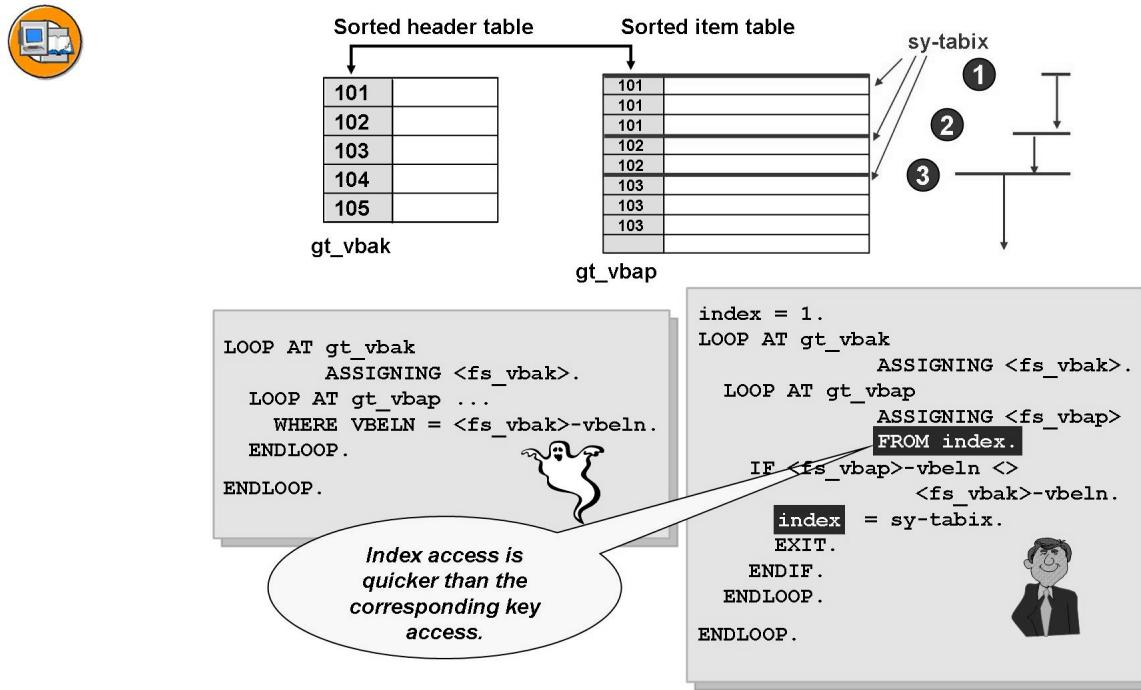
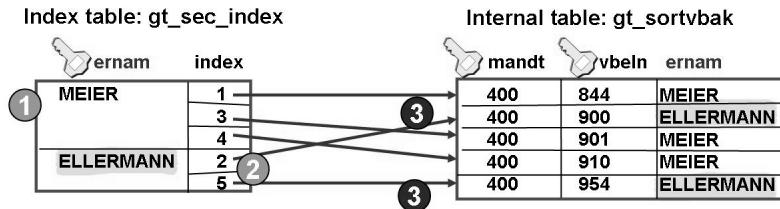


Figure 187: Mass Processing Using an Index Interval

In the example, the index access is used to process the dependent table partly sequentially, from respective starting point. To use this technique, the tables must be sorted and a criterion must exist to move the starting point in the dependent table. A typical example are header tables and item tables that are connected by common key fields. For each loop on the external table, the cost decreases for the internal loop.

In the left-hand example, the inner LOOP has a cost of  $O(\log n)$  where  $n$  is the number of rows in it\_vbap.



```
PARAMETERS: p_name TYPE vbak-ernam.  
DATA: line_number TYPE sy-tabix.
```

```
1 READ TABLE gt_sec_index INTO gs_sec_index  
  WITH TABLE KEY ernam = 'ELLERMANN' TRANSPORTING index.  
  
2 LOOP AT gs_sec_index-index INTO line_number.  
3   READ TABLE gt_sortvbak INTO gs_vbak INDEX line_number.  
  ...  
ENDLOOP.
```



→ Outlook: From SAP NW 7.1 there will be secondary indexes for internal tables!

Figure 188: Secondary Indexes: Reading Data

If you often need to access a field that is not specified by the key of the internal table, a full table scan is performed. It may be useful to create an internal hashed table <gt\_sec\_index> whose key consists exactly of the specified fields. The only non-key field consists of an internal table, via which you can access the corresponding fields of the original table.

This slide shows an approach that is useful for read accesses on an internal table. In contrast, for write accesses this approach causes a large cost when refreshing the secondary index.

Since you are accessing the internal table <gt\_sec\_index> via single record processing and the table key, a hashed table is useful. The internal table <gt\_sortvbak> is only accessed via an index. A standard table or a sorted table can be used.

This approach is especially useful for a sorted table, since you cannot keep resorting by different fields. For a standard table, it is generally better to sort according to the relevant fields and subsequently execute a READ... BINARY SEARCH.

For modifying accesses on an internal table, you can use a secondary index procedure like that for the database. Add the field <rowid> to your internal table. This field is then added to the index table.

Since you are accessing the internal table <gt\_sec\_index> via single record processing and the table key, a hashed table is useful. The internal table <gt\_sortvbak> is only accessed via the full key (field <rowid>). A hashed table is therefore also useful here.

## Additional Examples of the Use of Internal Tables



```

...
DATA: gt_knal TYPE HASHED TABLE OF knal ...
...
SELECT ... FROM vbak INTO gs_vbak ...
...
READ TABLE gt_knal WITH TABLE KEY
  kunnr = gs_vbak-kunnr INTO gs_knal.
IF sy-subrc NE 0. "record not yet in itab ?
  SELECT SINGLE ... FROM knal
    INTO (gs_knal-kunnnr, ...)
    WHERE kunnr = gs_vbak-kunnnr.
"possible also to work with additional "exist-info"
IF sy-subrc EQ 0.
  gs_knal-exist = 'X'.
ELSE.
  CLEAR: gs_knal-exist, gs_knal-name1, ...
ENDIF.
APPEND gs_knal TO gt_knal. "fill itab with knal-data
ENDIF.
...
ENDSELECT.

```

*Read data from DB when necessary*

*Fill itab with new customer number*

**Figure 189: Buffering in the Program Using Internal Tables**

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

In the above example, the intention is to read the suitable data for KNA1 (customer master data) from table VBAK (sales document headers) within a SELECT ... ENDSELECT. You can solve this with a SELECT SINGLE or by using a read routine.

The SELECT SINGLE would execute identical SQL statements in the above constellation. You should therefore encapsulate the reading of KNA1 data records in a read routine. In the above example, the table contents read from KNA1 are buffered in an internal table. Before the database is accessed, the system checks whether the corresponding table entry has already been read. The information that no appropriate entry exists in the database is also buffered in the internal table (in the EXIST field).

If the information will be buffered within the program and does not have to be reused, the read routine can be a subroutine. If you want to buffer in an internal session for multiple programs (call sequence), you should use a function module. Table contents are buffered in various places in the standard SAP system (for example, in function module STATUS\_CHECK).



```
SELECT vbeln posnr
FROM vbap
INTO ...
WHERE vbeln = '0000000001'.
```



```
SELECT posnr matnr
FROM vbap
INTO ...
WHERE vbeln = '0000000001'.

* better: buffering in itab
* and accessing itab later
```

### Avoid similar statements through buffering.

```
SELECT vbeln posnr
FROM vbap
INTO ...
WHERE vbeln IN (
    '0000000001',
    '0000000002'
).
```



```
SELECT vbeln posnr
FROM vbap
INTO ...
WHERE vbeln IN (
    '0000000001',
    '0000000002',
    '0000000003'
).

* better: buffering in itab...
```

**Figure 190: Avoiding Similar SQL Statements Through Buffering**

Within an SQL performance trace, individual similar SQL statements may not be expensive. However, in total, they may greatly increase system load. If this is the case, you should group the similar statements together.



**Hint:** The database differentiates SQL statements according to their texts. For example, a statement text for the database differs in the following cases:

- Different number of values in a WHERE clause with IN operation
- Different number of values in a WHERE clause with OR operation
- Different WHERE clauses (number, reference fields, order)
- Different field lists

If SQL statements differ only in the composition of the field lists or in the order of the WHERE clauses, they can often be grouped together. You can also group together SQL statements where one statement returns a subset of another statement (for example, statements with WHERE clauses with IN or OR operations).

In addition to reducing SQL statements for the particular single object, grouping SQL statements together affects the performance of the whole system. The database manages fewer SQL statements in the DB SQL cache. The CPU load on the database server is reduced (due to fewer parses). The memory load on the database server is also reduced (due to fewer SQL statements in the DB SQL cache).



## Exercise 10: Table Buffering

### Exercise Objectives

After completing this exercise, you will be able to:

- Initiate access of buffered database tables
- Use the SQL trace to verify whether the table buffer was used to handle database accesses

### Business Example

Mr. Jones, the developer, wants to optimize his database accesses even more. He knows that database tables can be buffered in certain cases.

#### Task 1: Accessing the Single Record Buffer

Using technical means, you want to check whether SELECT accesses of buffered database tables really are handled using the table buffer.

1. Create a new program, Z##\_SINGLEBUFFER.
2. Program a correct single-record access of table *T100*. This table is buffered by single record.  
Beforehand, go to the *Data Browser* and choose a record that you will access later.
3. Verify that the access is served by the buffer when performed correctly.  
Use the SQL trace tool to verify this.

#### Task 2: Optional: Accessing the Generic Table Buffer

Example of generic buffer access

1. Create a new program, Z##\_GENERICBUFFER.
2. Program a correct access of table *TCURR*. This table is buffered generically.  
Beforehand, go to the *Data Browser* and choose several records that you will access later.
3. Verify that the access is served by the buffer when performed correctly.  
Use the SQL trace tool to verify this.

*Continued on next page*

### Task 3: Optional: Accessing a Fully Buffered Table

Example for accessing a fully buffered table

1. Create a new program, Z\_##\_COMPLETEBUFFER
2. Program a correct access of table *T005*. This table is fully buffered.

Beforehand, go to the *Data Browser* and choose several records that you will access later.

3. Verify that the access is served by the buffer when the table is accessed.  
Use the SQL trace tool to verify this.

## Solution 10: Table Buffering

### Task 1: Accessing the Single Record Buffer

Using technical means, you want to check whether SELECT accesses of buffered database tables really are handled using the table buffer.

1. Create a new program, Z\_##\_SINGLEBUFFER.
  - a) Create the program as usual.
2. Program a correct single-record access of table *T100*. This table is buffered by single record.

Beforehand, go to the *Data Browser* and choose a record that you will access later.

- a) You start the *Data Browser* by entering transaction code SE16. Choose a record from the table that you will access later using SELECT SINGLE. Also note the corresponding slides in the course materials.
- See model solution.

3. Verify that the access is served by the buffer when performed correctly.

Use the SQL trace tool to verify this.

- a) To access the single record buffer, you have to use SELECT SINGLE and a fully qualified key; otherwise the operation will bypass the buffer.

In transaction ST05, you have to set the *TABLE BUFFER TRACE* flag in order to trace buffer accesses. When SELECT SINGLE is used, the second access of the single record is served from the buffer. The first time the program is executed, the record is read into the buffer from the database, which means it can only be read from the buffer during the second access onwards.

The access time for this single record will probably reduce by a factor of 40 to 100.

### Task 2: Optional: Accessing the Generic Table Buffer

Example of generic buffer access

1. Create a new program, Z\_##\_GENERICBUFFER.
  - a) Create the program as usual.
2. Program a correct access of table *TCURR*. This table is buffered generically.

*Continued on next page*

Beforehand, go to the *Data Browser* and choose several records that you will access later.

- a) To access the buffer, you at least have to qualify the generic key fully; otherwise the buffer will be bypassed. This involved four key fields in Release 4.6C and earlier; it only involves one key field (the client) in *SAP Web AS 6.10* and later.

Also note the corresponding slide in the course materials.

See model solution.

3. Verify that the access is served by the buffer when performed correctly.

Use the SQL trace tool to verify this.

- a) In transaction ST05, you have to set the *TABLE BUFFER TRACE* flag in order to trace buffer accesses.

When you use the *SELECT* statement with a fully qualified generic key, the buffer is accessed during the second table access. The first time the program is executed, the records are read into the buffer from the database, which means they can only be read from the buffer during the second access onwards.

### Task 3: Optional: Accessing a Fully Buffered Table

Example for accessing a fully buffered table

1. Create a new program, Z\_##\_COMPLETEBUFFER
  - a) Create the program as usual.
2. Program a correct access of table *T005*. This table is fully buffered.

Beforehand, go to the *Data Browser* and choose several records that you will access later.

- a) You can access individual records in the table or the entire table; the accesses are handled by the buffer (starting with the second access, of course).

Also note the corresponding slide in the course materials.

See model solution.

3. Verify that the access is served by the buffer when the table is accessed.

Use the SQL trace tool to verify this.

*Continued on next page*

- a) In transaction ST05, you have to set the *TABLE BUFFER TRACE* flag in order to trace buffer accesses.

When you use the SELECT statement, the buffer is accessed during the second table access. The first time the program is executed, the records are read into the buffer from the database, which means they can only be read from the buffer during the second access onwards.

## Result

```
*&-----*
*& Report SAPBC490_BUFF_BUFFERACCESS *
*&-----*
*& Accessing the tablebuffers *
*& Single record / generic / full table buffer *
*&-----*

REPORT sapbc490_buff_bufferaccess.

DATA: gs_t100 TYPE t100.
DATA: gt_tcurr TYPE STANDARD TABLE OF tcurr.
DATA: gt_t005 TYPE STANDARD TABLE OF t005.

START-OF-SELECTION.
*****  

* single record buffer access
*****
SELECT SINGLE * FROM t100
      INTO gs_t100
      WHERE sprsl = 'D'
      AND arbgb = '00'
      AND msgnr = '100'.

* generic buffer access - there is one generic key field
*****
* in releases higher 4.6 the MANDT is the generic buff-field
SELECT * FROM tcurr INTO TABLE gt_tcurr.

* complete tablebuffer access
*****
SELECT * FROM t005 INTO TABLE gt_t005.
```



# Exercise 11: Internal Tables

## Exercise Objectives

After completing this exercise, you will be able to:

- Program high-performance access to internal tables

## Business Example

Mr. Jones, a software developer at a major corporation that uses SAP, is supposed to develop a new business application to analyze mass data. He knows that reading data is a runtime-critical operation. In particular, the database – by accessing the selected set – has a critical effect on runtime behavior. He thus wants to reduce the load on the database by means of buffering across internal tables and access these internal tables optimally.

### Task 1: (Optional) COLLECT:

Copy the template program SAPBC490\_TEMP\_COLLECT to your own program Z##\_COLLECT. The program contains two subroutines. One of them has already been programmed completely. It shows the manual addition of numeric data in an internal table. The same requirement is now supposed to be met using a COLLECT command in the second subroutine.

1. Analyze the already completely programmed subroutine *uprog\_usual*.
2. Implement the second subroutine *uprog\_collect*. There, the total across NETWR is supposed to be formed across all identical data sets of ERNAM and WAERK. Use the COLLECT statement to do so.

### Task 2: (Optional) TRANSPORTING:

Optional exercise for COLLECT: Copy template program SAPBC490\_TEMP\_TRANSPORTING to your own program Z##\_TRANSPORTING. In this program, all index items (SY-TABIX) that contain material number 'M-01' are supposed to be determined for an already filled internal table (containing data from VBAP).

1. Hence, all occurrences of material number 'M-01' in the internal table *itab* are to be determined and saved in a second internal table *index\_itab*. This task should be solved with the best performance possible (access without reading data!).

*Continued on next page*

## Task 3: Correct Use of Table Types

Main exercise for table types Copy the template program SAPBC490\_TEMP\_3LOOPS to your own program Z\_##\_3LOOPS. In this program, data is read in nested form from three internal tables. A first solution uses three internal tables of type STANDARD to do so. The performance is now supposed to be improved by selecting a suitable table type.

1. Analyze subroutine *nested\_itabs\_wa\_bad*. It uses the STANDARD type for all three internal tables that are involved.
2. Optimize the performance by selecting a better table type. Implement this in subroutine *nested\_itabs\_wa\_better*

Take a look at the runtimes output at the end of the two subroutines. Which variant is faster?

## Task 4: (Optional) Multidimensional Nested Internal Tables

Optional exercise for multidimensional nested internal tables Copy the template program SAPBC490\_TEMP\_NESTED\_ITABS to your own program Z\_##\_NESTED. In this program, data is read from multidimensional internal tables. A first solution uses the conventional technique that uses a work area as the interface. Optimize the accesses via field symbols.

1. Analyze the already completely programmed subroutine *loop\_copy*. It uses a work area as the interface for reading data.
2. Optimize the performance by reading via field symbols. Implement the subroutine *loop\_assigning*.

You will find that both require field symbols *<fs\_outer>* and *<fs\_inner>* have already been prepared!

Take a look at the runtimes output at the end of the two subroutines. Which variant is faster?

*Continued on next page*

## Task 5: (Optional) Index Accesses Instead of Key Accesses

Replacing a key access with an index access: Copy template program SAPBC490\_TEMP\_SEQ\_INDEX\_ACCESS to your own program Z##SEQ\_INDEX\_ACCESS. In this program, data is read from two sorted internal tables. The outer driving itab contains document headers; the inner driven itab contains the dependent item entries.

