

English



FUJITSU Software BS2000

SESAM/SQL-Server V9.0

Performance

User Guide

Comments... Suggestions... Corrections...

The User Documentation Department would like to know your opinion on this manual. Your feedback helps us to optimize our documentation to suit your individual needs.

Feel free to send us your comments by e-mail to:

manuals@ts.fujitsu.com

Certified documentation according to DIN EN ISO 9001:2008

To ensure a consistently high quality standard and user-friendliness, this documentation was created to meet the regulations of a quality management system which complies with the requirements of the standard DIN EN ISO 9001:2008.

cognitas. Gesellschaft für Technik-Dokumentation mbH

www.cognitas.de

Copyright and Trademarks

Copyright © 2016 Fujitsu Technology Solutions GmbH.

All rights reserved. Delivery subject to availability; right of technical modifications reserved.

All hardware and software names used are trademarks of their respective manufacturers.

Content

1	Preface	9
1.1	Objectives and target groups of this manual	9
1.2	Summary of contents	10
1.3	Notational conventions	11
2	Tools for analyzing performance	13
2.1	Measurement and analysis tools	14
2.2	SESMON	15
2.3	SQL optimizer	17
2.4	SESCOS and SESCOSP	19
2.4.1	Differentiation from other performance analysis tools	19
2.4.2	Recommended approach for analysis using SESCOS and SESCOSP	20
2.5	View SYS_DML_RESOURCES	21
3	Application aspects relevant to performance	23
3.1	Structure and control of the optimizer	24
3.1.1	Plans	24
3.1.2	Phases in the creation of a plan	24
3.1.2.1	Phase 1: Front-end	25
3.1.2.2	Phase 2: Algebraic optimization	25
3.1.2.3	Phase 3: Selecting the access path	30
3.1.3	Overview of the effect of pragmas relevant to optimization	39
3.1.4	Annotations	42
3.1.4.1	JOIN annotation	42
3.1.4.2	CACHE annotation	43
3.1.4.3	IMMUTABLE annotation	44

3.2	Synchronization	45
3.2.1	Phenomena and isolation level	45
3.2.2	Synchronization of DDL and DML	48
3.2.3	Synchronization of utilities	49
3.3	Resource consumption	51
3.3.1	Request-specific space requirements in the communication buffer	52
3.3.2	Conversation-specific space requirements in the VGM	53
3.3.3	Space requirements in the prefetch buffer	54
3.4	Use of routines	55
4	Aspects of database design relevant to performance	57
4.1	Spaces with a size of more than 64 GB	57
4.2	Primary data	59
4.3	Metadata	61
4.4	Integrity constraints	62
4.5	Data distribution and allocation	65
4.6	Partitioned tables	67
5	Aspects of database operation relevant to performance	69
5.1	Diagnosis and corrective measures for performance problems	70
5.2	Work container	72
5.3	Transfer container	72
5.4	User and system data buffer	73
5.4.1	Dimensioning the user data buffer	74
5.4.2	Dimensioning the system data buffer	75
5.4.3	Example for the calculation of buffer sizes	76
5.5	DBH option RESTART-CONTROL	77
5.6	Cursor buffer	78
5.7	Number of threads	79
5.8	Number of DBH tasks and multitasking	80
5.9	Suborders	81

5.10	Plan buffer	82
5.10.1	Assignment of plans to SQL statements	82
5.10.2	Life-span of plans	84
5.10.3	Number of plans	84
5.10.4	Size of the plan buffer	86
5.10.5	Optimizing the size of the plan buffer	87
5.10.6	Estimating the minimum number of plans in a sample application	89
5.11	Prefetch buffer for block mode	92
5.11.1	Size of the prefetch buffer	92
5.11.2	Optimizing the size of the prefetch buffer	94
5.12	Service tasks	97
5.12.1	Influencing the service task strategy	97
5.12.2	Influencing sorting	98
5.12.3	Storage requirements for sort work files when constructing indexes	101
5.12.4	Summary of performance-enhancing measures	102
5.12.5	RECOVER/REFRESH options	103
5.13	Reorganization criteria	104
5.13.1	Physical DB structure and changes to it during operation	104
5.13.2	Indicators of the necessity of a reorganization	104
5.13.3	Space statistic from SESDIAG	105
5.13.4	Reducing the number of extents of a BS2000 file	106
5.13.5	Reorganization of a space together with reassignment of row numbers	107
5.14	Distributed processing	108
5.14.1	Dimensioning of the SESDCN memory pool	108
5.14.2	Starting multiple SESDCNs	109
5.15	Analysis of lock conflicts	110
6	Specific notes on tuning applications	113
6.1	Optimization options for SQL applications	114
6.1.1	General programming recommendations	114
6.1.2	Tuning SQL applications	118
6.1.2.1	Basic approach	118
6.1.2.2	Typical performance problems	119
6.2	Optimization options for CALL DML	128
6.2.1	Economic use of communication path lengths	128
6.2.2	Economic use of path lengths for syntax analysis	128
6.2.3	General programming recommendations	129
6.2.4	Acceleration of join processing for CALL DML	130
6.2.5	Order of processing queries for CALL DML	130

7	Explain component of the optimizer	131
7.1	Introduction	132
7.1.1	Terminology	132
7.1.2	Introductory example	133
7.2	Table-value operations (RELATION)	140
7.2.1	cast_rows_to_rel	140
7.2.2	cross	142
7.2.3	cursor_scan	143
7.2.4	Empty	144
7.2.5	eproject	145
7.2.6	full_outer_join	146
7.2.7	full_outer_mjoin	148
7.2.8	left_outer_join	150
7.2.9	left_outer_mjoin	151
7.2.10	mjoin	153
7.2.11	njoin	155
7.2.12	One	157
7.2.13	pre_rest	158
7.2.14	rest	159
7.2.15	sort	161
7.2.16	sorted_group	162
7.2.17	store_temp	164
7.2.18	table_function_scan	166
7.2.19	table_scan	167
7.2.20	union	169
7.2.21	virtual_scan	171
7.3	DML statements	172
7.3.1	UPDATE statement	172
7.3.2	INSERT statement	174
7.3.3	MERGE statement	177
7.3.4	DELETE statement	180
7.3.5	DECLARE CURSOR statement	181
7.3.6	Single SELECT statement	182
7.3.7	Dynamic SELECT statement	183
7.3.8	Internal EXPORT DATA statement	184
7.3.9	Internal SELECT FOR UNLOAD statement	186
7.3.10	CALL statement	188
7.4	Quantifiers	192
7.5	Value queries	194

7.6	Expressions, function calls, and predicates	195
7.7	Intermediate files	203
8	Performance-related aspects of utility functions and DDL statements.	205
8.1	LOAD - Load user data into a base table	205
8.1.1	Processing LOAD in SESAM/SQL	206
8.1.2	Performance-related aspects of LOAD	208
8.1.3	Advice on using LOAD	209
8.1.4	Continuing an aborted LOAD statement	211
8.2	UNLOAD - Unload user data from a table	213
8.3	Creating SESAM backup copies (COPY) and foreign copies	215
8.3.1	Processing the COPY statement in SESAM/SQL	215
8.3.2	Backup to disk	216
8.3.3	Backup with ARCHIVE	216
8.3.4	Backup with HSMS	217
8.3.5	Backup with a foreign copy	218
8.3.6	Criteria for selecting backup procedures	219
8.4	RECOVER - Repairing and resetting	220
8.4.1	Shortening read in time	220
8.4.2	Speeding up the application of the modifications logged in the log files	225
8.4.3	Repairing index spaces	227
8.5	REORG - Reorganizing spaces and base tables	228
8.6	Optimized index creation for partitioned tables	229
8.7	DROP TABLE DEFERRED / DROP INDEX DEFERRED	230
8.8	ALTER TABLE ... ADD COLUMN with index definition	231
8.9	Media definition for DDL-TA-LOG files	231

9	Performance-related aspects of administration statements	233
9.1	RECONFIGURE-DBH-SESSION and RELOAD-DBH-SESSION	233
9.2	SET-SQL-DB-CATALOG-STATUS (STATUS=FREE)	234
9.3	STOP-DBH	234
	Related publications	235
	Index	237

1 Preface

The functions and architectural features of the SESAM/SQL-Server database system meet all the demands placed on a powerful database server in today's world. These characteristics are reflected in its name: SESAM/SQL-Server.

SESAM/SQL-Server is available in a standard edition for single-task operation and in an enterprise edition for multitask operation.

For the sake of simplicity, we shall use the name SESAM/SQL throughout this manual to refer to SESAM/SQL-Server.

The following introductory descriptions are contained centrally in the [“Core manual”](#):

- Brief product description
- Structure of the SESAM/SQL server documentation
- Demonstration database
- Readme file
- Changes since the last editions of the manuals

1.1 Objectives and target groups of this manual

This manual is intended for experienced users of SESAM/SQL.

It describes how the user can identify performance bottlenecks in the operation of SESAM/SQL and the measures with which system behavior can be influenced.

1.2 Summary of contents

This manual contains specific information on how users can influence the performance of SESAM/SQL:

- How can the user detect the presence of a performance problem and how can the problem be analyzed?
- Which features of the overall SESAM/SQL system can be modified?
- How can the user eliminate performance problems?

The following topics, in particular, are dealt with in detail:




- The measurement and analytical tools available for tuning and the analysis of performance bottlenecks are discussed in chapter 2, which follows this preface.
- Chapter 3 describes the application-specific aspects that are relevant to performance: This includes the approach of the optimizer when generating SQL access plans, synchronization mechanisms, and the utilization of resources.
A thorough understanding of the approach used by the optimizer is crucial to all users who intend fine-tuning their applications with the aid of pragmas and additional indexes.
- Chapter 4 deals with performance-related aspects that can be considered early in the database design, i.e. questions such as: How are disk storage requirements for primary, secondary and metadata determined? What impact do testing for integrity constraints and the distribution of data have on performance?
- Chapter 5 discusses the performance-related aspects of database operation.
- Chapter 6 contains specific notes on tuning SQL and CALL DML applications.
- Chapter 7 describes the output of the Explain component of the optimizer.
- Chapter 8 discusses the performance-related aspects of utility functions and DDL statements.
- Chapter 9 discusses the performance-related aspects of administration statements.



The interfaces, processing sequences, and evaluations described in this manual are based on the current version of SESAM/SQL-Server and are subject to changes in subsequent versions.

1.3 Notational conventions

The following notational conventions are used in this manual:

UPPERCASE	SQL keywords
<u>underscored</u>	Default values
bold	Used for emphasis in running text
<i>italics</i>	Parameters in syntax definitions and running text
Fixed-space font	Program text in examples
,...	In syntax definitions, a comma followed by three dots means that you can repeat the preceding specification any number of times, separating each specification with a comma. If you do not repeat a specification, you must omit the comma.
...	In syntax definitions, an ellipsis means that you can repeat the preceding specification any number of times. In examples, the ellipsis means that the rest of the statement is of no significance to the example. The ellipsis is a metacharacter and must not be entered in an SQL statement.
⇒	The expressions or operations on the right-hand side of the sign or above the sign result in the same result table as the expressions or operations on the left-hand side of the sign or below the sign .
	Indicates notes that are of particular importance.
	Indicates warnings.
	The notation used by the Explain component of the optimizer is described separately in the chapter “Explain component of the optimizer” on page 131 .

2 Tools for analyzing performance

This chapter describes:

- the measurement and analytical tools available for tuning and for the analysis of performance bottlenecks
- the information returned by the SESAM/SQL performance monitor
- the information provided by the Explain component of the optimizer
- how transactions and statements can be logged and analyzed by means of the SESAM/SQL request logging facility SESCOS
- how a view of the SYS_INFO_SCHEMA enables “costly” DML statements , i.e. ones which consume a particularly large amount of resources, to be determined.

2.1 Measurement and analysis tools

A number of different measurement and analysis tools are available to the DB administrator for tuning and for the purpose of analyzing performance bottlenecks. These include:

- the BS2000 monitor openSM2
- the SESAM/SQL performance monitor SESMON
- the SQL optimizer and its Explain component
- the SESAM/SQL request logging component SESCOS with the evaluation program SESCOSP
- the view SYS_DML_RESOURCES

Each of the measurement instruments listed above has its own specific application scenario.

openSM2 and SESMON are monitors that collect and output statistics without any significant effect on 'normal' operations (2-3% CPU utilization).

The typical measurement values obtained with the aid of openSM2 include overloading of individual channels or disks, utilization of the DAB (disk access buffer), and the utilization of CPU resources by individual programs (see the "[openSM2 \(BS2000\)](#)" manual for details).

openSM2 does not, however, provide any information on the utilization of the SESAM/SQL buffer and other DBH resources. This is where the performance monitor SESMON comes in. Whereas SESMON enables you to answer questions such as: "Are my DBH options set 'reasonably'?", The question: "Is my disk distribution 'optimum' for the prevalent conditions in the given framework?" can only be answered by openSM2.

Besides dealing with explicit issues concerning utilization, SESMON also returns information on the current lock situation (see the "[Database Operation](#)" manual).

SESCOS can be used to analyze workflows in the system. Specific measurements can be made at clearly defined points (e.g. entry and exit of a request, change of thread, etc.) and managed for each segment of the request. The cost for each processing step can thus be determined as a proportion of the overall cost by using the evaluation program SESCOSP. This allows the processing of a statement to be examined in detail and optimized in combination with the Explain component of the SQL optimizer (see the "[Database Operation](#)" manual). Activating SESCOS places a considerable load on the DBH, however, and should therefore only be used selectively for tuning individual statements on a test DBH.

SESAM/SQL automatically logs particularly “costly” SQL-DML statements in an internal buffer. A DML statement is regarded as costly when the amount of resources used by it is very high compared to other SQL-DML statements. The view `SYS_DML_RESOURCES` of the `SYS_INFO_SCHEMA` enables information on particularly “costly” statements to be determined. The collected data contains for each statement, for example, the application, the user, the statement type, and the quantity of resources processed.

2.2 SESMON

The performance monitor SESMON obtains current operational data from the SESAM/SQL-DBH for SESAM/SQL-DCN and optionally outputs this information to the screen, to SYSLST, or to a file.

SESMON can be started in interactive or batch mode.

The SESAM/SQL-DBH, SESAM/SQL-DCN and the connection module DBCON maintain statistics counters (even if the monitor is not started up). SESMON runs in a separate task and handles the evaluation and output of statistics.

The performance of SESAM/SQL-DBH and SESAM/SQL-DCN is not adversely affected by the operation of the monitor.

The option of directing output to SYSLST or a file enables statistics to be recorded over an extended period and subsequently analyzed to determine appropriate tuning measures.

The data logged in such files can be evaluated with the aid of an application program.

To obtain information on the current situation, the user can start the monitor interactively and select output to the screen.

Differentiated statistics can be selected by the user from the output on the screen, since the data is arranged in different masks. A differentiated selection is also possible from the output to SYSLST.

Data on the SESAM/SQL-DBH is distributed in masks as follows:

- The “OPTIONS” mask returns information on the currently set DBH options.
- The “SYSTEM INFORMATION” mask provides information on the size and occupancy of the work and transfer containers, the number of cursors, the number of existing or used logical files, and on the progress of the DBH session restart.
- The “SQL INFORMATION” mask outputs information on SQL-DML statements, plans and calls to special SQL components.
- The “SERVICE TASKS” mask outputs information on service tasks and on requests to be processed.

- The “SERVICE ORDERS” mask contains information on the DDL and/or utility statements currently processing in the service tasks as well as the user ID for the job. It also displays the progress of a RECOVER/REFRESH statement.
- The “I/O” mask contains the information on I/O accesses to the spaces and the cursor files. It also includes information on the I/O behavior of the remaining files used by SESAM-DBH (DA-LOG, CAT-LOG, TA-LOG, AC-LOG and CATREC).
- The “TRANSACTIONS” mask shows the number of transactions (arranged by various transaction states), the number of statements (CALL DML and SQL), and the number of activities.
- The “STATEMENTS” mask provides an overview of all running statements. Statements that run within a transaction as well as those that run outside of a transaction are shown.
- The “SYSTEM THREADS” mask outputs statistics on write threads.
- The “TASKS” mask provides the multitasking system with statistical data that can be used to determine the utilization of individual DBH tasks.

Information on SESAM/SQL-DCN operation is arranged as follows:

- The “OVERVIEW” mask outputs general information on DCN operation. This includes values from the parameter file as well as values relating to the entire DCN session.
- The “CAPACITY” mask contains the values related to incoming messages and the utilization of the pool. It also includes information pertaining to resource bottlenecks.
- The “TRANSACTIONS” mask provides information on the status and behavior of transactions.
- The “APPLICATIONS” mask contains information on individual applications.

In addition, SESMON provides the “PREFETCH-BUFFERS” mask for each configuration. This contains information on the utilization of the prefetch buffer used to support the block mode functions at the SQL interface.

For more information on how database operation can be optimized with the aid of SESMON, see [chapter “Specific notes on tuning applications” on page 113](#).

2.3 SQL optimizer

SESAM/SQL creates an evaluation plan called the SQL access plan for most SQL statements. The SQL optimizer ensures that a highly efficient access plan using the least possible amount of system resources is created for the SQL-DML statements.

The Explain component of the SQL optimizer can be used to examine the individual steps in which an SQL-DML statement (INSERT, UPDATE, DELETE, SELECT, MERGE and CALL) is processed. The insertion of an EXPLAIN pragma in the SQL statement causes a readable representation of the internal access plan to be output to a file (see the [“SQL Reference Manual Part 1: SQL Statements”](#)).

The pragma EXPLAIN has the following format:

```
--%PRAGMA EXPLAIN INTO file
```

The output of the Explain component and the information contained in it is described in [chapter “Explain component of the optimizer” on page 131](#) of this manual. It includes, in particular, specific information on the indexes, join orders, and join algorithms used.

The criteria for selecting indexes and relational operations are described in [section “Structure and control of the optimizer” on page 24](#).

The following factors influence the internal access plan of the SQL optimizer:

- Pragmas and annotations
- Catalog contents
- Database contents

Pragmas

The following pragmas are significant for the optimization process (see also [section “Overview of the effect of pragmas relevant to optimization” on page 39](#)):

- OPTIMIZATION LEVEL *n*
- SIMPLIFICATION ON/OFF
- JOIN
- KEEP JOIN ORDER
- IGNORE/USE INDEX *indexname*
- IGNORE/USE SORT_INDEX *indexname*

Catalog contents

Optimization is tailored to the existing indexes and integrity constraints.

Database contents

Optimization is tailored to the estimated number of rows found. This estimation is primarily dependent on the available statistics. It is therefore essential to ensure that the statistics reflect the actual distribution of values in the database. If required, the REORG STATISTICS FOR INDEX statement can be used to update the statistics for an index.

When you modify the contents of the catalog or database and try to optimize the access plan, especially by using pragmas, the Explain component will allow you to trace exactly how the plan has changed.



Since the database content is also significant, all attempts to set pragmas and indexes should be made with the original database or a test database that reflects a real-life situation.

2.4 SESCOS and SESCOSP

SESCOS is a measurement and diagnostic aid with which transactions, statements, or the processing steps of a statement can be logged and analyzed over a specific period. It can thus be used to determine how long a transaction, a statement or a processing step lasts and which resources are used.

SESCOSP is a utility routine for the output of logged requests.

2.4.1 Differentiation from other performance analysis tools

SESCOS is not suitable for analyzing the system behavior and system utilization of SESAM/SQL-DBH. The performance monitor SESMON is the appropriate tool for this purpose.

It is likewise not possible to use SESCOS in order to determine the individual processing steps into which a statement is subdivided by the optimizer. This subdivision can, however, be determined with the aid of the Explain component. SESCOS can certainly be used as a supplement to the Explain component to verify the cost involved for each such processing step.

2.4.2 Recommended approach for analysis using SESCOS and SESCOSP

The following procedure is recommended when using SESCOS and SESCOSP for analysis:

- SESAM requests should be logged with SESCOS over a time period that is sufficient to make the performance problem evident in the evaluation. Logging such requests over an extended interval takes up too much space in the CO-LOG file and makes the evaluation unduly long and cumbersome. In fact, it makes sense only if the information written to the CO-LOG file can be restricted accordingly by means of a USER selection. Furthermore, it should be noted that request logging places an additional load on the SESAM-DBH. This additional load, in particular, also has an impact on the measurement results, but does not invalidate the general findings.
- Use the evaluation program SESCOSP to evaluate the data.
- If it is not known which user or which statement is causing the performance problem, it is advisable to first evaluate the transaction statistics for the entire time period that was logged. This will indicate which transactions use the most resources and to which users they belong. A specific interval to be examined more closely can then be selected accordingly.
- Particularly “costly” SQL-DML statements can be determined with the view SYS_DML_RESOURCES of SYS_INFO_SCHEMA, see [section “View SYS_DML_RESOURCES” on page 21](#).
- If an individual statement or a sufficiently small time interval that needs to be examined more closely is already known, the statement statistics for that statement or time interval can be evaluated. The editing formats “*STRING-FORMAT” and “*IO-STATISTICS” may be used in this context to prepare the statement for printing or to output the I/O behavior of each statement.
- If a single statement or a few statements are to be analyzed, the editing formats “*STEP-IO-STATISTICS” and “*STEP-COMPLEXITY”, which reflect the I/O behavior and complexity of each processing step, can be output for each of the statements.

2.5 View SYS_DML_RESOURCES

Performance improvements can be achieved by analyzing, evaluating, and enhancing particularly “costly” SQL-DML statements.

An SQL-DML statement is regarded as costly when the amount of resources used by it is very high compared to other SQL-DML statements.

The resources of a statement which are examined are the number of logical and physical inputs/outputs, the elapsed time, and the activity time of this statement in the DBH and in the service tasks. The activity time supplies more accurate information than the elapsed time as the times in which the SQL-DML statement was disabled (because of a transaction lock or waiting for a service task request to be enabled) are not also taken into account. From these resources, SESAM/SQL determines a `MEASURE_OF_COSTS` for the costs of the statement.

SESAM/SQL logs the SQL-DML statements with a particularly high measure of costs in an internal buffer which can be cyclically overwritten. Logging takes place automatically and requires no preparatory measures. It does not impair the performance.

The view `SYS_DML_RESOURCES` of the `SYS_INFO_SCHEMA` supplies information on particularly “costly” SQL-DML statements, see the manual “[SQL Reference Manual Part 1: SQL Statements](#)”. Basically it outputs the contents of the written buffer. The collected data contains for each statement, for example, the application, the user, the statement type, and the quantity of resources processed.

The SQL-DML statements concerned can, for instance, be identified by their start time and application name, and also by the user. When a critical statement is recognized, more detailed information can be ascertained, e.g. by using `SESCOS`.

3 Application aspects relevant to performance

This chapter describes some important application aspects that are relevant to performance and can be influenced by the user in some cases by suitable measures:

- Structure and control of the optimizer
- Synchronization mechanisms
- Resource consumption
- Use of routines

3.1 Structure and control of the optimizer

This section describes the approach of the optimizer when generating access plans for SQL statements and the optimization steps and techniques used in the process. The section is specially intended for users who wish to speed up the evaluation of SQL statements with the help of pragmas and additional indexes and for those who wish to understand why a particular access plan was selected by the optimizer.

CALL DML statements are not dealt with in this section, since the optimization techniques for them are not the same as those for SQL statements.

In order to understand this section, you will need to have a thorough knowledge of the usual optimization techniques for query expressions as described in the standard literature for SQL systems.

3.1.1 Plans

The main task of the optimizer is to create the most suitable SQL access plan in terms of response time and throughput from the textual representation of an SQL statement. An SQL access plan (or plan for short) is an evaluation rule for (parts of) SQL statements. A plan describes the type and sequence of individual evaluation steps such as “read next primary record”, “read next index entry” or “create column definition”.

The following must be observed:

- A plan is not generated for every SQL statement. For example, no plan is created for the statements COMMIT, ROLLBACK, SET TRANSACTION, SET SCHEMA, etc.
- Although plans are created for DDL, SSL and utility statements, no optimization of the plan is required here. Consequently, these statements will not be examined in detail in [section “Structure and control of the optimizer”](#).

3.1.2 Phases in the creation of a plan

The creation of a plan generally includes the lexical, syntactic, and semantic analysis of the SQL statement, the generation and assessment of multiple alternative evaluation rules and, finally, the selection of the most favorable plan (based on the cost model). The optimization of a plan occurs in several independent phases.

3.1.2.1 Phase 1: Front-end

The front-end performs the syntactic and semantic analysis of the SQL statement. It creates a so-called standard plan for the statement representing a complete and valid - though not very efficient - evaluation rule. The standard plan is then refined in steps in the following phases.

3.1.2.2 Phase 2: Algebraic optimization

The algebraic optimization techniques are still independent of the existing access paths (indexes), evaluation methods (e.g. special join algorithms), cost estimates, and the estimated number of hits. Algebraic optimization techniques are, in turn, further subdivided into the following phases:

- Normalization
- Simplification
- nesting

Normalization

The objective of normalization is to transform differently phrased SQL statements that are semantically equivalent into a uniform normalized internal representation so that the following optimization phases can use this uniform representation as a starting point.

Examples of normalization

1. Elimination of NOT, IN, BETWEEN:

```
NOT (R.a = 5)           ⇒ R.a <> 5
R.a IN (10,20,30)      ⇒ R.a = 10 OR R.a = 20 OR R.a = 30
R.a BETWEEN :var1 AND :var2 ⇒ R.a ≥ :var1 AND R.a ≤ :var2
```

2. Predicate normalization ⇒ conjunctive normal form (CNF):

```
R.a > 10 OR (R.b = 20 AND R.c = 5) ⇒
CNF (R.a > 10 OR R.b = 20 AND R.a > 10 OR R.c = 5)
```

3. View substitution:

```
DECLARE VIEW V AS SELECT R.a, R.b, R.c
                    FROM   R
                    WHERE  R.c > 20

SELECT V.a FROM V WHERE V.b = :var1 ⇒
SELECT R.a FROM R WHERE R.b = :var1 AND R.c > 20
```

Simplification

The goal of simplification is to collect all search conditions (of nested query blocks and views) at one location, i.e. in the outermost query block, and to then simplify the predicates and expressions (by the detection and elimination of contradictions or tautologies, for example). This is particularly useful for join expressions and queries on views.

Examples of simplification

1. Elimination of redundant predicates:

WHERE R.a > 20 AND R.a > 10 \Rightarrow WHERE R.a > 20

2. Detection of contradictions (empty set of hits):

WHERE R.a > :var1 AND R.a IS NULL \Rightarrow FALSE

3. Evaluation of constant expressions:

a) WHERE R.a > 3 + S.b + 4 \Rightarrow WHERE R.a > S.b + 7

b) WHERE R.a > 2 * :varx + 3 * :vary \Rightarrow WHERE R.a > p

The right relational operand p is evaluated once at the outset instead of being reevaluated for each row of R.

4. Propagation of constants in the case of a local or joint predicate

```
SELECT R.a, S.b
FROM   R, S
WHERE  R.c = 10
AND    R.c = S.c
 $\Rightarrow$ 
SELECT R.a, S.b
FROM   R, S
WHERE  R.c = 10 (1)
AND    S.c = 10 (2)
```

The join is eliminated in favor of a Cartesian product of the two smaller sets of rows found with the two local predicates (1) and (2). (No analogous propagation of parameter values occurs, since these values are unknown at the time of optimization. Similarly, no propagation takes place for the other relational operands (<>, >, \geq , <, <=).)

5. Reformulation of a join to an IN predicate with a subquery (only for special cases where the select list contains only DISTINCT columns of a single join operand):

```
SELECT DISTINCT R.a, R.b
FROM   R, S
WHERE  R.a = S.a
AND    R.b > 10
⇒
SELECT DISTINCT R.a, R.b
FROM   R
WHERE  R.b > 10
AND    R.a IN (SELECT S.a FROM S)
```

The advantage of this representation is that all S.a values need not be examined; instead, the scan on S can be stopped after the first hit with S.a=R.a.



Influencing the simplification

If the standard plan contains no views and no redundant expressions of predicates, and if there is no join expression to which the above-mentioned optimization steps apply, simplification will not produce any improvement in the standard plan. To save optimization time in such cases, the phase can be deactivated using the pragma “SIMPLIFICATION OFF”.

In the case of extremely “simple” query expressions, the system automatically turns off the simplification phase on the assumption that the optimization overhead is not justifiable. The pragma “SIMPLIFICATION ON” may be used to force simplification in such cases.

Nesting

The objectives of nesting are:

- to evaluate predicates early and to thus reduce the size of intermediate results
- to eliminate costly evaluation steps if possible or to compute them only once at the outset.

This will be illustrated in some examples which follow.

Examples of nesting

1. Preempting restrictions

```
SELECT R.a, S.a
FROM   R, S
WHERE  R.b = S.b   (1)
AND    R.c > 20   (2)
AND    S.c > 30   (3)
```

The local subpredicates (2) and (3) are first evaluated on R or S, and the join result according to (1) is then determined (not on all R and S relations, but on the lower number of hits from (2) and (3)).

2. Formation of precalculated expressions (“precomputables”)

Precomputables are costly expressions that are only evaluated once.

```
a) SELECT R.a FROM R
   WHERE R.b > (SELECT MAX(S.b)
                FROM   S
                WHERE  S.c = :var1)
```

The uncorrelated subquery on S is handled as a precomputable, i.e. the result value of the subquery is only computed once at the start of processing and not recalculated for each tuple in R.

```
b) WHERE R.b > 3 * :varx + :vary
```

The right reference value becomes a precomputable and thus an atomic reference value for the DBH kernel. If an index to R.b exists, this is what allows the index to be used (note that the reference value must not be an arithmetic expression).

3. Detection of predicates that can be evaluated in advance (“preconditions”)

```
SELECT R.a FROM R
WHERE R.a > :var1 (1)
AND :var2 > 30 (2)
```

Condition (2) is independent of the rows in R and can therefore be computed in advance:

- If it evaluates to FALSE, the set of rows found is empty (regardless of R, i.e. access to individual rows is no longer required).
- If it evaluates to TRUE, condition (1) must be tested for each row in R.

Preconditions are possible only if the subpredicate is linked to the remainder exclusively with AND (if the AND in the example is replaced by OR, no precondition can be extracted).

4. Elimination of DISTINCT via uniqueness constraint

```
SELECT DISTINCT R.a FROM R
```

If a uniqueness constraint on R.a exists, the otherwise required elimination of duplicates (incl. sorting) can be dispensed with.

3.1.2.3 Phase 3: Selecting the access path

The optimization techniques for selecting the access path take the existing access paths (indexes) and evaluation methods in the system specially into account. In contrast to the other phases, it is not sufficient to generate only one plan variant (“straight forward”) when selecting the access path. In many situations, a number of plan variants must be examined and evaluated in order to find the most suitable plan. Cost and hit estimates based on statistical data on the distribution of values in the database play a significant role in the evaluation. The REORG STATISTICS FOR INDEX statement should therefore be used from time to time to ensure that the available statistics are always current.

An SQL query usually consists of a number of nested query blocks (subqueries). Each query block comprises multiple tables which are combined with one another by means of joins. Each individual table is subject to additional local constraints.

This general structure of a request is the basis for the approach used to select the access path. A (complex) request is broken down stepwise into a number of successively simpler subrequests. The following intermediary steps are performed in the given order when subdividing each request into increasingly simpler subrequests:

- Subquery optimization
- Join optimization
- optimization of one-relation requests
- Minimization of sort operations
- Access to base tables

Optimization of a nested request (subquery optimization)

A nested request is split into its individual query blocks. Each query block is in turn optimized further. An attempt is made to evaluate the inner query blocks as seldom as possible. A distinction is made in this context between correlated and uncorrelated subqueries as explained below.

Examples of subquery optimization

1. Uncorrelated subquery:

isolation as a precomputable

```
SELECT R.a FROM R
WHERE R.b > (SELECT MAX(S.b)
             FROM S
             WHERE S.c = :var1)
```

The uncorrelated subquery with respect to S becomes a precomputable, i.e. the result value is only computed once at the start (and not repeated for each tuple in R).

Such optimization always occurs and cannot be influenced by a pragma.

2. Correlated subquery:

minimization of evaluations by sorting “outer values”

```
SELECT R.a FROM R
WHERE R.b > (SELECT S.b
             FROM S
             WHERE S.c > R.c)
```

If the evaluation of the subquery with respect to S is too “costly”, the tuples of R are returned sorted by R.c, so the result of the correlated subquery can be buffered and only needs to be recalculated for a new R.c value.

This optimization occurs only for outer references in the directly enclosing query block and only if the outer reference is not a column with a uniqueness constraint.

The user can suppress this optimization of a correlated subquery by specifying a low value for PRAGMA OPTIMIZATION LEVEL n ($n \leq 4$ at present). If the pragma is not specified, the optimization is performed, i.e. the evaluation costs of the subquery are estimated, and a sort operation is introduced if the cost of the subquery exceeds a specific threshold value.

Optimization of a query block (join optimization)

Several join orders are taken into account to determine the most favorable order. The individual operands of the join (one-relation requests) are in turn optimized further. The join order is defined by the following strategy:

1. application of all elementary equi-joins
(i.e. conditions of the type $R.a = S.b$, but not $R.a+1 = 2*S.a$)
2. if relevant, application of all elementary theta-joins
(i.e. the conditions $R.a \text{ OP } S.a$, OP in $\{>, \geq, <, \leq\}$).

All other join predicates are not optimized further and should therefore be avoided.

Example

```
SELECT R.d, S.d, T.d, U.d
FROM   R, S, T, U
WHERE  R.a = S.a
AND    S.b = T.b
AND    T.c > U.c
```

1. The join of (R, S, T) is planned first: The more cost-effective sequence from JOIN (R, JOIN(S,T)) and JOIN (JOIN(R,S), T) and the join algorithm to be applied in each case are selected.
2. The composite (R, S, T) becomes the outer relation of a nested-loop join (and is best evaluated only once); U is selected as the inner relation (possibly using an existing index on U.c).

It is only if the join predicate contains more than one elementary equi-join that this strategy will provide some latitude in the determination of the join order. The actual join order can then be defined in three ways:

- Without a pragma:
The optimizer generates some/all possible alternatives, runs a cost estimate for each plan variant, and selects the plan variant with the lowest cost.
- With PRAGMA OPTIMIZATION LEVEL n ($n \leq 8$):
The optimizer generates (with a quasi non-deterministic approach) only one join order (which must naturally follow the above strategy) and ignores all further alternatives right from the start. This allows the user to limit the optimization overhead at the expense of runtime performance.

- With PRAGMA SIMPLIFICATION OFF:
This enables the user to define a specific join order that must be followed by the optimizer. The use of this pragma requires a thorough understanding of the underlying set structure and distribution of values in the database.
- With PRAGMA KEEP JOIN ORDER:
For the order of the joins, it causes the optimizer to keep as closely as possible to the order of join operands defined in the SQL statement.

Furthermore, the join algorithm to be used for each binary join operation is defined as follows:

- for elementary equi-joins: sort-merge join or nested-loop join
- for elementary theta-joins: nested-loop join
- for other join predicates: determination of cross-product, followed by selection

In the case of a (recommended, since efficient) elementary equi-join, the user can force a sort-merge join by a pragma specification of a low OPTIMIZATION LEVEL n ($n \leq 5$); other than that, the user has no further direct control options. The selection is based on the cost and hit estimates for the two join operands and uses heuristic techniques to determine the most effective use of the existing schema information (especially the existence of indexes).

Example illustrating the problem of selecting the join algorithm

```
SELECT R.a, R.b, S.a, S.b
FROM   R, S
WHERE  R.a = S.a
AND    R.b = :var1
```

- Sort-merge join incorporating sort minimization
If R.a and S.a are both inverted or primary keys, the indexes are processed and the join rows are determined in the merge without needing to physically sort the rows found (costly); if an index is missing, the join operand in question must first be sorted physically.
- Nested-loop join with index to the inner relation
If S.a is an inverted or primary key, a nested-loop-join with R as the outer relation must be considered. For each row that is found in R, R.a serves as the index reference value in S.a to determine the join rows in S.

optimization of one-relation requests

From a subrequest whose predicate references only a base relation, the part of the predicate that is relevant for the access plan is first separated.

All subpredicates that do not contain the following constructs can be evaluated by the DBH kernel:

- correlated scalar subqueries (not precomputable)
- EXISTS, ANY, and ALL predicates
- IS CASTABLE predicates
- LIKE_REGEX predicates
- IN predicates with subquery as relational operand
- concatenation of strings
- CASE, CAST and all other functions provided these are independent of the database content, i.e. if they can be precompiled

A few further optimizations are performed on the separated subpredicate:

- Combine and minimize search ranges on one index

WHERE R.a > 10 AND R.a < 20 Indexed search on the range (10,20) (This is only possible if there is no existing OR relation with further predicates to other columns; it should therefore be used judiciously.)

- Schema reduction: output only the required columns of a table

```
SELECT R.a, R.b
FROM   R
WHERE  R.c > :var1           (1)
AND    R.d = :var2          (2)
AND    R.a IN (SELECT S.a FROM S (3)
              WHERE S.b = R.b)
```

R.c and R.d are read here by the DBH kernel only to evaluate the predicates (1) and (2). The DML interpreter, which checks the remaining predicate (3), is only passed the values required at that point, i.e. the values of columns R.a and R.b, thus minimizing the data transport.

For multiple columns, only the actually required variations are passed to the DML interpreter, not the entire column.

Minimization of sort operations

In tandem with the foregoing optimization techniques, an attempt is made to selectively eliminate physical sort operations, while guaranteeing the desired sort order of an (intermediate) set of rows found. This is achieved by treating the required sort criterion as an add-on condition when optimizing a subrequest.

A high optimization overhead is associated with minimizing sort operations, since this frequently involves the consideration of opposing criteria (trade-offs) and thus requires the analysis of several plan alternatives. In individual cases where such overhead does not seem justifiable, the user can set a low value in the pragma OPTIMIZATION LEVEL n ($n \leq 6$) to restrict sort minimization and thus the overall optimization overhead.

Examples for elimination of sort operations

1. Inheritance of sort order by the intermediate result relation

```
SELECT    R.a, SUM(R.b)
FROM      R
WHERE     R.a < :varx
GROUP BY  R.a
ORDER BY  R.a
```

- No new sorting is required for the ORDER BY clause, since the GROUP result is already returned correctly sorted.
- If R.a is NOT NULL and is also inverted (secondary index), R is read by an indexed scan, i.e. need not be sorted at all.

2. Reduction of sort operations by intelligent ordering of join operations

```
SELECT    *
FROM      R, S, T
WHERE     R.a = S.a
AND       S.b = T.b
ORDER BY  R.a
```

The join order sort-merge-join (R ORDER BY R.a, JOIN(S,T) ORDER BY S.a) eliminates the need for subsequent ordering by R.a, since the sort was already completed as a preparation for the sort-merge-join.

3. Use of indexes for sorting

```
SELECT    R.a, R.b
FROM      R
WHERE     R.a > 10
ORDER BY  R.a
```

- If R.a is secondary-index-inverted (or the primary key or its first component), then an index scan on R.a evaluates the WHERE clause, and the found rows are correctly sorted at the same time.
- If the condition R.a > 10 were omitted on the other hand, and if NULL values were also allowed, the index could not be used for sorting, since the insignificant rows would otherwise be lost. From a viewpoint of minimizing sort operations, indexes are therefore especially recommended for significant columns.

Access to base tables

The most suitable method for accessing an individual base table is determined on the basis of the distribution of column values and the reference values used in the request.

The following access methods are available (see also examples on [page 37](#)):

- Sequential read of the entire table
- Sequential read of a restricted primary key range of the table
- Use of one or more secondary indexes to determine the number of hits (or a superset thereof), followed by reading of the rows found in the table.
- Sequential read of a secondary index. This method is used to obtain the rows of the table, sorted by the index, or to evaluate the search conditions on a combined index which do not refer to the first column(s) of the index. In both cases, the secondary index may be merged with other indexes before the rows found (or a superset thereof) are read in the table.

Prerequisites for the use of an index:

1. The index evaluation must produce a superset of the actual number of hits. This is possible in the following cases:
 - if the subpredicate to be evaluated by indexing is linked with the remainder of the predicate only by AND
 - if the subpredicate to be evaluated by indexing is only ORed with other subpredicates which can, in turn, also be evaluated by indexing.

2. With relational conditions, the predicate must have the following format:
`spalte op wert`
 i.e. the inverted column must occur as a relational operand and must not be “hidden” in an arithmetic expression (such predicates should hence always be rephrased).
3. For NULL constraints: the operator must be “IS NOT NULL”.
4. The statistical estimate must project a sufficiently high level of selectivity for the subpredicate ($\leq 75\%$).
5. The index must not have been excluded by the user with PRAGMA IGNORE INDEX.

If the request contains parameters or subqueries, which amounts to the same thing, the reference values will only be known when executing the plan, not at the time it is created. The access path selection is partially repeated at the time of execution in this case to take the current reference values into account.



The estimated hit rate is based on the statistical distribution of index values. If these statistics do not correspond to the actual distribution of values in the table, it is advisable to update the statistics with the statement REORG STATISTICS FOR INDEX (or REORG SPACE).

Examples of index processing

Relation R with columns: a (secondary index); b,c (combined index); d (without Index) and the column 'key' as the primary key

(1) WHERE R.key = :var1 AND ...

Index processing:

direct access via the primary key; if relevant, the remaining predicate may then also be evaluated (method 2).

(2) WHERE R.a = :var1

Index processing:

complete predicate evaluation with the secondary index on “a” (method 3).

(3) WHERE R.a = :var1 AND R.d > :var2

Index processing:

determination of a superset of hits with the secondary index on “a”; text of “d” on primary data (method 3).

```
(4) WHERE R.b > :var1  
      WHERE R.b = :var1 AND R.c > :var2
```

Index processing:

use of the combined secondary index in both cases (method 3).

```
(5) WHERE R.c > 10
```

Index processing:

use of the combined secondary index (method 4) even if there is no further condition on R.d.

```
(6) WHERE R.a > 10 AND/OR R.b > 20
```

Index processing:

AND/OR merge of the two indexes (method 3).

```
(7) WHERE R.birthdate > 1900 AND ...
```

Index processing:

the index is not used if the statistical estimate projects a hit rate of more than 75% (the decision is made at the time of creating the plan). Consequently: sequential search in the plan (method 1).

```
(8) WHERE R.birthdate < 1900 AND R.zip BETWEEN 80000 AND 81999
```

Index processing:

the index is not used if the intermediate result calculated to this point contains few hits (decision at the time of executing the plan). Consequently: only index on "birthdate" used; second subpredicate tested on primary data (method 3).

3.1.3 Overview of the effect of pragmas relevant to optimization

The following pragmas are relevant for the actual optimization process:

- SIMPLIFICATION
- IGNORE/USE INDEX
- IGNORE/USE SORT_INDEX
- JOIN
- KEEP JOIN ORDER
- OPTIMIZATION LEVEL

These pragmas can also be used in routines and when calling procedures with the SQL statement CALL, see the manual "[SQL Reference Manual Part 1: SQL Statements](#)".

```
--%PRAGMA SIMPLIFICATION ON/OFF
```

All optimization techniques for simplification (part of the algebraic optimization) are collectively turned on or turned off, i.e. all (ON) or none (OFF) of the optimization steps outlined there are executed.

```
--%PRAGMA IGNORE/USE INDEX index
```

The specified index is ignored (IGNORE) or used (USE) when determining the join order and join algorithm, and when selecting the optimum access path (two base relations).

```
--%PRAGMA IGNORE/USE SORT_INDEX index
```

The specified index is ignored (IGNORE) or used (USE) when selecting the sort algorithm.

```
--%PRAGMA JOIN [SORT MERGE | NESTED LOOP]
```

Specifying this pragma selects the JOIN method to be used (sort-merge join or nested-loop join).

If the number of plan alternatives is limited by the OPTIMIZATION LEVEL pragma so that no more nested-loop joins are viewed (OPTIMIZATION LEVEL < 6), then the JOIN pragma cannot force a nested-loop join.

This pragma is also taken into account in the intermediate joins of multiple joins.

If the JOIN pragma is not specified, then the optimizer always selects the JOIN method to be used.

```
--%PRAGMA KEEP JOIN ORDER
```

The pragma defines how a multiple join is to be processed.

For the order of the joins, it causes the SQL optimizer to keep as closely as possible to the order of join operands defined in the SQL statement. This applies for explicit joins (e.g. *table_1 JOIN table_2 ON search_condition*) provided the JOIN annotation specifies no other algorithm.



The same effect can also be achieved with JOIN and SIMPLIFICATION OFF. However, other optimizations are then disabled in the process.

If the pragma KEEP JOIN ORDER is not specified, the optimizer selects the join order to be used provided no specific order is forced to be used by means of the pragma JOIN or SIMPLIFICATION OFF

The pragma can be used in the case of CALL, in DML statements and in routines.

```
--%PRAGMA OPTIMIZATION LEVEL n
```

The '*n*' option controls the number of plan alternatives that are created and evaluated in the course of selecting the access path.

All plan variants are examined, and heuristic techniques are used to select the most promising variants with the best potential to generally improve the evaluation costs. The number of planned variants to be examined further depends on the value of '*n*'. For the value of '*n*' any number between 1 and 10 may be selected; the default is 9. Starting with a value of $n \leq 5$, only one plan variant is examined in each optimization step.

The following individual levels can be differentiated:

- $n \geq 9$
Various join orders are taken into account.
- $n \geq 8$
In a nested-loop join, an attempt is made to check whether switching the order of two join partners would be beneficial.
- $n \geq 7$
Sort minimization is performed.
- $n \geq 6$
For join optimization, the sort-merge as well as the nested-loop join are considered. In the access selection, all methods of achieving a required sort (physical sort in the DBH kernel, sort by index scan) are considered.

- $n \geq 5$
Execution of subquery optimization and storage of intermediate result relations required more than once.
- $n \geq 4$
Execution of range construction, i.e. multiple atomic predicates on the same column are combined into a single index access.

This pragma also has an effect on the extent to which predicates are normalized. Since the normalization for complex predicates can be very CPU-intensive, a low value for ' n ' with complex predicates means that a complete conjunctive normal form will not be generated. This saving of CPU overhead during optimization generally leads to increased processing costs for the query expression, since certain optimization steps must be omitted in the case of degenerated normal forms.

Practical notes on the use of pragmas and other tuning options can be found in [section "Optimization options for SQL applications"](#) on page 114.



The foregoing pragmas are a very sensitive tool that should only be used in exceptional cases. Bear in mind that pragmas are designed to restrict the number of variants considered by the optimizer for the most favorable access plan. As a rule, this will generally result in less efficient access plans, especially after extensive changes in the database (set structure, value distribution, indexes, etc.).

3.1.4 Annotations

As described in the previous section, pragmas are placed at the start of a statement, and their effect extends over the entire statement. Thus, for example, an index excluded with the IGNORE INDEX pragma cannot be used in the entire statement.

Annotations are provided when the scope of optimization information is limited to the local sphere. Annotations are notes which are contained at particular positions within an SQL statement and are only used when optimizing the operations affected.

Annotations can also be specified in the definition of a view. Consequently optimization information can already be provided in the view definition for SQL statements which use this view.

When an annotation which can be evaluated by the optimizer is contained in an operation, this annotation has priority over any specified pragma which is relevant to optimization. This also applies when the annotation is integrated implicitly in the SQL statement via the definition of a view.

3.1.4.1 JOIN annotation

SESAM/SQL offers an annotation in order to influence join optimization of inner joins and outer joins:

```
T1 ... JOIN /*% {SORT MERGE | NESTED LOOP} [{LEFT|RIGHT} FIRST] %*/ T2 ON ...
```

This annotation notifies the optimizer of which join algorithm and (optionally) of which sequence of join operands is preferred for analyzing the join between T1 and T2. If the proposed strategy is possible, it is then also planned by the optimizer.

Annotations may be ignored if they cannot be executed in the context. For example, the annotation RIGHT FIRST is meaningless with a LEFT OUTER JOIN because in this case the LEFT table, as the dominant table, is always accessed first. Also, defining a low optimization level may reduce the number of considered plan variants in such a way, that certain join methods or join sequences - which may contain annotations - are ignored.

Multiple annotations can be used in a statement, e.g. in order to support the planning of multiple joins.

3.1.4.2 CACHE annotation

You can use this annotation to cache the result of the table function CSV() in a temporary file. This expedites multiple access to the same table. Thus enabling CSV tables to be processed in parallel.

Examples

If a CSV table is read more than once in an SQL statement (e.g. as a subquery or as an operand in a Join), caching the CSV table can reduce the execution time.

The example below ascertains the sum `namscore` of the `score` from the CSV file `csv.scoretab` for each name `nam` of the name table `namtab`. For each name the CSV file contains no, one or multiple lines with the score achieved.

```
SELECT nam, (SELECT SUM(CAST(S.score AS INT))
             FROM TABLE(CSV /*%CACHE%*/ ('csv.scoretab' DELIMITER ',' ,
                                         CHAR(20), VARCHAR(10)))
             AS S(nam, score)
            WHERE N.nam = S.nam)
AS namscore
FROM namtab AS N
....
```

Caching a CSV file also enables the restriction in the use of CSV files, which means that a CSV file cannot be read simultaneously in parallel, to be bypassed. If the table function contains the CACHE annotation, the time during which parallel reading is not possible is restricted to the time which is required to read the file into the cache. At any other transaction time, parallel reading of the CSV file is possible. Parallel reading of the CSV file is then possible in the remaining processing time.

The example below selects from the CSV file `csv.scoretab` all lines in which the `score` is less than the average score of the other names `nam` in the CSV file.

```
SELECT name, cast(score as int)
FROM TABLE(CSV /*%CACHE%*/ ('csv.scoretab' DELIMITER ',' ,
                              CHAR(20), VARCHAR(10)))
AS S(nam, score)
WHERE CAST(score AS INT) < (SELECT AVG(CAST(score AS INT))
                            FROM TABLE('csv.scoretab' DELIMITER ',' ,
                                         CHAR(20), VARCHAR(10)))
                            AS X(nam, score)
                            WHERE X.nam <> S.nam)
```

3.1.4.3 IMMUTABLE annotation

The calculation of functions with constant input values (uncorrelated functions) depends on the call context concerned. These functions are, for example, recalculated in particular for each record in SELECT lists. See the description in the “Routines” chapter of the manual [“SQL Reference Manual Part 1: SQL Statements”](#).

When the application is designed, IMMUTABLE annotation can be used to ensure that the function value of uncorrelated functions is only calculated once. However, it must be guaranteed that the function value does not change when the input values remain the same.

In the case of uncorrelated functions whose calculation requires costly database accesses, this can lead to an appreciable enhancement in performance.

Example

```
CREATE FUNCTION maxs( ) RETURNS INTEGER READS SQL DATA
  BEGIN
    RETURNS (SELECT max(c) FROM S);
  END
```

The `maxs()` function is uncorrelated, but would always be recalculated within a SELECT list. This can be prevented by the IMMUTABLE annotation:

```
SELECT x - maxs /*% IMMUTABLE %*/ ( )
  FROM   T
 WHERE  col = ...
```

3.2 Synchronization

This section discusses the following topics:

- Phenomena and isolation level
- Synchronization of DDL and DML
- Synchronization of utilities

3.2.1 Phenomena and isolation level

The following three phenomena are used in the definition of the SQL2 standard:

- P1 (dirty read)
- P2 (non-repeatable read)
- P3 (phantom)

P1 (dirty read)

(SQL) transaction T1 updates a row. (SQL) transaction T2 reads this row before T1 has executed a COMMIT. If T1 is now rolled back, then T2 will have read a row that was never “committed” and may therefore be treated as one that never existed.

P2 (non-repeatable read)

(SQL) transaction T1 reads a row. (SQL) transaction T2 modifies or updates this row and executes a COMMIT. If T1 now tries to read this row again, it receives the updated row or detects that the row was deleted.

P3 (phantom)

(SQL) transaction T1 reads a set of rows N that satisfies a particular query expression. (SQL) transaction T2 then executes (SQL) statements that generate one or more rows which match the query expression of T1. If T1 then repeats the original read operation with the same query expression, it will receive a different set of rows.

Based on these three phenomena, the isolation or consistency level is defined as follows:

SQL		CALL DML		Phenomena		
Isolation level	Consistency levels	RNL ¹	RNW ²	P1	P2	P3
READ UNCOMMITTED	0	x	x	x	x	x
-	1		x	x	x ³	x
READ COMMITTED	2	x		-	x	x
REPEATABLE READ	3			-	-	x
SERIALIZABLE	4			-	-	-

Table 1: Isolation level, consistency level and associated phenomena

¹ Read without locking

² Ignore the lock

³ The phenomenon "non-repeatable read" may occur if a row was read earlier with a dirty read.

This isolation level is implemented by the object-locking mechanism, which recognizes the following lock objects and their hierarchy:

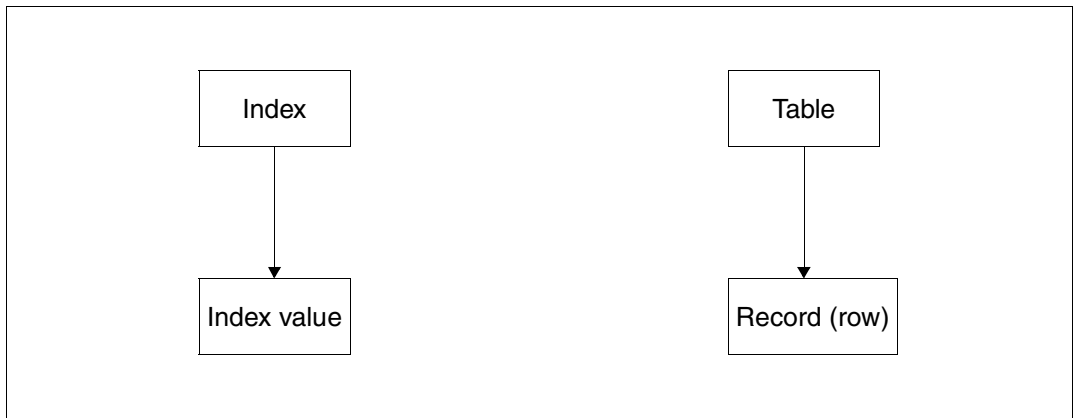


Figure 1: Hierarchy of lock objects

The following conditions apply:

- A lock on an object implies a lock on all objects below it in the hierarchy.
- If there are several locked objects that all belong to the same “parent object”, these locks are raised to a lock on the “parent object”.

The locking mechanism provides shared and exclusive locks. Either an exclusive lock or ≥ 1 shared locks may be active for an object. As soon as an exclusive lock is acquired, no further shared lock may be requested.

The isolation level is implemented with this object-locking mechanism as follows:

1. Records and secondary index blocks (SIS0) that are updated are always provided with an exclusive lock. If this is not possible, the request is deactivated until the locking transaction has terminated. The update thus operates independently of the isolation level.
2. Data retrieval generally operates with shared locks. The following exceptions apply:
 - CALL DML
The use of exclusive locks can be defined in the Open statement.
 - SQL:
Rows are provided with an exclusive lock when reading if they are updated or deleted in the same statement or if they belong to a “FOR UPDATE” cursor or the SQL statement contains %PRAGMA LOCK MODE EXCLUSIVE.
3. Only record locks are used for reads in the isolation levels READ UNCOMMITTED, READ COMMITTED and REPEATABLE READ and in consistency level 1:
 - Isolation level “READ UNCOMMITTED” (consistency level 0):
Qualifying rows are not locked; rows locked by another transaction are output with a corresponding status (SQLSTATE 01SA1, CALL-DML status 9S).
 - Consistency level 1
As far as possible, matching rows are locked; rows locked by another transaction are output with a corresponding status (see above).
 - Isolation level “READ COMMITTED” (consistency level 2):
Matching rows are not locked. If a row is locked by another transaction, the request is deactivated until the locking transaction has terminated.

- Isolation level “REPEATABLE READ” (consistency level 3):
Matching rows are locked; if a row is locked by another transaction, the request is deactivated until the locking transaction terminates.
- Isolation level “SERIALIZABLE”:
It is not sufficient to lock individual rows here; instead, entire “spaces” must be locked.
 - Search via secondary index: all evaluated SIS0 blocks are locked.
 - In addition, not only the matching rows but also the non-matching rows are locked.
 - In the case of a sequential search, the 1st record which does not belong to the primary-key group is also always locked. With each row, the logical “space” to its predecessor is also locked.
 - For tables without primary keys, the entire table must be locked.

3.2.2 Synchronization of DDL and DML

The synchronization of DDL and DML occurs on two levels: on the level of the plans affected by DDL and on the level of the spaces involved.

SQL-DML plans include catalog information on the tables addressed in the DML and on the access paths (indexes) used. DDL updates the catalog information. Consequently, when a table or an index of the table is updated with DDL/SSL, the plans containing catalog information on the table must be invalidated, and the table must not be updated so long as such plans exist. The appropriate synchronization is effected by means of a transaction lock on the row that defines the table in the catalog table TABLES. A shared lock on the row is requested for generating and executing a plan. An exclusive lock on the row is requested for the DDL. When this exclusive lock is acquired, i.e. when all transactions in which a shared lock was requested have terminated, the existing plans are invalidated, and no new plans can be generated so long as the lock exists, i.e. until the DDL transaction has terminated.

DDL updates not only the definitions in the catalog space, but also the structures of the user space (file) on which the object (table or index) affected by the DDL is located. Due to the management of free space, in particular, DML calls to other objects of the space must also be coordinated with the DDL structural updates. The synchronization required for this purpose is achieved via transaction locks on the space. This synchronization also works with CALL DML, for which the above-mentioned plans do not exist.