1. Analyze the already completely programmed subroutine *uprog\_usual*. It uses a key access for the partially sequential reading of the inner table.
2. Optimize performance by changing the access to the inner table to an index access. What would the coding have to look like here?

Implement the subroutine *uprog\_tricky*.

Take a look at the runtimes output at the end of the two subroutines. Which variant is faster and why?

## Task 6: Read-on-Demand: Avoiding DB Accesses Through Buffering

Avoiding DB accesses through buffering in internal tables: Continue using program Z##JOINS3 (if that is no longer available, copy template SAPBC490\_TEMP\_JOINS3). Then implement variant “read on demand”. Data from the KNA1 table is to be read to internal table *it\_kna1* on demand.

1. Recall the already programmed subroutines with various solution approaches. The solutions with nested joins were very ineffective. The triple join appears to have been the best solution so far.
2. Implement the subroutine *litab\_ondemand*

Here, READ is to be used to access the internal table with customer data. The customer number is already in the outer loop over the VBAK data. If there is nothing corresponding to the customer in internal table *it\_kna1*, then read the data via a SELECT SINGLE from KNA1 (on demand) and then buffer it in *it\_kna1*. Further queries regarding the same customer number can then be fulfilled via the internal table and are many times faster.

## Solution 11: Internal Tables

### Task 1: (Optional) COLLECT:

Copy the template program SAPBC490\_TEMP\_COLLECT to your own program Z\_##\_COLLECT. The program contains two subroutines. One of them has already been programmed completely. It shows the manual addition of numeric data in an internal table. The same requirement is now supposed to be met using a COLLECT command in the second subroutine.

1. Analyze the already completely programmed subroutine *uprog\_usual*.
  - a) Proceed in the usual manner.
2. Implement the second subroutine *uprog\_collect*. There, the total across NETWR is supposed to be formed across all identical data sets of ERNAM and WAERK. Use the COLLECT statement to do so.
  - a) See sample solution source code and execute it directly.

### Task 2: (Optional) TRANSPORTING:

Optional exercise for COLLECT: Copy template program SAPBC490\_TEMP\_TRANSPORTING to your own program Z\_##\_TRANSPORTING. In this program, all index items (SY-TABIX) that contain material number 'M-01' are supposed to be determined for an already filled internal table (containing data from VBAP).

1. Hence, all occurrences of material number 'M-01' in the internal table *itab* are to be determined and saved in a second internal table *index\_itab*. This task should be solved with the best performance possible (access without reading data!).
  - a) See the source code in the model solution

*Continued on next page*

## Task 3: Correct Use of Table Types

Main exercise for table types Copy the template program SAPBC490\_TEMP\_3LOOPS to your own program Z\_##\_3LOOPS. In this program, data is read in nested form from three internal tables. A first solution uses three internal tables of type STANDARD to do so. The performance is now supposed to be improved by selecting a suitable table type.

1. Analyze subroutine *nested\_itabs\_wa\_bad*. It uses the STANDARD type for all three internal tables that are involved.
  - a) Proceed in the usual manner. Note the output of the runtime at the end of the subroutine. It is the time in milliseconds. The dimension is decisive for the interpretation of the number.
2. Optimize the performance by selecting a better table type. Implement this in subroutine *nested\_itabs\_wa\_better*

Take a look at the runtimes output at the end of the two subroutines. Which variant is faster?

- a) See the extract from the model solution. Note the output of the runtime at the end of the subroutine. It is the time in milliseconds. The dimension is decisive for the interpretation of the number.

## Task 4: (Optional) Multidimensional Nested Internal Tables

Optional exercise for multidimensional nested internal tables Copy the template program SAPBC490\_TEMP\_NESTED\_ITABS to your own program Z\_##\_NESTED. In this program, data is read from multidimensional internal tables. A first solution uses the conventional technique that uses a work area as the interface. Optimize the accesses via field symbols.

1. Analyze the already completely programmed subroutine *loop\_copy*. It uses a work area as the interface for reading data.
  - a) Proceed in the usual manner. Note the output of the runtime at the end of the subroutine. It is the time in milliseconds. The dimension is decisive for the interpretation of the number.
2. Optimize the performance by reading via field symbols. Implement the subroutine *loop\_assigning*.

You will find that both require field symbols *<fs\_outer>* and *<fs\_inner>* have already been prepared!

*Continued on next page*

Take a look at the runtimes output at the end of the two subroutines. Which variant is faster?

- a) See the extract from the model solution. Note the output of the runtime at the end of the subroutine. It is the time in milliseconds. The dimension is decisive for the interpretation of the number.

### Task 5: (Optional) Index Accesses Instead of Key Accesses

Replacing a key access with an index access: Copy template program SAPBC490\_TEMP\_SEQ\_INDEX\_ACCESS to your own program Z\_##\_SEQ\_INDEX\_ACCESS. In this program, data is read from two sorted internal tables. The outer driving itab contains document headers; the inner driven itab contains the dependent item entries.

1. Analyze the already completely programmed subroutine *uprog\_usual*. It uses a key access for the partially sequential reading of the inner table.
  - a) Proceed in the usual manner. If necessary, debug this template. Note the output of the runtime at the end of the subroutine. It is the time in milliseconds. The dimension is decisive for the interpretation of the number.
2. Optimize performance by changing the access to the inner table to an index access. What would the coding have to look like here?

Implement the subroutine *uprog\_tricky*.

Take a look at the runtimes output at the end of the two subroutines. Which variant is faster and why?

- a) See the extract from the model solution. Note the output of the runtime at the end of the subroutine. It is the time in milliseconds. The dimension is decisive for the interpretation of the number.

*Continued on next page*

## Task 6: Read-on-Demand: Avoiding DB Accesses Through Buffering

Avoiding DB accesses through buffering in internal tables: Continue using program Z##\_JOINS3 (if that is no longer available, copy template SAPBC490\_TEMP\_JOINS3). Then implement variant “read on demand”. Data from the KNA1 table is to be read to internal table *it\_kna1* on demand.

1. Recall the already programmed subroutines with various solution approaches. The solutions with nested joins were very ineffective. The triple join appears to have been the best solution so far.
  - a) Proceed in the usual manner.
2. Implement the subroutine *litab\_on-demand*

Here, READ is to be used to access the internal table with customer data. The customer number is already in the outer loop over the VBAK data. If there is nothing corresponding to the customer in internal table *it\_kna1*, then read the data via a SELECT SINGLE from KNA1 (on demand) and then buffer it in *it\_kna1*. Further queries regarding the same customer number can then be fulfilled via the internal table and are many times faster.

- a) See the extract from the model solution. Note the output of the runtime at the end of the subroutine. It is the time in milliseconds. The dimension is decisive for the interpretation of the number.

### Result

**Source text of the model solution:**

#### Exercise for the COLLECT Command

```
*&-----*
*& Report  SAPBC490_ITAB_COLLECT
*&-----*
*& Demo to show how the collect statement adds numerical values
*& inside an internal table.
*&-----*

REPORT  sapbc490_itab_collect.

TYPES: BEGIN OF ty_rec,
        ernam TYPE vbak-ernam,
        waerk TYPE vbak-waerk,
```

*Continued on next page*

```

        netwr TYPE vbak-netwr,
END OF ty_rec.

TYPES: ty_itab TYPE HASHED TABLE OF ty_rec
      WITH UNIQUE KEY ernam waerk.

FIELD-SYMBOLS <fs>  TYPE ty_rec.

DATA: wa_rec TYPE ty_rec.
DATA: itab TYPE ty_itab.

DATA: counter TYPE i.
DATA: t1 TYPE i, t2 LIKE t1.

START-OF-SELECTION.
*#####
*# PERFORM uprog_usual.
*# PERFORM uprog_collect.

*-----*
*&      Form  uprog_usual
*-----*
*      collect simulated with usual programming
*-----*
FORM uprog_usual.

GET RUN TIME FIELD t1.

SELECT * FROM vbak
      INTO CORRESPONDING FIELDS OF wa_rec
      WHERE vbeln between '0000002000' and '0000009000'.

READ TABLE itab FROM wa_rec
      ASSIGNING <fs>.

IF sy-subrc <> 0.
  INSERT wa_rec INTO TABLE itab.
ELSE.
  <fs>-netwr = <fs>-netwr + wa_rec-netwr.
ENDIF.
ENDSELECT.

```

*Continued on next page*

```

        GET RUN TIME FIELD t2.
        t2 = t2 - t1.
        WRITE: / 'Runtime = ', t2.
        counter = LINES( itab ).
        WRITE: counter.

ENDFORM.          "uprog_usual

*&-----*
*&      Form uprog_collect showing the usage of the COLLECT statement
*&-----*
FORM uprog_collect.

        GET RUN TIME FIELD t1.

        SELECT ernam waerk netwr FROM vbak
           INTO wa_rec
           WHERE vbeln between '0000002000' and '0000009000'.
           COLLECT wa_rec INTO itab.
        ENDSELECT.

        GET RUN TIME FIELD t2.
        t2 = t2 - t1.
        WRITE: / 'Runtime = ', t2.
        counter = LINES( itab ).
        WRITE: counter.

ENDFORM.          "uprog_collect

```

### **Exercise for the TRANSPORTING Addition**

```

*&-----*
*& Report SAPBC490_ITAB_TRANSPORTING
*&-----*
*& Demo to show how the transporting clause works
*&-----*

REPORT sapbc490_itab_transporting.
*
```

*Continued on next page*

```

* just shows the usage of the TRANSPORTING technique
* very nice too: TRANSPORTING NO FIELDS
* a kind of existancy check !

DATA: count TYPE i, wa_index TYPE sy-tabix.
DATA: index_itab TYPE STANDARD TABLE OF sy-tabix
      WITH NON-UNIQUE DEFAULT KEY INITIAL SIZE 10,
      itab TYPE STANDARD TABLE OF vbap.

START-OF-SELECTION.
*#####
*#####
  SELECT * FROM vbap INTO TABLE itab
    WHERE vbeln < '1000020000'.

LOOP AT itab TRANSPORTING NO FIELDS
  WHERE matnr = 'SD000001'.
  APPEND sy-tabix TO index_itab.
  ADD 1 TO count.
ENDLOOP.

LOOP AT index_itab INTO wa_index.
  WRITE: / wa_index.
ENDLOOP.

ULINE.
WRITE: / count.

```

## Main Exercise for Table Types

```

*&-----*
*& Report  SAPBC490_ITAB_3LOOPS *
*&-----*
*& Demo to show good performance when using the wright
*& table type -> SORTED / HASHED
*&-----*
*
* This report shows the usage of Internal Table types
* Just to use the correct type can really improve performance a lot!
* You will be astounded what happens in the third routine!
* --> as a fourth step you could change the content of the itabs
* add the modify to routines 1,2 and compare again with field-symbols!
* --> again you will be astounded!

```

*Continued on next page*

```

*&-----*
REPORT  sapbc490_itab_3loops.

START-OF-SELECTION.
*#####
PERFORM nested_itabs_wa_bad.      "use standard tabletypes
PERFORM nested_itabs_wa_better. "use perfect tabletypes
PERFORM nested_itabs_the_best.   "really the best ?

*&-----*
*&      Form  nested_itabs_wa_bad
*&-----*
FORM nested_itabs_wa_bad.
DATA: t1 TYPE i, t2 TYPE i.
DATA: it_saplane TYPE STANDARD TABLE OF saplane
      WITH KEY planetype.
DATA: it_spfli TYPE STANDARD TABLE OF spfli
      WITH KEY carrid connid.
DATA: it_sflight TYPE STANDARD TABLE OF sflight
      WITH KEY carrid connid fldate.
DATA: wa_saplane TYPE saplane.
DATA: wa_spfli TYPE spfli.
DATA: wa_sflight TYPE sflight.

***** BLACK BOX - we just read data into itabs quick and dirty....*****
SELECT * FROM saplane INTO TABLE it_saplane.
SELECT * FROM spfli INTO TABLE it_spfli.
SELECT * FROM sflight INTO TABLE it_sflight.

*****Read from itabs *****
GET RUN TIME FIELD t1.
LOOP AT it_spfli INTO wa_spfli.
LOOP AT it_sflight INTO wa_sflight
      WHERE carrid = wa_spfli-carrid
            AND connid = wa_spfli-connid.
READ TABLE it_saplane INTO wa_saplane
      WITH TABLE KEY planetype = wa_sflight-planetype.
*      WRITE: / wa_spfli-carrid, wa_spfli-connid,
*                  wa_sflight-fldate, wa_saplane-planetype.
ENDLOOP.

```

*Continued on next page*

```

ENDLOOP.

GET RUN TIME FIELD t2.
t2 = t2 - t1.
WRITE: / 'runtime = ', t2.

ENDFORM.                                     "nested_itabs_wa_bad

*-----*
*&      Form nested_itabs_wa_better; this is much better!
*-----*
FORM nested_itabs_wa_better.

DATA: t1 TYPE f, t2 TYPE f.
DATA: it_saplane TYPE HASHED TABLE OF saplane
      WITH UNIQUE KEY planetype.

DATA: it_spfli TYPE STANDARD TABLE OF spfli
      WITH KEY carrid connid.
DATA: it_sflight TYPE SORTED TABLE OF sflight
      WITH UNIQUE KEY carrid connid flddate.
DATA: wa_saplane TYPE saplane.
DATA: wa_spfli TYPE spfli.
DATA: wa_sflight TYPE sflight.

**** BLACK BOX - we just read data into itabs quick and dirty...****
SELECT * FROM saplane INTO TABLE it_saplane.
SELECT * FROM spfli INTO TABLE it_spfli.
SELECT * FROM sflight INTO TABLE it_sflight.

*****Read from itabs ****
GET RUN TIME FIELD t1.
LOOP AT it_spfli INTO wa_spfli.
  LOOP AT it_sflight INTO wa_sflight
    WHERE carrid = wa_spfli-carrid
      AND connid = wa_spfli-connid.
    READ TABLE it_saplane INTO wa_saplane
      WITH TABLE KEY planetype = wa_sflight-planetyp.
*     WRITE: / wa_spfli-carrid, wa_spfli-connid,
*           wa_sflight-flddate, wa_saplane-planetyp.
  ENDLOOP.
ENDLOOP.

```

*Continued on next page*

```

GET RUN TIME FIELD t2.
t2 = t2 - t1.
WRITE: / 'runtime = ', t2.

ENDFORM.           "nested_itabs_wa_better

*&-----*
*&      Form nested_itabs_the_best; is this really the best? -> no!
*&      afterwards you could modify data in the inner itabs and
*&      then you will see that field-symbols win finally!
*&      this is not shown here anymore...
*&-----*

FORM nested_itabs_the_best.
DATA: t1 TYPE f, t2 TYPE f.
DATA: it_saplane TYPE HASHED TABLE OF saplane
      WITH UNIQUE KEY planetype.

DATA: it_spfli TYPE STANDARD TABLE OF spfli
      WITH KEY carrid connid.
DATA: it_sflight TYPE SORTED TABLE OF sflight
      WITH UNIQUE KEY carrid connid flddate.
FIELD-SYMBOLS: <fs_saplane> TYPE saplane.
FIELD-SYMBOLS: <fs_spfli> TYPE spfli.
FIELD-SYMBOLS: <fs_sflight> TYPE sflight.

***** BLACK BOX - we just read data into itabs quick and dirty....*****
SELECT * FROM saplane INTO TABLE it_saplane.
SELECT * FROM spfli INTO TABLE it_spfli.
SELECT * FROM sflight INTO TABLE it_sflight.

*****Read from itabs ****
GET RUN TIME FIELD t1.
LOOP AT it_spfli ASSIGNING <fs_spfli>.
  LOOP AT it_sflight ASSIGNING <fs_sflight>
    WHERE carrid = <fs_spfli>-carrid
      AND connid = <fs_spfli>-connid.
  READ TABLE it_saplane ASSIGNING <fs_saplane>
    WITH TABLE KEY planetype = <fs_sflight>-planetyp.
*  WRITE: / <fs_spfli>-carrid, <fs_spfli>-connid,

```

*Continued on next page*

```

*           <fs_sflight>-fldate, <fs_saplane>-planetype.
ENDLOOP.
ENDLOOP.

GET RUN TIME FIELD t2.
t2 = t2 - t1.
WRITE: / 'runtime = ', t2.

ENDFORM.          "nested_itabs_the_best.