Before the execution of the DDL statement on a space, an exclusive transaction lock on the space of the table is requested. If an index is involved (CREATE INDEX or DROP INDEX) and the index is located on some other space as the table, the exclusive lock is requested on both spaces. The acquired lock initiates a wait state until all transactions that have shared or exclusive locks on the space are completed. New lock requests on a space for which an exclusive lock has been requested are placed in a wait state, i.e. the lock is only allocated after the DDL transaction terminates. Resources for the table affected by the DDL are also released that are not coupled to a transaction (CALL DML Opens on tables and SQL cursors that were saved with STORE). In following accesses in the applications that reserved these resources, this leads to CALL DML Status or SQLSTATE.

Any long-running transaction blocking a DDL statement can be rolled back by means of an administration command. The locking transaction can be determined with the performance monitor SESMON or with SESADM.

To execute a DDL statement, information is read from the metadata of the catalog in order to verify authorization and to supplement and validate parameters of the statement.

Following the execution, the metadata in the catalog is updated. The catalog accesses required for this purpose are executed in the highest consistency level. This may produce wait states on transaction locks of other DDL or DML statements and could also lead to a deadlock. If a deadlock occurs, a competing DML statement is rolled back. In the case of a deadlock between two DDL statements, the deadlock is broken by rolling back one of the two DDL statements. It is advisable to avoid using DDL concurrently with a DML high load, since plans, cursors and CALL DML resources are lost as a result. Similarly, concurrent DDL on the same spaces should also be avoided, since they are serialized in the best case, but cause one of the statements to be rolled back in unfavorable situations.

3.2.3 Synchronization of utilities

Let us begin with the synchronization of utilities for which only catalog accesses are required, namely MODIFY and CREATE / DROP / ALTER MEDIA DESCRIPTION. These statements are processed with DML plans for the catalog space. The synchronization mechanisms of DML are used.

Synchronization between the other utilities on one hand, and DDL and DML on the other, is handled using transaction locks and utility locks on the spaces (space usage lock).

From the viewpoint of the application, utility statements are executed outside transactions. To enable the synchronization of utilities via transaction locks, transactions are opened internally. After the utility statement has been processed, the transaction is closed.

The transaction lock can be shared or exclusive. A shared lock is set for utilities in which the space is only accessed for reading, this includes: CHECK FORMAL, COPY OFFLINE and UNLOAD. Shared locks are also placed for LOAD OFFLINE and MIGRATE statements on the tables and indexes of a space that are not affected by the statement. The lock for REFRESH, RECOVER and REORG and for the table specified with LOAD OFFLINE or MIGRATE is exclusive. Other statements that find a transaction lock when accessing a space are placed in a wait state. Resources that are not bound to a single transaction are also released for the tables affected by the utility (CALL DML Opens on tables and SQL cursors that were saved with STORE). In following accesses in the applications that reserved these resources, this leads to CALL DML Status or SQLSTATE.

For CHECK CONSTRAINTS, an SQL-DML plan is generated. This plan is processed within a transaction. Synchronization for CHECK CONSTRAINTS *integrity-constraint-name,...* is handled by the synchronization mechanisms of DML. For a CHECK CONSTRAINTS ON TABLE / ON SPACE, an exclusive transaction lock is first requested on the spaces which are specified in the statement or the spaces on which the specified tables are located.

For COPY ONLINE, concurrent updates on the space with DML are possible. For updated blocks, before images are written and backed up on completion of the COPY. DDL/SSL and utilities that affect the space are placed in a wait state. Only one online copy can be active for a catalog at one time.

Using LOAD ONLINE or UNLOAD ONLINE you can read and modify tables and any related indexes in parallel with DML (or another additional (UN)LOAD ONLINE). In this case, LOAD ONLINE behaves like a DML update statement. In this case, UNLOAD ONLINE behaves like a DML read statement.

As for DDL, deadlocks in transaction phases may also occur in the case of utilities. Utilities are given preference when breaking such deadlocks. If there is a deadlock between two utility statements, however, one of the utilities is aborted, since a utility statement cannot be completely rolled back. The spaces affected by that utility statement must be recovered using the utility statement RECOVER. When considering whether a deadlock may occur, catalog accesses for authorization checks and information retrieval or updates must also be taken into account. Concurrent utilities on the same spaces should therefore be avoided as far as possible.

3.3 Resource consumption

This section describes:

- the conversation and request-specific resources that are available on the side of the SQL application program,
- the factors which influence the utilization of these resources, and
- the measures that can be implemented to eliminate a resource bottleneck.

3.3.1 Request-specific space requirements in the communication buffer

The application program and the DBH communicate (in independent mode) via a communication buffer that has a size of up to 64 Kbytes. The currently used space must therefore be less than this upper limit; furthermore, since the send length also has an impact on the performance, this space should be utilized as little as possible.

The size of the required space in the communication buffer between the application program and the DBH is essentially dependent on the following:

- Number and data type of parameters used in the statement
During execution, space in the communication buffer is used for values of I/O parameters (plus administrative information) when transferring data to and from the DBH.
Resource bottlenecks in the communication buffer can be prevented by modifying the length of the data type of parameters (e.g. CHAR(20) instead of CHAR(200)) or by reducing the number of parameters (e.g. specific entry of columns in the SELECT list instead of "SELECT *").
- Application type
Depending on the statement type, the complete statement text (plus administration information) is passed to the DBH via the communication buffer at the time of precompilation and execution.
At precompile time, this includes the DML statements SELECT, INSERT, UPDATE, DELETE, MERGE, and CALL, the cursor specification with DECLARE CURSOR, and all DDL, SSL, and utility statements.
At execution time, this includes the statements SELECT, INSERT, UPDATE, DELETE, MERGE, and CALL, the cursor specification with OPEN; all DDL, SSL, and utility statements, and the statement to be prepared with PREPARE or EXECUTE IMMEDIATE.

Bottlenecks in the communication buffer resource can be likewise prevented by reducing the statement text (e.g. by using permanent views, avoiding comments, using correlation names, splitting a CREATE SCHEMA statement, etc.).

3.3.2 Conversation-specific space requirements in the VGM

Whenever an SQL program is precompiled or executed, a conversation-specific memory area (VGM) of a restricted size is used.

The area reserved by the VGM is saved at the PEND time under openUTM. The maximum size of the area that openUTM can store is specified by the KDCDEF control statement MAX in the VGM-SIZE=number parameter when generating the UTM application.

Utilization of the VGM at the time of precompilation

The main information stored in the VGM on precompiling a compilation unit is as follows:

- specification (statement text) and administration of static cursors
- administration of dynamic cursors and preparable statements

If required, a bottleneck in the VGM resource can be prevented by reducing the statement text of static cursors or temporary views (e.g. by avoiding indentations and comments, using correlation names, etc.), by using permanent views, or by splitting the program code into multiple compilation units.

Utilization of the VGM at runtime

Besides various administrative information, the main information stored in the VGM at the time of executing an application is as follows:

- SQL descriptor areas (administrative data and contents):
The space requirement for a descriptor area depends **only** on the value of the WITH MAX clause of the ALLOCATE DESCRIPTOR statement. Space is reserved in the VGM for both administrative information as well as values of average length for this number of descriptor entries.
If required, a bottleneck in the VGM resource can be prevented by selectively releasing or reusing descriptors. Furthermore, the space required for an SQL descriptor area can be decreased by reducing the number of entries in an SQL descriptor area (WITH MAX clause) and by modifying the length of the data types (e.g. CHAR(20) instead of CHAR(200)).
- Prepared statements (administration):
In the case of prepared statements, it is not the statement text but an internal execution rule that is stored. The space for this code depends on the type of prepared statement and the number of dynamic parameters in the statement. If required, a bottleneck in the VGM resource can be prevented by reducing the number of parameters (e.g. by explicitly specifying columns in the SELECT list instead of "SELECT *") or by selectively reusing statement names.

Note

In the case of UTM applications, a bottleneck could also be eliminated by increasing the size of the VGM (with the parameter VGMSIZE at UTM generation). This will, however, adversely affect all program units of the UTM application (because openUTM saves the used portion of the VGM at every new dialog step). It is therefore advisable to first examine the above aspects for each specific program unit.

3.3.3 Space requirements in the prefetch buffer

If SQL users are working in block mode, all the partial result sets (blocks) from cursors opened in block mode are buffered in the prefetch buffer. The user specifies the size of the prefetch buffer with the PREFETCH-BUFFER parameter in the configuration file for the application.

Occupancy of the prefetch buffer

The space available in the prefetch buffer is occupied by the following data:

- user data (blocks)
- memory management data (for managing the memory areas for the user data)

The memory space required for a block depends on the following factors:

- number of columns in the cursor
- data types of the individual columns
- current values of the buffered result rows
- number of result rows in a block.

The space required for memory management depends on the number of blocks to be managed simultaneously (see also [section “Size of the prefetch buffer” on page 92](#)).

Information on the utilization of the prefetch buffer can be obtained from the PREFETCH-BUFFERS mask in the SESMON performance monitor (see the [“Database Operation”](#) manual).

If the size of the prefetch buffer proves to be too small, the user can increase the value of the PREFETCH-BUFFER parameter in the configuration file. The new value takes effect the next time the application is started.

3.4 Use of routines

Routines (procedures (stored procedures) and User Defined Functions (UDFs)) contain sequences of SQL statements which are stored and managed in a database.

Routines and their use in SESAM/SQL are described in detail in the manual "[SQL Reference Manual Part 1: SQL Statements](#)".

This section describes the aspects of using routines which are relevant to performance.

The use of routines enables the sequence of SQL statements in the DBH to be implemented without communicating with the application program.

This saves the **communication overhead** for the various statements and shortens **runtimes**. This effect occurs in particular when the application does not run on the same computer as the DBH.

With short statements, communication accounts for a significant share of a statement's runtime. When, for example, a cursor is processed, the FETCH statements mostly take a relatively short time, but they nevertheless require communication with the application program. Encasing cursor processing in a routine has a positive effect on the performance provided no contacts to the application program are required.

It must be noted that no COMMIT WORK (and also no ROLLBACK) is possible in routines, and that the transaction therefore comprises the entire run of the routine. This must be taken into account when defining routines.

Using routines also results in savings on **CPU time**.

Privilege checks need only be performed for the EXECUTE privilege of the routine. Only one plan is provided. I/O values are processed only once for each routine.

It must be noted that the use of routines in favor of the application creates additional effort in the DBH. The logic (control statements) of a processing process executes with the routine in the DBH. Without a routine it would execute in the application.

A performance-neutral but sensible aspect of routines is the encasing of work processes. As long as the interface of a routine does not change, the processing contained in it can be modified as required without the application program needing to be modified, compiled or linked.

4 Aspects of database design relevant to performance

This chapter deals with disk storage requirements for primary, secondary and metadata. It also describes some performance-related aspects to be observed when using integrity constraints and for data distribution and allocation.

4.1 Spaces with a size of more than 64 GB

User spaces (and also the catalog space) can be up to 4 TB in size (“large space”) if they are created with SESAM/SQL V7.0 or higher on a pubset which supports “large files”. This is done with CREATE SPACE, CREATE CATALOG, REORG SPACE or RECOVER TO. Otherwise spaces can be up to 64 GB in size.

You can store tables or indices containing a very large volume of data on large spaces. A non-partitioned table can thus be up to 4 TB in size. A partitioned table with 16 partitions can thus be up to 64 TB in size.

A large space consists of up to 1,073,741,822 blocks. All references to block numbers are stored in a field which is four bytes long. This slightly increases the space requirements of a database.

Examples

Space requirements of a base table with 250 million records and an average gross record length of 2,000 bytes. Length of the primary key: 20 bytes, density: 80%.

Primary data blocks	approx. 155 million 4KB blocks
Primary key index blocks	approx. 1.3 million 4KB blocks
Blocks for disk record number assignment table	approx. 0.25 million 4KB blocks
Total	approx. 156.55 million 4KB blocks = 597 GB

Space requirements for a secondary index with 1 million values, 250 hits per value, length of 20 bytes, 80% density.

Secondary index base level	166,667 4KB blocks
Secondary index index level	1,254 4KB blocks
Statistics block	1 4KB block
Total	167,922 4KB blocks = 656 MB

Space requirements of a secondary index for a unique index with 250 million values, 1 hit per value, length of 20 bytes, 80% density.

Secondary index base level	2.57 million 4KB blocks
Secondary index index level	19,381 4KB blocks
Statistics block	1 4KB block
Total	approx. 2.6 million 4KB blocks = 9.9 GB

The backup and recovery times depend both on the size of the spaces and on the peripherals used. When the volumes of data are extremely large, you should consider whether a partitioned table with several partitions would not be better suited than a single large table.

Fundamental information on non-partitioned and partitioned tables is provided in the [“Core manual”](#). Performance information on partitioned tables is provided in the [section “Partitioned tables” on page 67](#).

4.2 Primary data

The disk storage requirements for primary data can be calculated as follows:

Block size	4096
Block header	52
Block checkinfo	4
Record header	17
per significant value	3
Length: CHAR	Length without blanks at the end
Length: VARCHAR	≤ 253 bytes --> length value + 3 > 253 bytes --> $(\lceil \text{length value} / 4040 \rceil + 1) * 4 \text{ Kb}$
Length: NCHAR	$(\text{Length in code units without national blanks at the end}) * 2$
Length: NVARCHAR	Length in code units ≤ 126 --> Length in code units * 2 + 3 Length in code units > 126 --> $(\lceil \text{Length in code units} * 2 / 4040 \rceil + 1) * 4 \text{ KB}$
Length: NUMERIC	Number of positions without leading zeros
Length: DECIMAL	$\lceil \text{number of positions} / 2 \rceil$, where [...] means “next higher integer” (DECIMAL values are stored without leading zeros for spaces created for SESAM/SQL V2.2 and higher)
Length: INTEGER	4
Length: SMALLINT	2
Length: REAL	4
Length: DOUBLE PRECISION	8
Length: TIME	8
Length: DATE	6
Length: TIMESTAMP	14

Table 2: Disk storage requirements for primary data

When calculating the disk storage requirements for BLOBs, special conditions apply. A BLOB value consists of several rows which are stored in a specially structured table, known as a BLOB table (see the “[SQL Reference Manual Part 1: SQL Statements](#)”).

A BLOB table essentially consists of the following columns:

- OBJ_NR of data type INTEGER
- SLICE_NR of data type INTEGER
- SLICE_VAL of data type VARCHAR(31000)

- OBJ_REF of data type CHAR(237)

OBJ_NR and SLICE_NR together form the primary key of the BLOB table.

Within the table, a BLOB value is identified by its OBJ_NR. Each BLOB value has a row with the SLICE_NR 0 (see example). This row contains:

- an attribute description in the SLICE_VAL column (may be specified, for example, in a MIME or USAGE clause in CREATE TABLE OF BLOB)
 - the REF value for this BLOB in the OBJ_REF column. The length of OBJ_REF depends on the current table name, and must lie somewhere in the range 46 to 237 bytes.
- Each BLOB value also has a row with the SLICE_NR 2147483647 (high value), which merely contains the primary key. This row is exactly 28 bytes in length.

The BLOB value itself is split into segments of up to 31000 bytes, which appear in the SLICE_VAL column starting at SLICE_NR 1. The OBJ_REF column is not significant in these rows.

Apart from the space in the BLOB table, each BLOB also reserves storage space for its REF value in the referenced table. This REF value is stored in a column defined with “FOR REF”. It has a length of between 46 and 237 bytes.

Example

A BLOB value has a length of 100,000 bytes. The associated REF value is 70 bytes in length, and the attribute description 150 bytes. This value is stored in the BLOB table in the form of six rows:

OBJ_NR	SLICE_NR	SLICE_VAL	OBJ_REF
nnn	0	attribute_description	REF value
nnn	1	BLOB_value (31000 bytes)	NULL
nnn	2	BLOB_value (31000 bytes)	NULL
nnn	3	BLOB_value (31000 bytes)	NULL
nnn	4	BLOB_value (31000 bytes)	NULL
nnn	2147483647	NULL	NULL

Table 3: Example of a BLOB table

In accordance with VARCHAR conventions, SLICE_VAL occupies 8 * 4 KB in slices 1 through 3, but only 2 * 4 KB in slice 4. If we add the remaining columns and rows, we find that the BLOB occupies a total of 104 KB + 397 bytes in the BLOB table, and 73 bytes in the referenced table.

4.3 Metadata

The current version of SESAM/SQL includes a total of 28 system tables that define a database and the user data.

User definition of catalog	approx. 1500	4KB pages
per space	approx. 400	bytes
per table	approx. 1300	bytes
per column	approx. 700	bytes
per index	approx. 300	bytes

Table 4: Disk storage requirements for metadata

Example

10 table definitions with 100 columns require approx. 700 KB. 200 index definitions require approx. 60 KB.

4.4 Integrity constraints

Integrity constraints can be used to specify restrictions on the permitted values for columns. A distinction is made between the following types of integrity constraints:

- Referential constraint
- Check constraint
- NOT NULL constraint
- UNIQUE constraint
- PRIMARY KEY constraint

Following an update, (with the SQL statements INSERT, UPDATE or DELETE for example), the updated data must be in compliance with the restrictions defined by the integrity constraints. In SESAM/SQL, only the integrity constraints that are actually and directly affected by the data update are checked. This includes all integrity constraints defined on the updated table as well as all referential constraints in which that table is involved (as a referencing or referenced table).

An explanation of how integrity constraints are checked and of some specific aspects to be observed from a performance viewpoint are provided below. In general, the following assertions can be made:

- As far as possible, the primary key constraint must be defined for every table. This has no adverse effect on performance and, in fact, enables many requests to be very efficiently processed via the primary key index.
- Integrity constraints should be otherwise used only selectively and sparingly in the schema definition. As far as possible, only those integrity constraints that are absolutely essential for data consistency should be defined, and all other checks should be performed within the framework of the processing logic of an application program.

Referential constraint

This section explains the process by which referential constraints are checked. It also shows how the definition of an index can improve performance in some cases when checking a referential constraint.

In the following example, a foreign key has been defined for a referential constraint in the ORDER table, and the columns of a second CSTMR table (i.e. the referenced table) are assigned to it (e.g. ... FOREIGN KEY cstmr_no REFERENCES CSTMR (no) ...).

When the referential constraint is checked, a distinction is made depending on whether the referencing or referenced table is updated (update of a column, deletion or insertion of rows). The following overview shows how a referential constraint (that is not self-referencing) is checked for various statements.

Key to the used notation:

old indicates the value before the update

new indicates the new value

Example

Check for a referential constraint when the referenced ORDER table is updated:

1. UPDATE of ORDER.cstmr_no
The following is checked for each updated row of ORDER:
Does a row exist in the CSTMR table with
CSTMR.no = ORDER.cstmr_no(new)
2. DELETE
No check
3. INSERT
The following is checked for each inserted row of ORDER:
Does a row exist in the CSTMR table with
CSTMR.no = ORDER.cstmr_no(new)

Example

Check for a referential constraint when the referenced CSTMR table is updated

1. UPDATE of CSTMR.no
The following is checked for each row of ORDER:
Does a row exist in the CSTMR table with CSTMR. no = ORDER. cstmr_no

2. UPDATE ... WHERE CURRENT OF for CSTMR.no
The following is checked for the updated row of ORDER:
In the CSTMR table, does a row exist with ORDER.cstmr_no = CSTMR.no(old)
3. DELETE and does no secondary index exist for ORDER.cstmr_no
The following is checked for each row of ORDER:
Does a row exist in the CSTMR table with CSTMR. no = ORDER. cstmr_no
4. DELETE and does a secondary index exist for ORDER. cstmr_no
The following is checked for each row of ORDER which is to be deleted: Does a record exist in the ORDER table with ORDER.cstmr_no
5. INSERT
No check

The search for a row in the CSTMR table, where the value of the column number is given, can always be performed more efficiently than when using the primary key index. A comparison of (3) and (4) shows that a check for a referential constraint in the case of a DELETE statement on the referenced table can be improved significantly by first defining a secondary index for the referencing columns.

Check and NOT-NULL constraints

Check and NOT-NULL constraints are tested immediately (i.e. on-the-fly) on updating each individual row. Consequently, the test for these integrity constraints has no critical impact on performance.

UNIQUE constraint

The DBH automatically creates a suitable index for a uniqueness constraint. This index can also be an index that was explicitly defined by the user. The check for a uniqueness constraint is performed when updating the index in connection with an SQL update statement (INSERT, UPDATE, DELETE, MERGE). There are therefore no additional costs as opposed to any normal index.

PRIMARY KEY constraint

The check for a primary key constraint is performed implicitly when updating a row in the base table and is therefore not critical to performance.

4.5 Data distribution and allocation

The distribution of data on the various available disk devices must be taken into account for all files accessed during a SESAM/SQL session. In other words database files (catalog space and user spaces), backup copies for the database files, DA-LOG files, CAT-LOG files, CATREC file, and the DBH files, WA-LOG file, TA-LOG files, cursor files, and temporary work files.

The following aspects must be considered for the distribution of data:

- Efficient I/O
- Backup concept
- Lock granularity for DDL, SSL and utilities
- Allocation of indexes

Efficient I/O

TA-LOG files are I/O-intensive; in exceptional cases I/O bottlenecks can occur. TA-LOG files should therefore be located on a separate device which is as fast as possible. The size of the write unit for the TA-LOG files depends on the disk type. SESAM/SQL uses the maximum possible input/output length (64 to 160 KB).

Information on the maximum input/output length in half pages (2KB) can be obtained using the BS2000 command

```
/SHOW-PUBSET-CONFIGURATION PUBSET=<catid>,INFORMATION=*PUBSET-FEATURES.
```

If significant numbers of writes to the WA-LOG file are observed (with the performance monitor), the WA-LOG file should also be stored on a separate device which is as fast as possible. However, you should first try to reduce the number of these accesses to the WA-LOG file by increasing the values of the settings for the user data buffer and system data buffer (see [section “Dimensioning the user data buffer” on page 74](#) and [section “Dimensioning the system data buffer” on page 75](#))

The distribution of tables and indexes on spaces and of the spaces on devices must be uniform, i.e. spread the load evenly on all devices. The appropriate distribution will depend on the application, for example:

- Large tables that are accessed frequently and more or less uniformly across the table should be placed in a separate space (file) that is distributed over multiple devices.
- When several tables are accessed uniformly, such tables should also be placed in different spaces, and the spaces should then be distributed over multiple devices.

The allocation for the DA-LOG, CAT-LOG and CAT-REC files will basically depend on the backup concept. All three files may be located on the same device. The CAT-LOG and CAT-REC files do not present any I/O problems. I/O bottlenecks on temporary work files and cursor files can be prevented by using DBH options to set a sufficiently large buffer. If this is not possible, the temporary work files should be placed on a fast device.

Backup concept

To allow data to be restored in the case of a disk failure, the backup files (CAT-REC file, CAT-LOG files, DA-LOG files and backup copies of the spaces) should not be on the same devices as the data (catalog space, user spaces) (see also the [“SQL Reference Manual Part 2: Utilities”](#)).

Lock granularity for DDL, SSL and utilities

The lock granularity for DDL, SSL, and utilities is generally the space, and thus a file. Tables that are independent of one another, i.e. without referential relationships between them, can be placed on different spaces to enhance parallel processing. A DDL, SSL or utility on one table will not hinder access to a table in another space in such cases.

Allocation of indexes

As far as the lock granularity is concerned, it may be practical to place the indexes for a table on the same space as the table itself. From a performance viewpoint, however, it is probably better to allocate them to spaces on other devices.

In terms of recovery, a table and its associated indexes should be located on one space set, i.e. a set of collectively saved spaces that can also be collectively restored.

There is a further aspect to be considered in the context of recovery. Index updates are logged in the course of logical data saving. Subsequently the modifications logged in the logging files are applied to a backup during recovery. This index logging takes up CPU and I/O time, but can be bypassed if an index is on a space for which logical logging has been deactivated. In the case of a RECOVER, however, the index would need to be recreated after the space on which the table is located has been repaired.

4.6 Partitioned tables

You can distribute base tables to more than one user space. These tables are referred to as partitioned tables. Fundamental information on partitioned tables is provided in the “[Core manual](#)”.

The partitions of a partitioned table can be distributed to up to 16 user spaces (a non-partitioned table is stored on one user space). The size of all user spaces in a partitioned table is approximately the same as the size of the user space of the corresponding non-partitioned table if no further tables, partitions or indexes are stored on the user spaces.

Partitioned tables are generally used to structure user data via the primary key in accordance with particular criteria, e.g. on a monthly basis. A partition then contains the user data for a month. In many cases only the data of the current month is accessed. The data of the previous months need only be retained for auditing reasons. However, normally this data is not accessed.

Partitioned tables enable the data resource which is currently accessed to be reduced.

The smallest backup unit and repair unit in SESAM/SQL is a user space. The time required for backing up and repairing a partitioned table is reduced as multiple (small) user spaces can be backed up or repaired in parallel using one statement.

If, for example, only one partition is affected by a repair, the user space belonging to this can be repaired specifically.

In a partitioned table the primary key or part of the primary key can also be used as an “automatic count field” (COUNTING_FIELD), see the “[Core manual](#)”.

The lone count field or constant part of the primary key together with the count field is referred to as a “primary key area”. To enable the value of the count field to be determined for a primary key area in the INSERT and LOAD ONLINE statements, a search is made backward from the highest possible primary key value in this primary key area to the previous highest used primary key value. The value in the count field is incremented by 1. In conjunction with the constant part of the key this produces the new primary key value. If a primary key area extends beyond partition borders, determining the new value for an automatic count field can thus require more effort as multiple partitions must be read.

5 Aspects of database operation relevant to performance

The following sections describe the aspects of database operation which influence performance and which can be controlled by the system administrator when starting the DBH session or in ongoing DBH session.

5.1 Diagnosis and corrective measures for performance problems

The [table 5](#) shows an overview of the possible causes of various performance problems, together with the diagnostic options and the corrective measures that can be taken by the system administrator to eliminate each such problem.

Possible cause	Diagnostic option	Action
Prefetch buffer too small	SESMON mask PREFETCH-BUFFERS: hit rates (on memory requests) < 100%	Increase value of PREFETCH-BUFFER parameter in the configuration file
SQL plan buffer too small	SESMON mask SQL INFORMATION (plans)	DBH option SQL-SUPPORT/ Increase PLANS parameter
Transfer container too small	SESMON mask SYSTEM INFORMATION	Adapt DBH option (in ongoing operation with MODIFY-STORAGE-SIZE)
Work container too small	SESMON mask SYSTEM INFORMATION	Adapt DBH option (in ongoing operation with MODIFY-STORAGE-SIZE)
Block mode not active on prefetch cursor	Check the following items: 1. DECLARE CURSOR must contain a (correct) pragma --% PREFETCH n 2. DECLARE CURSOR must not contain a FOR UPDATE clause 3. The application's configuration file must contain the PREFETCH- BUFFER parameter 4. The maximum message length must be sufficient to transport more than one row	1. and 2.: Modify SQL application 3.: Specify parameter 4.: Increase message length

Table 5: Cause, diagnosis and corrective measures for performance problems

(part 1 of 2)

Possible cause	Diagnostic option	Action
Lock conflicts due to unnecessary indexes (e.g. index: sex; possible values: male or female)	SESMON mask TRANSACTIONS (Locked Transactions), view SYS_INFO_SCHEMA. SYS_LOCK_CONFLICTS	DROP index with no selectivity
Lock conflicts due to excessive TA nesting	SESMON mask TRANSACTIONS, view SYS_INFO_SCHEMA. SYS_LOCK_CONFLICTS	Try to reduce TA nesting; check whether consistency level is appropriate
Lock conflicts due to parallel utilities or parallel DDL	SESMON mask TRANSACTIONS, view SYS_INFO_SCHEMA. SYS_LOCK_CONFLICTS	Avoid utilities or DDL during peak loads
Front masking usually leads to performance degradation during Index processing	SESCOS - SESCOSP I/O statistics: Too many accesses on the system data buffer; SYS_DML_RESOURCES view of the SYS_INFO_SCHEMA	Avoid front masking or define additional restrictive conditions
Long I/O waiting periods	BS2000 software monitor openSM2: SYS_DML_RESOURCES view of the SYS_INFO_SCHEMA	Reduce load on disk device
Several physical I/Os, though same data constantly accessed	SESMON mask I/O; SYS_DML_RESOURCES view of the SYS_INFO_SCHEMA	Check DBH options SYSTEM and USER-DATA-BUFFER or BUFFER STRATEGY
Secondary index defective	INFORMATION_SCHEMA or SYS_INFO_SCHEMA (INDEXES or SYS_INDEXES view)	Utility statement RECOVER INDEX
SQL plans are not reused even though the same statements are sent	SESMON mask SQL INFORMATION (plans): plans generated despite available space in plan buffer.	Possibly same statements from different compilation units --> restructure modules

Table 5: Cause, diagnosis and corrective measures for performance problems

(part 2 of 2)

5.2 Work container

The “SYSTEM INFORMATION” mask of the performance monitor SESMON includes notes on the effectiveness of the work container setting. An inappropriate setting can be corrected during ongoing operation with the administration statement MODIFY-STORAGE-SIZE or using the DBH option WORK CONTAINER at the next DBH cold start.

SESMON outputs the following: The current container size and the current relative utilization as well as the maximum size according to the options and the previous maximum utilization relative to the option.

If utilization is relatively low, the size of the work container can be reduced.

When the utilization is very close to 100%, it is possible that statements cannot be executed due to a lack of space in the WORK CONTAINER and will be aborted with a SQLSTATE or CDML status. In this case the size of the WORK CONTAINER should be increased.

5.3 Transfer container

The “SYSTEM INFORMATION” mask of the performance monitor SESMON includes notes on the effectiveness of the transfer container setting. An inappropriate setting can be corrected during ongoing operation with the administration statement MODIFY-STORAGE-SIZE or using the DBH option TRANSFER-CONTAINER at the next DBH cold start.

SESMON outputs the following: The current container size and the current relative utilization as well as the maximum size according to the options and the previous maximum utilization relative to the option.

If utilization is relatively low, the size of the container can be reduced.

When the utilization is very close to 100%, it is possible that statements cannot be executed due to a lack of space in the TRANSFER CONTAINER and will be aborted with a SQLSTATE or CDML status. In this case the size of the TRANSFER CONTAINER should be increased.

5.4 User and system data buffer

The buffering technique of the DBH allows blocks that are required often in a session to be physically read from the disk just once and then made available as required. Since all blocks of a database cannot usually be buffered, a displacement mechanism is used to overwrite seldom used blocks when a bottleneck in the buffer occurs.

The quality of the buffer setting is reflected in the hit rate, i.e. the percentage of blocks found in the buffer. The use of a buffer improves performance only if a block is required often before it is displaced. The probability that a specific block will be requested often depends on the application on one hand, and on the function of the block (index block, administration block, user data block) on the other. For example, index blocks are required far more often than user data blocks.

In the case of a sequential search spanning a large user data resource, all index blocks could be displaced. To prevent this from occurring, SESAM/SQL makes a distinction between the user data buffer, in which the user data of tables is stored, and the system data buffer, which contains all access and administration data.

The DBH options `SYSTEM-DATA-BUFFER` and `USER-DATA-BUFFER` can be used by the system administrator to set the size of each buffer as required when starting the DBH. The size of each buffer can be modified by the system administrator in ongoing DBH session by means of the administration statement `RECONFIGURE-DBH-SESSION`.

5.4.1 Dimensioning the user data buffer

Under normal circumstances, a high hit rate cannot be achieved in the user data buffer, since access to the same user data is usually infrequent. The goal is to ensure that blocks read for READ and WRITE operations are still resident in the buffer at the time of the WRITE. This leads to a simple calculation:

Number of required blocks = $n * m$

where the placeholders have the following meaning:

n : Number of user data blocks read in a second

m : Number of seconds that pass between the time of reading and rewriting the block

A small additional reserve is required as a safety margin since n and m are not exact quantities.

Buffering of tables

If small frequently used tables or table sections are to be held resident in the user data buffer, their sizes must be added to the above formula. The space requirements for a row can be determined with the help of [table 2 on page 59](#) by adding 17 bytes for the record header, 3 bytes per significant value, and the value lengths of the individual columns depending on the data type.

A maximum of 4040 bytes are available per block.

Avoiding redundant I/Os on the WA-LOG file

If a block is displaced prematurely, i.e. before the next consistency point, the DBH will first write the physical before-image to the WA-LOG file. To avoid unnecessary I/Os on the WA-LOG file, the user data buffer must be large enough to accommodate the user data blocks of all parallel transactions (physical before-images and after-images).

5.4.2 Dimensioning the system data buffer

The goal of dimensioning the system data buffer is to achieve the highest possible hit rate. If possible, complete indexes should be made memory-resident; if not, at least their access paths.

The size of a secondary index (SI) can be roughly estimated as follows:

Number of blocks, SI base level =

$$n * (\text{value-length} + 7 + 2 * a) / (4040 * \text{density})$$

For spaces with a size up to 64 GB: Number of blocks, SI-index(*i*)=

$$\text{number of blocks SI-index}(i-1) * (\text{value-length} + 3) / (4040 * \text{density})$$

For spaces with a size of more than 64 GB: Number of blocks, SI-index(*i*)=

$$\text{number of blocks SI-index}(i-1) * (\text{value-length} + 4) / (4040 * \text{density})$$

The placeholders *n*, *a* and *i* stand for the following:

n = number of values

a = average number of references per value

i = index level; *i*=0 corresponds to the base level

The following rule of thumb applies to the size of the primary key index:

For spaces with a size up to 64 GB: Number of blocks of the base level =

$$\text{Number of ZD blocks} * (\text{length of primary key} + 6) / (4040 * \text{density})$$

For spaces with a size of more than 64 GB: Number of blocks of the base level =

$$\text{Number of ZD blocks} * (\text{length of primary key} + 7) / (4040 * \text{density})$$

The following applies to the additional PSI levels:

For spaces with a size up to 64 GB: Number of blocks of PSI level *n*+1 =

$$P_n * (\text{length of primary key} + 6) / (4040 * \text{density})$$

For spaces with a size of more than 64 GB: Number of blocks of PSI level *n*+1 =

$$P_n * (\text{length of primary key} + 7) / (4040 * \text{density})$$

where:

P_n = Number of blocks of PSI level *n*

Avoiding redundant I/Os on the WA-LOG file

As in the case of the user data buffer (see [page 74](#)), a small reserve is also required for the system data buffer to prevent unnecessary I/Os on the WA-LOG file.

5.4.3 Example for the calculation of buffer sizes

The following sample computation shows how the user data and system data buffers must be dimensioned when a specific table is to be held resident in memory.

Example

A table with 5000 rows of 20 columns and a net data length of 500 bytes is to be maintained in the buffer. The block utilization is equal to eighty percent; the length of the primary key is 20 bytes.

The number of user data blocks required can be calculated as follows:

$$5000 * (17 + 20 * 3 + 500) / (4040 * 0.8) = 893 \text{ blocks}$$

The DBH option USER-DATA-BUFFER must be increased to the value $893 * 4 = 3572$.

The number of secondary data blocks required is calculated as follows:

For spaces with a size up to 64 GB:

$$\text{Database translation table: } 5000 / 1346 + 1 = 5$$

For spaces with a size of more than 64 GB:

$$\text{Database translation table: } 5000 / 1010 + 1 = 6$$

Primary key index:

For spaces with a size up to 64 GB:

$$\text{lowest level: } 893 * (20 + 6) / (4040 * 0.8) = 8$$

$$\text{highest level: } 8 * (20 + 6) / (4040 * 0.8) = 1$$

For spaces with a size of more than 64 GB:

$$\text{lowest level: } 893 * (20 + 7) / (4040 * 0.8) = 8$$

$$\text{highest level: } 8 * (20 + 7) / (4040 * 0.8) = 1$$

The DBH option SYSTEM-DATA-BUFFER must be increased to the following value:

For spaces with a size up to 64 GB:

$$(5+8+1) * 4 = 56$$

For spaces with a size of more than 64 GB:

$$(6+8+1) * 4 = 60$$

5.5 DBH option RESTART-CONTROL

With the TALOG-LIMIT and BUFFER-LIMIT parameters the DBH option RESTART-CONTROL offers a way of influencing the following performance factors:

- The frequency of physical write I/Os to the spaces
- The duration of a potential restart
- The runtime of some administration commands

Selecting as low a value as possible for TALOG-LIMIT and BUFFER-LIMIT enables the runtime of some administration commands (see also [chapter "Performance-related aspects of administration statements" on page 233](#)) or of a potential restart to be shortened.

Consequently blocks which are logically written are physically written more quickly to disk. Fewer logical write operations are collected in a block before the block is physically written. The regular write load for the disks on which the spaces reside increases.

This effect can be observed in the SESMON mask I/O in the number of "Phys. Write" in the columns "System Data Buffer" and "User Data Buffer", see the "[Database Operation](#)" manual. The values are actually made available per space in the SYSLST output.

5.6 Cursor buffer

The cursor buffer in combination with the internal cursor files provides a temporary storage medium for intermediate result sets. The following information is stored in the cursor buffer and/or the cursor files:

- database keys; for large matching quantities, in bitlist representation
- result set when sorting
- values for the index update for aggregate functions
- information on recovery units for the RECOVER function

The size of the entire cursor buffer is specified in the DBH option BUFFER-SIZE and the size of a buffer frame is specified in the DBH option FRAME-SIZE. The default size of the buffer frame is 4 Kb. BUFFER-SIZE/FRAME-SIZE buffer frames are created with the size specified by FRAME-SIZE.

Increasing the size of the buffer frames (DBH option FRAME-SIZE) without simultaneously increasing the size of the entire cursor buffer (DBH option BUFFER-SIZE) leads to a reduction in the number of buffer frames.

If the number of buffer frames is reduced, more data per SVC is written when the buffer frame size is larger.

A larger buffer frame should be specified if large sets of data are expected to be returned when selecting using inverted attributes.

Unconditional writes to the cursor file

In the following cases, the DBH will write to an internal cursor file even if the cursor buffer is large enough:

- when sorting if the set to be sorted is larger than a buffer frame.
- when passing values to the service task during a Recover.

“I/O” output mask of the performance monitor

The “I/O” mask of the performance monitor SESMON contains information on logical and physical read and write access operations (I/O).

If the indicated hit rate is low, the system administrator can try to improve it by increasing the size of the cursor buffer (DBH option BUFFER-SIZE) at the next DBH cold start or with the administration statement RECONFIGURE-DBH-SESSION.

If the system administrator has increased the buffer frame size (DBH option FRAME-SIZE) and the number of physical I/Os increases because of it, then the FRAME-SIZE should be set to its original value or the size of the cursor buffer should be increased (DBH option BUFFER-SIZE) because more buffer frames have been forced out.

“Service Orders” output mask of the performance monitor

The SESMON performance monitor provides information in the “Service Orders” mask on the sorting requests and DDL and utility statements processed in the service task.

If too many sorting requests are processed in the service task, the system administrator should increase the size of the buffer frames (DBH option FRAME-SIZE) so that more sorts can be executed in the DBH task. If the set of data to be sorted is smaller than a buffer frame, the sorting is executed in the DBH task (see [section “Influencing sorting” on page 98](#)).

5.7 Number of threads

The current utilization of threads is displayed in the “SYSTEM INFORMATION” mask of the performance monitor SESMON. If this mask consistently shows the existence of free threads, the load size of the DBH can be reduced by specifying a lower number of threads (with the DBH option THREADS). On the other hand, if the load size of the DBH does not present a problem, the presence of unused threads does not degrade performance.

The number of threads determines the number of requests that the DBH can process concurrently. It should be taken into account that in SESAM/SQL even interrupted requests, e.g. due to a transaction lock, can occupy a thread. Too few threads can lead to a backlog of requests. The transaction is rolled back when the last thread reaches a transaction lock.

When CREATE INDEX statements are executed for partitioned tables, the DBH option THREADS should be greater than or equal to the number of partitions. See [section “Optimized index creation for partitioned tables” on page 229](#).

The number of threads can be corrected at the next DBH cold start or using the administration statement RELOAD-DBH-SESSION.

5.8 Number of DBH tasks and multitasking

Balancing the load

The load balance, i.e. the distribution of the load amongst the DBH tasks by the requests, is done automatically in SESAM/SQL and cannot be changed by the administration.

When doing so, the load is not distributed equally across all DBH tasks, but an attempt is made to process the existing load with as few tasks as possible. If the load increases, the start task will include additional DBH tasks to process the requests. When the load drops down again, these tasks deactivate themselves.

DBH-TASKS option

The SESMON performance monitor provides information on the quality of the setting of this option in the "TASKS" DBH mask. An unfavorable setting can be corrected during the next cold start of the DBH or with the administration statement RELOAD-DBH-SESSION using the DBH option "DBH-TASKS".

SESMON outputs the following for each DBH task:

- the TSN of the task
- the number of threads currently bound to this task
- the number of requests waiting for this task
- the number of wait states due to task I/O in the last measurement interval
- the number of I/Os triggered in the task in the last measurement interval
- the quotients of the two values
- the CPU time utilized by the task (the value can only be output if SESMON is running under the same user ID as the DBH due to technical reasons).

If DBH tasks that have hardly used any CPU resources are shown in the DBH tasks mask at the end of the table, this indicates that these tasks are superfluous.

If requests are waiting for the task in the DBH tasks mask for all tasks shown, this indicates that the number of DBH tasks is currently too low.

The following should be noted in order to make optimum use of multiprocessors:

- The number of DBH tasks should generally (when there are more than 3 CPUs) be smaller than the number of CPUs.
- The ratio of the number of CPUs to the number of DBH tasks depends on the distribution of the CPU time required by the DBH and the application.
- If the number of DBH tasks is lower than the number of CPUs, a fixed priority is to be assigned to the DBH tasks.
- The user tasks are to be assigned a fixed priority, if necessary, but it should be 3 to 5 percentage points higher than the priority of the DBH tasks so that the DBH tasks will always find work to do.

5.9 Suborders

The number of suborders used and the maximum number of suborders available are shown in the "SYSTEM INFORMATION" mask of the performance monitor SESMON. If the number of available suborders is significantly greater than the maximum number used, there will be no runtime losses for the DBH; however, it means that main memory usage is unduly large.

5.10 Plan buffer

Before SQL statements are first executed, they are transformed into a uniform representation called the SQL access plan (see [section “Structure and control of the optimizer” on page 24](#)). This internal representation of the SQL statement (referred to in brief as a plan below) is stored in a DBH-specific memory area called the plan buffer. When the same SQL statement is executed again, the already created plan can thus be reused. Since the cost of creating a plan for an SQL statement is relatively high as opposed to its execution, the generation of plans for SQL statements must be reduced to a bare minimum.

The size of the plan buffer depends on the following DBH options:

- DBH option COLUMNS
- PLANS parameter of the DBH option SQL-SUPPORT
- DBH option USERS (to a lesser degree)

In order to fine-tune the dimensions of the plan buffer, it is useful to know some inter-relationships. These are explained in the following sections.

5.10.1 Assignment of plans to SQL statements

Every plan belongs to one or more SQL statements. Conversely, there may be one, several, or no plan for an SQL statement. The number of plans generated per SQL statement depends on the type of SQL statement, as indicated below:

- No plans are created for transaction-control statements (COMMIT WORK, ROLLBACK WORK, etc.) and session-control statements (SET CATALOG, SET SCHEMA etc.).
- There is exactly one plan for static DML, DDL, SSL, USER and UTILITY statements that do not reference cursors.
- There is exactly one plan for a static cursor. This plan can be accessed with the associated cursor statements (OPEN, FETCH, CLOSE, etc.), i.e. one plan belongs to several statements.
- There are two plans for update statements on a cursor (DELETE ... WHERE CURRENT OF ..., UPDATE ... WHERE CURRENT OF ...): one plan for the statement plus one plan for the cursor (see above).
- In the case of the dynamic statements PREPARE and EXECUTE IMMEDIATE, plans are generated for the SQL statements to be prepared. There is, however, no plan for the dynamic SQL statement itself (PREPARE, EXECUTE IMMEDIATE, EXECUTE). The plan generated via PREPARE can be accessed with EXECUTE or with statements for dynamic cursors (OPEN, FETCH etc.) if a cursor plan was created using PREPARE.

Two plans are created for two static SQL statements within a single SQL module even if the statements are absolutely identical. On the other hand, two different dynamic statements in a module can be assigned to the same plan if they refer to identical statement texts. Two statement texts are regarded as being identical if all the characters of the texts including the pragmas are identical and the same default settings apply with regard to the database name and schema name. No normalization (e.g. compression of spaces) is performed.

Shareable plans

An SQL module can be executed by several different requesters.

Static SQL statements are generally not requester-specific. That is why it is enough to generate only one plan per statement that will be used by all requesters.

The dynamic statements PREPARE and EXECUTE IMMEDIATE depend on the current context of the requester (on the contents of the statement variables). The SQL statement contained in a statement variable of this type can refer to the database name or schema name currently set or to temporary views.

Two dynamic statements whose statement variables contain identical text can also use the same plan if the preset database and schema names are identical.

This then makes all plans shareable.

5.10.2 Life-span of plans

A plan is retained in the plan buffer as long as possible so that it can be reused when the associated SQL statement is executed again. However, if the underlying catalog information for a plan is modified by SQL-DDL, SQL-SSL or utility statements, the plan must be deleted. When a database is physically closed, all plans for this database are deleted.

Depending on the associated SQL statement, a plan may be required for various periods:

- For static cursors, from OPEN CURSOR until CLOSE CURSOR and, in addition, as long as at least one requester (per STORE) has stored a cursor position.
- For other static SQL statements and EXECUTE IMMEDIATE only while this statement is being processed.
- For dynamic cursors until the end of the transaction and if no further cursor position is stored.
- For other dynamic SQL statements until the end of the transaction and, in addition, as long as at least one requester (per STORE) has stored a cursor position.

A plan that is no longer required at the moment may be deleted to make space for a new plan. Wherever possible, SESAM/SQL selects the plan which has not been used for the longest period of time.

5.10.3 Number of plans

The system administrator uses the PLANS parameter of the DBH option SQL-SUPPORT to specify the minimum number of parallel SQL access plans for a DBH session. With respect to the number of plans, two questions can be asked.

1. What is the total number of plans in existence, taking all requesters of the DBH session into account?
2. What is the number of plans required simultaneously?

The answer to the question regarding the total number of plans (1) gives a maximum value for the PLANS parameter. In this case it is assumed that each plan created within a DBH session occupies memory which had been free up to that point and that it remains in the plan buffer until the end of the DBH session, irrespective of the length of time it has already existed and whether it has ever been reused. Depending on the application scenario, it is possible that the plan buffer will be far too large and will occupy resources unnecessarily.

The answer to the question regarding the maximum number of plans required simultaneously (2) gives a lower threshold value for the PLANS parameter. If the plan buffer is smaller than this value, it can be occupied fully by plans which must not be deleted, for instance because they belong to a cursor which is currently open. An attempt to create a plan for a further SQL statement would then be rejected because of a resource bottleneck

(plan buffer overflow). It is, however, also possible that the following undesirable effect is observed: a plan has to be displaced and then is required again shortly thereafter and has to be regenerated, thus displacing another plan etc.

It is therefore necessary to take a middle path to avoid the need to constantly regenerate plans which are required regularly. The following should be taken into account:

- The number of plans reused at regular intervals. One should take into account both those plans reused by the same requester and shareable plans used by different requesters (e.g. by a UTM application with a large number of terminals).
- The number of plans which are not reused, but which can delete a plan which is still needed if the interval between accesses to the required plan becomes too large.
- Number of plans which have a long life span (see [section “Life-span of plans” on page 84](#)).

Static SQL statements

To determine the number of plans for static SQL statements, it is necessary to have some knowledge of all the application programs in a DBH session. In particular, details must be known regarding which SQL applications run in parallel in the configuration concerned.

The following values must be determined:

- the number of SQL statements in a module
- the number of plans which belong to this SQL module (see also [section “Assignment of plans to SQL statements” on page 82](#))
- the number of requesters who use the SQL module in the configuration
- the resulting number of plans required for a module
- the number of plans (for static statements) for the entire configuration

Dynamic SQL statements

In the case of dynamic statements, it is only possible to estimate the total number of plans. Since the statement text is generally not known beforehand, a user can, for instance, create any number of different plans in succession with a single EXECUTE IMMEDIATE statement. It is, however, possible to imagine applications where the SQL statement texts are not entered spontaneously at the terminal, but are rather taken from a restricted pool of statements.

PLANS parameter

Before finally setting an appropriate value for the PLANS parameter of the DBH option SQL-SUPPORT, you should define the resulting plan buffer size (see [section “Size of the plan buffer”](#)). The size of the plan buffer can then be optimized using the performance monitor (see [section “Optimizing the size of the plan buffer” on page 87](#)).

If the available information is not sufficient to make a theoretical estimation of the PLAN parameter, it is, of course possible to first use the standard options and to determine the optimum size with the performance monitor.

The PLANS parameter of the DBH option SQL-SUPPORT is, however, only one factor affecting the number of plans which can be stored in the plan buffer. The average amount of memory made available for a plan is based on the DBH option COLUMNS and is generally sufficient, even for complex plans. If relatively simple SQL statements are involved, or if the COLUMNS options has been set too high, considerably more plans can be accommodated in the plan buffer than specified in the PLANS parameter.

5.10.4 Size of the plan buffer

The size of a plan depends on the SQL statement to which it belongs and to the associated schema information. In general, the greater the number of tables and columns addressed by an SQL statement, the larger the associated plan will be. A plan will also increase in size in proportion to the number of integrity constraints which need to be checked when the relevant SQL statement is executed.

SESAM/SQL calculates the approximate size of the plan buffer from the average plan size to be expected, taking into account the DBH options COLUMNS, USERS and the PLANS parameter of SQL-SUPPORT. Depending on the precise situation surrounding the application, the actual plan sizes can vary considerably.

The plan buffer is split into two memory areas: the primary buffer containing the plans and the secondary buffer containing the related administration information.

The [table 6 on page 87](#) shows some guideline values for the size of the plan buffer, depending on the DBH options described above (under the assumption that USERS=24). The size of the plan buffer for the current DBH session can be roughly interpolated from these guideline values. The values can subsequently be optimized using the performance monitor (see also [section “Optimizing the size of the plan buffer”](#)).

PLANS	COLUMNS			
	256		1024	
	Primary buffer	Secondary buffer	Primary buffer	Secondary buffer
1	150 KB	27 KB	591 KB	92 KB
10	468 KB	74 KB	1.78 MB	268 KB
70	1.73 MB	260 KB	6.28 MB	930 MB
100	2.36 MB	353 KB	8.53 MB	1.26 MB
1000	21.3 MB	3.14 MB	76.0 MB	11.2 MB
10000	210 MB	31.0 MB	751 MB	111 MB

Table 6: Size of the plan buffer depending on the DBH options

In the default values of the DBH options (SQL-SUPPORT/PLANS=70, COLUMNS=256, USERS=24), the total size of the plan buffer is 2.0 Mb. All the values apply if the USERS option is set to 24. The size of the secondary buffer increases or decreases by approximately 200 bytes for each user above or below this value (the primary buffer is not dependent on the value of the DBH option USERS).



In subsequent versions of SESAM/SQL, it is possible that the plan sizes will be affected by internal changes. This could have a corresponding effect on the size of the plan buffer. For this reason, the values shown in [table 6](#) are subject to possible changes in future versions.

5.10.5 Optimizing the size of the plan buffer

The SQL-INFORMATION mask in the SESMON performance monitor allows you, among other things, to determine the following values:

- the size of the plan buffer (primary buffer)
- the number of plan accesses (corresponds to the number of times the relevant SQL statement is executed)
- the number of plan generations
- the number of plans in the plan buffer (primary buffer)
- the space occupied in the plan buffer (primary buffer)
- the size of the secondary buffer

The first time an SQL statement is executed, the corresponding plan is generated. When the SQL statement is subsequently executed, the plan generated previously can be reused, provided that the space occupied by this plan has not been reserved by another plan in the interim.

The plan buffer has the optimum size if, after an initial phase, the number of plan generations is significantly lower than the number of plan accesses. In this case, a large number of plans are generated only once and then executed a number of times without being deleted. These can be either plans that are used by several requestors or plans that are executed a number of times by the same user, e.g. in loops.

It is, for instance, an entirely different situation if the majority of applications running involve entering SQL statements spontaneously in a dialog. In this case, it is likely that the dynamic statements will be processed using different statement texts, so that new plans constantly need to be generated and it is not possible to optimize the reuse of plans.

When dimensioning the plan buffer, you should make sure that sufficient space is available for all the plans which are required simultaneously, so that no SQL statement is rejected as a result of a resource bottleneck. In particular, it should be noted that a plan generated with PREPARE remains in memory at least until the end of the transaction (if the same requester does not execute a further PREPARE for the same statement name). Transactions which are kept open for an excessive period are thus just as bad as the parallel preparation of a number of statements in cases where consecutive processing (using the same statement name) would be perfectly possible.

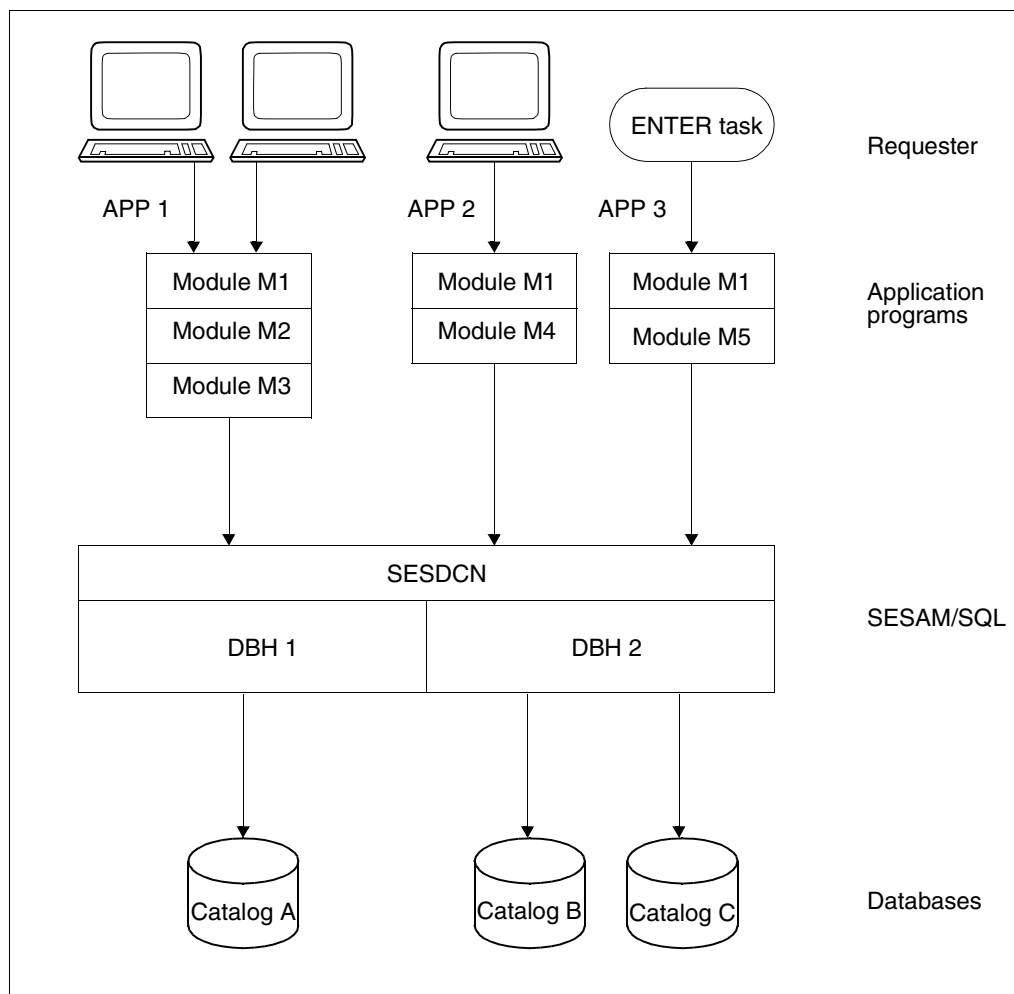
Plans which are not used for a long time are only deleted if the space they occupy is needed by another plan. This means that a fully occupied plan buffer does not necessarily indicate that the buffer is too small. The space occupied may be equal to the overall size of the plan buffer (primary buffer) or marginally lower than the overall size.

If the SQL statements or the schema information of the databases are changed, the PLANS parameter of the DBH option SQL-SUPPORT may need to be adapted accordingly. In the case of special application scenarios, the number of plans and the average plan size can be determined from the above indicators.

5.10.6 Estimating the minimum number of plans in a sample application

Example

The following illustration shows a sample application in which two DBHs are involved. An optimum value for the PLANS parameter of the DBH option SQL-SUPPORT is to be estimated in each case.



The application program APP1 is an OLTP application used by a maximum of 300 terminals concurrently.

The application program APP2 is a decision support application used by 20 terminals, each with a variety of individual SQL queries.

The application program APP3 is a batch application started by one BS2000 ENTER process only.

The OLTP application and the decision support application are used primarily during working hours. The batch application is started once each night.

The database catalog A is assigned to DBH 1 and catalog B and catalog C are assigned to DBH2.

Modules M1, M2 and M3 process only catalog A using static SQL statements. Module M4 contains only dynamic SQL statements which can address any of the three catalogs, depending on the statements prepared. Module M5 contains statements which reference catalogs A and B, in a ratio of 50:50.

For static statements, the number of plans can be determined simply by counting them. In the case of dynamic statements, it is only possible to estimate the number of plans required. (A single EXECUTE IMMEDIATE statement executed in a loop can issue any number of different statement texts and thus different plans.)

In the above example, the assumption is made for module M4 that each requester enters an average of 60 (20 per catalog) spontaneously formulated, and thus different statements in a dialog. The rest of the SQL statements are taken from a pool of 30 statement texts defined in the program (10 for each catalog).