```

### Exercise for Nested Multidimensional Internal Tables

```

*&-----*
*& Report  SAPBC490_ITAB_NESTED
*&-----*
*& Demo to show how accesses to multidimensional itabs
*& can be improved. Use field-symbols!
*&-----*

REPORT  sapbc490_itab_nested.

START-OF-SELECTION.
*#####
TYPES:
* types for the inner table
BEGIN OF inner_type,
    fldate  TYPE sflight-fldate,
    seatsmax TYPE sflight-seatsmax,
    seatsocc TYPE sflight-seatsocc,
END OF inner_type,

inner_table_type TYPE STANDARD TABLE OF inner_type
WITH NON-UNIQUE DEFAULT KEY,

* types for the outer table
BEGIN OF outer_type,
    carrid   TYPE spfli-carrid,
    connid   TYPE spfli-connid,
    airpto   TYPE spfli-airpto,
    airpfrom TYPE spfli-airpfrom,

```

*Continued on next page*

```

        capacity TYPE inner_table_type,
END OF outer_type,

outer_table_type TYPE STANDARD TABLE OF outer_type
WITH NON-UNIQUE DEFAULT KEY.

DATA:
* data objects
flight_info TYPE outer_table_type,
runtime      TYPE i.

START-OF-SELECTION.
*#####
* fill internal table
PERFORM fill_table CHANGING flight_info.

PERFORM loop_copy      CHANGING flight_info.
PERFORM loop_assigning  CHANGING flight_info.

*&-----*
*&      Form  loop_copy
*&-----*
*      text
*-----*
*      <-P_ITAB    text
*      <-P_RUNTIME  text
*-----*
FORM loop_copy CHANGING p_itab    TYPE outer_table_type.

* local data
DATA: t2 TYPE i, t1 LIKE t2.
DATA:
wa_outer TYPE outer_type,
wa_inner TYPE inner_type.
GET RUN TIME FIELD t1.

* nested loop
LOOP AT p_itab INTO wa_outer.
LOOP AT wa_outer-capacity INTO wa_inner.
ADD 1 TO wa_inner-seatsocc.
MODIFY wa_outer-capacity FROM wa_inner.
ENDLOOP.

```

*Continued on next page*

```

        CLEAR wa_inner.
        MODIFY p_itab FROM wa_outer.
    ENDOLOOP.

    GET RUN TIME FIELD t2.
    t2 = t2 - t1.
    WRITE: / t2.

ENDFORM.                                     " loop_copy
*-----*
*&      Form  loop_assigning
*-----*
*      <--P_ITAB  text
*      <--P_RUNTIME  text
*-----*
FORM loop_assigning CHANGING p_itab      TYPE outer_table_type.

* local data
FIELD-SYMBOLS:
<fs_outer> TYPE outer_type,
<sf_inner> TYPE inner_type.
DATA: t2 TYPE i, t1 LIKE t2.

GET RUN TIME FIELD t1.

LOOP AT p_itab ASSIGNING <fs_outer> .
  LOOP AT <fs_outer>-capacity ASSIGNING <fs_inner>.
    ADD 1 TO <fs_inner>-seatsocc.
  ENDLOOP.
ENDLOOP.

GET RUN TIME FIELD t2.
t2 = t2 - t1.
WRITE: / t2.

ENDFORM.                                     " loop_assigning
*-----*
*&      Form  fill_table
*-----*
*      <--P_FLIGHT_INFO  text
*-----*
FORM fill_table CHANGING p_itab TYPE outer_table_type.

```

*Continued on next page*

```

* local data
DATA:
    wa_outer LIKE LINE OF p_itab,
    wa_inner TYPE inner_type.

SELECT carrid connid airpfrom airpto
FROM spfli
INTO (wa_outer-carrid, wa_outer-connid,
      wa_outer-airpfrom, wa_outer-airpto)
* where ...
.

SELECT fldate seatsmax seatsocc
FROM sflight
INTO CORRESPONDING FIELDS OF wa_inner
WHERE carrid = wa_outer-carrid
AND connid = wa_outer-connid.

INSERT wa_inner INTO TABLE wa_outer-capacity.

ENDSELECT.

INSERT wa_outer INTO TABLE p_itab.
CLEAR wa_outer.

ENDSELECT.
ENDFORM.                               " fill_table

```

## Exercise for Index and Key Accesses

\*&-----  
\*& change access via key to an indexaccess and you can improve  
\*& performance; in this example we have two sorted itabs  
\*& on is the header itab (vbak) on the other the position-itab (vbap)  
\*&-----\*

```
REPORT SAPBC490_ITAB_SEQ_INDEX_ACCESS.
```

```

FIELD-SYMBOLS: <fs_vbak> TYPE vbak.
FIELD-SYMBOLS: <fs_vbap> TYPE vbap.
SELECT-OPTIONS vbeln FOR <fs_vbak>-vbeln
      DEFAULT '1000000000' TO '1000022000'.

```

*Continued on next page*

```

TYPES: ty_itvbak TYPE SORTED TABLE OF vbak
       WITH UNIQUE KEY vbeln.
TYPES: ty_itvbap TYPE SORTED TABLE OF vbap
       WITH UNIQUE KEY vbeln posnr.
DATA: it_vbak TYPE ty_itvbak.
DATA: it_vbap TYPE ty_itvbap.
DATA: index TYPE i.

DATA: t1 TYPE i, t2 LIKE t1.

START-OF-SELECTION.
*#####
  PERFORM fill_table.

  PERFORM uprog_usual.
  PERFORM uprog_tricky.

*-----*
*&      Form  uprog_usual
*-----*
FORM uprog_usual.
  GET RUN TIME FIELD t1.
  LOOP AT it_vbak ASSIGNING <fs_vbak>.
    LOOP AT it_vbap ASSIGNING <fs_vbap>
      WHERE vbeln = <fs_vbak>-vbeln.
*     WRITE: / <fs_vbap>-vbeln, <fs_vbap>-posnr.
    ENDLOOP.
  ENDLOOP.
  GET RUN TIME FIELD t2.
  t2 = t2 - t1.
  WRITE: / 'Runtime = ', t2.
ENDFORM.          "uprog_usual

*-----*
*&      Form  uprog_tricky
*-----*
FORM uprog_tricky.

  GET RUN TIME FIELD t1.

```

*Continued on next page*

```

index = 1.
LOOP AT it_vbak
ASSIGNING <fs_vbak>.
LOOP AT it_vbap
ASSIGNING <fs_vbap>
FROM index.
IF <fs_vbap>-vbeln <> <fs_vbak>-vbeln.
index = sy-tabix.
EXIT.
ENDIF.
WRITE: / <fs_vbap>-vbeln, <fs_vbap>-posnr.
ENDLOOP.
ENDLOOP.

GET RUN TIME FIELD t2.
t2 = t2 - t1.
WRITE: / 'Runtime = ', t2.

ENDFORM.                               "uprog_tricky

*-----*
*&      Form fill_table (quick and dirty )
*-----*
FORM fill_table .
SELECT * FROM vbak INTO TABLE it_vbak
WHERE vbeln IN vbeln.
SELECT * FROM vbap INTO TABLE it_vbap
WHERE vbeln IN vbeln.
ENDFORM.                               " fill_table

```

### Main Exercises for READ ON DEMAND

```

*** see exercises on joins3
*** this is an additional form-routine showing the usage
*** of itab-techniques
***...
*-----?-----*
*      FORM itab_ondemand; is this better ?
*-----*  


```

*Continued on next page*

```

FORM itab_ondemand.

DATA: it_kna1 TYPE HASHED TABLE OF rec_kunnr WITH UNIQUE KEY kunnr,
      wa TYPE rec_kunnr.

GET RUN TIME FIELD t1.

***  

*** Useful technique, when does it work best ?  

***  

*** even much better would be combining this with 2-join, but this  

*** example is good to get analysed in the ST05 and understand how  

*** accessing the itab works; you see less and less KNA1-accesses

SELECT vbeln kunnr erdat FROM vbak
      INTO (rec-vbeln, rec-kunnr, rec-erdat)
      WHERE vbeln IN vbeln.
SELECT posnr matnr FROM vbap
      INTO (rec-posnr, rec-matnr)
      WHERE vbeln = rec-vbeln.

*** the more accesses to the same kunnr the less db-accesses you see
*** in the trace -> the KNA1 accesses disappear!
READ TABLE it_kna1 INTO rec-name1
      WITH TABLE KEY kunnr = rec-kunnr.
IF sy-subrc <> 0.
  SELECT SINGLE name1 FROM kna1
    INTO rec-name1
    WHERE kunnr = rec-kunnr.
  wa-kunnr = rec-kunnr.
  wa-name1 = rec-name1.
  INSERT wa INTO TABLE it_kna1.
ENDIF.
*   WRITE: / rec
ENDSELECT.
ENDSELECT.
GET RUN TIME FIELD t2.
t2 = t2 - t1.  WRITE: / 'runtime ondemand = ', 40 t2.
ENDFORM.          "itab_ondemand

```



## Lesson Summary

You should now be able to:

- Explain the different buffering strategies
- Use buffering strategies to avoid database accesses



## Unit Summary

You should now be able to:

- Explain the different buffering strategies
- Use buffering strategies to avoid database accesses

# Unit 7

## Summary and Additional Tips

### Unit Overview

#### Content:



- Summary
- The golden rules of performance
- Additional tips for analyzing expensive statements - cursor cache analysis
- Access times for modularization units
- Recommendations for object-oriented programming
- Roadmaps



### Unit Objectives

After completing this unit, you will be able to:

- Explain all the important aspects of efficient programming

### Unit Contents

Lesson: Summary and Additional Tips..... 288

# Lesson: Summary and Additional Tips

## Lesson Overview

### Content: Summary



- Summary
- The golden rules of performance
- Additional tips on analyzing expensive statements



## Lesson Objectives

After completing this lesson, you will be able to:

- Explain all the important aspects of efficient programming

## Business Example

### Brief Summary



SAP Web Application Server



- **Reduce database load**
- **Reduce network load**
- **Only read necessary data**
- **No multiple reads**
- **Application of buffering techniques**
- **Minimize search effort for the database (indexes)**
- **Use bundling techniques to fill data packages**
- **Favor mass accesses**

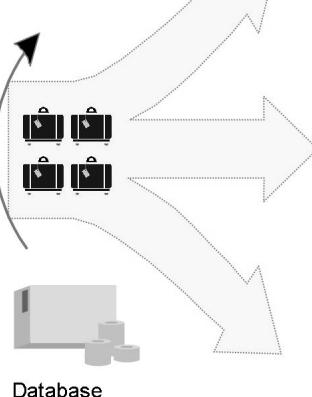
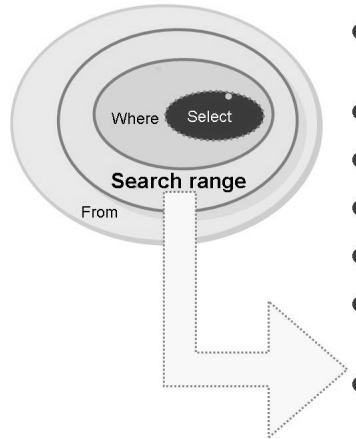


Figure 191: General Rules



### SELECT: Possible dataset



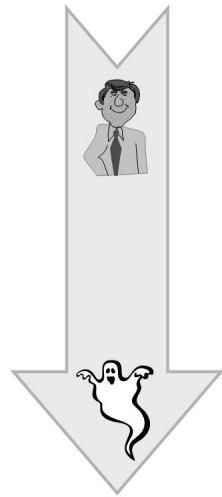
- Few indexes per table (4-7)
- Fewer short fields in the index
- Selective fields in front
- Disjunctive indexes
- No indexes of transaction data
- Change existing indexes before you build new ones
- Under certain circumstances, it can even be advantageous to delete indexes
- Do not change any SAP indexes or SAP tables (unless instructed to do so by SAP; see the SAP Notes in SAPNet)

**Figure 192: Rules for Index Design**

The selectivity analysis tool also assists you in designing indexes.



### Using Operators



SELECT * FROM SFLIGHT INTO WA WHERE CARRID ...	=	'AA'
	IN	( 'AA', 'LH', 'UA' ).
	BETWEEN	'AA' AND 'LH'.
	LIKE	'A%'
	NOT	CARRID IN ( 'AA', 'LH', 'UA' ).

**Figure 193: Quality of Operators in the WHERE Clause**

The operators < and > are not shown in the diagram. They are comparable to the costs of a BETWEEN. The following applies for the operator OR: external ORs are better than inner ORs (in other words, ORs in the brackets of the where clause for the select are not as advantageous as external ORs). However, it is better to avoid ORs and to replace them with the IN operator and an argument list (if possible).



- Native SQL
- Subqueries, ABAP joins
- SELECT ... BYPASSING BUFFER
- SELECT FOR UPDATE
- Aggregate functions (COUNT, MIN, MAX, SUM, AVG)
- SELECT DISTINCT ...
- WHERE condition with "IS NULL"
- ORDER BY, GROUP BY (HAVING)
- For single record-buffered tables:
  - All SQL statements except SELECT SINGLE ...
- For generically buffered tables:
  - All SQL statements except SELECT \* .. if WHERE clause is field = value for all fields included in generic area

Figure 194: SQL Statements Bypassing the Table Buffer

## Expensive Statements & Analyzing the Cursor Cache

The database SQL cache (also known as the cursor cache) is an overview of all SQL statements that were executed by the database in a given time period. You can use threshold values to identify expensive SQL statements and then classify them as having an index problem or another problem. The indicated threshold values only provide a guideline.

The SAP system and the database must have been running for several days prior to performing a database SQL cache analysis. To perform a database SQL cache analysis, you require the number of data blocks logically read since database startup (reads) and the number of physically read data blocks (physical reads).

- Reads (data blocks logically read from the database buffer)
- Physical reads (data reads physically read from the hard disk)
- Data buffer size and Data buffer quality (size and quality of the database buffer)
- User calls (number of SQL statements sent to the database)
- Reads / User calls (relevant ratio) The ratio Reads / User calls is an indicator for whether it would be worthwhile to perform a database SQL cache analysis. If the ratio is significantly larger than 20:1, an analysis is urgently required.
- Total Execution (Oracle): Number of executions of the SQL statement

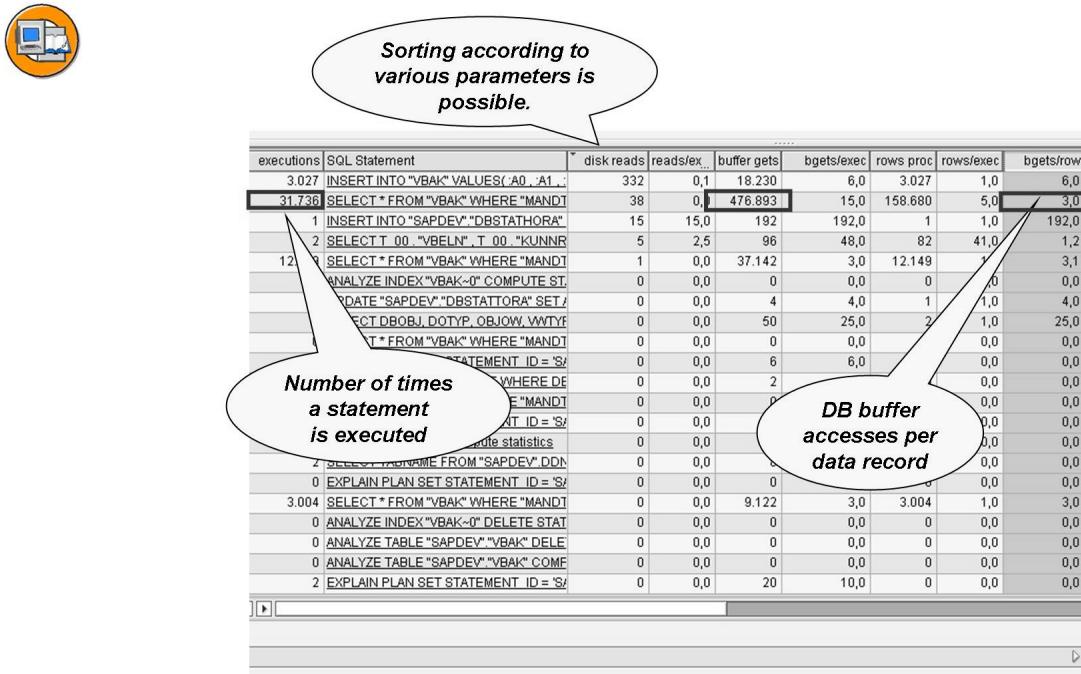


Figure 195: Identifying Expensive Burdensome Statements

Navigation: From the initial screen of the database monitor, choose Detailed Analysis. To access the database SQL cache for the respective database system, choose SQL request (Oracle), SQL statement (Informix), Stored proc. stats (MS SQL Server) or Diagnosis monitor (ADABAS D).

- Buffer Gets (Oracle): Number of logical reads required for the execution
- Records processed (Oracle): Number of transferred (read) data records
- Bufgets/row (Oracle): Average number of logical reads required for each data record (also Bufgets/record, depending on the release)
- Bufgets/execution (Oracle): Logical reads per execution of the statement

As a rule of thumb, you can identify an SQL statement as expensive if it causes more than 5% of all logical reads, or more than 3% of all physical reads.

This is where the problem characterization (as either an index or other problem) comes in. The above example shows a database SQL cache for the VBAK table (for Oracle) sorted by the number of disk reads. For each data record, the number of logical reads is **less than 5**. Therefore, you can regard the access path as suitable. It is not possible to optimize the SQL statement using database methods (creating or modifying indexes). Many data records are read (see Records processed) and the SQL statement is executed frequently (see Total Executions). An optimization can only be attempted in terms of the ABAP coding or at the business process level.

If there were **more than 5 logical reads for each data record**, you should attempt to optimize using database methods (index design).

Begin by double-clicking the line containing the SQL statement or choose *Detail*. In the resulting detail view, choose *DDIC Info* to view data on the relevant dictionary object (table or view), choose *Explain* to view the access path, or choose *ABAP display* to view the source code of the last ABAP program that triggered a PREPARE for this SQL statement.