The key values described are determined before the minimum number of plans can be estimated:

Number of SQL statements		Plans		User	Generated plans	
		Total	Individual		DBH 1	DBH 2
M1	30	20	-	321	20	-
M2	200	150	-	300	150	-
M3	12	10	-	300	10	-
M4	100	90	60	20	410	820
M5	1200	1000	-	1	500	500
				Total	1090	1320

Table 7: Key figures for estimating the minimum number of plans

In [table 7](#), the number of requesters is derived by adding together all the application programs in which the modules are used. The number of generated plans, on the other hand, is derived from the number of individual plans multiplied by the number of requesters plus the number of shared user plans.

Number of plans in the configuration for “DBH 1”

The total number of plans which belong to the configuration for “DBH 1” is, in theory 1090. However, not all the plans are required at the same time (see above). The plan buffer only needs to provide sufficient space for the plans which are required simultaneously. Since the plan buffer would be rather large if the DBH option PLANS was assigned the value 1090 (approx. 18 Mbytes for the primary and secondary buffer together if COLUMNS=256 and USERS=321), a smaller value should be determined.

A reasonable approach to finding an initial setting is to take the number of shared plans which are used relatively regularly. In our example, these are the shareable plans in modules M1, M2 and M3 (a total of 180) and the shareable user plans in M4 which refer to catalog A (a total of 10), giving an overall total of 190. In addition, it must be assumed that a further plan, caused by a spontaneously formulated SQL statement and which is unlikely to be accessed by any other user, will be used by each terminal of the decision support application. If you make a further allowance for concurrently used user plans of the batch application, 220 should be a suitable value for the PLANS parameter.

On the other hand, the average amount of space made available for a plan is based on the DBH option COLUMNS and is generally sufficient, even for complex plans. If relatively simple SQL statements are generally used or if the setting for COLUMNS is too high, there is room for a greater number of plans in the buffer than indicated by the PLANS parameter.

Number of plans in the configuration for “DBH 2”

The number of plans generated in the configuration for “DBH 2” comes to a total of 1320. The correct approach to finding an initial setting for the PLANS option is to a large degree dependent on the structure of the batch program (or module M5). However, since the program is only used by one requester, the overall number of plans is not critical with regard to optimization. Only the minimum number of plans used simultaneously in the batch application (see [section “Life-span of plans” on page 84](#)) is to be determined. If you add the 20 clearly defined plans from module M4 which reference catalogs B and C (which are probably used regularly), this gives a reasonable lower limit for the PLANS parameter in the configuration for “DBH 2”. However, the only reliable way to decide whether the plan buffer really has a sensible size is to use the performance monitor.

5.11 Prefetch buffer for block mode

The prefetch buffer manages all the partial result sets (blocks) from cursors opened in block mode. A block contains the values of the result rows and management information on these values. The user specifies the size of the prefetch buffer with the PREFETCH-BUFFER parameter in the configuration file for the application.

5.11.1 Size of the prefetch buffer

The space available in the prefetch buffer is occupied by the following data:

- user data (blocks)
- memory management data (for managing the memory areas for the user data)

The memory requirements for a block depend on the following factors:

- number of columns in the cursor
- data types of the individual columns
- current values of the buffered result rows
- number of result rows in a block.

The memory area required for a block can be broadly estimated on the basis of the following formula:

$$376 \text{ bytes} + sw + f * (40 \text{ bytes} + 10 \text{ bytes} * s)$$

where f = number of columns in the cursor

s = number of result rows in the block

sw = memory requirements for all the values of the result rows contained in the block

The value for sw can be determined as follows:

1. For cursors with fixed-length columns:

$$sw = s * \left(\sum_{i=1}^f l(i) \right)$$

$l(i)$ = length of the column value in column i

2. For cursors with variable-length columns:

$$sw = \sum_{i=1}^f \sum_{j=1}^s l(i,j)$$

$l(i,j)$ = length of the column value in column i in result row j

A part of the prefetch buffer is required for memory management. For this reason, it is possible that SESAM/SQL will identify a part of the buffer as being occupied (by SESMON, for instance) although the buffer does not as yet contain any user data. This can be expected if only a very small buffer (a few Kb) is provided.

The space required for memory management depends on the number of blocks to be managed simultaneously. A rough estimate of the space required for management gives the value $a * 32$ bytes, where a is the number of blocks to be managed concurrently.

To be more precise, the following applies:

The memory area required for memory management comprises two parts:

- a central data structure with a fixed length of 320 bytes
- a variable length area for managing the individual blocks. A structure with a length of 32 bytes is required for each block.

When the prefetch buffer is initialized, one percent (at least 96 bytes and at most 4 Kbytes) of the memory is used for the variable length area.

The variable part of the memory management is extended dynamically as required. The degree of expansion depends on the free memory available and the mean memory requirements for a block at the time of the expansion. The degree of expansion can be calculated as follows:

$$32 \text{ bytes} * \frac{\text{free memory}}{\text{average memory requirement for one block}}$$

In this case also, the memory area is expanded by at least 96 bytes and at the most 4 Kbytes. Once the memory management area has been expanded, it cannot be reduced again.

In a TIAM environment, each SQL user has their own prefetch buffer. In this environment, it is possible to calculate the memory requirements for the prefetch buffer precisely on the basis of the cursors used in block mode.

In a UTM environment, all SQL users for an application have a common prefetch buffer. This means that the memory requirements are not determined by the structure of the program unit, but also by the dynamic load on the application. In this environment in particular, it is recommended that you optimize the size of the prefetch buffer using the performance monitor SESMON.

The user can resize the prefetch buffer as required using the value of the PREFETCH-BUFFER parameter in the configuration file. The new value takes effect the next time the application is started. It is not possible to resize the prefetch buffer while the application is running.

5.11.2 Optimizing the size of the prefetch buffer

The prefetch buffer for an application has an ideal size when all memory requests can be satisfied immediately and when maximum memory utilization is as close as possible to 100%. This can be seen in the SESMON mask PREFETCH-BUFFERS if the hit rate for an application in all memory classes where requests have been made is 100% and the value for the maximum memory utilization (in the "Max. Occ." column) is in the region of 100%.

The ideal size for the prefetch buffer for a TIAM application or UTM application can be determined as follows using the SESMON performance monitor:

1. The prefetch buffer is initially dimensioned large enough (e.g. 4 Mb) to ensure that as many memory requests as possible can be satisfied immediately.
2. After the task or application has been started, it is left to work using prefetch until the value of "Max. Occ." (maximum memory utilization) in the SESMON mask PREFETCHBUFFERS no longer changes.
3. If the hit rate in one or more memory classes is less than 100%, the prefetch buffer was not large enough to satisfy all memory requests and should, if possible, be enlarged (e.g. doubled).
 - If it is possible to enlarge the prefetch buffer, you should do so and return to step 2.
 - If it is not possible to enlarge the prefetch buffer, the procedure is complete.

4. If the hit rates in all memory classes in which memory requests occur are 100%, it is possible to determine the ideal size for the prefetch buffer on the basis of the value for the maximum memory utilization ("Max. Occ."):
 - If the value for "Max. Occ." is already ideal (e.g. > 90%), the procedure is complete.
 - If the value for "Max. Occ." is between 10% and 90%, the ideal size for the prefetch buffer is the same as the percentage of the size currently set for the prefetch buffer as indicated in the column "Max. Occ.". An additional safety margin should also be added. With UTM applications, this safety margin should be generous, since the prefetch buffer cannot be reorganized with UTM applications and the free memory could therefore be heavily fragmented, with the result that it may not be possible to satisfy buffer requests despite sufficient free memory space. Once the prefetch buffer size has been changed, you should return to step 2.
 - If the value for "Max. Occ." is between 1% and 9%, it is not possible to determine the ideal value for the buffer size accurately, since the value for "Max. Occ." only has one significant digit. In this case, it is recommended that the prefetch buffer is reduced to one tenth of its current size and that you return to step 2.
 - If the value for "Max. Occ." is 0%, the prefetch buffer should be reduced to one hundredth of its current size and you should return to step 2.

Example

A prefetch buffer size of 1 Mbyte is set for the UTM application "MYAPPL". After the application has run for a certain length of time, the SESMON mask PREFETCH-BUFFERS displays the following:

```
=====
>>> PREFETCH-BUFFERS <<<   <ver>                               CNF: X      Time: <time>

Appl./                               H i t r a t e                               Perc. Max.
TSN      T <=1KB <=2KB <=4KB <=8KB <=16KB <=32KB <=64KB >64KB  Occ. Occ.
.
.
.
MYAPPL  U  --%  100%  100%  --%  100%  --%  --%  --%  3%  7%
.
.
.

>>> INTERRUPT WITH KEY K2 <<<
=====
```

Since the value of "Max. Occ." is below 10%, 100 Kb is set as the new size of the prefetch buffer. After the application has been restarted using this value and has run for a certain period of time, SESMON displays the following values:

```
=====
>>> PREFETCH-BUFFERS <<<   <ver>                               CNF: X       Time: <time>

Appl./                               H i t r a t e                               Perc. Max.
TSN      T <=1KB <=2KB <=4KB <=8KB <=16KB <=32KB <=64KB >64KB  Occ. Occ.
.
.
.
MYAPPL  U  --% 100% 100%  --% 100%  --%  --%  --%  47% 76%
.
.
.

>>> INTERRUPT WITH KEY K2 <<<
=====
```

76% of the current prefetch buffer size of 100 Kb (i.e. 76 Kb) should now be the ideal size. If a safety margin of approx. 10% is added, the resulting prefetch buffer size is 84 Kb. After the application has been restarted with this new prefetch buffer size, SESMON returns the following values:

```
=====
>>> PREFETCH-BUFFERS <<<   <ver>                               CNF: X       Time: <time>

Appl./                               H i t r a t e                               Perc. Max.
TSN      T <=1KB <=2KB <=4KB <=8KB <=16KB <=32KB <=64KB >64KB  Occ. Occ.
.
.
.
MYAPPL  U  --% 100% 100%  --% 100%  --%  --%  --%  83% 91%
.
.
.

>>> INTERRUPT WITH KEY K2 <<<
=====
```

This shows that a very good memory utilization has been achieved and that the procedure for determining the ideal memory size can be completed.

5.12 Service tasks

CPU-intensive actions such as the sorting of intermediate result sets or the execution of some utility functions are relocated by the DBH to service tasks. The system administrator can influence the strategy of the DBH by means of the DBH option SERVICE-TASKS or during ongoing operation with the administration statement MODIFY-SERVICE-TASKS.

5.12.1 Influencing the service task strategy

The INITIAL parameter enables the system administrator to set how many service tasks are to be initiated immediately on starting up the DBH or during ongoing operation. If the value selected for this parameter is too high, the loading procedure will be delayed, since the tasks compete with one another on starting up for accesses on the memory pool and module libraries. The default value of INITIAL=1 ensures that one service task is made available immediately.

The MAXIMUM parameter enables the system administrator to set the maximum number of service tasks that can be concurrently used by the DBH in the session. The utilization of service tasks can be monitored by the system administrator via the "SERVICE TASKS" output mask of the performance monitor. If the value for "Orders-Not-Processed" exceeds 0, there is a backlog of orders. The value of MAXIMUM should be raised during ongoing operation with the administration statement MODIFY-SERVICE-TASKS or in the next session.

The JOBCLASS parameter can be used by the system administrator to specify the job class with which the service tasks are to be started by the DBH. The job class can also be set during ongoing operation with the administration statement MODIFY-SERVICE-TASKS. The selected job class should be one with a time limit of at least 1000 CPU seconds; otherwise, the service tasks crash too often due to the CPU timeout. Affected SQL statements are then aborted with SQLSTATE 81SS0.

If the JOIN entry of the DBH ID is NO-CPU-LIMIT=YES, the service tasks are started with CPU-LIMIT=*NO and therefore have no limit to the CPU time they can use.

The JOIN entry of the DBH ID should allow START-IMMED=YES to ensure that the service tasks can be started with START=*IMMED and do not languish in a wait state for too long in order type T1.

When CREATE INDEX statements are executed for partitioned tables, a sufficient number of service tasks must be available for parallel processing. See [section "Optimized index creation for partitioned tables" on page 229](#).

5.12.2 Influencing sorting

When a statement is processed, input data, intermediate result sets, result cursor files, etc., must be sorted in the DBH. To relieve the load on the DBH, these sorts are performed in service tasks.

The type and speed of sorting depends on the following factors:

- Number of sort criteria
- sorting order in the module SESFS90 (see the “Database Operation” manual)
- WORK-FILES parameter
- RECORDS-PER-CYCLE parameter
- the CORE parameter of the SORT default procedure
- JOIN entry of the DBH ID

Number of sort criteria

The DBH obtains the sort criteria from the ORDER BY specification of an SQL statement or the S subquery of a CALL DML search request. Sorting with up to two sort criteria is performed with the internal SORT module of SESAM/SQL, but only if the following conditions are satisfied: The following is checked beforehand in the DBH:

1. If the sum of the length of the sort criteria smaller than or equal to 512 bytes.
2. If the sorting order in the module SESFS90 has not been changed.
3. If the amount to be sorted (number of rows times length of row) is smaller than a buffer frame of the cursor buffer (DBH option FRAME-SIZE).

If these checks are all positive, sorting in the DBH is carried out using SESAM/SQL's own SORT module. If they are negative, sorting is passed to a service task. A check is also conducted there. In addition to the first two DBH checks stated above, a check is performed to see if the amount to be sorted (number of rows times length of row) is smaller than or equal to 96 Kb. If this check is positive, sorting (now in a service task) is carried out using SESAM/SQL's own SORT module. If it fails, sorting uses the software product SORT. In all cases, however, sorting is performed in the service task using the SORT software product if there are more than 2 sort criteria.

WORK-FILES parameter

The work files and scratch files for BS2000 sorting are usually created on the default pubset of the ID. The WORK-FILES parameter of the DBH option SERVICE-TASKS allows the system administrator to choose a pubset on which these files are to be created (PUBLIC-DISK or PRIVATE-DISK). The pubset should have about 100,000 free PAM pages available and should be reorganized so that the fewest possible extents are created when files are expanded. By allocating a sufficiently large number of PAM pages in the PRIMARY-ALLOCATION parameter, the system administrator can prevent the files from being extended during sorting. How to calculate the size of the work and scratch files for a BS2000 sort run is described in the “[SORT \(BS2000\)](#)” manual.

If several private disks were specified for VOLUME in the DBH option PRIVATE-DISK, then the working and scratch files are created such that they begin in a different VOLUME in each service task (DBH option MAXIMUM).

Example

DBH option VOLUME

```
VOLUME=(DISK01,DISK02,DISK03)
```

Working and scratch files are created in the service tasks in the following manner:

in the 1st service task: VOLUME=(DISK01,DISK02,DISK03)

in the 2nd service task: VOLUME=(DISK02,DISK03,DISK01)

in the 3rd service task: VOLUME=(DISK03,DISK01,DISK02)

In the 4th service task: VOLUME=(DISK01,DISK02,DISK03)

etc.

RECORDS-PER-CYCLE parameter

The SORT software product includes an option that allows the quantity to be sorted to be distributed evenly over up to 9 sort subtasks. The number of rows to be sorted in each subtask can be specified by the system administrator in the RECORDS-PER-CYCLE parameter of the DBH option SERVICE-TASKS or during ongoing operation with the administration statement MODIFY-SERVICE-TASKS (see also the RECORDS-PER-CYCLE operand in the “[SORT \(BS2000\)](#)” manual).

A multi-task sort by the SORT software product requires up to double the disk storage space needed for a single-task sort, since the sort subtasks exchange data via scratch files. The DBH parameter RECORDS-PER-CYCLE should therefore only be specified if there is sufficient free space on the pubset and if true parallel (multiprocessor) processing of the sort subtasks is possible.

The value of the RECORDS-PER-CYCLE parameter must not be less than 1,000,000. The selection of a too low number of rows to be sorted in a cycle by a sort subtask increases the communication overhead and effectively offsets all the performance benefits of multi-task sorting.

The following must also be noted when using multitask sorting:

If the data to be sorted is passed to the service task in a cursor file, e.g. for SQL statements with “ORDER-BY” clauses, then the amount of data to be sorted is known at the time when the SORT software product is initialized.

The number of sort cycles is calculated as follows:

- Number of sort cycles = number of rows / RECORDS-PER-CYCLE
- Number of sort subtasks = number of sort cycles - 1

Work files are created for the number of sort cycles calculated and scratch files are created for the number of sort subtasks calculated. The maximum number of sort cycles is limited to 9 cycles. A maximum of 8 sort subtasks are therefore started.

If the amount of data to be sorted is determined in the service tasks during processing, e.g. for DDL statements such as “MIGRATE”, “CREATE INDEX”, etc., the amount to be sorted is not known at the time when the SORT software product is initialized. The number of sort cycles is always set to the maximum value of 9. The work and scratch files are created accordingly.

the CORE parameter of the SORT default procedure

The default settings can be set for each server and user ID separately for SORT using the command MODIFY-SORT-DEFAULTS. The job class for the sort subtasks can also be set. The default settings are stored in the file SYSPAR.SORT.vvv (vvv = version of SORT). See the “[SORT \(BS2000\)](#)” manual for a detailed description of the command.

JOIN entry of the DBH ID

The JOIN entry of the DBH ID should allow START-IMMED=YES and NO-CPU-LIMIT=YES (see [section “Influencing the service task strategy” on page 97](#)).

The selection of ADDRESS-SPACE-LIMIT=32767 is recommended to ensure that the SORT software product can make full use of the value specified in CORE.

5.12.3 Storage requirements for sort work files when constructing indexes

The disk storage requirement for an index is equal to (in half pages):

$(\text{number} * \text{rowlength} / 2048)$

`number` is the number of rows to be sorted.

From each row of the table in which at least one value of a column (from which the index is constructed) is significant, one sort row is created. The maximum value is thus equal to the number of rows in the table.

`rowlength` is the length of a sort row.

It consists of a fixed component of length 8 (length column, index ID, row number) plus the sum of the inverted lengths of the columns from which the index is composed. The inverted length is the length that was specified when defining the index or, if not specified, the length of the column. If the only column or the last column of the index is of type CHARACTER, blanks at the end of the sort row are not included in the length.

If multiple indexes are being defined, the requirement for each individual index must be added.

5.12.4 Summary of performance-enhancing measures

The following section summarizes the criteria and measures that have a positive effect on performance when processing service requests:

- Parameters of the DBH option SERVICE-TASKS:

INITIAL: 1

MAXIMUM: value determined with the performance monitor

PRIMARY-ALLOCATION: sufficiently large number of PAM pages

RECORDS-PER-CYCLE: value greater than 1,000,000 (on multi-processor servers)

WORK-FILES: (rapid) volume for work and scratch files with
approx. 100,000 free PAM pages
(PUBLIC-DISK or PRIVATE-DISK)

- JOIN entry of the DBH ID:

START-IMMED = YES

NO-CPU-LIMIT = YES

ADDRESS-SPACE-LIMIT = 32767

- SORT parameter file:

CORE = 32767

- Number of sort criteria: ≤ 2
- Length of sort criteria: ≤ 512 bytes
- Number of rows * length of row: ≤ 96 Kbytes
- The sort order in the SESFS90 module must not be changed. The module contains a table that specifies the sort order (see the “[Database Operation](#)” manual).

5.12.5 RECOVER/REFRESH options

You use the DBH option RECOVER-OPTIONS to set the following options which are used in the case of a RECOVER or REFRESH run for the DBH in the service task:

- The size of the buffer for system-access data
- The size of the buffer for user data
- The storage information for the transaction log files (TA-LOG files)
- The storage information for the restart log file (WA-LOG file)

You can adjust the values of the parameters during the DBH session using the administration statement MODIFY-RECOVER-OPTIONS.

In comparison with changes to other DBH options during ongoing operation you must note that changing RECOVER-OPTIONS using MODIFY-RECOVER-OPTIONS only applies for future RECOVER or REFRESH runs.

The success of a change to RECOVER-OPTIONS can best be checked using a sequence of comparable RECOVER or REFRESH runs. This is, for example, the case during regular maintenance of a replication.

You can also evaluate IO statistics regarding the buffer access hit rates which are contained in the recover run's SYSLST log. This enables you to check the quality of the buffer sizes used for the recovery (see [section "Dimensioning the user data buffer" on page 74](#) and [section "Dimensioning the system data buffer" on page 75](#)).

With the specifications for the TA-LOG file you must take into account that not only the memory sizes must be large enough, but that, when private disks are used, the device employed is fast enough (DEVICE-TYPE). Otherwise a RECOVER or REFRESH run would be slowed by input/output bottlenecks since the TA-LOG file is regularly written to. If the primary allocation is large enough, there is also the advantage that no or only few secondary allocations are required. This prevents the TA-LOG file from being fragmented.

The settings for the WA-LOG file are less critical because normally only few inputs/outputs are required for it.

However, if regular WA-LOG file inputs/outputs are observed, this indicates premature displacement of changed blocks on account of the buffer being too small for system-access data or user data. This situation can be improved by increasing SYSTEM-DATA-BUFFER and USER-DATA-BUFFER.

5.13 Reorganization criteria

This section describes how to recognize when it makes sense to reorganize a space and what things should be noted during a volume reorganization with SPACEOPT.

5.13.1 Physical DB structure and changes to it during operation

After migrating or loading a table using the MIGRATE or LOAD OFFLINE utility statement the database blocks are filled in accordance with the block utilization setting. The rows in the table are sorted in ascending order according to its primary key and stored in the data blocks. The logical linking of the blocks generally corresponds to their physical location.

If rows are only added to the end of the table using DML, then nothing change. The table remains reorganized. If, on the other hand, rows are inserted between the existing rows or the length of existing rows is increased so that the space reserved in the block is not large enough, then a relocation block is logically inserted. This means that the logical link does not point to the block with the next highest block number, rather it points to the relocation block.

The same thing can happen when many rows are deleted. If rows are deleted and blocks are released due to the deletions, then the blocks are taken out of the linkage and given to the free space administration. If a new block is needed when rows are added or a row is changed, the blocks managed by the free space administration are used.

Inopportune deletions and additions of rows can lead to poorer utilization of the blocks, especially when the block utilization was set too high, so that blocks are constantly being released and relocation blocks are created.

5.13.2 Indicators of the necessity of a reorganization

The following clues can be used to recognize the necessity for a reorganization:

- The buffer hit rate (SESMON DBH mask I/O) drops although the set of data (number of rows) does not increase and the application profile has not changed.
- The size of the space increases although the amount of data has not increased.
- The SESDIAG space statistic (see the following subsection) shows that many blocks are only being marginally utilized.

5.13.3 Space statistic from SESDIAG

The SESDIAG space statistic function can be used to output the contents of the space statistic block to SYSLST. The space statistic block shows the block utilization of the blocks read into the DBH for this space. Only the primary data blocks and the blocks of the lowest hierarchy level of secondary index are taken into account. The data is output in the following form:

- time period over which the data was acquired using two time stamps
- output of the absolute and relative number of blocks and their utilization in intervals of 10 percent.

Data for the last 54 sessions in which the data has been changed is output. (The information is only written in the space when changes have been made.) A session is the time period between the physical opening and physical closing of a space. If the data is reorganized, the time of the reorganization is recorded with a time stamp.



When a space is passed for utilities from the DBH task to the service task, the space is physically closed.

For a short description of SESDIAG, see the SESAM/SQL library SIPANY.SESAM-SQL.090.TOOLS

5.13.4 Reducing the number of extents of a BS2000 file

The repeated creation, deletion, expansion and contraction of files during current operation leads to ever increasing fragmentation of the free storage space and new files in pubset volumes. As it increases, fragmentation has an increasingly negative influence on data access performance and on the uniform distribution the I/O load over all the volumes of the pubset. The extent lists in the catalog entries are increased by the forced creation of multiple small file extents when files are increased in size.

The software product SPACEOPT (purchased separately) cleans up fragmentation by reorganizing the file extents in the volumes of a pubset; for more details, see the “[SPACEOPT \(BS2000\) manual](#)”. The purpose of SPACEOPT is to create the largest possible, contiguous free storage spaces so that large files can be created with a small number of extents.

SPACEOPT will also reorganize open files. You should note the following:

- File reorganization involves the actual physical reorganization of file extents. You should therefore only use this function when the I/O load of the volume being reorganized is low.
- During file reorganization, SPACEOPT temporarily locks the corresponding BS2000 catalog entries.

If a file is opened or expanded while reorganization is in progress, SPACEOPT will interrupt the reorganization and release the lock in order to allow the transaction to pass. After it has obtained the catalog lock, SPACEOPT resumes processing. This procedure is repeated by up to three times per file before the file is excluded from the reorganization.

5.13.5 Reorganization of a space together with reassignment of row numbers

SESAM/SQL allocates row numbers when adding rows to the table (using the INSERT and LOAD functions). When rows are deleted SESAM/SQL administers the row numbers internally in the row number administration blocks. The row numbers and the database translation table (DBTT) are used to create the link between the values in the secondary index and the primary data rows. To do this, the row number is stored in the secondary index behind the corresponding value. The row number also forms the access index inside the DBTT which is used to find the reference to the corresponding primary data row. The size of the DBTT is defined by the currently highest allocated row number. Over a period of time, the deletion and insertion of rows can lead to a wide distribution of the row numbers across the primary key spectrum. If the table is of the "history table" type, where data are stored for a set time and then deleted cyclically (e.g. in the booking procedure for a bank account), it is advisable to set shorter times between reorganization cycles with correspondingly lower data amounts. Higher cycle times with correspondingly higher data amounts will lead to an unnecessary enlargement of the DBTT and row number administration. This results in a worsening of the buffer hit rate in the SYSTEM DATA buffer.

A row number administration with many free row numbers can also lead to a worsening of the recovery times for INSERT statements during a RECOVER. This is because the recovery must use the same row numbers as the original and because the row number must be searched for sequentially where necessary in the row number administration.

If you already have a situation where there are too many free row numbers or the distribution of existing row numbers is very high, the function REORG SPACE with the parameter NEW ROW_IDS can be useful. However, you should note that when the row numbers are reassigned, all the indexes will be set to defective and the logical data saving will be interrupted.

Use the tool SESDIAG function 3 (output table information) to show how many free and administered row numbers exist for a particular table.

For a short description of SESDIAG, see the SESAM/SQL library SIPANY.SESAM-SQL.090.TOOLS

5.14 Distributed processing

The following options are available to the system administrator to influence performance for distributed processing with SESAM/SQL-DCN:

- Dimensioning of the SESDCN memory pool
- Starting multiple SESDCNs

5.14.1 Dimensioning of the SESDCN memory pool

The size of the SESDCN memory pool can be controlled by the system administrator via the `USERS` parameter of the `SET-DCN-OPTIONS` control statement. Depending on the value for `USERS`, a specific number of containers of up to 256 bytes each (approx. $4 * \text{USERS}$ value, i.e. 1 Kb per user) is created when the pool is initialized by the master DCN. This is, however, only an orientation value. The actual space requirement may differ depending on the request profile of a user.

If the pool dimensions are set too high, an unnecessary amount of address space will be used in the tasks (SESDCNs, DBHs, application programs) attached to this pool. If the dimensions are too low, it will lead to resource bottlenecks that will be evident from the frequent occurrence of `SQLSTATE 91SC5` or status `2B/UG`.

The actual number of containers being used and the number of free containers are shown in the `CAPACITY` mask of the performance monitor (used pool elements and free pool elements). This information can be used by the system administrator to define the pool dimensions as required.

5.14.2 Starting multiple SESDCNs

Messages from remote configurations that are directed to a DBH of a specific configuration (C) are received by an SESDCN of that configuration C and then forwarded to the addressed DBH. This process of receiving and forwarding messages is carried out in asynchronous contingency tasks of the SESDCN task. A high request load could cause an SESDCN task to be fully utilized by these activities and could even lead to a backup of messages before the SESDCN task. In such cases, there would be no time available to the SESDCN task for other activities such as administration and lock monitoring.

To prevent such a bottleneck, the system administrator can start multiple SESDCN tasks in a configuration. In order to achieve such a load distribution, however, the data space belonging to this configuration must be distributed over multiple DBHs so that each SESDCN task can be assigned to at least one DBH.

The CAPACITY mask of the performance monitor shows the extent to which the SESDCN task(s) of a configuration is/are utilized by asynchronous activities (elapsed time in contingency). This information can serve as a useful indicator when deciding whether further SESDCN tasks should or need to be started in the configuration in order to distribute the load.

5.15 Analysis of lock conflicts

The SESMON performance monitor (see the "[Database Operation](#)" manual) and the SYS_LOCK_CONFLICTS view can be used to diagnose lock conflicts, see [section "Diagnosis and corrective measures for performance problems"](#) on page 70.

Various information on locked transactions can be found in the SYSLST output of the SESMON mask TRANSACTIONS.

More extensive information is available in the SYS_LOCK_CONFLICTS view, which can be read with SQL means. This view contains the 1,000 lock conflicts which have occurred most recently, with detailed information on lock types, locked objects and causers. However, when the view is accessed, only the internal IDs of the objects affected are displayed, which means that the associated object names must be ascertained.

Combining the SYS_LOCK_CONFLICTS and SYS_DBC_ENTRIES views (information on the active databases) enables the lock information to be displayed with the object names. A SELECT applied to the view defined below then supplies lock information on the last 1,000 locks which have occurred in the DBH. All objects with their corresponding names are displayed here.

The view below, for example, is defined:

```
CREATE VIEW LOCK_INFO AS
  SELECT ALL TIME_OF_CONFLICT, OBJECT_TYPE, DBC_NUMBER,
    GENERAL_LOCKS.CATALOG_NAME AS CATALOG_NAME, SPACE_ID,
    CASE WHEN DBC_NUMBER IS NOT NULL AND SPACE_ID < 32767
      THEN COALESCE (
        (SELECT SPACE_NAME FROM SYS_INFO_SCHEMA.SYS_SPACES
         WHERE SPACE_ID = GENERAL_LOCKS.SPACE_ID), '???' )
      ELSE '---'
    END AS SPACE_NAME,
    TABLE_ID,
    CASE WHEN DBC_NUMBER IS NOT NULL AND TABLE_ID < 30720
      AND SPACE_ID < 32767
      THEN COALESCE (
        (SELECT TABLE_NAME FROM SYS_INFO_SCHEMA.SYS_TABLES
         WHERE TABLE_ID = GENERAL_LOCKS.TABLE_ID
         AND SPACE_ID = GENERAL_LOCKS.SPACE_ID), '???' )
      ELSE '---'
    END AS TABLE_NAME,
    INDEX_ID,
```

```

CASE WHEN DBC_NUMBER IS NOT NULL AND INDEX_ID IS NOT NULL
THEN COALESCE (
  (SELECT INDEX_NAME FROM SYS_INFO_SCHEMA.SYS_INDEXES
   WHERE INDEX_ID = GENERAL_LOCKS.INDEX_ID
   AND SPACE_ID = GENERAL_LOCKS.SPACE_ID
   AND ORDINAL_POSITION = 1), '???'')
ELSE '---'
END AS INDEX_NAME,
CASE WHEN DBC_NUMBER IS NOT NULL AND INDEX_ID IS NOT NULL
THEN COALESCE (
  (SELECT TABLE_NAME FROM SYS_INFO_SCHEMA.SYS_INDEXES
   WHERE INDEX_ID = GENERAL_LOCKS.INDEX_ID
   AND SPACE_ID = GENERAL_LOCKS.SPACE_ID
   AND ORDINAL_POSITION = 1), '???'')
ELSE '---'
END AS INDEX_TABLE_NAME,
ROW_ID, SI_VALUE, PLAN_ID, META_SCHEMA, META_TABLE, HOST_NAME,
APPLICATION_NAME, CUSTOMER_NAME, CONVERSATION_ID, TAC_NAME,
MODULE_NAME, STATEMENT_NAME, STATEMENT_TYPE, LOCK_MODE,
LOCK_TYPE, REQUEST_ANNOUNCED, LOCKING_OBJECT_TYPE,
LOCKING_HOST_NAME, LOCKING_APPLICATION_NAME,
LOCKING_CUSTOMER_NAME, LOCKING_CONVERSATION_ID,
LOCKING_LOCK_MODE

FROM (SELECT SYS_INFO_SCHEMA.SYS_LOCK_CONFLICTS.*, CATALOG_NAME
FROM SYS_INFO_SCHEMA.SYS_LOCK_CONFLICTS
LEFT JOIN SYS_INFO_SCHEMA.SYS_DBC_ENTRIES
ON SYS_INFO_SCHEMA.SYS_LOCK_CONFLICTS.DBC_NUMBER
= SYS_INFO_SCHEMA.SYS_DBC_ENTRIES.DBC_NUMBER)
AS GENERAL_LOCKS

WHERE DBC_NUMBER IS NULL
OR DBC_NUMBER = (SELECT DBC_NUMBER
FROM SYS_INFO_SCHEMA.SYS_DBC_ENTRIES
WHERE CATALOG_NAME = CURRENT_REFERENCED_CATALOG)

```

6 Specific notes on tuning applications

This chapter is divided into two separate sections containing specific notes for tuning SQL and CALL DML applications.

6.1 Optimization options for SQL applications

This section begins with some general recommendations on programming SQL applications and then deals with the tuning of SQL statements.

6.1.1 General programming recommendations

There are some programming recommendations for SQL applications that can have a positive effect on performance if observed. These programming recommendations are related to the following topics:

- descriptors and prepared SQL statements
- number of columns in SELECT lists and update statements
- SQL cursors in block mode
- maximum length of VARCHAR

descriptors and prepared SQL statements

Descriptors and prepared statements utilize resources of the SQL- RTS (conversation memory of openUTM) and of the SQL-DBH (plan buffer).

The following measures reduce the resource requirements:

- As far as possible, it is better to use:
 - static SQL instead of dynamic SQL, and
 - “USING *user-variable* ...” instead of “USING SQL DESCRIPTOR ...” for dynamic SQL.
- SQL descriptor areas should be allocated sparingly; whenever possible, only a single input and output descriptor should be defined and used for all prepared statements.
- Large descriptor areas should be released before the end of the dialog step (also for PEND KP). Note that conversation memory is saved by openUTM on changing the dialog step.
- Statement names for prepared SQL statements should be reused (since this implicitly releases the previously prepared SQL statement).

Number of columns in SELECT lists and UPDATE statements

The number of columns in a SELECT list has a significant impact on resource consumption (see above) and performance.

Whenever possible, the user should therefore:

- avoid a SELECT list of the form * and
- always specify only the required columns.

This also applies to UPDATE statements: Only the columns to be actually updated should be specified in the SET clause of an UPDATE statement.

SQL cursors in block mode

If a number of result rows of a cursor are to be determined with a sequence of FETCH statements in an ESQL-COBOL application program, the use of block mode (PREFETCH pragma) permits a significant increase in performance (see the [“SQL Reference Manual Part 1: SQL Statements”](#)). In block mode, sets of result rows of a prefetch cursor are transported from the DBH to the application task in a single job. This means that communication steps between the application task and the DBH are no longer necessary for subsequent FETCH jobs.

The gain in performance resulting from block mode depends primarily on the overhead associated with the communication between the application and the DBH. Even in a local environment (where the application and the DBH are on the same computer), improvements can be seen. Block mode is, however, of real significance in a distributed environment where the application program and the DBH are located on different computers.

Restrictions

- The prefetch cursor must not be used in the statements UPDATE ... WHERE CURRENT OF *cursor* and DELETE ... WHERE CURRENT OF *cursor*.
- The prefetch cursor must not be declared with SCROLL.
- The prefetch cursor must not be processed with the STORE *cursor* statement.

Conditions

- The FETCH request for the first row of a block is somewhat slower than a single FETCH request. Subsequent FETCH requests, on the other hand, are faster.
- The result blocks are buffered in the user task. This results in increased administrative overheads and increased main memory requirements for the user task.

Settings

- Transfer of the blocks from the DBH to the application program requires an appropriate message length. The PUF parameter in the configuration file for the UTM application should therefore be set as high as possible.
- The blocks returned by cursors are administered in the prefetch buffer. The user defines the size of the prefetch buffer in the configuration file of the application program (see [section “Size of the prefetch buffer” on page 92](#)). The utilization of the prefetch buffer can be checked with the performance monitor.
- The user specifies the number of rows in a block as the blocking factor (n) using the PREFETCH pragma. The blocking factor should suit the requirements of the SQL application program. If the results of cursors are further processed in subsets of the same size, the cardinality of the subset would, for instance, be an appropriate blocking factor.

If the logic of the application program does not provide a suitable blocking factor, the following aspects should be taken into account:

- In an SQL application, all cursors opened in block mode compete for space in the prefetch buffer. Space which has been occupied remains assigned to the cursor until the cursor is closed. The size of the memory area required is derived from the length of the result row and the value defined as the blocking factor.
If a bottleneck prevents a prefetch cursor from occupying space in the prefetch buffer, the PREFETCH pragma is ignored and block mode is not activated. The result rows of the corresponding cursor are then processed using single FETCH requests.
If the prefetch buffer is not a bottleneck, the blocking factor can be generously dimensioned.
- Irrespective of the blocking factor specified, only the number of result rows which fit into a message buffer can be transported in one block. If the blocking factor has already reached this size, it is no longer possible to improve performance by increasing the blocking factor.
- The blocking factor defined in the PREFETCH pragma should not be used to request more result rows than are processed by the application program.

Row comparison (“lexicographical comparison”)

In certain cases it is necessary to compare rows with more than one column such as:

```
(SURNAME, FIRST_NAME) >= ('Smith', 'John')
```

In this example a search is made for rows where the tuple (SURNAME, FIRST_NAME) is lexicographically the same as or greater than the tuple ('Smith', 'John'). In other words,

1. SURNAME column has a value >'Smith' or
2. SURNAME column has a value = 'Smith' and at the same time FIRST_NAME column has a value >='John'

The following predicate can be used for this purpose:

```
SURNAME > 'Smith' OR (SURNAME = 'Smith' AND FIRST_NAME >= 'John')
```

However, the following formulation is simpler and more straightforward:

```
(SURNAME, FIRST_NAME) >= ('Smith', 'John')
```

In certain cases this notation has performance benefits:

- No linking takes place with OR
- It is easier to use appropriate indexes (also primary keys)

An SQL-INSERT statement for multiple rows

Performance can also be improved when adding rows. This can be done by specifying multiple rows in a single SQL INSERT statement (see the manual [“SQL Reference Manual Part 1: SQL Statements”](#)). In this case, just as in the case of block mode reading, we can reduce the number of communication steps required between the application program and the DBH. Once again, the performance improvement depends on how expensive the saved communications steps are.

Use of routines

The use of routines can lead to enhancements in performance, see [section “Use of routines” on page 55](#).

6.1.2 Tuning SQL applications

This section contains specific notes on how the performance and thus the response time of individual SQL-DML statements can be improved. A single badly formulated SQL statement can potentially degrade performance of the entire DBH session. The optimization of SQL statements should be dealt with first before trying to improve the performance of the DBH (e.g. by modifying DBH options).

6.1.2.1 Basic approach

Before one can start tuning, it is first necessary to determine which SQL statement is causing the performance problems. The following tools are useful for analyzing performance:

- **SYS_DML_RESOURCES** view:
The **SYS_DML_RESOURCES** view of the **SYS_INFO_SCHEMA** supplies information on particularly “costly” SQL-DML statements, see [section “View SYS_DML_RESOURCES” on page 21](#) and the manual “[SQL Reference Manual Part 1: SQL Statements](#)”.
- The “SQL INFORMATION” mask of **SESMON**:
This mask contains a global overview of the frequency of SQL-DML statements, plan generations and calls to SQL components.
- **EXPLAIN**:
The Explain component contains an edited representation of the access plan for an SQL-DML statement. It shows the individual steps in which the SQL statement is processed and the access paths which can be used for this purpose. The Explain component can provide valuable hints as to where in the execution of the SQL statement performance problems could occur. It is therefore advisable to check the access plans of any new SQL application with the aid of this tool.
- **SESCOS**:
Transactions, SQL statements and individual processing steps can be logged over a specific period and subsequently evaluated and analyzed with **SESCOSP**.

These performance analysis tools are described in detail in [chapter “Tools for analyzing performance” on page 13](#). Details on the optimization of SQL statements can be found in [section “Structure and control of the optimizer” on page 24](#). The edited representation of an access plan as generated by the Explain component is described in [chapter “Explain component of the optimizer” on page 131](#).

On isolating an SQL statement that is critical to performance, the Explain component can be used to obtain the necessary information on the access plan selected by the optimizer. This access plan is the starting point for all further considerations as to how the plan can be specifically improved. In order to do this, it is first necessary to develop some concept as to what the optimum plan for that SQL statement could be. The following measures can be considered in the context of improving the access plan:

- creating indexes
- controlling the optimizer with the pragmas OPTIMIZATION LEVEL, SIMPLIFICATION, JOIN, KEEP JOIN ORDER, and IGNORE INDEX
- rephrasing the SQL statement or restructuring the SQL application.

The optimum or almost optimum access plan can be achieved by selectively using these measures.

6.1.2.2 Typical performance problems

The following section deals with some typical cases where performance problems could potentially occur when processing SQL statements. Each problem is described together with an explanation of how that problem can be detected from the output of the Explain component and the possible responses in each case.

Problem

The rows that satisfy the search condition are determined by sequentially reading the entire base table.

Explain:

There is either no “used index” attribute at the “table_scan” node, or the attribute “used index: <Primary Key>” exists, but the selector with the search condition on the columns of the primary key is too weak. One indicator for this is the attribute “cost_ratio: n / m”. Check if the value of n is too high (approximately equal to the number of blocks of the base table).

Response:

Create one or more indexes on the columns of the search condition. In the case of search conditions of the form `col1 = value1 AND col2 = value2 AND ...` in particular, a compound index on the columns (col1, col2, ...) should be considered.

Problem

An index with low selectivity is used for evaluating a search condition on a base table.

Explain:

The index in question is listed under “used index:”. The value m in the attribute “cost_ratio: n / m ” is approximately equal to the number of rows in the base table.

Response:

The pragma IGNORE INDEX can be used to exclude the index in question from the evaluation. In some cases, it may even be worth deleting the index with DROP INDEX.



It is conceivable that only the statistics for that index is outdated. This can be remedied with a REORG STATISTICS on the index.

Problem

Several rows need to be sorted for the evaluation of the SQL statement.

Explain:

The contents of the access plan may vary in this case:

- a) One “sort” node
- b) The “table_scan” node may contain the attribute “sortorder:”, without the possibility of using a “sort index:”.

Response:

In case (a), a distinction must be made based on whether the “sort” node is required due to an ORDER BY clause or a sort-merge join. It may be more efficient to replace the sort-merge join by a nested-loop join (see below) so that the sorting can be dropped. If the “sort” node is the result of an ORDER BY clause in the SQL statement, this clause should be omitted if it is not essential to the application logic.

In case (b), an index that covers at least the columns listed under “sortorder:” should be created.

Problem

The optimizer selects the algorithm for calculating a join result set so that the determination of the entire (!) result set is made to perform as quickly as possible. If, however, you are not interested in the entire set found by the join but only in the first *n* rows or maybe even only the first row, then the plan created by the optimizer may not be the optimal solution for achieving this goal. If, for example, the entire join result set consists of a large number of rows and access to the inner relation is not aided by suitable indices for the nested-loop join, then it is better from the point of view of the optimizer to sort the two join relations and to determine the result using a sort-merge join. However, for an application that is interested in only one row of the large result set, you would only need to search in the inner table for one single partner of the first matching row of the outer table during the evaluation. This search would be much faster using the nested-loop join.

Explain:

Sort-merge join in the Explain and a large set of data is expected for the complete join result.

Response:

Force the nested-loop join using the pragma JOIN NESTED LOOP.

Problem

A nested-loop join is very efficient if a suitable index on the left (inner) operand exists for the join condition and the join condition is very selective; this is especially true if the join condition is on the primary key of the inner base table. If the join condition is less selective, however, a nested-loop join is much worse than a sort-merge join.

Explain:

The “table_scan” node that is specified under “left:” at the node “njoin” should be observed in the Explain representation. There are two possibilities:

- a) No suitable index for the join condition exists under “used index:”.
- b) The value of *m* in the attribute “cost_ratio: *n* / *m*” is less than the actual number of rows that satisfy the join condition by several orders of magnitude.

Response:

In case (a), a suitable index can be defined.

In case (b), REORG STATISTICS should first be used to update the existing statistics.

In both cases, the pragma JOIN SORT MERGE can be used to force a sort-merge join. It is often sufficient to simply turn off the index used for evaluating the join condition (not the primary key index) by specifying the pragma IGNORE INDEX.

Problem

Sort-merge joins are generally more efficient than nested-loop joins. However, if one join operand consists of only a few rows, and if the other join operand consists of several rows, the nested-loop join may be better under certain conditions (see above).

Explain:

The Explain representation provides no further assistance for this situation. The size of each respective join operand must be used as a basis to determine whether a nested-loop join is preferable.

Response:

A suitable index should be defined on the larger table for the evaluation of the join condition. Furthermore, the existing statistics must be up-to-date. If the desired nested-loop join is not produced anyway, it can be forced using the pragma JOIN NESTED LOOP.

Example

```

SELECT ....          SELECT ....
FROM R,S             FROM R
WHERE R.a = S.a      WHERE R.a IN (SELECT S.a FROM S)

```

Note: following this conversion, the Explain representation will no longer have an “nljoin” node. The modified processing strategy does, however, correspond to a nested-loop join, even in terms of the response time.

Problem

The Cartesian product is generated instead of a join.

Explain:

The Explain representation contains the “cross” node. There is, however, one situation in which the Cartesian product is more efficient than a join:

```

SELECT ...
FROM R, S
WHERE R.a=wert AND R.a=S.a

```

In this case, an additional condition S.a=value is generated by the optimizer and the join condition R.a=S.a is eliminated instead. In the plan, the “table_scan” node of the base table S contains the additional condition and is linked via a “cross” node with the “table_scan” of the base table R.

Response:

Check whether all tables in the FROM clause are connected to one another with at least one join condition of the form `table1.column1=table2.column2`. Other join conditions should be avoided, e.g. for the following (frequently leads to the "nljoin" node):

```
R.a > S.a
R.a = S.a * 1.15
R.a + S.a = 10
```

Problem

An unsuitable join order was selected for a join on several tables.

Explain:

The Explain representation clearly shows in which order and with which join methods the tables are linked.

Response:

1. The decision as to which join order is most efficient can be made based on the table sizes and the intermediate results obtained after each paired join. This join order can be forced on the optimizer by using the pragma `SIMPLIFICATION OFF` to turn off normalization of SQL statements and by using explicit join expressions in the SQL statement, or if the join order is specified directly with the pragma `KEEP JOIN ORDER`.

Example

```
SELECT ...
FROM R, S, T, U
WHERE R.a=S.a AND S.b=T.b AND T.c=U.c AND ...
```

```
⇒
--%PRAGMA SIMPLIFICATION OFF
SELECT ...
FROM ( (R JOIN S ON R.a=S.a) JOIN
(T JOIN U ON T.c=U.c) ON S.b=T.b )
WHERE ...
```

In the second, rephrased SQL statement, the joins of R on S and T on U are performed first, followed by the joins on the individual intermediate results.

2. When the pragma `OPTIMIZATION LEVEL n` ($n < 9$) is used, the optimizer selects a join order which can be influenced by converting the join conditions in the WHERE clause.

Problem

The check for referential constraints slows down the response time.

Explain:

The subtree under “check_after_all” or “check_on_the_fly” at the “insert_stmt”, “delete_stmt” or “update_stmt” nodes contains a quantifier (“some” or “all” nodes), and this in turn contains a “table_scan” on the referenced or referencing base table. The response time can be adversely affected, especially if the referential constraint under “check_after_all” needs to be tested. This applies if a self-referencing integrity constraint (i.e. the referencing table is also the referenced table) is involved or if an UPDATE or DELETE statement refers to the referenced table.

Response:

In the case of a DELETE statement, the check can be speeded up by defining an index on the referencing columns (see [section “Integrity constraints” on page 62](#)).

Problem

Performance problems can occur if non-correlated subqueries occur in predicates.

Explain:

The table evaluated for the predicate is read sequentially and the predicate is represented as a “some” predicate of a “rest” node.

Response:

Possibly reformulate query as a join.

Example

```

SELECT *
FROM R
WHERE R.key = ANY ( SELECT S.b
                    FROM S
                    WHERE S.c = :var )

```

oder auch

```

SELECT *
FROM R
WHERE EXISTS ( SELECT S.b
              FROM S
              WHERE S.c = :var AND S.b = R.key )

```

⇒

```

SELECT R.*
FROM R, (SELECT DISTINCT S.b
        FROM S
        WHERE S.c = :var ) AS T(x)
WHERE R.key = T.x

```

Whereas in the first two formulations, the table R must be read sequentially, the condition `R.key = T.x` is evaluated via an index defined on R.key.

Problem

In the case of EXISTS / NOT EXISTS predicates, frequent evaluation of the subquery can lead to performance problems.

Explain:

This predicate is generally indicated by means of a “some” or “all” node in the output of the Explain. However, it is not evident from the Explain how often a subquery needs to be evaluated. Consequently it cannot generally be recognized whether performance problems occur.

Response:

If the conditions in the subquery are suitable for join evaluation (equi-join predicates; see [section “Phase 3: Selecting the access path” on page 30](#)), it may be advantageous to rephrase the statement into an outer join in the case of a NOT EXISTS predicate.

Example

```
SELECT ...
FROM   R
WHERE  NOT EXISTS ( SELECT ...
                   FROM   S
                   WHERE  S.b = R.key )
```

⇒

```
SELECT ...
FROM   (R LEFT OUTER JOIN S on S.b = R.key)
WHERE  S.key IS NULL
```

The subquery must be reevaluated every time here, since the column R.key is unique as a key; the outer join variant would be more suitable in this case if a sort of S on column b (e.g. using an index) exists. A more efficient “left_outer_mjoin” could be planned in that case.

Problem

Aggregate functions (AVG,MAX,MIN,SUM,COUNT(*),COUNT) for all the rows of a base table, are determined by sequential reading of all the rows in a base table, e.g.

- SELECT MIN(column) FROM T
- SELECT MAX(column) FROM T
- SELECT COUNT(*) FROM T

Explain:

A “sorted_group” node occurs when identifying the aggregate function.

Response:

The column must not be of the data type VARCHAR. An index must be set up for the argument of the aggregate function. Column should be the first (possibly only) component of the index for determining MIN(column) or MAX(column). Any index on a column with the NOT NULL property must be set up to determine COUNT(*). The entire length of the column must be included in the index.

Problem

Aggregate functions for a selected set of rows, are determined by sequential reading of all the rows in a base table, e.g.

```
SELECT MIN(co11),MAX(co12),AVG(co13),SUM(co14),COUNT(*),...
      FROM T
      WHERE search-condition
```

Each row is checked to see whether it fulfils the search condition.

Explain:

A “sorted_group” node occurs when identifying the aggregate function.

Response:

- Indexes must be set up for the arguments of the aggregate function. The arguments of the aggregate function must not be of the data type VARCHAR and the entire length of the argument must be included in the index.
- The search condition must be evaluated using indexes.

To do this, an index must be set up for each column of the search condition. The entire length of the columns must be included in the index.

Only the following predicates (linked with AND or OR) may occur as predicates of the search condition:

- comparison of two values in the form “column comp_op value”
- range queries “column BETWEEN value_1 AND value_2”
- pattern comparison “column LIKE value”
- element query “column IN (value_1,...,value_n)”

where column must not be a multiple column, comp_op is a comparison operator, value is a literal, a host variable or an expression which can be calculated beforehand.

6.2 Optimization options for CALL DML

This section describes the optimization options that are recommended for CALL DML applications:

- Economic use of communication path lengths
- Economic use of path lengths for syntax analysis
- compliance with general programming rules
- acceleration of join processing
- influencing the processing sequence for queries

6.2.1 Economic use of communication path lengths

Economic use of communication path lengths can be made in the CALL DML by means of the following options:

- Concatenating CALL DML requests. The following chains are permitted:
 - Begin of Transaction with an UPDATE or retrieval statement
 - n Open statements
 - n Open statements preceded by Begin of Transaction
 - End of Transaction or Rollback of Transaction linked to Begin of Transaction (not permitted in UTM environments)
 - End of Transaction or Rollback of Transaction linked to a Close statement
- Use of block mode

6.2.2 Economic use of path lengths for syntax analysis

The following options are available in the CALL DML to economize on path lengths during syntax analysis:

- use of block mode
- use of continuation statements
- specification of ranges for multiple attributes instead of the output of all individual variations
- use of an implicit Close on logical files at End of Transaction for all logical files opened within that transaction

- if all logical files of a user are to be closed, the User Close is more economical than multiple file Closes.

6.2.3 General programming recommendations

The following recommendations should be observed when programming CALL DML applications:

- Restrict the number of attributes in the projection to those actually required. Each unnecessarily output attribute involves path-length costs for conversion from the internal format to the external, for the transfer, and for syntax analysis.
- Restrict the number of attributes in an UPDATE statement to those actually updated. Each attribute that only represents a pseudo update (e.g. color from red to red) generates redundant path lengths at the time of syntax analysis, during processing (the attribute is actually exchanged), and at the time of logging for transaction management and media recovery.
- Combine multiple identical search conditions for a SAN with one relational condition in an Or group into a single search condition with multiple relational conditions.

Example

```
coll=wert1 v coll=wert2 v coll=wert3 v... v coll=wertn (san5010san501...)
⇒
coll=wert1 v =wert2 v =wert3 v... v =wertn (san50101...01)
```

Since SESAM/SQL does not perform any optimization at the CALL DML that affects the search condition, substantial savings in the path length can be realized by the equivalent reformulation of the search query. This is especially true if a secondary index for the relevant attribute exists. (The secondary index is evaluated for each search condition.)

- Avoid front masking and insignificant conditions. Subqueries that contain values with front masking cannot normally be evaluated efficiently via the secondary index. The insignificant condition cannot be evaluated via the index, since an insignificant value is represented in SESAM/SQL by the non-existence of the value.

6.2.4 Acceleration of join processing for CALL DML

Join processing for CALL DML can be accelerated by switching from nested-loop joins to merge joins at an appropriate time and in cases where the number of rows found is suitable for this purpose.

The main factor that effects the switch is the number of matched rows returned by the two join components. The following cases are differentiated:

- One or both components return at most one row using the specified primary key function (4/8).
- The number of rows returned by one of the components via the preceding index evaluation does not exceed the optimization value. The optimization value can be controlled by means of an optional Rep (see the release notice of SESAM/SQL). The default value is 10.

6.2.5 Order of processing queries for CALL DML

In the case of a sequential search (index evaluation neither desirable nor possible), it is usually necessary for subqueries to be processed in the order defined in the statement.

The following section explains why this is not the case and how the processing order can be influenced by the statement or by the sequence of individual subqueries in the statement.

The restrictive subqueries (conditions) of a query are internally represented in a postfix notation (Polish list). This Postfix representation is created from left to right during the analysis of the statement. However, since a Postfix notation is involved, it is processed from right to left. In other words, the statement is processed in reverse order.

To accelerate the sequential search, the selectivity of the individual subqueries is used in order to force optimum processing. The individual subqueries are assigned a so-called static weight, which represents the sort criterion within the same nesting level.

In other words, subqueries with the same selectivity are not sorted again and are hence not processed in the specified order.



Conclusion: the most selective subqueries within the same nesting level should be specified to the extreme right within that nesting level.

7 Explain component of the optimizer

The Explain component of the optimizer is designed to provide users with clues concerning the approach selected by the optimizer to execute their queries. It is always directly related to the implementation of a database query or, in more specific terms, to an SQL-DML statement. The pragma EXPLAIN is defined only for SQL-DML statements (see [section “SQL optimizer” on page 17](#)).

In the [section “Introduction”](#) section below, first the Explain component is presented using an example.

The remaining sections describe the working and representation of the Explain component for individual SQL elements:

- [“Table-value operations \(RELATION\)” on page 140](#)
- [“DML statements” on page 172](#)
- [“Quantifiers” on page 192](#)
- [“Value queries” on page 194](#)
- [“Expressions, function calls, and predicates” on page 195](#)
- [“Intermediate files” on page 203](#)



Rights to changes explicitly reserved:

As in the other parts of this manual, the contents of the sections that follow are based on the current status of the implementation.

The order of attributes within a node cannot be influenced. The examples illustrated here may therefore deviate from the output of the product in this regard.

As the optimizer is developed further, nodes or attributes may be added, dropped, or changed in meaning.

Finally, the layout of the explanation could change or the Explain component may be replaced by a better tool.

7.1 Introduction

At first glance, the output of the Explain component bears little resemblance to an SQL statement, since it is directly derived from the operator tree that is passed to the interpreter for execution. This makes the explanation somewhat unclear at first, but has the advantage that the actual processing overhead can be very precisely estimated.

As you will see, some operators have no equivalent representations in SQL, e.g. because the order of processing operands is significant for them.

This introduction will show how the explanation is to be read and how the effects of performance-enhancing measures can be gleaned from it.

7.1.1 Terminology

Some of the terms used to describe the Explain component occasionally reflect a mathematical model of an attribute graph. Within this context, operations (both relational and arithmetic) are called nodes. Their operands and options are collectively referred to as attributes. Some attributes represent a variable number of operands, in which case they are designated as sequences.

A node is represented as follows:

```
nodename [  
    attributename1: attributvalue1,  
    attributename2: <  
        attributvalue2.1,  
        ...  
    >  
    ...  
]
```

If you occasionally notice that attributes which actually belong to a node seem to be missing, this means that the Explain component has omitted them because they were not used.

An explanation is preceded by the associated source text. Additional pragmas generated when editing the statement are shown at the start.)