## Performance of Modularization Units

In the following examples, the basis of measurement involved testing modularization units when transferring a parameter with Call By Reference (the measurements were taken on what is now considered a slow machine with a PII processor - 300MHz).



■ Access times for modularization units (1 par. with call by reference)

	PERFORM	Class Method	Instance Method	Function Module
Local	1.3 µs	1.3 µs	1.8 µs	3.5 µs
Global	4.5 µs	3.1 µs	3.5 µs	5.3 µs



- Give preference to CALL BY REFERENCE (Call By Value only if necessary).
- CALL BY VALUE only for smaller import parameters (<100Byte)



- Dynamic calls are about 3 to 6 times slower than static calls.
- No dynamic calls in time-critical coding
- Use fully qualified parameters and field symbols.



Figure 196: Access Times for Modules & Tips

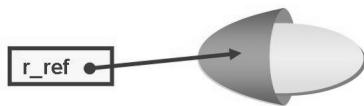
“Global” is used here to understand calls from external subroutines or function modules from an external new function group.

When transferring with Call By Value, the system requires a new data control block internally. Even if only one byte is transferred, this is better than a Call By Reference.

However, there are cases where a Call by Value is used or has to be used: returning parameters of methods or all parameters of handling methods, non-TABLES parameters for RFCs and update functions.

As of release SAP Web AS 6.10, “table sharing” is used when copying internal tables (but only in the case of flat itabs). The actual copy process is delayed until the copied itab is changed. You can trace this operation in the debugger under the memory usage display. (Nested itabs are always duplicated immediately. As mentioned before, sharing only affects flat itabs.)

Avoid any unnecessary dynamic calls. Static calls are always 3 to 6 times faster. You would lose the most time with the following dynamic code: ('CLASS')=>('METHOD').



- Avoid unnecessary instantiations of objects.
- Try to reuse objects instead of generating new ones (an OO object requires 40 bytes for administration).
- Do not design too granularly; objects should “do something” (initialization and garbage collection overhead).
- Avoid too many nested method calls. (cascading method calls)
- Avoid too many calls of atomic methods (Methods with only one internal assignment).
- References to objects in function groups, handler tables that are no longer required.
- Use the Memory Inspector to determine and analyze the size of the used objects, internal tables or strings.

**Figure 197: Recommendations Regarding OO**

In the object-oriented ABAP (ABAP OO), you can define an attribute as PUBLIC and READ-ONLY. This allows a client a direct read access from outside and prevents the need for method calls. However, encapsulation as the principle of the object-oriented approach is bypassed here. It behaves the same way with regard to the “friendship concept”, which is also used in the C++ programming language (but not in JAVA). Friendly classes can directly access the private encapsulated components in the class that grants them friendship.

To delete an object, all references to the object have to be removed, including the objects stored in internal tables or any event-triggering objects in the handler tables, for example (it is also possible to trace here in the debugger or Memory Inspector).

## Roadmaps - Overview Diagrams for Performance Analysis



### **Explanation of symbols for the following roadmap:**

- Analysis action or initial situation (usually start of roadmap)
- ? Decision node (if “yes”, proceed as described in subsidiary node; if “no”, continue with next step in sibling node )
- ! Subsidiary step (after this step, proceed with next step in sibling node)
- ⌚ Concrete action (end of the step sequence)
- ➔ Reference to another roadmap (end of the step sequence)

**Figure 198: Explanation of Symbols in the Roadmaps**

The following diagrams provide you with hints and advice about how to approach the analysis of performance problems in a structured manner. The starting point is identifying performance-critical single objects with the workload analysis.

These roadmaps use the following symbols: top symbol – the **Monitor** – is the starting point for a roadmap. Typically, you will have just completed an analysis task and want to apply an optimization measure.

**Question mark:** A yes/no question you must answer. If the answer is affirmative, apply the procedure indicated in the subsidiary node. If the answer is negative, proceed as described in the next sibling node.

**Exclamation mark:** Indicates an interim result. Apply the optimization procedure in the next sibling node.

**Tool icon:** A concrete action is required (end of the optimization path).

**Arrow:** Refers to another roadmap (end of the optimization path)

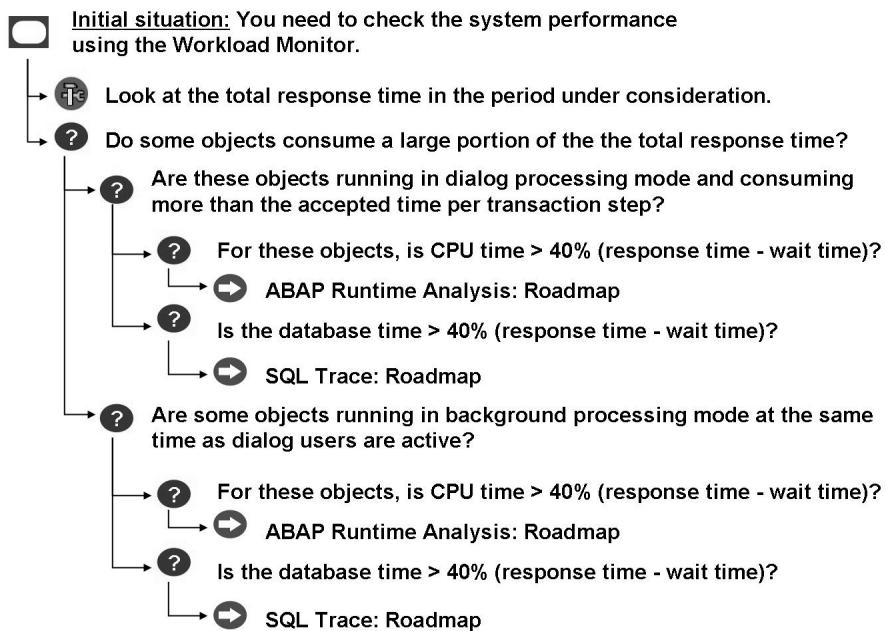


Figure 199: Roadmap: Identifying Single Objects with a Poor Performance

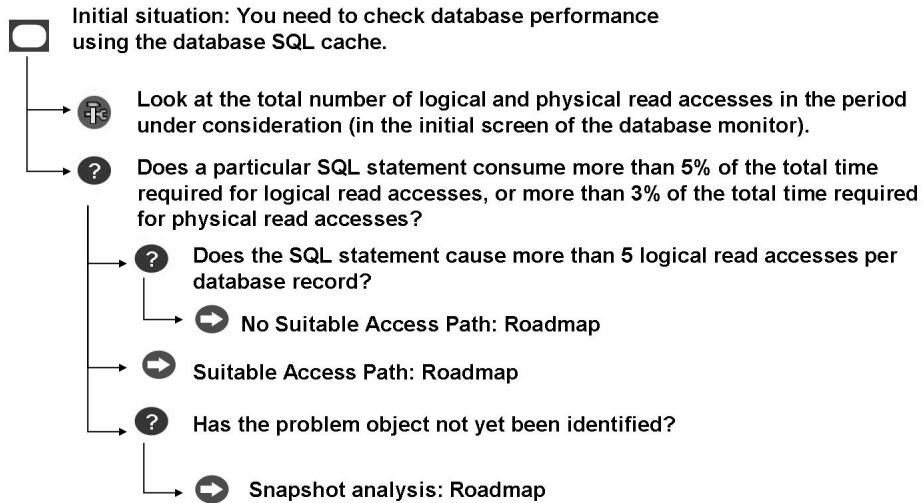
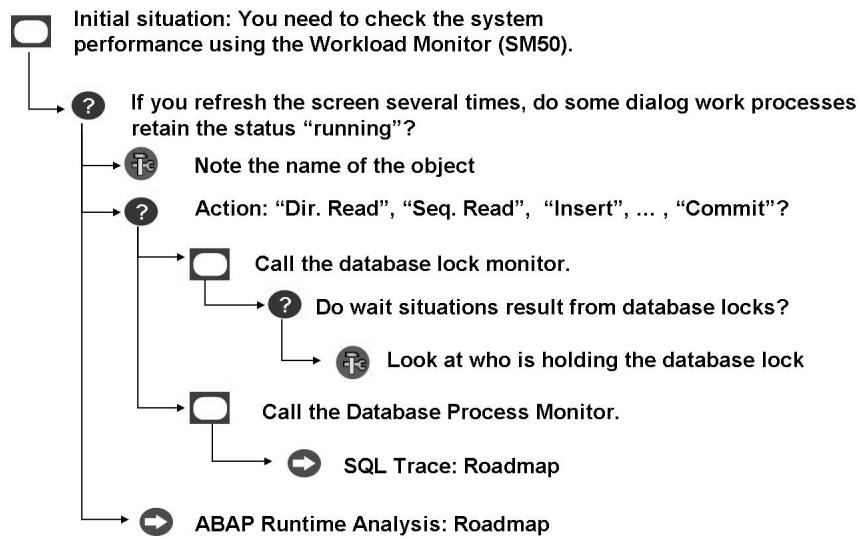
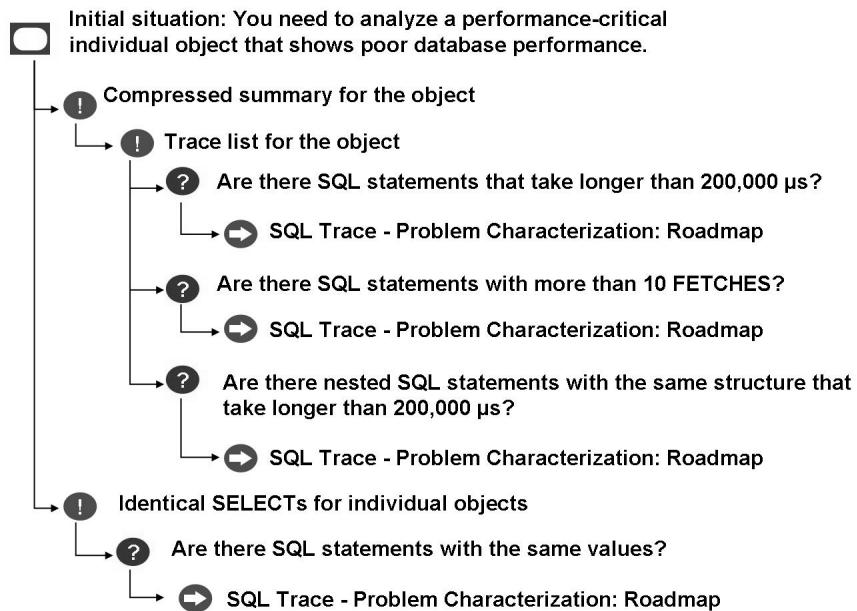


Figure 200: Roadmap: DB SQLCache (Cursor Cache) Analysis



**Figure 201: Roadmap: Snapshot Analysis, SM50**



**Figure 202: Roadmap: SQL Trace**

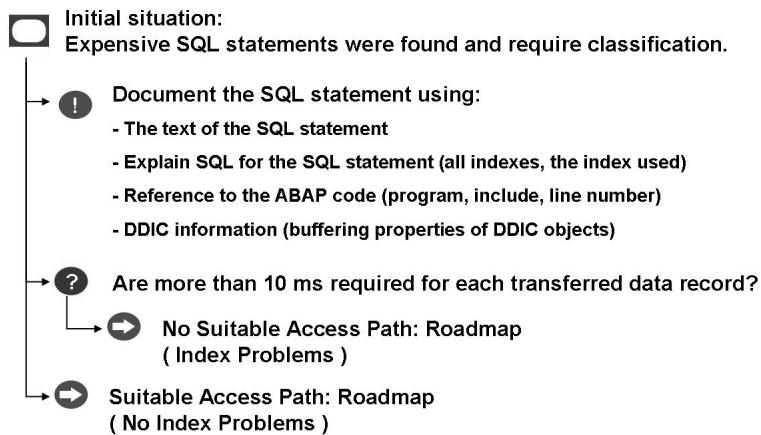


Figure 203: Roadmap: SQL Trace - Problem Characterization

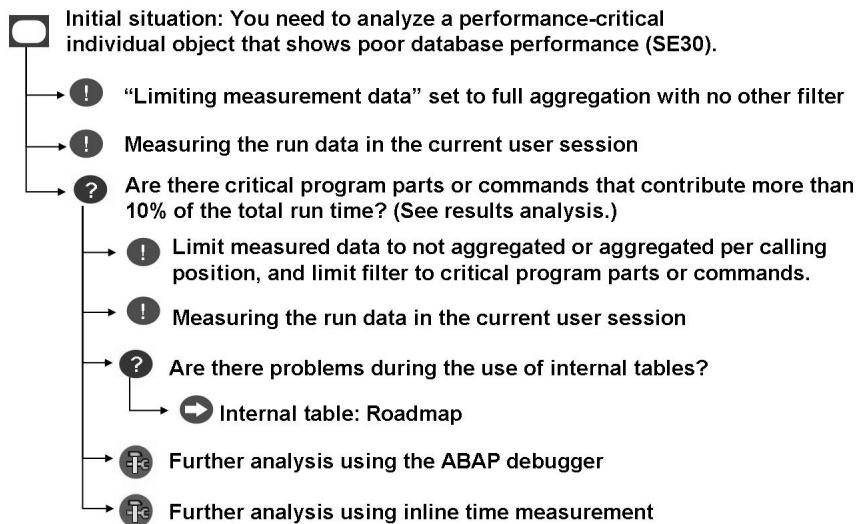
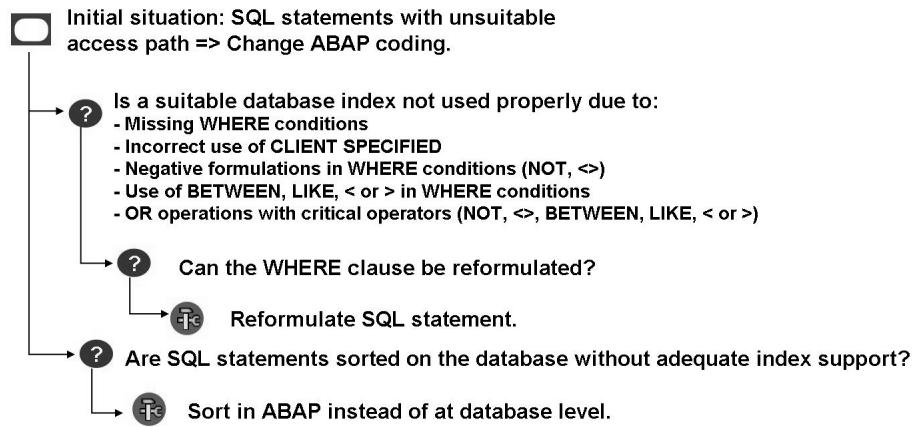
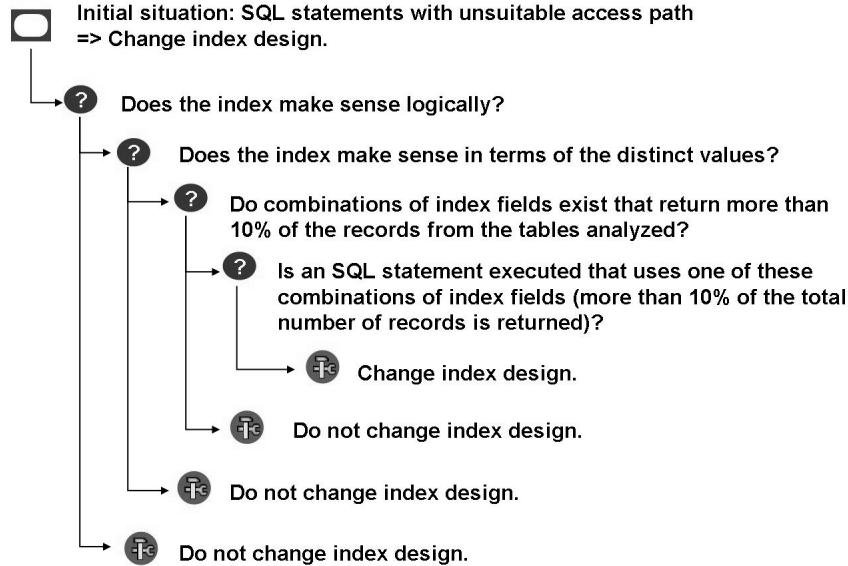


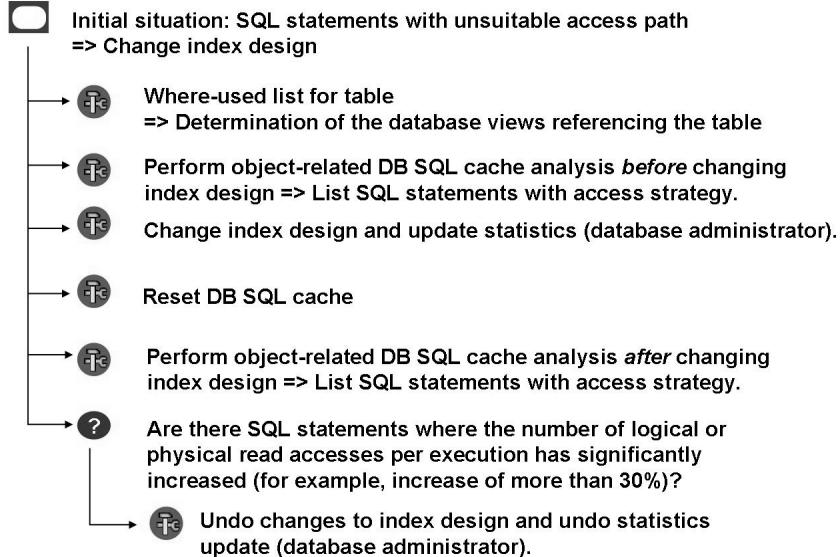
Figure 204: Roadmap: Runtime Analysis (SE30)



**Figure 205: Roadmap: Changing ABAP Code (Unsuitable Access Path)**

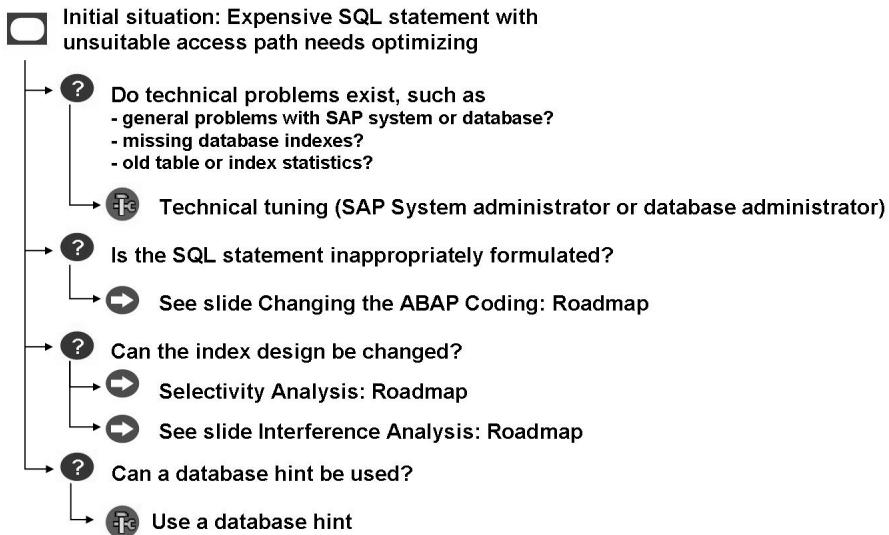


**Figure 206: Roadmap: Selectivity Analysis**

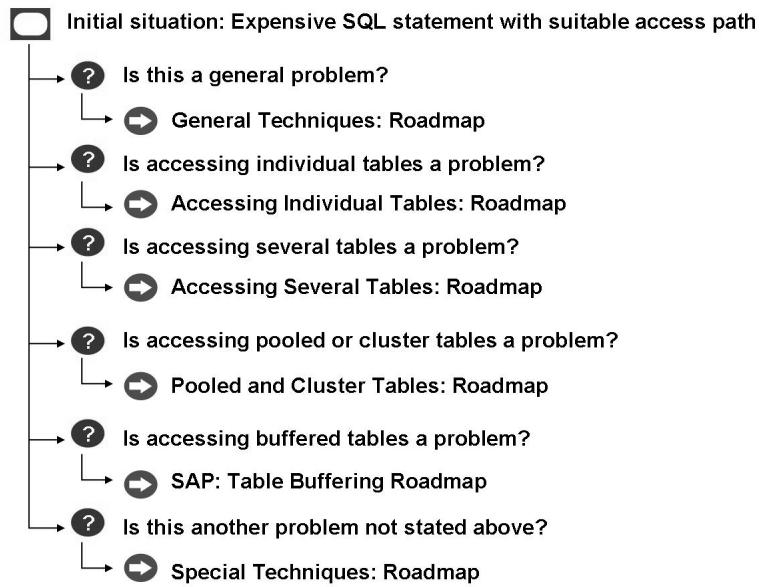


**Figure 207: Roadmap: Interference Analysis**

The technical term interference analysis describes a procedure used to verify a performance improvement after you change the index design. It is a very complicated procedure that developers rarely use in practice; it is normally carried out by database specialists.

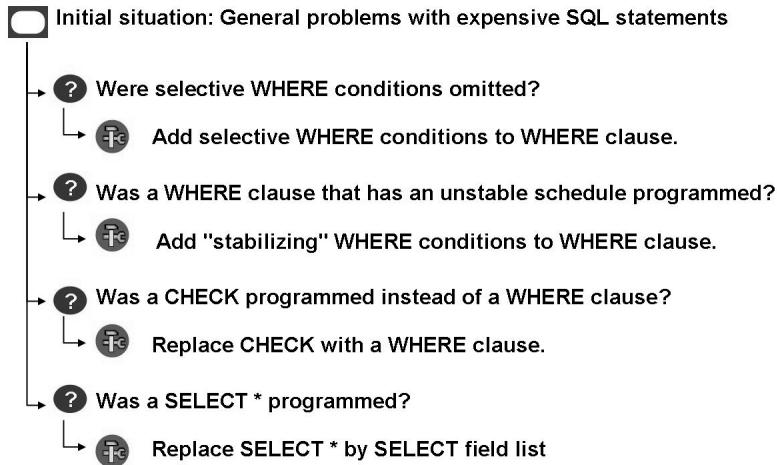


**Figure 208: Roadmap: Unsuitable Access Path (Index Problem)**



**Figure 209: Roadmap: Suitable Access Path**

The Suitable Access Path roadmap shows options for improving the index design after it has already been optimized.



**Figure 210: Roadmap: General Techniques (for Expensive Statements)**

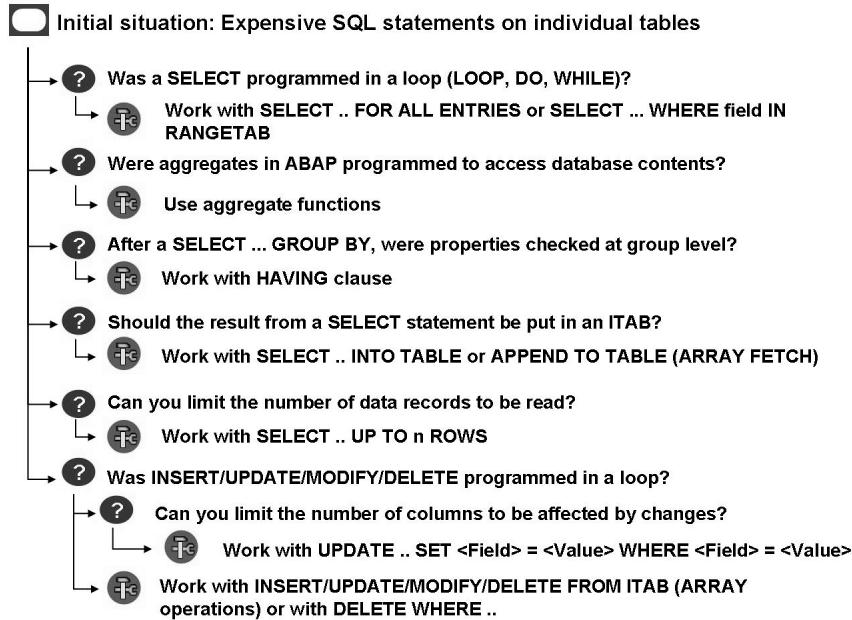


Figure 211: Roadmap: Accessing Single Tables

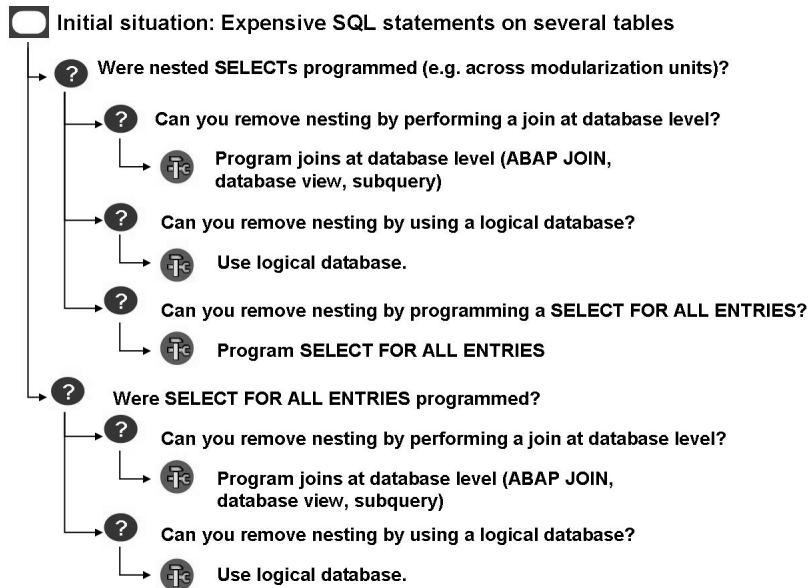
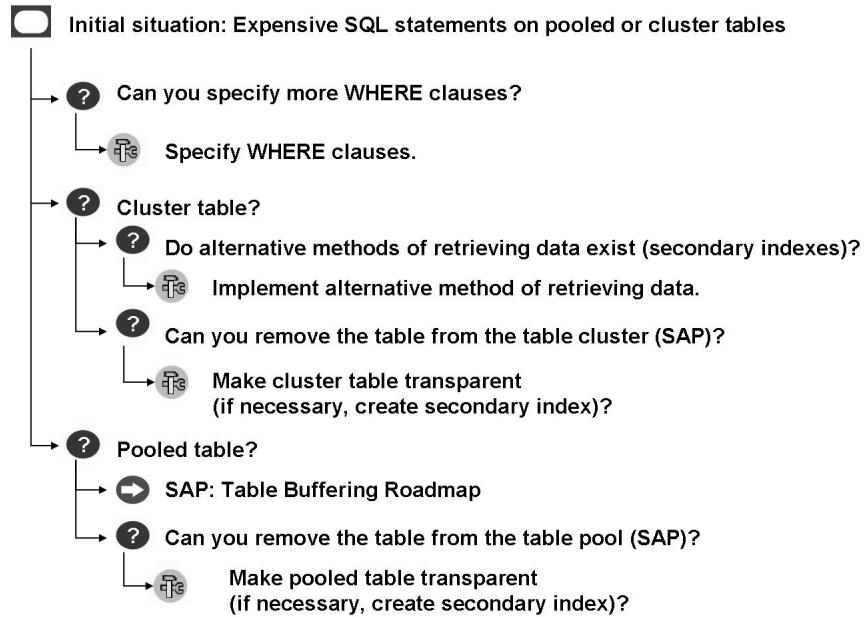
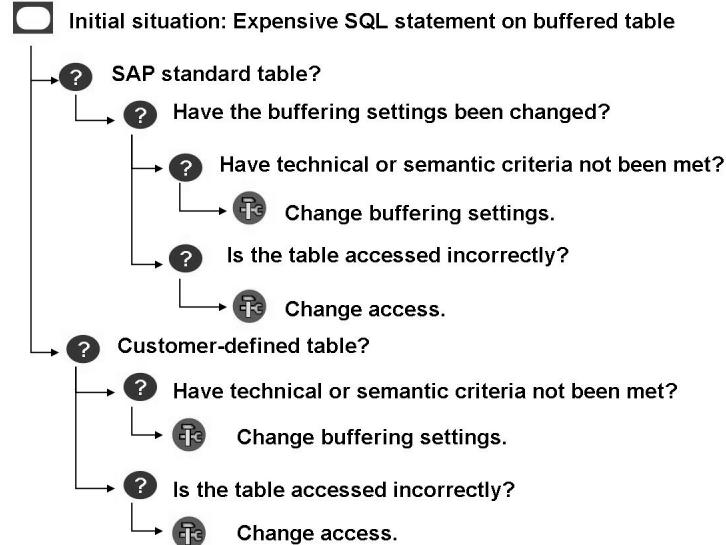


Figure 212: Roadmap: Accessing Multiple Tables



**Figure 213: Roadmap: Pooled and Cluster Tables**



**Figure 214: Roadmap: SAP Table Buffering**

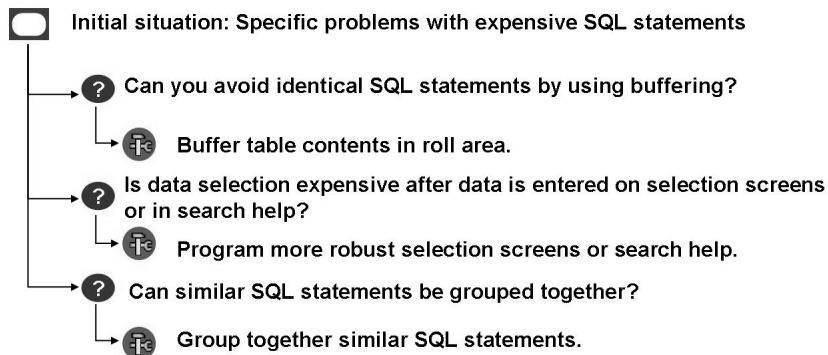


Figure 215: Roadmap: Special Techniques

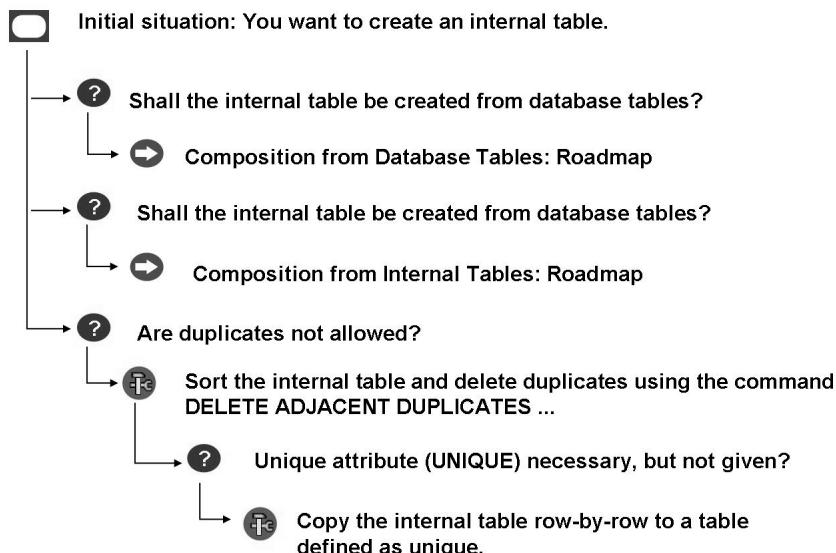
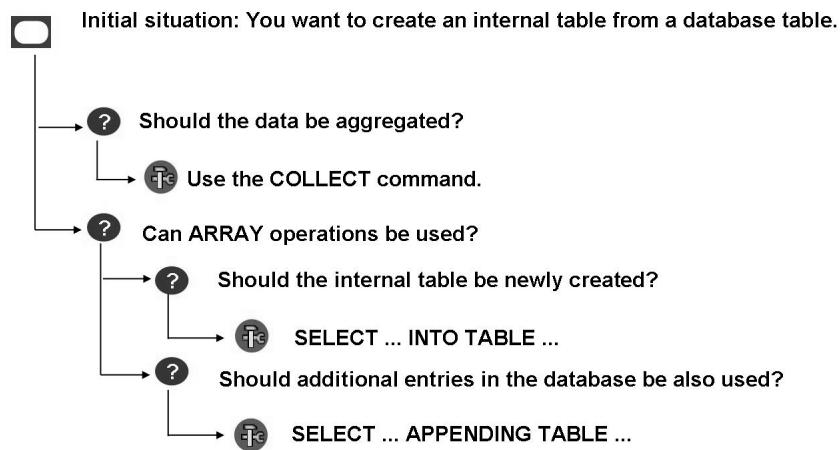
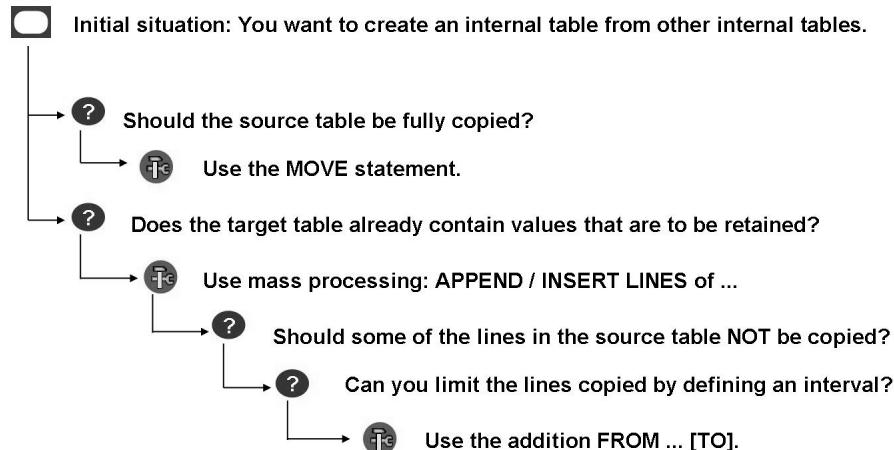