7.1.2 Introductory example

Let us assume the following two tables exist:

- The table DEPT contains the columns DEPT_NO (primary key), DEPT_NAME and CITY. The uniqueness constraint DEPT applies to the columns DEPT_NAME and CITY.
- The table EMP contains the columns EMP_NO (primary key), DEPT, EMP_NAME and SAL. The referential constraint F1_DEPT is defined for the column DEPT and refers to the primary key DEPT.DEPT_NO.

The tables can be created by the following DDL statements:

```
CREATE TABLE dept
    (dept_no    CHAR(5),
     dept_name  CHAR(30) NOT NULL,
     city       CHAR(30) NOT NULL,
     CONSTRAINT dept_no PRIMARY KEY (dept_no),
     CONSTRAINT dept UNIQUE (dept_name, city) )

CREATE TABLE emp
    (emp_no    CHAR(5),
     dept      CHAR(5),
     emp_name  CHAR(30),
     sal       DEC(9,0),
     CONSTRAINT emp_no PRIMARY KEY (emp_no),
     CONSTRAINT f1_dept FOREIGN K (dept)
       REFERENCES dept (dept_no) )
```

We will study the following DML statement, which calculates the average salary for each department:

```
--%PRAGMA EXPLAIN INTO 'tm.expl'
--%PRAGMA SIMPLIFICATION ON

SELECT dept_name, AVG(sal)
FROM emp, dept
WHERE emp.dept = dept.dept_no
GROUP BY dept_name
```

The explanation is output if the statement contains the pragma EXPLAIN.

Since the statement is extremely simple, the pragma SIMPLIFICATION ON was also specified. This is generally not required, since simplification has relatively few benefits for simple statements and is initiated automatically for complex queries.

(It is required in this example, however, because some redundant proj operations are dropped, thus making the explanation shorter.)

One way of determining the result would be

- to begin by creating the cross product between EMP and DEPT
- and then discarding all pairs in which EMP.DEPT and DEPT.DEPT_NO differ
- the remaining pairs can then be sorted by DEPT_NAME
- and the average can be calculated via SAL for all rows with the same DEPT_NAME.

This is certainly not the best method, and you have every right to expect more from the SESAM/SQL optimizer. The approach selected by the optimizer can be seen with the aid of the Explain component.



The following examples were created using a sparsely filled test database. Since optimization depends on the estimated number of matches, the same statement could also be implemented differently.

Processing strategy of the example

The output of the Explain component for the named statement is shown below. To enable better orientation, brief explanations in italics are included in the output in right-justified format.

```

dynamic_select_stmt [                               Statement optimized as dyn. SELECT statement
  output: <
    DEPT_NAME,                                     column of result table
    (column #2)                                    column of the result table
  >
  query: sorted_group [                            description of result table starts here
    from: nljoin [                                  grouping operation
      left: table_scan [                            based on nested-loop join
        outer_reference_s: <                        left join operand: table scan
          (expr #1) ::= DEPT_NO                     only rows from EMP
        >                                           for dept=dept.dept_no and
        base: ROCAT.DDLOPT.EMP                       current value of dept.dept_no
        column_s: <                                  underlying base table
          SAL,                                       emp with the columns
          DEPT                                       sal and
        >                                           dept
        isolation level: set at runtime              isolation level
        lock mode: shared
        cost_ratio: 6.000000E+00/4.000000E+00
        pred:                                        column values must satisfy
          ((DEPT = (expr #1)))                       (emp.dept=dept.dept_no)
      ]
      right: table_scan [                          right join operand: table scan
        base: ROCAT.DDLOPT.DEPT                     underlying base table
        column_s: <                                  with the columns
          DEPT_NAME #3,                             dept_name and
          DEPT_NO                                    dept_no
        >
        sortorder: <                                sort by DEPT_NAME,
          DEPT_NAME #3 ascending                    read in ascending order,
        >                                           and use the index for
        sort_index: UI9941102110408000             the uniqueness constraint
        isolation level: set at runtime
        lock mode: shared
        cost_ratio: 0.000000E+00/0.000000E+00
      ]
    ]
    group_col_s: <
      DEPT_NAME ::= DEPT_NAME #3                    grouping column
    >
    aggregate_s: <                                  calculate average AVG()
      (column #2) ::= all_avg(SAL)                  for each group
    >
  ]
  is_for_update: False
  read_only: True
]

```

The explanation is fairly comprehensive, since it includes all the information on the planned execution of the statement. It can be read as follows:

The statement was optimized as a dynamic SELECT statement (`dynamic_select_stmt`), i.e. with the default value 9 for the pragma `OPTIMIZATION LEVEL`.

The result of the statement can be found starting with the line:

```
query: sorted_group [
```

The result table is described from this point up to the complementary closing brackets. It constitutes an operation in which all rows with the same values are collected in the following grouping columns:

```
group_col_s: <
  DEPT_NAME ::= DEPT_NAME #3
>
```

In addition, the corresponding aggregate functions are calculated for each group:

```
aggregate_s: <
  (column #2) ::= all_avg(SAL)
>
```

The argument of the grouping operation is a table of a so-called nested-loop join operation (`nljoin`). Here all rows of the left operand which satisfy the join condition are searched for the right operand. The right operand here is a table scan. The SESAM kernel system is to read the table `ROCAT.DDLOPT.DEPT` and sort the table by `DEPT_NAME` at the same time. To do this, the system uses an auxiliary index that was created to implement the uniqueness constraint:

```
sortorder: <
  DEPT_NAME #3 ascending
>
sort_index: UI9941102110408000
```

The left operand of the nested-loop join is also a table scan. However, it also includes the following predicate:

```
pred:
  ((DEPT = (expr #1)))
```

Only the rows of `ROCAT.DDLOPT.EMP` which satisfy the predicate are to be returned. The expression (`expr #1`) refers to the outer reference:

```
(expr #1) ::= DEPT_NO
```

This references the current value of the `DEPT_NO` column in the right operand.

To evaluate the nested-loop join, the left operand is calculated for each row of the right operand. Since the left operand only includes those rows which satisfy the predicate, the result consists of all pairs of rows from the right and left operands. This table is sorted by DEPT_NAME and thus satisfies the conditions of sorted_group.

Summary

The table DEPT is read one row at a time. Here an index is used to sort the rows by DEPT_NAME. For each row of DEPT, the table EMP is searched for all rows that have a DEPT column containing the same value as the DEPT_NO column in the current row of DEPT. Each found combination of rows from DEPT and EMP now constitutes one row. All rows with the same value for DEPT_NAME (which will now be arranged in succession) are combined to form a separate group. A new row is then constructed from it with columns containing the value of DEPT_NAME and the average of SAL in that group.

An improved attempt

As the above analysis has shown, the table EMP is searched for each department from the start to the end. The number of rows to be read could be significantly reduced here by creating an index to EMP.DEPT as follows:

```
CREATE INDEX ddlopt.empdept(dept) ON TABLE ddlopt.emp
```

On optimizing our statement again, we would receive a different explanation, an extract of which is shown below:

```
...
left: table_scan [
  outer_reference_s: <
    (expr #1) ::= DEPT_NO
  >
  base: ROCAT.DDLOPT.EMP
  column_s: <
    SAL,
    DEPT
  >
  used index: <EMPDEPT>
  isolation level: set at runtime
  lock mode: shared
  cost_ratio: 3.000000E+00/2.000000E+00
  pred:
    ((DEPT = (expr #1)))
]
...
```

The index EMPDEPT is now used, so only those new rows of EMP which were matched are read. The predicate is very selective, so the overhead for reading the index is more than compensated for.

A completely different result

It can hardly be emphasized often enough that the results of the optimization also depend on the statistics of the database. It is therefore important to work with the original database or with realistic test data when analyzing access plans. Even relatively minor changes can produce a completely different picture. This is illustrated below by a simple trick (which introduces a small variation in the above statement):

```
SELECT dept_name, AVG(sal)
FROM emp, (SELECT * FROM dept WHERE dept_name <= 'zzz') AS d
WHERE emp.dept = d.dept_no
GROUP BY dept_name
```

The variation consists of an inserted predicate, which the optimizer assumes will reduce the number of hits for the table DEPT. The cost and hit estimates now result in a totally different execution plan (extract):

```

dynamic_select_stmt [
  query: sorted_group [
    from: sort [
      from: mjoin [
        left: table_scan [
          base: ROCAT.DDLOPT.EMP
          sortorder: <
            DEPT ascending
          >
          sort_index: EMPDEPT
          pred:
            ((DEPT IS NOT NULL))
        ]
        right: table_scan [
          base: ROCAT.DDLOPT.DEPT
          sortorder: <
            DEPT_NO ascending
          >
          sort_index: Primary Key
          pred:
            ((DEPT_NAME #2 <= 'zzz'))
        ]
        joincond: (row(DEPT) = row(DEPT_NO))
      ]
      sort_spec_s: <
        DEPT_NAME #2 ascending
      >
    ]
    group_col_s: <
      DEPT_NAME ::= DEPT_NAME #2
    >
    aggregate_s: <
      (column #1) ::= all_avg(SAL)
    >
  ]
]

```

In this case, the index EMPDEPT of EMP and the primary key of DEPT are used to sort the two join operands by the columns in the join condition. The join operation used here is the so-called merge join (mjoin). This means that the two operand tables are read concurrently (in ascending order), and it is only if one of the operands has duplicates in the join condition columns that the other is reset accordingly.

The result of the merge join is, however, sorted by DEPT_NO and not by DEPT_NAME, so an explicit sort by DEPT_NAME is required for the sorted group operation.

7.2 Table-value operations (RELATION)

The nodes dealt with in this section have a table as a result. They collectively belong to the class RELATION.

The interpreter processes the tables by rows. A base table is read one row at a time using predicates on indexes to keep the number of hits as small as possible. For the other tables, a distinction is essentially made in the interpreter between an “open” phase and a “read” phase. The read phase is row-oriented. In other words the interpreter returns control to the caller when the next row, which in turn is the result of one or multiple reads on rows of the underlying table, has been determined.

7.2.1 cast_rows_to_rel

Meaning: Table constructor, i.e. generation of an (intermediate) table from a set of rows. Usually occurs when row comparisons are used or views are used which are defined via a VALUES clause.

Attributes:

column_s	Columns required from the table which is generated.
rows	The rows from which the table is generated.

SQL fragment:

```
SELECT    PK1,PK2,PK3,PK4,PK5
FROM      T
WHERE     (C11, C12, C13) in (1,11,111),(1,11,122),(1,11,133) )
```

Extract from the explanation:

```

query: rest [
  from: table_scan [
    base: MYCAT.MYSCHEMA.T
    column_s: <
      ...
    >
    isolation level: set at runtime
    lock mode: shared
    cost_ratio: 8.000000E+00/1.000000E+00
  ]
  pred:
    some [
      outer_reference_s: <
        (expr #1) ::= C11,
        (expr #2) ::= C12,
        (expr #3) ::= C13
      >
      arg: rest [
        from: cast_rows_to_rel [
          column_s: <
            (column #4),
            (column #5),
            (column #6)
          >
          rows: <
            row(1, 11, 111),
            row(1, 11, 122),
            row(1, 11, 133)
          >
        ]
        pred: (row((expr #1), (expr #2), (expr #3)) =
              row((column #4), (column #5), (column #6)))
        cost_ratio: 1.000000E+00/2.999999E-01
      ]
    ]
  pred: true
]

```

7.2.2 cross

Meaning: cross product.

A single row is created by concatenating rows from all operands. This is done by applying the first operand on each row of the second, which in turn is applied on each row of the third, etc.

Attributes:

`from_s` List of operands, each of which represents a RELATION.

`outer_reference_s` List of expressions and predicates that can be evaluated independently of the remaining attributes of the node.

The schema of cross is produced by merging the schemata of the subtables.

SQL fragment:

```
SELECT t1.tellerid, t2.tellerid
FROM teller t1, teller t2
```

Extract from the explanation:

```
query: eproj [
  from: cross [
    from_s : <
      table_scan [
        column_s: <
          TELLERID #3
        >
      ],
      table_scan [
        column_s: <
          TELLERID #4
        >
      ]
    >
  ]
  column_s: <
    TELLERID ::= TELLERID #4,
    TELLERID #2 ::= TELLERID #3
  >
]
```

7.2.3 cursor_scan

Meaning: The current row of a cursor.

The row on which the cursor is positioned is read again. cursor_scan occurs only in conjunction with UPDATE... CURRENT and DELETE... CURRENT statements.

Attributes:

base, column_s, isolation level and lock mode have the same meaning as in table_scan.

SQL fragment:

```
UPDATE t
SET c1 = x
WHERE CURRENT OF c
```

Extract from the explanation:

```
update_stmt [
  object_rows:  cursor_scan [
    base : cat.schema.t
    column_s : <
      C1
    >
    isolation level : at least repeatable read
    lock mode : exclusive
  ]
  cursor_id_ref: MY_MODULE.C
  ...
]
```

7.2.4 Empty

Meaning: The empty table
(the null set in relational algebra).

SQL fragment:

```
--%PRAGMA SIMPLIFICATION ON  
DECLARE any_cursor CURSOR FOR  
SELECT * FROM account WHERE 4 = 5
```

Extract from the explanation:

```
cursor_select_stmt [  
  query: Empty  
  is_for_update: False  
  read_only: False  
]
```


7.2.5 eproj

Meaning: Table containing a row, for each row of the operand, with columns constructed in accordance with the expressions in `column_s`.

In most cases, `eproj` is blended into another relation with an explicit schema during the simplification phase of the optimizer. Redundant `eproj` nodes indicate that no simplification has occurred. This can be forced with:

```
--%PRAGMA SIMPLIFICATION ON
```

Attributes:

<code>column_s</code>	List of expressions that specify how individual columns are evaluated.
<code>from</code>	Operand of the unary operation
<code>outer_reference_s</code>	List of expressions and predicates that can be evaluated independently of the remaining attributes of the node.

SQL fragment:

```
SELECT balance*balance*balance FROM account
```

Extract from the explanation:

```
eproj [
  from : table_scan [
    ...
  ]
  column_s : <
    (column #1) ::= ((BALANCE * BALANCE) * BALANCE),
  >
]
```

7.2.6 full_outer_join

Meaning: Cross product with condition, and all the rows of both tables (padded in each case with NULL) that are not included in the cross product.

The left operand is read sequentially. For each row of the left operand, the right operand is set to the starting position and read sequentially. All right rows that satisfy the join condition with the current left row are searched. A result row is then constructed for each pair of left and right rows found.

If no right row can be found for a left row, a result row is constructed from the left row and padded on the right with NULL. The right operand is then read sequentially. For each row of the right operand, the left operand is set to the starting position and is read sequentially. If no left row that satisfies the predicate together with the right row can be found, a result row is constructed from the right row and filled on the left with NULL.

Attributes:

column_s	List of expressions that specify how individual columns are evaluated.
left, right	Operands, each of which represents a RELATION.
outer_reference_s	List of expressions and predicates that can be evaluated independently of the remaining attributes of the node.
pred	Join condition

SQL fragment:

```
SELECT p1.ps_partkey, p1.ps_availqty + p2.ps_availqty
FROM   partsupp AS p1 FULL JOIN partsupp AS p2
ON     p1.ps_partkey = p2.ps_partkey AND
       p1.ps_suppkey <> p2.ps_suppkey AND
       p1.ps_availqty + p2.ps_availqty <= :maximum
```

Extract from the explanation:

```
full_outer_join [
  left : table_scan [
    column_s: <
      PS_PARTKEY #7,
      PS_SUPPKEY,
      PS_AVAILQTY
    >
  ]
  right : table_scan [
    column_s: <
      PS_PARTKEY #8,
      PS_SUPPKEY #9,
      PS_AVAILQTY #10
    >
  ]
  column_s : <
    PS_PARTKEY #11 ::= PS_PARTKEY #7,
    PS_AVAILQTY #12 ::= PS_AVAILQTY,
    PS_AVAILQTY #13 ::= PS_AVAILQTY #10
  >
  pred : (
    (PS_PARTKEY #8 = PS_PARTKEY #7)
    AND (PS_SUPPKEY <> PS_SUPPKEY #9)
    AND ((PS_AVAILQTY + PS_AVAILQTY #10) <= :MAXIMUM\1)
  )
]
```

7.2.7 full_outer_mjoin

Meaning: Cross product with condition, and all the rows of both tables (padded in each case with NULL) that are not included in the cross product.

The interpreter begins with a sort of the two operands according to the join predicate. This corresponds to an equals condition.

Both operands are read sequentially until a pair that satisfies the join condition is found. A result row is then constructed from the left and right row in that pair. For rows of the left operand that do not appear in any pair, a result row is created from the left row and padded with NULL. Similarly, for rows of the right operand that do not appear in any pair, a result row is created from the right row and padded with NULL.

If a row of the right operand has the same values (in the columns defined by the join condition) as the row which precedes it, the left operand is backtracked by the number of rows that were read for the preceding row of the right operand.

Attributes:

column_s	List of expressions that specify how individual columns are evaluated.
joincond	The equi-join condition between the two operands. The row components of "joincond" are arranged by the guaranteed sort order of the join columns. It should be noted that a comparison of row expressions, which are constructed with the value operator, is performed here.
left, right	Operands, each of which represents a RELATION, sorted by the columns that appear in joincond.
outer_reference_s	List of expressions and predicates that can be evaluated independently of the remaining attributes of the node.
right_is_unique	Is true if the left operand never needs to be backtracked.

SQL fragment:

```
SELECT professors.pname , papers.title
FROM   professors FULL JOIN papers
      ON professors.pnr = papers.pnr
```

Extract from the explanation:

```
full_outer_mjoin [
  left : table_scan [
    column_s: <
      PNR,
      PNAME #1
    >
  ]
  right : table_scan [
    column_s: <
      PNR #2,
      TITLE #3
    >
  ]
  column_s : <
    PNAME ::= PNAME #1,
    TITLE ::= TITLE #3
  >
  joincond : (row(PNR #2) = row(PNR))
  right_is_unique : False
]
```

7.2.8 left_outer_join

Meaning: Cross product with condition, and all rows of the left table (padded with NULL) that are not included in the cross product.

The condition can be found under the right operand.

The left operand is read sequentially. For each row of the left operand, the right operand is set to the starting position and read sequentially. Only those rows which satisfy the join condition with the current left row are returned by the right operand in each case. A result row is then constructed for each pair of left and right rows found. If no right row can be found for a left row, a result row is constructed from the left row and padded on the right with NULL.

Attributes:

column_s	List of expressions that specify how individual columns are evaluated.
left, right	Operands, each of which represents a RELATION.
outer_reference_s	List of expressions and predicates that can be evaluated independently of the remaining attributes of the node.

SQL fragment:

```
SELECT dept.city, emp_city
FROM   dept LEFT JOIN emp
       ON dept.city <> emp_city
```

Extract from the explanation:

```
left_outer_join [
  left : table_scan [
    base: ...DEPT
  ]
  right : rest [
    from: table_scan [
      base: ...EMP
    ]
    pred: (CITY #1 <> EMP_CITY #2)
  ]
  column_s : <
    CITY ::= CITY #1,
    EMP_CITY ::= EMP_CITY #2
  >
]
```

7.2.9 left_outer_mjoin

Meaning: Cross product with condition, and all rows of the left table (padded with NULL) that are not included in the cross product.

The interpreter begins with a sort of the two operands according to the join predicate. This corresponds to an equals condition.

Both operands are read sequentially until a pair that satisfies the join condition is found. A result row is then constructed from the left and right row in that pair. For rows of the left operand that do not appear in any pair, a result row is created from the left row and padded with NULL.

If a row of the right operand has the same values (in the columns defined by the join condition) as the row which precedes it, the left operand is backtracked by the number of rows that were read for the preceding row of the right operand.

Attributes:

column_s	List of expressions that specify how individual columns are evaluated.
joincond	The equi-join condition between the two operands. The row components of "joincond" are arranged by the guaranteed sort order of the join columns. It should be noted that a comparison of row expressions, which are constructed with the value operator, is performed here.
left, right	Operands, each of which represents a RELATION, sorted by the columns that appear in joincond.
outer_reference_s	List of expressions and predicates that can be evaluated independently of the remaining attributes of the node.
right_is_unique	Is true if the left operand never needs to be reset.

SQL fragment:

```
SELECT professors.pname , papers.title
FROM   professors LEFT JOIN papers
      ON professors.pnr = papers.pnr
```

Extract from the explanation:

```
left_outer_mjoin [
  left : table_scan [
    column_s: <
      PNR,
      PNAME #1
    >
  ]
  right : table_scan [
    column_s: <
      PNR #2,
      TITLE #3
    >
  ]
  column_s : <
    PNAME ::= PNAME #1,
    TITLE ::= TITLE #3
  >
  joincond : (row(PNR #2) = row(PNR))
  right_is_unique : False
]
```


7.2.10 mjoin

Meaning: Cross product with condition.

The interpreter begins with a sort of the two operands according to the join predicate. This corresponds to an equals condition.

Both operands are read sequentially until a pair that satisfies the join condition is found. If a row of the right operand has the same values (in the columns defined by the join condition) as the row which precedes it, the left operand is backtracked by the number of rows that were read for the preceding row of the right operand.

Attributes:

joincond	The equi-join condition between the two operands. The row components of "joincond" are arranged by the guaranteed sort order of the join columns. It should be noted that a comparison of row expressions, which are constructed with the value operator, is performed here.
left, right	Operands, each of which represents a RELATION, sorted by the columns that appear in joincond.
outer_reference_s	List of expressions and predicates that can be evaluated independently of the remaining attributes of the node.
right_is_unique	Is true if the left operand never needs to be reset.

SQL fragment:

```
DECLARE c1 CURSOR FOR
    account INNER JOIN history
    ON accnr = hacnr
```

Extract from the explanation:

```
cursor_select_stmt [  
  output: <  
    ACCNR,  
    ...  
  >  
  query: mjoin [  
    left : table_scan [  
      column_s: <  
        HACCNR,  
        ...  
      >  
      sortorder: <  
        HACCNR ascending  
      >  
      ...  
      pred: ((HACCNR IS NOT NULL))  
    ]  
    right : table_scan [  
      column_s: <  
        ACCNR,  
        ...  
      >  
      sortorder: <  
        ACCNR ascending  
      >  
      ...  
    ]  
    joincond : (row(HACCNR) = row(ACCNR))  
    right_is_unique : true  
  ]  
  ...  
]
```

7.2.11 njoin

Meaning: Nested-loop join on the left and right operands.

The right operand is read one row at a time. For each row of the right operand, the left operand is opened, and each of its rows are combined with the right row.

Attributes:

left, right	Operands, each of which represents a RELATION. The left operand contains the join condition with references to the right operand.
outer_reference_s	List of expressions and predicates that can be evaluated independently of the remaining attributes of the node.

SQL fragment:

```
SELECT accnr, balance, amount
FROM (
    account AS a
    INNER JOIN
    history AS h
    ON a.accnr = h.haccnr
) AS x
```

Extract from the explanation:

```
query: eproj [  
  from: nljoin [  
    left : table_scan [  
      outer_reference_s: <  
        (expr #1) ::= ACCNR #2  
      >  
      column_s: <  
        AMOUNT #3,  
        HACCNR  
      >  
      pred: (HACCNR = (expr #1))  
    ]  
    right : table_scan [  
      column_s: <  
        BALANCE #4,  
        ACCNR #2  
      >  
    ]  
  ]  
  column_s: <  
    ACCNR ::= ACCNR #2,  
    BALANCE ::= BALANCE #4,  
    AMOUNT ::= AMOUNT #3  
  >  
]
```

7.2.12 One

Meaning: Table with exactly one empty row
(the unary element in relational algebra).

SQL fragment:

```
--%PRAGMA SIMPLIFICATION ON
SELECT COUNT (*)
FROM   t
WHERE  1 = 2
```

Extract from the explanation:

```
eproject [
  from:  One
  column_s: <
    (column #2) ::= 0
  >
]
```

7.2.13 pre_rest

Meaning: The operand table is restricted to the rows that satisfy the restriction predicate. It is independent of the contents of the table. The result is therefore either the same as the operand or the empty table.

Before the first row is read, the restriction predicate is evaluated. If it is satisfied, the rows of the operand are read. Otherwise, the table is empty. The attributes are the same as those listed under rest.

SQL fragment:

```
SELECT sal
   FROM emp
  WHERE sal <= :p AND :p <= 10000;
```

Extract from the explanation:

```
pre_rest [
  outer_reference_s : <
    (pred #2) ::= (:P\1 <= 10000)
  >
  from : table_scan [
    base: SICAT.SIETEC_DDLOPT.EMP
    pred:
      ((SAL <= :P\1))
  ]
  pred : (pred #2)
]
```

7.2.14 rest

Meaning: The operand table is restricted to the rows that satisfy the restriction predicate.

Attributes:

from	Operand of the unary operation.
outer_reference_s	List of expressions and predicates that can be evaluated independently of the remaining attributes of the node.
pred	The restriction predicate.

SQL fragment:

```
SELECT balance FROM account
WHERE balance > ALL (SELECT hacnr FROM history)
```

Extract from the explanation:

```

rest [
  from : table_scan [
    base: SICAT.SIETEC_DC.ACCOUNT
    column_s: <
      BALANCE
    >
  ]
  pred :
  all [
    outer_reference_s: <
      (expr #1) ::= BALANCE,
      (pred #2) ::= ((expr #1) IS NULL)
    >
    arg: table_scan [
      outer_reference_s: <
        (expr #3) ::= (expr #1)
        (pred #4) ::= (pred #2)
      >
      base: SICAT.SIETEC_DC.HISTORY
      pred:
        (
          ((HACCNR >= (expr #3)) OR (HACCNR IS NULL))
          OR (pred #4)
        )
    ]
    pred: false
  ]
]

```


7.2.15 sort

Meaning: Sorted table.

Before the first row is read, the operand table is written to an intermediate file and sorted. The sorted intermediate file is read thereafter.

Attributes:

from	Operand of the unary operation.
outer_reference_s	List of expressions and predicates that can be evaluated independently of the remaining attributes of the node.
sort_spec_s	Columns by which sorting is performed.

SQL fragment:

```
SELECT tellerid+1 AS tid
FROM teller
ORDER BY tid
```

Extract from the explanation:

```
sort [
  from : eproj [
    from: table_scan [
      ...
    ]
    column_s: <
      TID ::= (TELLERID + 1)
    >
  ]
  sort_spec_s : <
    TID ascending
  >
]
```

7.2.16 sorted_group

Meaning: Grouping operation.

A group consists of all rows of the operand table that have the same values in the columns specified in `group_col_s`. For each group, a result row is determined containing the columns from `group_col_s` and `aggregate_s`, the latter of which are formed for the group. If `aggregate_s` is empty, the operation degenerates to duplicate elimination. If `group_col_s` is empty, the entire operand table is seen as a single group.

The rows of the operand table must be sorted by the grouping columns. The rows are read sequentially. The columns to be aggregated are counted, summed etc., as required. On changing the group, the result row is formed by evaluating the aggregate functions.

Attributes:

<code>aggregate_s</code>	Defines new columns from the elements of a group. The following special operators are available for this purpose: card, all_count, all_min, all_max, all_sum, all_avg, dist_count, dist_sum, dist_avg
<code>from</code>	Operand of the unary operation, sorted by the <code>group_col_s</code> .
<code>group_col_s</code>	The columns that define the grouping.
<code>outer_reference_s</code>	List of expressions and predicates that can be evaluated independently of the remaining attributes of the node.

SQL fragment:

```
SELECT t1_dec_1,
       COUNT(*),
       COUNT(DISTINCT t1_num_1),
       MIN (t1_num_1),
       MAX (t1_num_1),
       SUM (t1_num_1),
       AVG (t1_num_1)
FROM t1
GROUP BY t1_dec_1
```

Extract from the explanation:

```
sorted_group [
  from : table_scan [
    ...
    sortorder: <
      t1_DEC_1 ascending,
      T1_NUM_1 ascending
    >
  ]
  group_col_s : <
    T1_DEC_1 #24 ::= T1_DEC_1
  >
  aggregate_s : <
    (column #25) ::= card,
    (column #26) ::= dist_count(T1_NUM_1),
    (column #27) ::= all_min(T1_NUM_1),
    (column #28) ::= all_max(T1_NUM_1),
    (column #29) ::= all_sum(T1_NUM_1),
    (column #30) ::= all_avg(T1_NUM_1)
  >
]
```

7.2.17 store_temp

Meaning: Identity.

The operating table is buffered in an intermediate file.

Since the creation of an intermediate file is an expensive operation, it is not executed twice if possible. The interpreter checks the attribute `outer_reference_s` before reopening the table `store_temp`. It reverts to the existing intermediate file if no values of `outer_reference_s` have changed. Among other things, this means that without the `outer_reference_s`, `store_temp` is always evaluated only once.

Attributes:

<code>from</code>	Operand of the unary operation.
<code>outer_reference_s</code>	List of expressions and predicates that can be evaluated independently of the remaining attributes of the node.