Figure 216: Roadmap: Composition of Internal Tables



**Figure 217: Roadmap: Composition from Database Tables**



**Figure 218: Roadmap: Composition from Internal Tables**

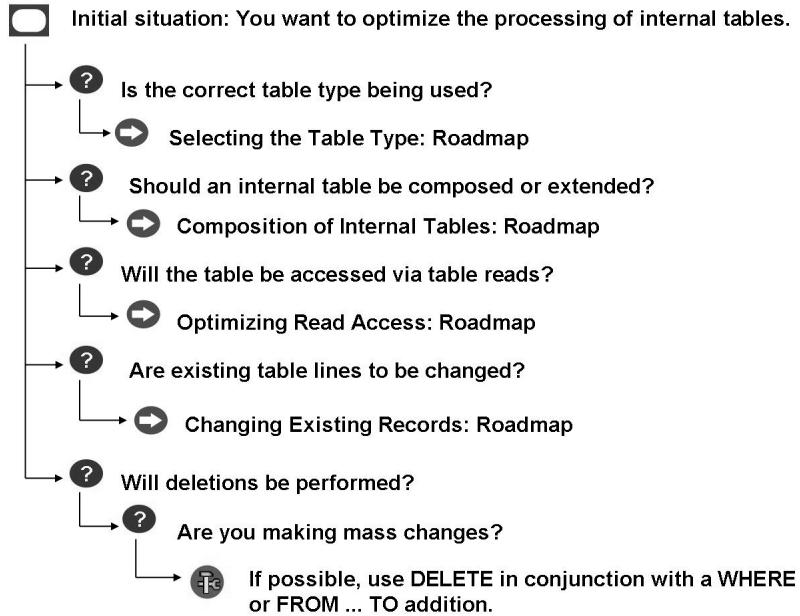


Figure 219: Roadmap: Optimizing Access for Internal Tables

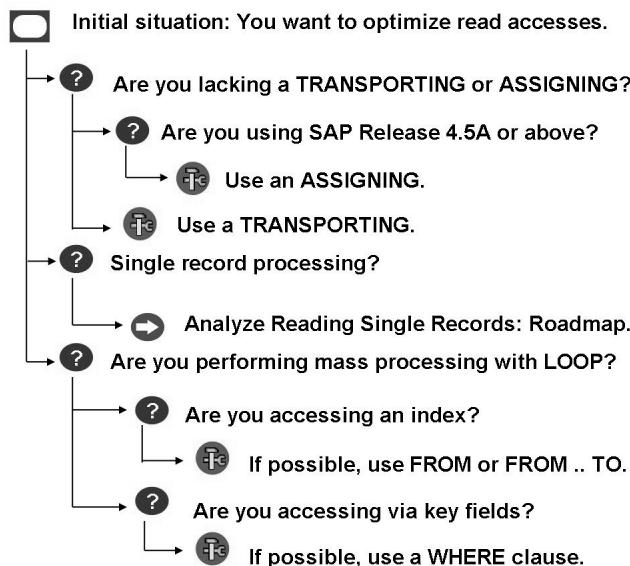
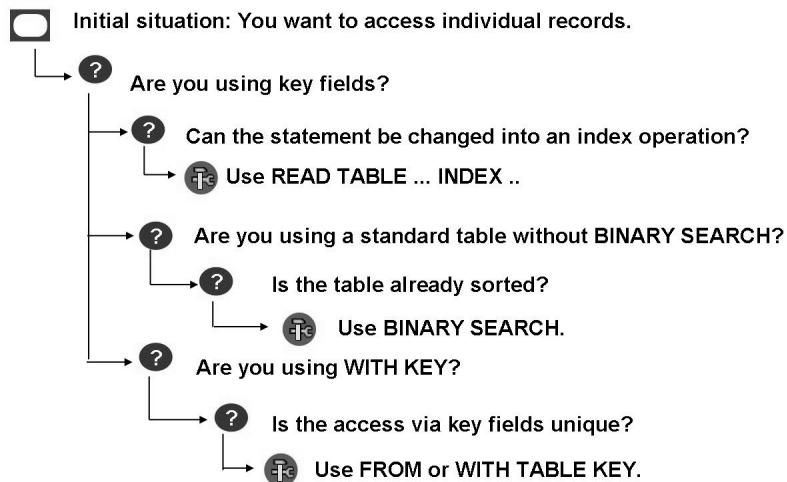
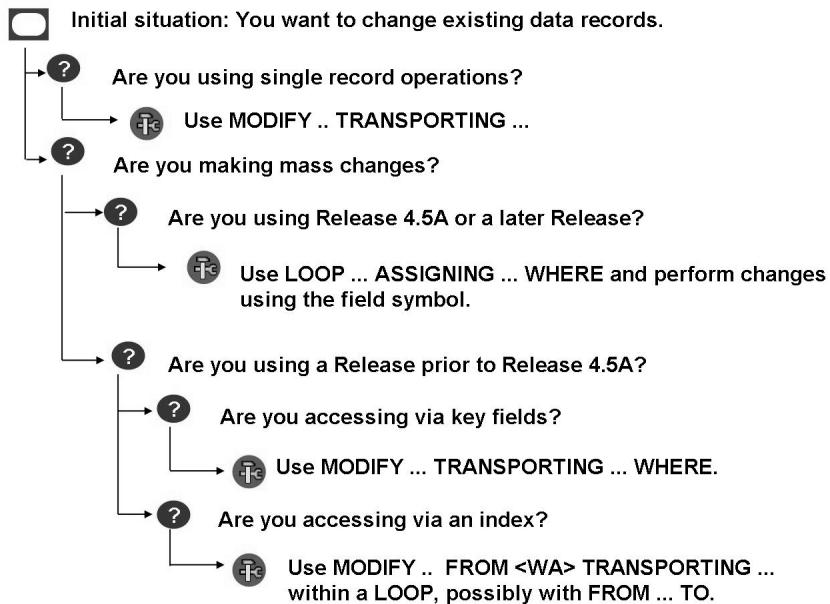


Figure 220: Roadmap: Optimizing Read Access



**Figure 221: Roadmap: Analyze Reading Single Records**



**Figure 222: Roadmap: Changing Existing Records**

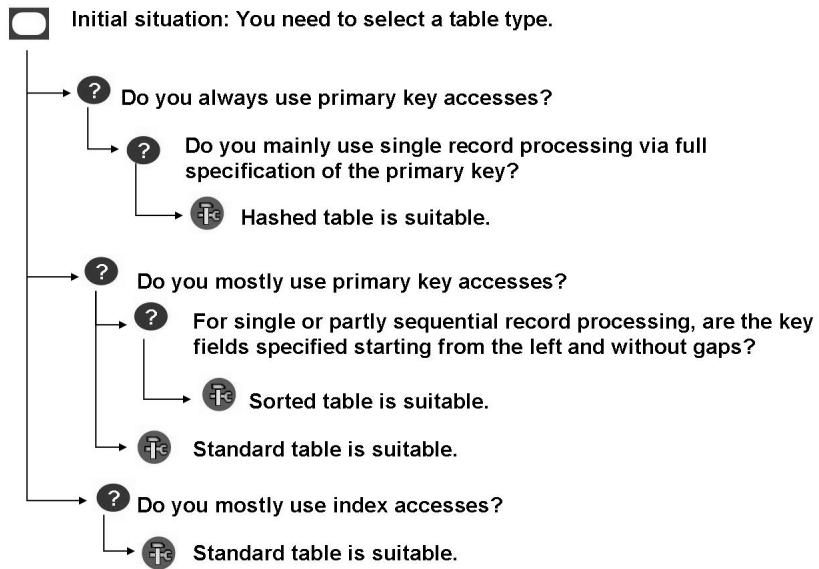


Figure 223: Roadmap: Selecting the Table Type



## Lesson Summary

You should now be able to:

- Explain all the important aspects of efficient programming



## Unit Summary

You should now be able to:

- Explain all the important aspects of efficient programming

# Unit 8

## Special Topics (Optional)

### Unit Overview

#### Content:



- Analysis and trace options for remote function calls and UI (user interface) programming



### Unit Objectives

After completing this unit, you will be able to:

- Analyze RFC accesses and UI techniques from performance aspects

### Unit Contents

Lesson: Special Topics (Optional) .....	312
---	-----

# Lesson: Special Topics (Optional)

## Lesson Overview

### Content:



- Analysis and trace options for remote function calls and UI (user interface) programming



## Lesson Objectives

After completing this lesson, you will be able to:

- Analyze RFC accesses and UI techniques from performance aspects

## Business Example

A company programs its own RFC modules and wants to optimize these for performance reasons. Furthermore, the various used UI techniques are supposed to be analyzed from a performance perspective.

## Performance for Remote Function Calls

 **Note:** Take into account that the previously explained rules for high-performance ABAP programming also apply in programming RFCs.

The focus of the following lesson is not on the programming of ABAP code but rather on the monitoring and analyzing of RFC connections and RFC calls. This subject therefore affects mostly administrators. Nonetheless, developers should also be familiar with the analysis options here.

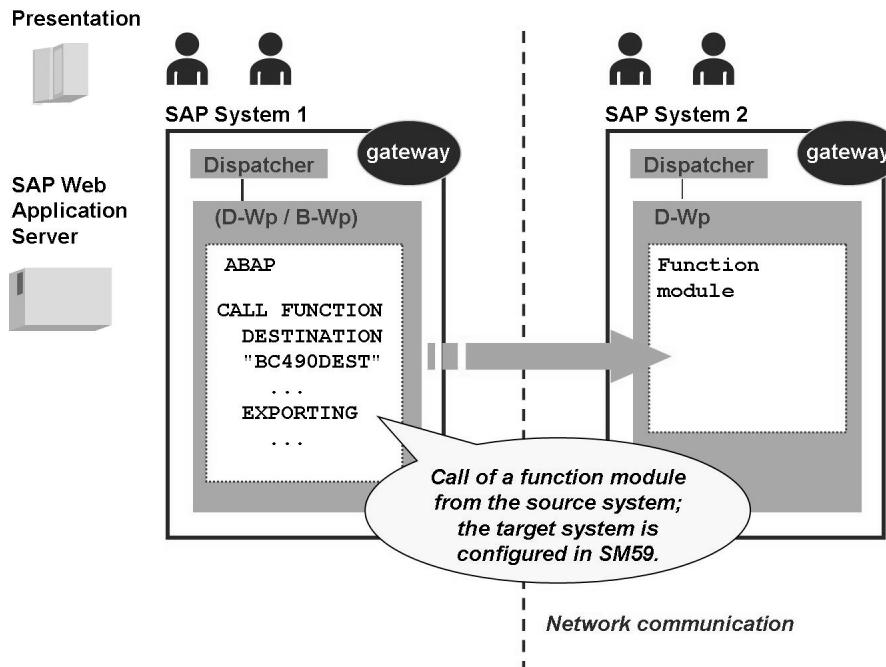


Remote Function Calls

UI Techniques

Figure 224: Special Topics - RFC

In today's business world, SAP systems are more and more connected to both SAP and non-SAP systems. There is a wide variety of software interfaces available. Remote function calls (RFCs) are a special technology for connecting SAP systems and external systems (via RFC libraries).



**Figure 225: The Remote Function Call (RFC)**

#### Uses for RFC calls:

- Communication between two SAP systems, or between an SAP system and a non-SAP system
- Communication between the SAP application layer and the SAP GUI
- Starting processes in parallel within one SAP system

According to their calling or execution mode, remote function calls can be separated into four types:

- Synchronous RFC (sRFC): for the communication between systems and the communication between SAP application servers and the SAP GUI
- Asynchronous RFC (aRFC): for the communication between systems and parallel processing.
- Transactional RFC (tRFC): for the transaction communication between systems (bundling of calls in the target system)
- Queued RFC (qRFC): a special type of tRFC for multiple communication steps in a specified order.



**Test Function Module: Result Screen**

Call of FM  
RFC\_READ\_TABLE via  
function builder  
SE37

Import parameters	Value
QUERY_TABLE	SFLIGHT
DELIMITER	
NO_DATA	
ROWSKIPS	0
ROWCOUNT	0
Tables	Value
OPTIONS	Result: 0 Entries
FIELDS	Result: 14 Entries
DATA	Result: 408 Entries

Result:

0 Entries	MA	0 Entries
800AA 001720070321	422.94 USD 747.400	385 375 194032.34
800AA 001720070418	422.94 USD 747.400	385 371 191181.92
800AA 001720070516	422.94 USD 747.400	385 374 194002.73
800AA 001720070613	422.94 USD 747.400	385 366 190724.98
800AA 001720070711	422.94 USD 747.400	385 373 194286.13
800AA 001720070808	422.94 USD 747.400	385 373 192137.68
800AA 001720070905	422.94 USD 747.400	385 373 194429.85
800AA 001720071003	422.94 USD 747.400	385 365 189189.61
800AA 001720071031	422.94 USD 747.400	385 365 190872.97
800AA 001720071128	422.94 USD 747.400	385 38 19565.22
800AA 001720071226	422.94 USD 747.400	385 136 69996.63
800AA 001720080123	422.94 USD 747.400	385 78 39862.11
800AA 001720080220	422.94 USD 747.400	385 103 52723.74
800AA 001720080319	422.94 USD 747.400	385 51 26488.73
800AA 001720080416	422.94 USD 747.400	385 30 15796.83
800AA 006420070323	422.94 USD A310-300	280 271 134254.01
800AA 006420070420	422.94 USD A310-300	280 263 128578.18
800AA 006420070518	422.94 USD A310-300	280 265 130718.28

Figure 226: Function Builder: Example of RFC\_READ\_TABLE



ST05: RFC trace of  
RFC\_READ\_TABLE

Duration	Obj. name	Dp.	Recs.	RC	Statement
147.692	t36tdc00_4	Client	4	0	twdf1902_DEV_00 t36tdc00_T36_50 C RFC_READ_TABLE 608 434.419

RFC Statement

Function module name: RFC\_READ\_TABLE  
 Source IP Address: 10.22.16.203  
 Source Server: twdf1902\_DEV\_00  
 Destination IP Address: 10.22.40.27  
 Destination Server: t36tdc00\_T36\_50  
 Client/Server: Client  
 Conversation ID: 17005513  
 RFC Trace Rec. Status: 4  
 Sent Bytes: 608  
 Received Bytes: 434.419

Total Sent Bytes: 608  
 Total Received Bytes: 434.419  
 ABAP program name: RFC\_READ\_TABLE=====  
 RFC Time: 147.686

Details of the trace record

Figure 227: ST05: RFC Trace

RFCs can also be analyzed via the SQL trace (ST05) and the system trace (ST01). Please note the checkmark in the initial screen of the transaction in ST05. If you double-click the RFC record, the details on the call such as the sent and received bytes are displayed. In the example, RFC\_READ\_TABLE was used to transfer the data of a table in the target system back to the source system (see entries for client-server and destination-server).

#### Used terms in the communication by RFC:

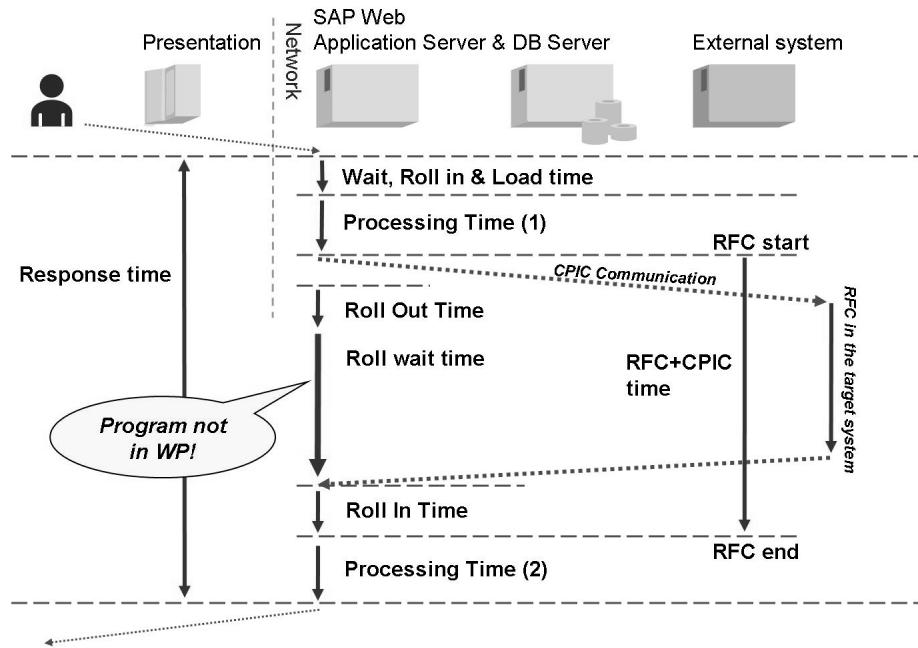
- RFC Client

The RFC client is the system that establishes the RFC communication. Possible alternative terms are client system or calling system. Because RFC communication is almost always initiated from within ABAP programs, the terms client program or calling program are used sometimes.

- RFC Server

The RFC server is the system that gets called by another system, using RFC. Possible alternative terms are server system or called system. Because RFC communication is almost always targeted at other (ABAP) programs, sometimes the terms server program or called program are used.

Dialog response time spent during synchronous RFCs is particularly relevant, because the client program waits for the response and proceeds with processing only after the called system returns data. During a synchronous RFC, the ABAP program in the calling system is rolled out by the work process (roll-out time mounts up). The time span during which the code is rolled out at the client is called **roll-wait time**.



**Figure 228: Roll Wait Time**

When data arrives from the server, roll-in at the client occurs and correspondingly produces roll-in time. RFC+CPIC time consists of the time needed for establishing the RFC communication, the time needed for roll-out, the roll wait time and the subsequent roll-in time.

During asynchronous RFC, no program roll-out occurs at the RFC client. Accordingly, RFC+CPIC time only comprises network time for establishing the connection to the RFC server. The calling ABAP program continues its work after starting the asynchronous RFC.



**Note:** Also see notes 552845 and 721170.

It documents many important parameters in connection with the RFCs in the statistics records

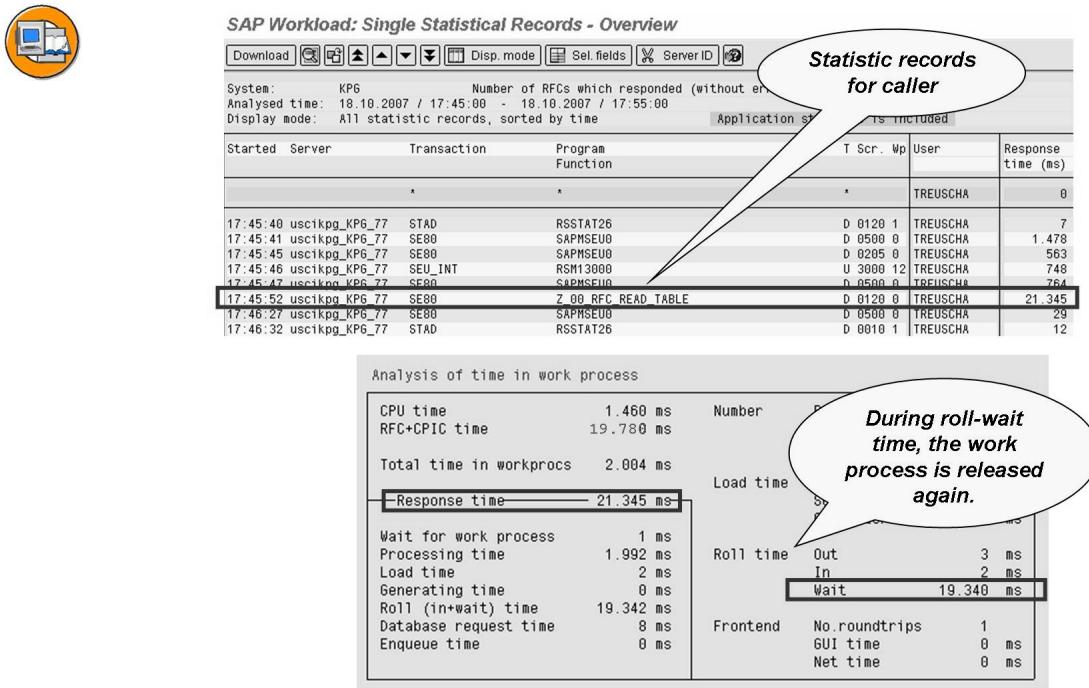


Figure 229: STAD: RFC Statistics Records

Transaction **STAD** allows access to individual statistical records. When displaying the detailed view of a single RFC record, important RFC-related data is shown in the Time and RFC sub-records. In particular, RFC+CPIC time cumulates time spent in RFC calls, including network time for establishing the communication between local and remote destinations. Large RFC+CPIC time is often connected with significant roll-wait time due to roll-outs.

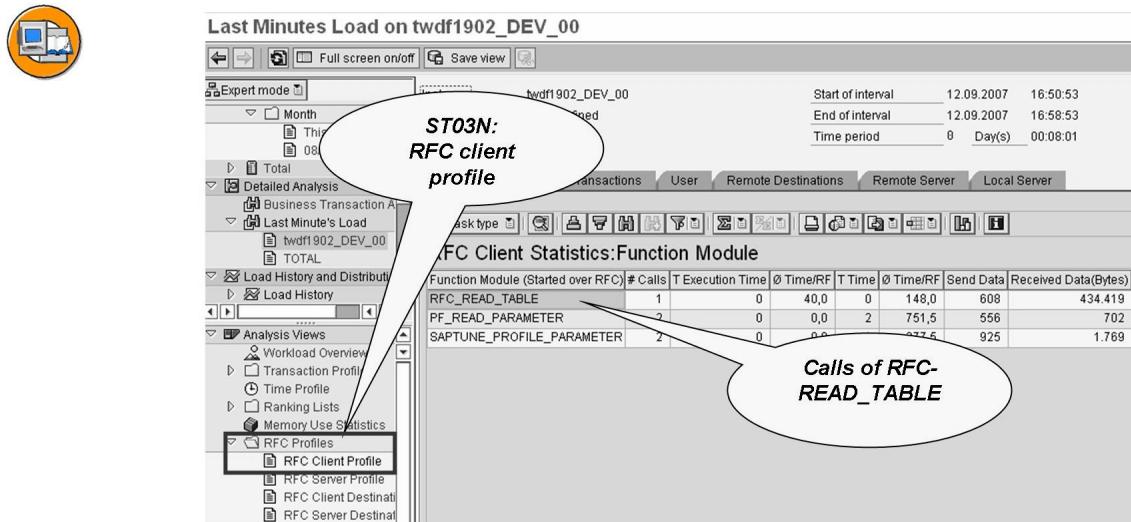


**Hint: If communication between systems is fast, both time intervals are very similar. If RFC+CPIC time exceeds roll-wait time by a large amount, then a communication problem exists.** The problem might be due to a slow network connection or a lack of free work processes at the server.

The RFC communication is also used for communication with the front-end (GUI round trips). Hence, long roll-wait time can also be caused by expensive front-end communication with too many GUI round trips. A GUI time that is too long is then observed.

In a dialog step with a single synchronous RFC, RFC+CPIC time should be always greater than roll-wait time. The sender sees the hourglass during the entire RFC+CPIC time period.

In asynchronous calls, the sending work process is free as soon as all function input data is transmitted and the transmission is confirmed by the receiver. For asynchronous RFCS, the RFC+CPIC time only shows the time for setting up the connection to the receiver and starting the RFC, but not the runtime of the RFC. The statistical record shows RFC+CPIC time but no roll-wait time.



**Figure 230: ST03N: RFC Client Profile**



**Hint:** In STAD, note the “Last ten minutes” function, if you want to analyze the programs or RFC that were executed recently. Otherwise, you can only see actions that are about one hour in the past.

The RFC client profile and client destination profile of transaction ST03N return an aggregation of individual RFC client sub-records for each function module called. Here you will find, for example, information on function modules called, calling and execution time, bytes transferred, destinations, and users. The programs or transactions that trigger RFCS are also displayed (in the client destination profile).

The RFC server profile is an aggregation of individual RFC server sub-records for each function module called. Here you will find, for example, information on function modules called, calling and execution time, and bytes transferred.

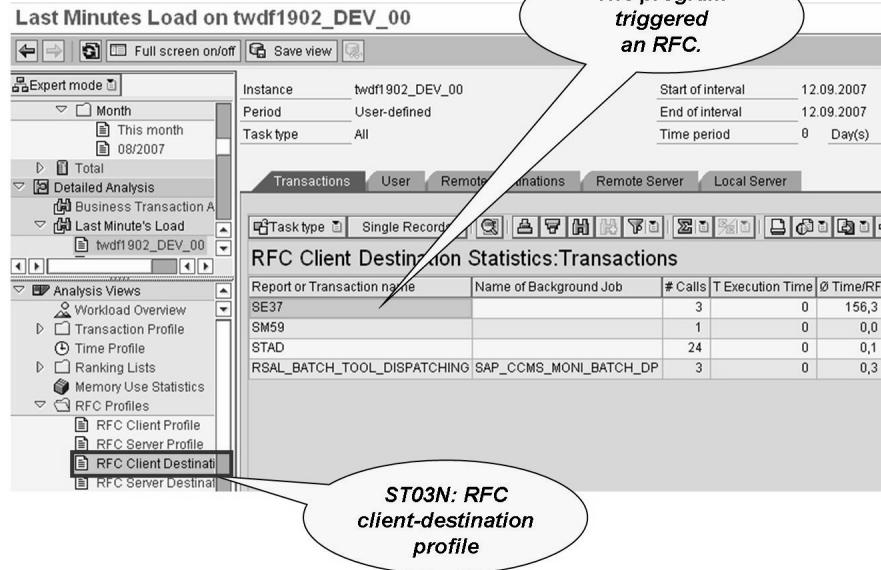


Figure 231: RFC Client Destination Profile

If you navigated to the RFC statistics records from the Workload Analysis ST03N (via Single Records button) you get additional details for the RFC connection (see above graphic)

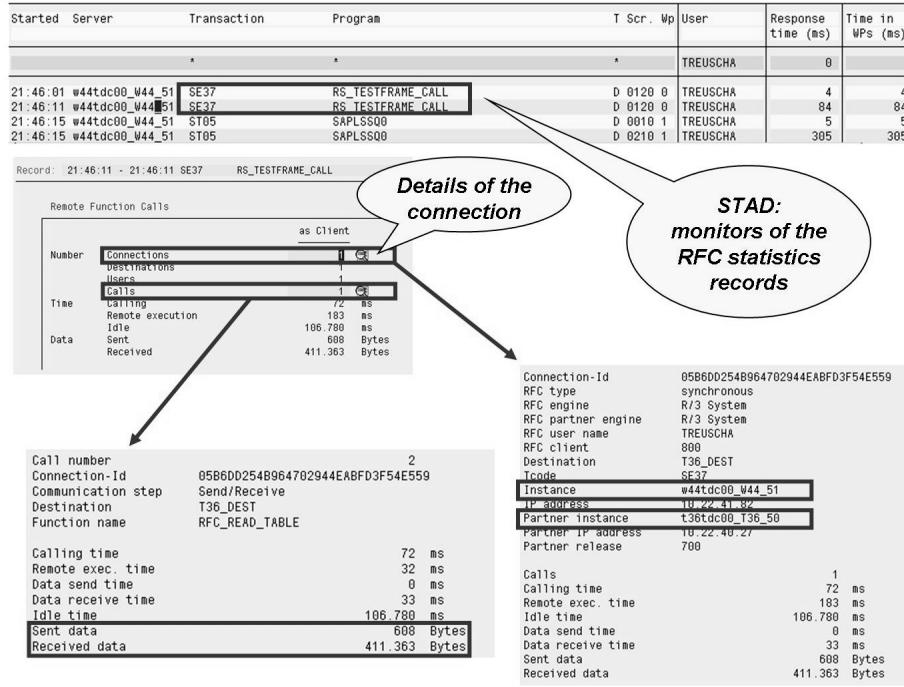


Figure 232: STAD: RFC Statistics Records (2)

From Release 4.6, roll wait times also occurs in the communication of the SAP system with Enjoy controls on the front-end. While the data is processed in the front-end, the user context is rolled out and the work process is thus released. This can happen several times during a transaction step. If Office applications such as MS Word are started on the front-end and only closed again after a long time (>minutes), a long roll wait time also occurs here.

## Performance in Programming User Interfaces



### Remote Function Calls



### UI Techniques

Figure 233: Special Topics - UI Techniques

UI techniques refer to the programming of user interfaces. Depending on the release used, SAP offers several options for implementing interaction with the user:

- Classic dynpro
- SAP Enjoy Controls (from Release R/3 4.6)
- IACs or Flow-Logic via ITS (from Release R/3 4.6)
- Business Server Pages (from Release SAP Web AS 6.10)
- ABAP Web Dynpro (from SAP NW AS 7.0)

In the design of UIs you have to pay close attentions of the good structuring and reusability of the individual components.

From the performance aspect, all principles for high-performance programming in ABAP that have been listed and discussed in course still apply, of course.

As previously introduced, this also applies for RFCs:

- Read only the data that is required
- Do not read the same data several times
- Bundled reading of data, that is, mass accesses are generally faster (only if the structure of the application allows this)
- Use buffering techniques
- and so on

In these performance analyses it therefore does not matter whether the accesses for controls, BSP or web dynpro programming are encapsulated in a method or whether modules are still used via dynpros. Of course, this does not affect a database access, that is, using SELECT. The listed principles of high performance ABAP programming apply to all UI techniques.

The following explanations are supposed to help with analyzing and tracing the programming of UIs.

### SAP Enjoy Controls Trigger RFC-Calls

With SAP Rel. 4.6 the SAPGUI Controls (Tree-Control, ALV Grid Control, Text Edit Control, HTML Control) were introduced. In parts, these controls take over tasks that used to be processed by the application server (such as line and page scrolling in lists, navigation in a tree and so on) and make functions available that used to have to be implemented separately by each application (sorting, sum formation in columns, and so on). At the same time, they offer totally new options of data presentation and enable the transactions to be redesigned in a user-friendly manner. Some transactions that use the controls intensively now require fewer dialog steps for the transaction than they used to. Of course, comparisons are difficult due to the significant influence of

Customizing and user responses. Furthermore, more "one screen transactions" are offered in Release 4.6. If the business operation has not changed, this results in an increased average response time per dialog step, naturally.



```

REPORT z_00_alv.

DATA: r_alv TYPE REF TO cl_salv_table,
      itab TYPE STANDARD TABLE OF sflight.

SELECT * FROM sflight INTO TABLE itab.

TRY.
  CALL METHOD cl_salv_table=>factory
    IMPORTING
      r_salv_table = r_alv
    CHANGING
      t_table      = itab.
  CATCH cx_salv_msg .
ENDTRY.

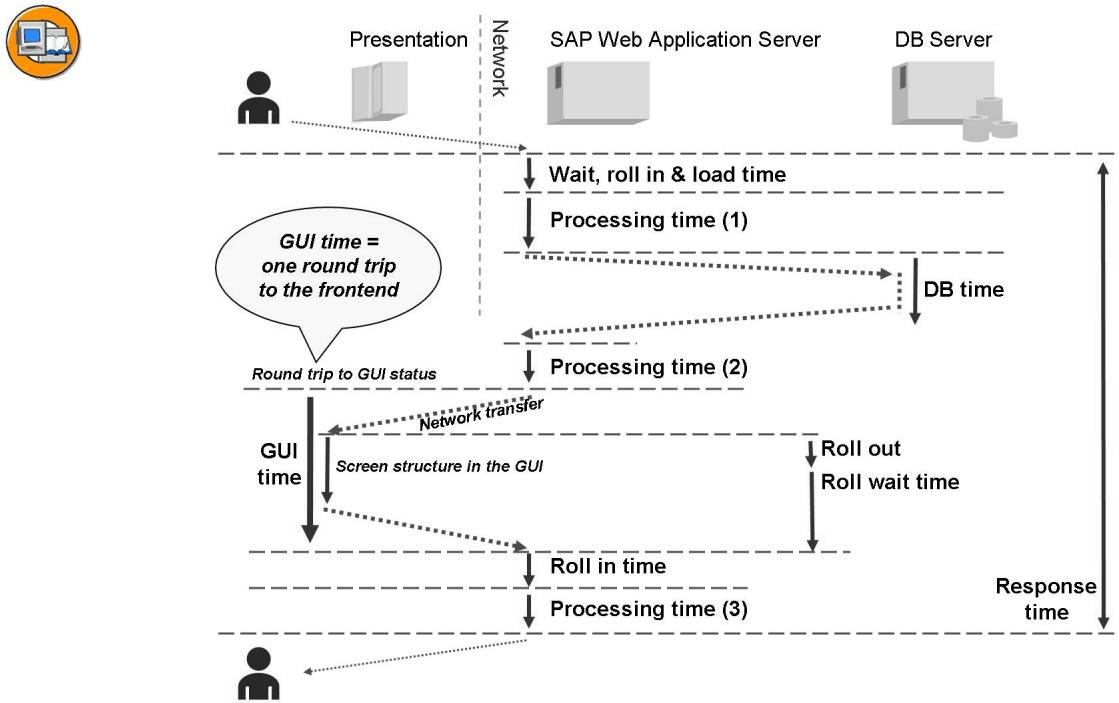
k_alv->display( ).
```

Ma	Air	Flu	Flugdatum	Flugpreis	Währ	Flugzeugtyp	Kapazität	Belegt	Akt. Buchungssumme	Kapazität
900 AA	17	03.01.2007	422,94 USD	747-400	385	373		193.389,50	31	
900 AA	17	31.01.2007	422,94 USD	747-400	385	367		190.949,16	31	
900 AA	17	28.02.2007	422,94 USD	747-400	385	364		188.787,88	31	
900 AA	17	28.03.2007	422,94 USD	747-400	385	370		192.848,05	31	
900 AA	17	25.04.2007	422,94 USD	747-400	385	365		190.555,91	31	
900 AA	17	23.05.2007	422,94 USD	747-400	385	368		190.885,77	31	
900 AA	17	20.06.2007	422,94 USD	747-400	385	373		194.032,43	31	
900 AA	17	18.07.2007	422,94 USD	747-400	385	373		191.938,92	31	
900 AA	17	15.08.2007	422,94 USD	747-400	385	370		192.890,45	31	
900 AA	17	12.09.2007	422,94 USD	747-400	385	107		54.952,80	31	
900 AA	17	10.10.2007	422,94 USD	747-400	385	166		88.368,88	31	
900 AA	17	07.11.2007	422,94 USD	747-400	385	91		46.954,89	31	
900 AA	17	05.12.2007	422,94 USD	747-400	385	73		37.654,40	31	
900 AA	17	02.01.2008	422,94 USD	747-400	385	76		39.337,70	31	
900 AA	17	30.01.2008	422,94 USD	747-400	385	15		7.790,59	31	

Figure 234: Example of an ALV Grid

**Response time and GUI time:** The application server sends information in chunks to the front-end PC during “round trips”. The time spent for a round trip is called GUI time; this time is measured at the application server. GUI time comprises time spans for network communication and the time needed to generate the screen at the front end. During GUI time, roll-wait time amounts at the application server (including complementary roll-out and roll-in phases).

Roll-wait time due to round-trips is just listed as roll-wait time, combined with roll-wait time due to other reasons, in the workload analysis. Hence, GUI time can be smaller or larger than the reported roll-wait time for aggregated values. In general, there is no numerical identity of roll-wait and GUI time by definition, although sometimes observable. If so, this is just by sheer coincidence.



**Figure 235: GUI Time**

Process: Time measurement in the SAP system starts when the request from the front-end is received by the application server. It ends when the last data package leaves the application server for the front-end.

During the transaction step, several communication steps can take place between application server and front-end.

During a round trip, the application server transfers data to the controls of the front-end. The user context is usually rolled out in doing so and this time is measured as roll wait time.

The entire time for all round trips is measured in GUI time in the application server. This GUI time can be used to determine the influence of network and front-end performance on the SAP response time. The SAPGUI also performs a time measurement.

From the sending of a request to the application server until the receipt of a response in the front-end, the time is measured by SAPGUI and transferred to the application server with the next request. There, the difference between this time and the response

time in the SAP system is displayed as the front-end network time, that is, the total of the network time for the first and the last communication step between the front-end and application server. The front-end network time is not included in the GUI time.

- **Note:** For details on response time with GUI time or without GUI time, see SAP Note 376148:

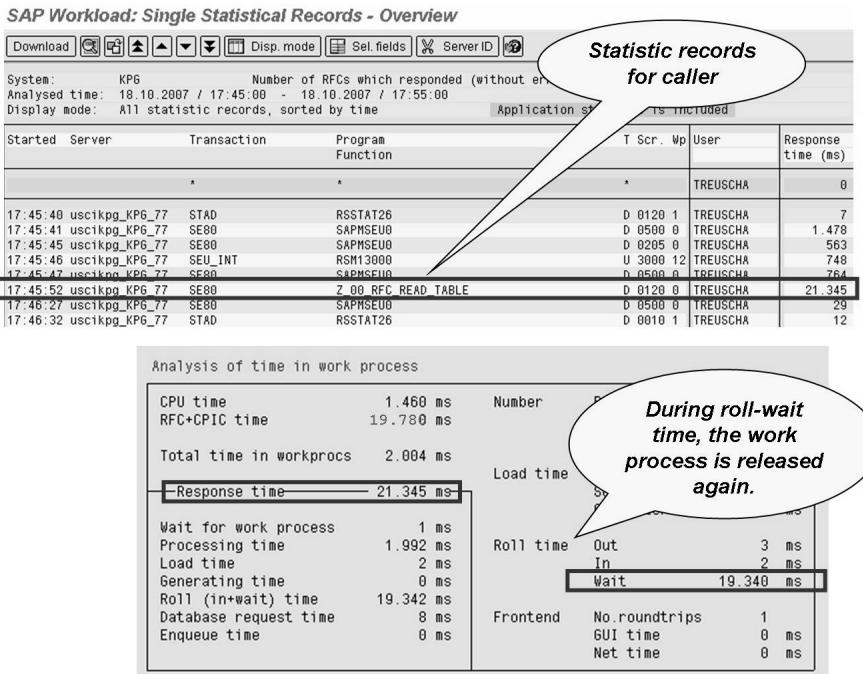


Figure 236: STAD: Statistical Records for Calling the Control

- **Note:** Most of the communication between the application server and the front-end is included in the response time from Release 4.6. About 100 - 200 ms of GUI time per dialog step are typical.

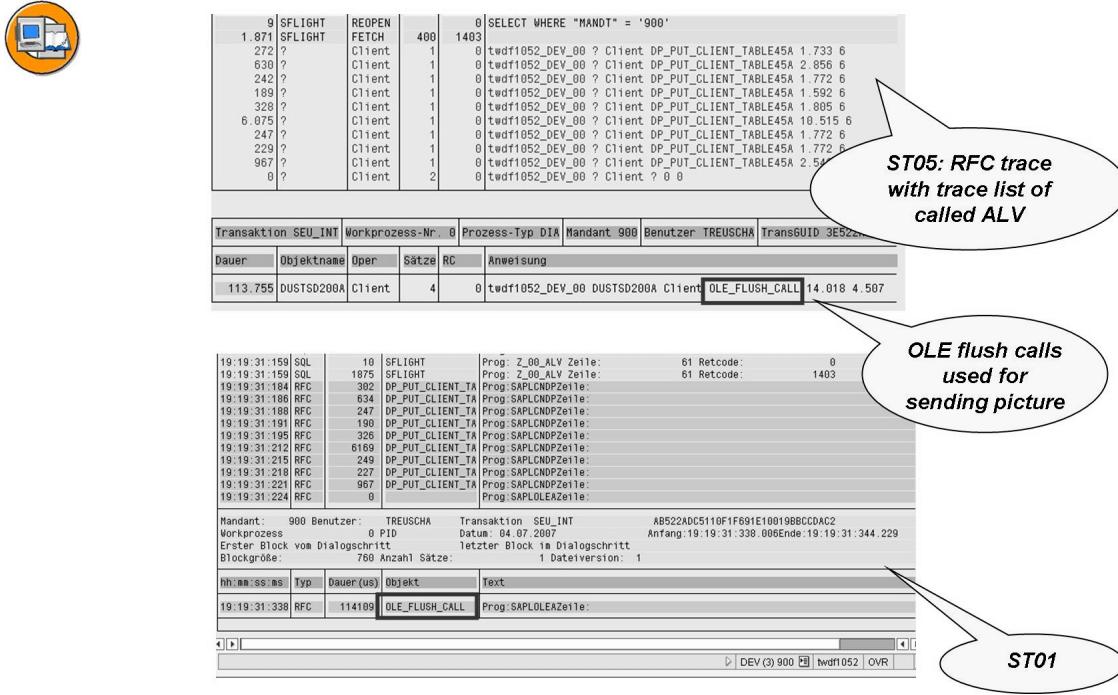


Figure 237: ST05 - ST01 Trace List for Calling up the ALV Grid

An RFC statement referring to **OLE\_FLUSH\_CALL** indicates a round trip to the front-end PC. If more than 32KB have to be transferred, function module **GUICORE\_BLOB\_DIAG\_PARSER** is executed.

→ **Note:** On average, there should be only one round trip per dialog step. Two are acceptable for “normal” screens. During a round trip, a maximum of 32 KB of data can be transferred. The average network transfer is between 5 and 8 KB. For a more complex application, the round trips for a screen should not be more than 2-10 Kb (that is, per dialog step).

### Tips for Analyzing Additional UI Techniques

The following explanations are supposed to help with analyzing and tracing the programming of BSPs.



*Setting an external break point*

*Example of a BSP application*

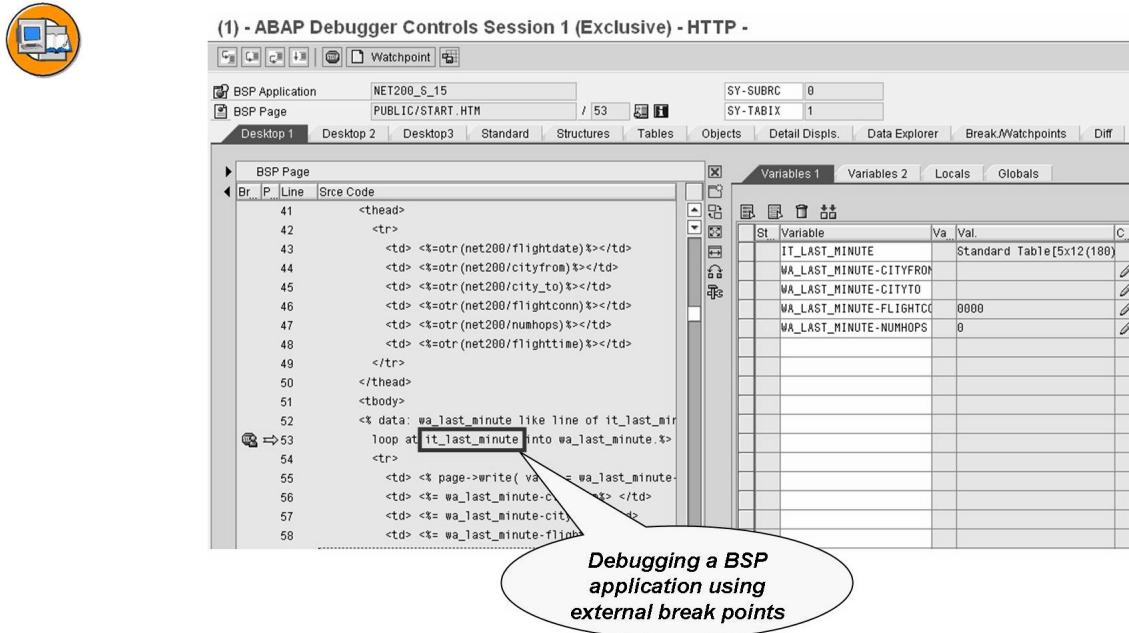
```

39   <table>
40     <thead>
41       <tr>
42         <td> <#= otr(net200/flightdate) %></td>
43         <td> <#= otr(net200/cityfrom) %></td>
44         <td> <#= otr(net200/cityto) %></td>
45         <td> <#= otr(net200/flightconn) %></td>
46         <td> <#= otr(net200/numhops) %></td>
47         <td> <#= otr(net200/flighttime) %></td>
48     </tr>
49   </thead>
50   <tbody>
51     <tr>
52       <td colspan="6"><%: wa_last_minute like line of it last minute.  
loop at it last minute into wa_last_minute.%>

```

Figure 238: External Debugging of a BSP Application (1)

As already mentioned: when analyzing the performance, it does not matter whether accesses are encapsulated or, as in the example of a BSP application here, are within “tags” in the HTML coding. This does not affect a database access using SELECT for example. The LOOP statement programmed in the example is also subject to the usual performance rules for programming internal tables.



**Figure 239: External Debugging of a BSP Application (2)**

The application could also be analyzed by means of the runtime analysis and the SQL trace.

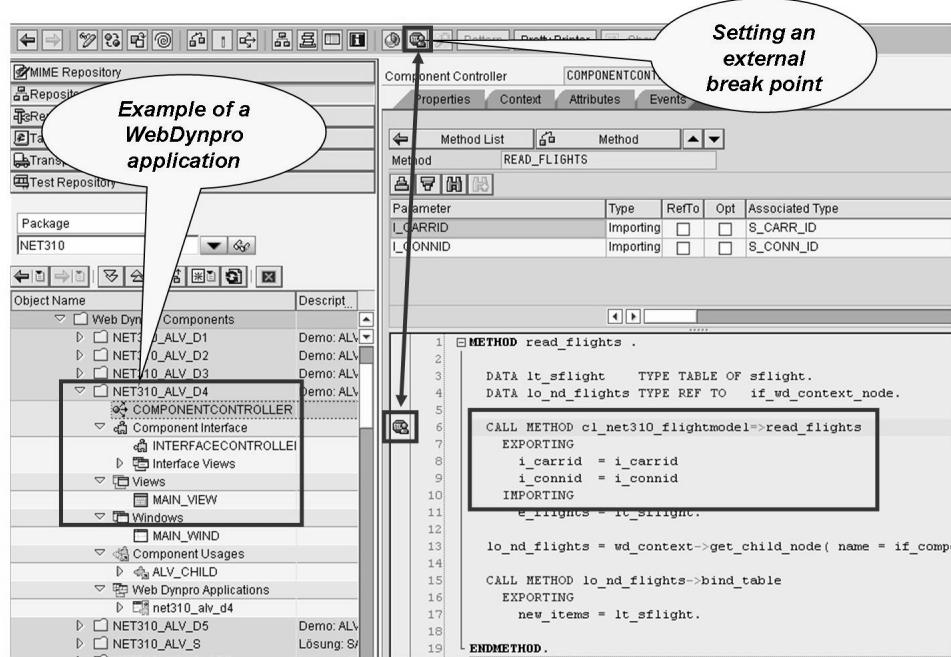
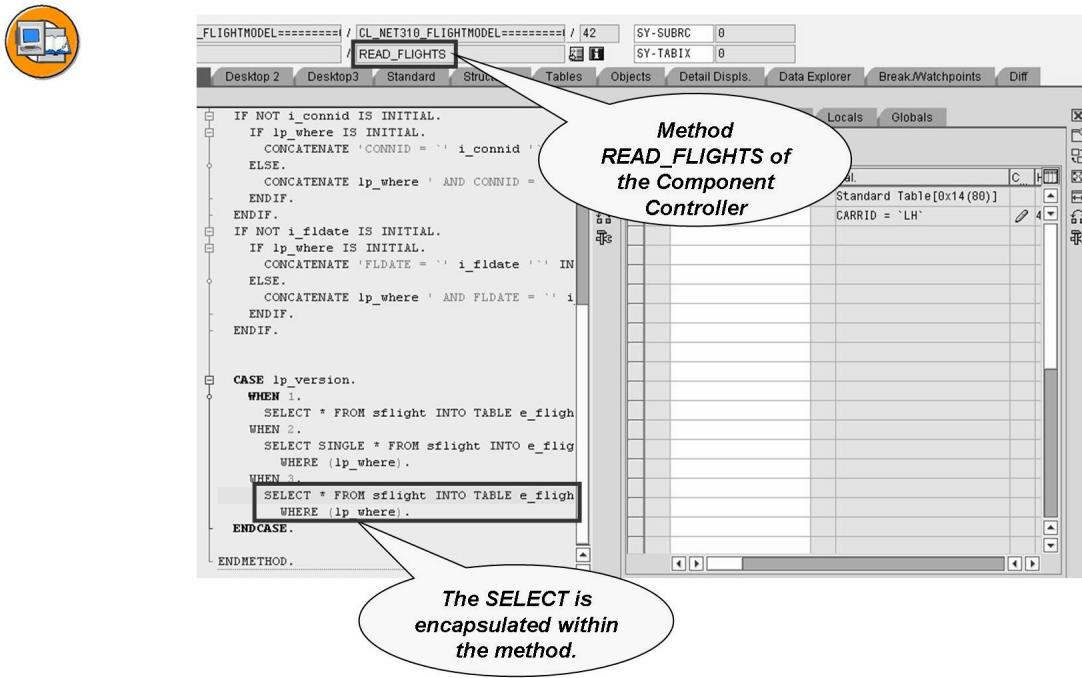


Figure 240: External Debugging of a Web Dynpro Application (1)

Encapsulated within method READ\_FLIGHTS, there is a SELECT access to database table SFLIGHT in the example. To evaluate the performance of this select, all rules of high-performance programming that have been mentioned in the course apply.



**Figure 241: External Debugging of a Web Dynpro Application (2)**

Evaluate the potential for optimization of the SELECT call in the Web Dynpro example.

## Traces for Browser-Based UIs

Besides the option of setting external break points in order to analyze the processing of the ABAP coding, you can still use various trace techniques. But this cannot be used to analyze the involved ABAP coding, this is rather about the transferred HTTP requests and a monitoring from the administrator perspective.

- Monitoring in CCMS (Web Dynpro) - displayed attributes: ApplicationCount, SessionCount, Roundtrips, CpuTime, Data
- Trace toll (Web Dynpro): supports the support with the analysis of problem cases and error causes.
- ICM tracing: tracing the data streams to and from the Web Application Server. The ICM is the component that regulates HTTP data traffic. The ICM is managed via transaction SMICM
- HTTP browser tracing as an alternative to ICM tracing
- SICF trace: different functions for monitoring, analysis and error handling in the ICF environment (transaction SICF is used for managing and administering HTTP based services).

 **Note:** For further information, refer to the *help.sap.com* → SAP NW .. → SAP NW Library → Application Platform by Key Capabilities → ABAP Technology → UI Technology → Web UI Technology → Web Dynpro ABAP. Similar functions for BSP are available under ... → UI Technology → BSP.



## Lesson Summary

You should now be able to:

- Analyze RFC accesses and UI techniques from performance aspects



## Unit Summary

You should now be able to:

- Analyze RFC accesses and UI techniques from performance aspects



## Course Summary

You should now be able to:

- Use SAP tools for analyzing ABAP performance
- Systematically analyze ABAP performance problems
- Solve ABAP performance problems that cause high database loads (database accesses) or high CPU loads (accesses of internal tables), and analyze performance problems in special areas



# Index

## A

Analyze, 44

## D

DBSS, 38

## G

Gross time, 62

## I

index search string, 106

INNER JOIN, 173

## J

JOINs, 171

## M

memory

ABAP memory, 233

SAP memory, 235

Memory, 232

## N

Net time, 62

## O

OUTER JOIN, 173



# Feedback

SAP AG has made every effort in the preparation of this course to ensure the accuracy and completeness of the materials. If you have any corrections or suggestions for improvement, please record them in the appropriate place in the course evaluation.