SQL fragment:

```
SELECT ...
FROM ...
GROUP BY ...
HAVING xxx IN ( SELECT SUM ( sa1 )
                FROM   emp AS e2
                GROUP BY dept
                HAVING yyy > MIN ( e2.sa1 ) )
```

Extract from the explanation:

```

some [
  outer_reference_s: <
    (expr #1) ::= Ausdruck für xxx
    (expr #2) ::= Ausdruck für yyy
  >
  from: rest [
    outer_reference_s: <
      (expr #7) ::= (expr #1),
      (expr #8) ::= (expr #2)
    >
    from: store_temp [
      from : sorted_group [
        from: ...
        group_col_s: <
          DEPT #24 ::= DEPT #21
        >
        aggregate_s: <
          (column #25) ::= all_min(SAL #22),
          (column #26) ::= all_sum(SAL #23)
        >
      ]
    ]
    pred: (
      ((expr #7) = (column #26))
      AND ((expr #8) > (column #25))
    )
  ]
  pred: true
]

```

Note on the example

The two HAVING clauses are combined into a predicate. The sorted_group table under “rest” does not depend on any outer values. Due to the inserted operator store_temp, it is only evaluated once (since there are no outer_reference_s) and then buffered.

7.2.18 table_function_scan

Meaning: Reads column values from a CSV file using the table function CSV().

CSV files can be read using the table function CSV(), and their values can be interpreted using SESAM/SQL means. The access to CSV files is output as the “table_function_scan” node in the EXPLAIN pragma.



Such file accesses can be critical to performance because when files are accessed no access paths such as primary keys or secondary indexes can be used. Files are always read sequentially.

Attributes:

column_s	The columns required from the CSV input file (the column numbers output are generated internally).
input file	Name of the CSV input file.
field separator	Delimiter character for the column values of the CSV file.
quote character	Quote character in which the column values in the CSV file can be enclosed.
escape character	Escape character for escaping special characters.

SQL fragment:

```
SELECT *
FROM TABLE (CSV(...))
WHERE ...
```

Extract from the explanation:

```
from: table_function_scan [
  column_s: <
    _Tb1CSVCo100001_,
    _Tb1CSVCo100007_,
    _Tb1CSVCo100004_,
    _Tb1CSVCo100003_
  >
  input file: 'MYFILE.CSV'
  field separator: ';'
  quote character: '"'
  escape character: '\'
]
```

...

7.2.19 table_scan

Meaning: All rows of a base table that satisfy the specified predicate, but only including the specified columns.

If the predicate includes values that are only known at the time of interpretation (parameters, references to columns of other tables, etc.), the indexes to be used are selected before reading the first row. The rows of the base table are then read in accordance with the selected access path.

Attributes:

base	The base table, specified in the form: catalog-name.schema-name.table-name
column_s	List of required columns of the base table
cost_ratio	The indicated numbers help developers to fine-tune the optimizer
has_at_most_one_row	Is true if there can be no more than one row
isolation level	Predefined values: uncommitted consistency level 1 read committed repeatable read serializable Value determined at time of interpretation: set at runtime at least repeatable read (see also section “Synchronization” on page 45)
lock mode	shared exclusive
outer_reference_s	List of expressions and predicates that can be evaluated independently of the remaining attributes of the node.
pred	The selected rows must satisfy the predicate.
sortorder	Columns for which the order (ascending or descending) is to be guaranteed.
used index	List of indexes that could be used for access to the base table (see section “Access to base tables” on page 36).

SQL fragment:

```
UPDATE account
SET balance = balance + :betrag
WHERE accnr >= :acc_id + 1
```

Extract from the explanation:

```
update_stmt [
  object_rows: table_scan [
    outer_reference_s : <
      (expr #1) ::= (:ACC_ID\2 + 1)
    >
    base : DCCAT.DC_SCHEMA.ACCOUNT
    column_s : <
      ACCNR,
      BALANCE #3
    >
    sortorder : <
      ACCNR ascending
    >
    used index : < Primary Key >
    isolation level : at least repeatable read
    lock mode : exclusive
    cost_ratio : 0.000000E+00/0.000000E+00
    pred : (ACCNR >= (expr #1))
  ]
  ...
]
```


7.2.20 union

Meaning: Union between the two operands left and right.

The left and right operand are read in succession sequentially.

Attributes:

column_s Defines the columns of the new table as pairs of columns of the operands.

left, right Operands, each of which represents a RELATION.

outer_reference_s List of expressions and predicates that can be evaluated independently of the remaining attributes of the node.

SQL fragment:

```
SELECT *
FROM (
    SELECT dept FROM emp
    UNION
    SELECT dept_no AS dept FROM dept
) AS u
```

Extract from the explanation:

```

sorted_group [
  from: sort [
    from: union [
      left : eproj [
        from: ...
        column_s: <
          DEPT #2 ::= ...
        >
      ]
      right : eproj [
        from: ...
        column_s: <
          DEPT #3 ::= ...
        >
      ]
      column_s : <
        DEPT #4 ::= (DEPT #2, DEPT #3)
      >
    ]
    sort_spec_s: <
      DEPT #4 ascending
    >
  ]
  group_col_s: <
    DEPT ::= DEPT #4
  >
]

```

Note on the example

Since duplicates must be eliminated for the SQL operation UNION, the operations sort and sorted_group are applied on the union.

7.2.21 virtual_scan

Meaning: Reads metadata which is not stored in the catalog space.

When accessing certain INFORMATION_SCHEMA views, you may require metadata which is not stored in the catalog space. For instance, the information determined using the SQL_FEATURES view cannot be stored in the catalog space. This information is managed on a version-specific basis by the SESAM/SQL Database Handler. In access plans for INFORMATION_SCHEMA views that use this type of internal information, access operations are represented by the virtual_scan operator. This operator represents access to a virtual metadata table in the catalog space, which cannot be addressed explicitly.

Attributes:

base	Addressed virtual table, specified in the format <i>catalog</i> .DEFINITION_SCHEMA. <i>table</i> .
column_s	Columns required by the virtual table.
cost_ratio	Estimated cost of access; required by the optimizer when creating plans.

SQL fragment:

```
SELECT *
FROM   INFORMATION_SCHEMA.SQL_FEATURES
WHERE  FEATURE_ID < 'E020'
```

Extract from the explanation:

```
...
  from: virtual_scan [
    base: TESTCAT.DEFINITION_SCHEMA.SQL_FEATURES
    column_s: <
      SUB_FEATURE_NAME,
      SUB_FEATURE_ID,
      IS_VERIFIED_BY,
      IS_SUPPORTED,
      FEATURE_SUBFEATURE_PACKAGE_CODE,
      FEATURE_NAME,
      FEATURE_ID,
      COMMENTS
    >
    cost_ratio: 1.000000E+00/3.590000E+02
  ]
...
```

7.3 DML statements

7.3.1 UPDATE statement

Meaning: Updates rows of a base table.

The rows of `object_rows` are determined and modified. `check_on_the_fly` is verified and the row is written back.

`check_after_all` is verified at the end.

Attributes:

<code>check_after_all</code>	A predicate whose truth value must be tested as a result of integrity constraints. The test is performed at the end of the statement, since it references several/all rows of the base table. (If true is entered here, nothing needs to be tested.)
<code>check_on_the_fly</code>	A predicate whose truth value must be tested as a result of integrity constraints. The test is performed before updating each individual row. (If true is entered here, nothing needs to be tested.)
<code>const_col_s</code>	See <code>updated_col_s</code> .
<code>cursor_id_ref</code>	If the <code>object_rows</code> contain <code>cursor_scan</code> , this entry will contain a reference to the cursor in the form <code>module-name.cursor-name</code>
<code>object_rows</code>	Table that determines the rows to be updated.
<code>target</code>	The base table to be updated. The node <code>base-table</code> is always shown in the form <code>catalog-name.schema-name.table-name</code> .
<code>updated_col_s, const_col_s</code>	You define the columns of the base table to be updated. <code>const_col_s</code> contains the values which do not change during execution, i.e. values that are the same for each row to be updated. The updates to be performed appear in the explanation as a list of “ <code>column_set</code> ”s: the column to be updated is on the left; the value to be entered is on the right. (For more details on node formats, see section “Expressions, function calls, and predicates” on page 195.) All types of expressions are possible as values, including the following entities, which are determined by the interpreter: user , system_user , current_catalog , current_isolation_level , current_referenced_catalog , current_schema , current_date , current_time , current_timestamp for <code>CURRENT_USER</code> , <code>SYSTEM_USER</code> etc.

SQL fragment:

```
UPDATE account
SET balance = balance + :amount
WHERE accnr = :acc_id
```

Extract from the explanation:

```
update_stmt [
  check_after_all : true
  target : DCCAT.DC_SCHEMA.ACCOUNT
  object_rows : rest [
    from: ...
    pred: (ACCNR = :ACC_ID\2)
  ]
  check_on_the_fly : true
  cursor_id_ref : Void
  const_col_s : <>
  updated_col_s : <
    BALANCE ::= (BALANCE #2 + :AMOUNT\1)
  >
]
```

7.3.2 INSERT statement

Meaning: Inserts rows into a base table.

check_on_the_fly is verified for each row of the table object_rows. The row is then inserted into the target.

check_after_all is verified at the end.

Attributes:

check_after_all	A predicate whose truth value must be tested as a result of integrity constraints. The test is performed at the end of the statement, since it references several/all rows of the base table. (If true is entered here, nothing needs to be tested.)
check_on_the_fly	A predicate whose truth value must be tested as a result of integrity constraints. The test is performed before updating each individual row. (If true is entered here, nothing needs to be tested.)
const_col_s	See inserted_col_s.
inserted_col_s, const_col_s	You define the columns of the base table to be set. const_col_s contains the values which do not change during execution, i.e. values that are the same for each row to be inserted. The insertions to be performed appear in the explanation as a list of "column_set"s: the column to be set is on the left; the value to be entered is on the right. (For more details on node formats, see section "Expressions, function calls, and predicates" on page 195.) All types of expressions are possible as values, including the following entities, which are determined by the interpreter: user , system_user , current_catalog , current_isolation_level , current_referenced_catalog , current_schema , current_date , current_time , current_timestamp for CURRENT_USER, SYSTEM_USER etc. The expression generate_key may also appear in the Insert statement. It causes a key to be generated automatically.
object_rows	Table that determines the rows to be inserted.
output	If "*" is specified as a value for a column, a reference to this column, whose value is assigned to the output parameter, is entered here.
target	The base table to be updated. The node base-table is always shown in the form catalog-name.schema-name.table-name.

SQL fragment:

```
INSERT INTO account
VALUES (*, :abranchid      )
      RETURN INTO :gen
```

Extract from the explanation:

```
insert_stmt [
  output : <
    ACCNR
  >
  check_after_all : true
  target : DCCAT.DC_SCHEMA.ACCOUNT
  object_rows : One
  check_on_the_fly : true
  const_col_s : <
    ACCNR := generate_key,
    ABRANCHID := :ABRANCHID\1
  >
]
```

SQL fragment:

```
INSERT INTO history (haccnr, amount, htellerid, hrest)
SELECT * FROM account
```

Extract from the explanation:

```
insert_stmt [
  check_after_all : true
  target : DCCAT.DC_SCHEMA.HISTORY
  object_rows : eproj [
    from: table_scan [
      base: DCCAT.DC_SCHEMA.ACCOUNT
      ...
    ]
    column_s: <
      ACCNR #1 ::= ACCNR,
      ABRANCHID #2 ::= ABRANCHID,
      BALANCE #3 ::= BALANCE,
      AREST #4 ::= AREST
    >
  ]
  check_on_the_fly : true
  inserted_col_s : <
    HACCNR ::= ACCNR #1,
    AMOUNT ::= ABRANCHID #2,
    HTELLERID ::= BALANCE #3,
    HREST ::= AREST #4
  >
]
```


7.3.3 MERGE statement

Meaning:

Inserts rows into a base table or modifies rows in a base table.

Initially the object_rows are calculated; then, for each row of the object_rows, either the INSERT into the target table or the UPDATE for the target table is performed from the source table depending on whether or not rows exist in the target table with correspond to a row in the source table with respect to the ON condition.

Attributes:

check_after_all	A predicate whose truth value must be tested as a result of integrity constraints. The test is performed at the end of the statement, since it references several/all rows of the base table. (If true is entered here, nothing needs to be tested.)
target	The base table to be updated. The node base-table is always shown in the form catalog-name.schema-name.table-name.
object_rows	Calculated intermediate table from which values for UPDATE or INSERT can be taken and on the basis of which it is decided which records are inserted into or modified in the target table.
inserted_col_s, inserted_const_col_s	You define the columns of the base table to be set when INSERT applies for MERGE. inserted_const_col_s contains the values which do not change during execution, i.e. values that are the same for each row to be inserted. The insertions to be performed appear in the explanation as a list of "column_set"s: the column to be set is on the left; the value to be entered is on the right. The expression generate_key may also appear in the statements. It causes an integer field to be generated automatically.
ins_check_on_the_fly	A predicate whose truth value must be tested, as a result of integrity constraints, for every record which was inserted by means of INSERT. (If true is entered here, nothing needs to be tested.)

updated_col_s, updated_const_col_s

You define the columns of the base table to be updated in the case of UPDATE.

updated_const_col_s contains the values which do not change during execution, i.e. values that are the same for each row to be updated.

The updates to be performed appear in the explanation as a list of "column_set"s: the column to be updated is on the left; the value to be entered is on the right.

upd_check_on_the_fly

A predicate whose truth value must be tested, as a result of integrity constraints, for every record which is modified by means of UPDATE. (If true is entered here, nothing needs to be tested.)

SQL fragment:

```
merge into t10
  using t9
    on t10.c001 = t9.c001
 when matched then update set c004 = t10.c004 + t9.c004
                          , c005 = 17
 when not matched then insert (c001, c003) values (t9.c007, 27)
```

Extract from the explanation:

```
merge_stmt [
  check_after_all: true
  target: CAT.SCH.T10
  object_rows: sort [
    from: left_outer_join [
      left: table_scan [
        base: CAT.SCH.T9
        column_s: <
          C001 #3,
          C007,
          C004 #4
        >
        isolation level: set at runtime
        lock mode: shared
        cost_ratio: 6.000000E+00/1.000000E+01
      ]
    ]
  ]
```

```

right: table_scan [
  outer_reference_s: <
    (expr #5) ::= C001 #3
  >
  base: CAT.SCH.T10
  column_s: <
    C001 #6,
    row_id,
    C004 #7
  >
  used index: <Primary Key, NoName>
  ...
  has_at_most_one_row: True
  pred:
    ((C001 #6 = (expr #5)))
]
column_s: <
  C004 #8 ::= C004 #7,
  C004 #9 ::= C004 #4,
  C007 #10 ::= C007,
  (column #11) ::= row_id
>
]
sort_spec_s: <
  (column #11) descending
>
]
...
inserted_const_col_s: <
  C003 ::= 27
>
inserted_col_s: <
  C001 ::= C007 #10
>
ins_check_on_the_fly: true
updated_const_col_s: <
  C005 ::= 17
>
updated_col_s: <
  C004 ::= (C004 #8 + C004 #9)
>
upd_check_on_the_fly: true
]

```

7.3.4 DELETE statement

Meaning: Deletes rows of a base table.

The rows of the table `object_rows` are to be deleted. For each row, the truth value of `check_on_the_fly` without that row is verified, and the row is then deleted. `check_after_all` is verified at the end.

Attributes:

<code>check_after_all</code>	A predicate whose truth value must be tested as a result of integrity constraints. The test is performed at the end of the statement, since it references several/all rows of the base table. (If true is entered here, nothing needs to be tested.)
<code>check_on_the_fly</code>	A predicate whose truth value must be tested as a result of integrity constraints. The test is performed before updating each individual row. (If true is entered here, nothing needs to be tested.)
<code>cursor_id_ref</code>	If the <code>object_rows</code> contain <code>cursor_scan</code> , this entry will contain a reference to the cursor in the form <code>module-name.cursor-name</code>
<code>object_rows</code>	Table that determines the rows to be deleted.
<code>target</code>	The base table to be updated. The node <code>base-table</code> is always shown in the form <code>catalog-name.schema-name.table-name</code> .

SQL fragment:

```
DELETE FROM account
WHERE accnr > :acc_id
```

Extract from the explanation:

```
delete_stmt [
  check_after_all : true
  target : DCCAT.DC_SCHEMA.ACCOUNT
  object_rows : rest [
    from: ...
    pred: (ACCNR > :ACC_ID\1)
  ]
  check_on_the_fly : true
  cursor_id_ref : Void
]
```

7.3.5 DECLARE CURSOR statement

Meaning: Declaration of a cursor.

If the cursor was opened with SCROLL, an intermediate file is opened for complex queries (see [section “Intermediate files” on page 203](#)).

On positioning, the table “query” is read at the appropriate position.

If an intermediate file exists:

if the table is being read sequentially, each row that is read from it is written to the intermediate file. For any type of positioning other than sequential, the entire unread portion of the table is first written to the intermediate file. The intermediate file is then positioned and read from that position.

Attributes:

is_for_update	True if and only if the FOR UPDATE clause is specified.
output	A list of expressions that are assigned to the output parameters in order.
query	The table to be processed.
read_only	True if the cursor is not permitted for UPDATE... CURRENT and DELETE... CURRENT.

SQL fragment:

```
DECLARE any_cursor CURSOR FOR
SELECT accnr FROM account
```

Extract from the explanation:

```
cursor_select_stmt [
  output : <
    ACCNR
  >
  query : table_scan [
    ...
  ]
  is_for_update : false
  read_only : false
]
```

7.3.6 Single SELECT statement

Meaning: Reads a table with a single row.

A row of the table “query” is read, and a test is performed to verify that no further rows exist.

Attributes:

output A list of expressions that are assigned to the output parameters in order.

query The single-row table.

SQL fragment:

```
SELECT accnr
INTO :accnr
FROM account
WHERE balance = 2000
```

Extract from the explanation:

```
single_select_stmt [
  output : <
    ACCNR
  >
  query : rest [
    ...
  ]
]
```

7.3.7 Dynamic SELECT statement

A `dynamic_select_stmt` is generated by the optimizer only if no distinction between `cursor_spec` and `single_select_stmt` is possible with dynamic SQL at compile time. The use of the statement is only evident when it is run. The semantics and processing are then determined.

The attributes are the same as for `cursor_select_stmt` or `single_select_stmt`, depending on how the statement is used.

SQL fragment:

```
SELECT accnr FROM account WHERE ? = 1
```

Extract from the explanation:

```
dynamic_select_stmt [
  output : <
    ACCNR
  >
  query : pre_rest [
    ...
  ]
  is_for_update : false
  read_only : false
]
```

7.3.8 Internal EXPORT DATA statement

Meaning: Defines an access plan for determining the rows to be exported in an EXPORT statement with a WHERE clause.

The WHERE clause of an EXPORT statement is used to select which table rows are to be exported. During execution of the EXPORT statement, it triggers an internal EXPORT DATA statement in order to determine the rows affected. The optimizer also generates an access plan for this internal statement. By specifying the EXPLAIN pragma in the EXPORT TABLE utility statement, you can force the output of this access plan to a file.



In the access plan of an EXPORT DATA statement, the underlying table_scan outputs the row numbers of all rows found (in the special attribute row_id), as these will be required for the internal processing of the statement.

The plan for the internal EXPORT DATA statement is not generated until the EXPORT utility statement is actually executed, and not while it is being prepared.

Attributes:

output	Consists exclusively of the special attribute row_id described above.
query	Derived table consisting of all rows found.
target	Table from which data is to be exported.

SQL fragment:

```
EXPORT TABLE T10 INTO FILE 'EXPORTFILE.T10' WHERE C001 < 100
```

Extract from the explanation:
(of the internal EXPORT DATA statement):

```
export_data_stmt [  
  output: <  
    _ROW_ID  
  >  
  query: eproj [  
    from: table_scan [  
      base: CAT.SCH.T10  
      column_s: <  
        row_id,  
        ...  
      >  
      sortorder: <  
        C001 ascending  
      >  
      sort_index: Primary Key  
      used index: <Primary Key>  
      ...  
      pred: ((C001 < 100))  
    ]  
    column_s: <  
      _ROW_ID ::= row_id,  
      C001 #1 ::= C001  
    >  
  ]  
  target: CAT.SCH.T10  
]
```

7.3.9 Internal SELECT FOR UNLOAD statement

Meaning: Determines the records to be unloaded in the case of a utility statement UNLOAD ONLINE with WHERE clause.

In the case of a utility statement UNLOAD ONLINE with the WHERE clause the records to be unloaded can be selected from a table or a view. In this case a SELECT FOR UNLOAD statement which is used to determine the records affected is called internally when the statement is processed. The optimizer also generates an access plan for this internal statement. When the EXPLAIN pragma is specified in UNLOAD ONLINE, the plan for the internal SELECT FOR UNLOAD statement is output to a file.

The plan for the internal SELECT FOR UNLOAD statement is only generated while UNLOAD ONLINE executes, not when it is prepared.

Attributes:

output	Values which are to be unloaded.
query	Derived table consisting of all rows found.

SQL fragment:

```
UNLOAD ONLINE TABLE T INTO FILE 'MYFILE.TXT'  
DELIMITER_FORMAT TERMINATED BY ';' WHERE C001 < 100
```

Extract from the explanation:

```
unload_data_stmt [  
  output: <  
    C001,  
    ...  
  >  
  query: eproj [  
    from: table_scan [  
      base: CAT.SCH.T  
      column_s: <  
        C001 #1,  
        ...  
      >  
      used index: <Primary Key>  
      ...  
      pred:  
        ((C001 #1 < 100))  
    ]  
    column_s: <  
      C001 ::= C001 #1,  
      ...  
    >  
  ]  
]
```

7.3.10 CALL statement

Meaning: Calls a procedure (Stored Procedure).

The myproc procedure is defined as follows:

```
create procedure myproc(in par1 integer, in par2 integer, out par3 integer)
modifies sql data
begin
  declare var1 integer default -1;
  declare var2 integer default null;
  set var2 = (select max(c001) from t1) + par1 * par2 * var1;

  if ((select min(c001) from t2) + var2 > 0)
  then set par3 = var2 * 1000;
  elseif (var2 = 0)
  then set par3 = 0;
  else set par3 = 1000000 * var2;
end if;
end
```

This procedure is called with:

```
--%pragma explain into 'MYPROC.EXPLAIN'
call myproc(17,18,?)
```

The explanation below is generated. This contains the usual explanations of the DML statements INSERT, UPDATE, DELETE, MERGE, SELECT and DECLARE CURSOR contained in the procedure.

It also contains the explanations of the procedure statements IF, LOOP, LEAVE and SET:

if_stmt

Explanation of the IF statement with an `if` block, with no, one or more than one `elseif` blocks, and possibly one `else` block.

An `if` or `elseif` block consists of a condition and the associated statements in the `then` block. These are executed if the condition is satisfied.

The `else` block contains the statements which are executed if no condition is satisfied in the `if` or `elseif` blocks.

assign_stmt

Explanation of the SET statement. In addition to the `target` assignment, an explanation of the `source` expression is supplied. `source` can contain subqueries which are then explained like ordinary DML statements.

loop_stmt

Explanation of the LOOP statement. This contains the sequence of statements `stmt_s` of the loop body.

leave_stmt

Explanation of the LEAVE statement. This defines that a loop is exited. In the case of nested loops, it also defines which loops are exited.

The output variables of the procedure, the local variables with default values and the input values of the CALL statement are also displayed. The input values can contain subqueries whose calculation is then explained as in other DML statements.

Extract from the explanation:

```
--%pragma explain into 'MYPROC.EXPLAIN'
call myproc(17,18,?)
call_stmt [
  output: <
    :par->PAR3
  >
  stmt: compound_stmt [
    is_atomic: False
    variable_s: <
      :var->VAR1,
      :var->VAR2
    >
    stmt_s: <
      assign_stmt [
        target: :var->VAR2
        source: plus [
          left: cast_rel_to_row [
            from: eproj [
              from: sorted_group [
                from: table_scan [
                  base: THUCAT.JOINSCHEMA.T1
                  column_s: <
                    C001
                  >
                  isolation level: set at runtime
                  lock mode: shared
                  cost_ratio: 1.000000E+01/9.200000E+01
                ]
                aggregate_s: <
                  (column #1) ::= all_max(C001)
                >
              ]
            ]
          column_s: <
            (column #2) ::= (column #1)
          >
        ]
      ]
    ]
  ]
]
```

```

    right: ((:par->PAR1 * :par->PAR2) * :var->VAR1)
  ]
],
if_stmt [
  if:
    gt [
      left: cast_rel_to_row [
        from: sorted_group [
          from: table_scan [
            base: THUCAT.JOINSCHEMA.T2
            column_s: <
              C001 #3
            >
            isolation level: set at runtime
            lock mode: shared
            cost_ratio: 1.000000E+00/1.000000E+00
            has_at_most_one_row: True
          ]
          aggregate_s: <
            (column #4) ::= all_min(C001 #3)
          >
        ]
      ]
      right: (- :var->VAR2)
    ]
  then: <
    assign_stmt [
      target: :par->PAR3
      source: (:var->VAR2 * 1000)
    ]
  >,
  elseif: (:var->VAR2 = 0)
  then: <
    assign_stmt [
      target: :par->PAR3
      source: 0
    ]
  >
  else: <
    assign_stmt [
      target: :par->PAR3
      source: (1000000 * :var->VAR2)
    ]
  >
]
>

```

```
        proc_var_default_s: <
            -1,
            null
        >
    ]
    value_s: <
        17,
        18,
        :\1
    >
]
```

7.4 Quantifiers

some, all quantifiers. A test is performed to ascertain whether a predicate applies to at least one row or to all rows of a table.

If all `outer_reference_s` in a subsequent evaluation have the same values as before, reevaluation of the quantifier is suppressed, and the old result is used. If the table EMP in the following example has multiple rows with the same SAL value, the quantifier for it will be evaluated only once. (A quantifier without `outer_reference_s` is evaluated once only.)

Attributes:

<code>arg</code>	The table to be tested.
<code>outer_reference_s</code>	List of expressions and predicates that can be evaluated independently of the remaining attributes of the node.
<code>pred</code>	The test condition. The optimizer attempts to insert the condition into the argument with the goal of reducing the number of rows found for all tables to a minimum. Consequently, the quantifier “some” has the predicate “true” in most cases, i.e. evaluates to true if the table is not empty. In the case of “all”, an attempt is made to insert the negated predicate into the argument, so “all” usually has the predicate “false”, which means that the quantifier evaluates to true if the table is empty.

SQL fragment:

```
SELECT sal
FROM emp
WHERE sal = ANY ( SELECT 10*s FROM ... )
```


Extract from the explanation:

```
rest [
  from: table_scan [
    ...
  ]
  pred: ((SAL IS NOT NULL)
        AND
        some [
          outer_reference_s : <
            (expr #1) ::= SAL
          >
          arg : rest [
            ...
            pred: (
              ...
              AND ((expr #1) = (10 * S))
            )
          ]
          pred : true
        ]
  )
]
```

7.5 Value queries

A value query is an expression that is evaluated from the database contents. It appears in the explanation as the node **cast_rel_to_row**.

Attributes:

outer_reference_s List of expressions and predicates that can be evaluated independently of the remaining attributes of the node. If all **outer_reference_s** in a subsequent evaluation have the same values as before, reevaluation of the value query is suppressed, and the old result is used.

from Table to be evaluated (1 row, 1 column).

SQL fragment:

```
WHERE ( SELECT subject
        FROM lectures AS l
        WHERE l.dnr = d.dnr
      )
=
( SELECT ... )
```

Extract from the explanation:

```
eq [
  left: cast_rel_to_row [
    outer_reference_s : <
      (expr #1) ::= DNR
    >
    from : rest [
      ...
      pred: (DNR #2 = (expr #1))
    ]
  ]
  right: cast_rel_to_row [
    ...
  ]
]
```

7.6 Expressions, function calls, and predicates

Expressions, function calls, and predicates are usually not critical to performance, so their presence in the explanation is primarily intended to allow specific parts of the SQL statement to be found in the explanation.

Expressions, predicates, etc., appear in the explanation in a format that resembles the original SQL statement. There are, however, three exceptions:

1. Multiple column references with identical names

If there are references to different columns with the same name or if different tables reference a single column, a number is appended to the column name to make the reference unique.

Example

```
ACCOUNT #3
```

2. References to common subexpressions/subpredicates

References to common subexpressions/subpredicates are defined in the `outer_reference_s` or `column_s`.

Example

```
(expr #1) ::= SAL
(pred #2) ::= ((expr #1) IS NULL)
(column #3) ::= ('mnopqr' || T1_CHAR)
```

Other common subexpressions/subpredicates are provided with a label consisting of the node name and a number.

Example

```
atom_def #11: (PNAME = :PROF1\\1)
(SAL / scalar_literal #2: 4) > 5
null #46: null = AMOUNT
```

3. Value queries

When a value from the database contents is computed, the corresponding expression contains a table-value node that explodes the syntax of a simple expression (see [section “Value queries” on page 194](#)). The expression is then output in node format.

Example

```
balance + (SELECT btota1 FROM branch WHERE branchid = 1)
plus [
  left: BALANCE #2
  right: cast_rel_to_row [
    from : rest [
      ...]
    ]
  ]
]
```

Node formats in connection with value queries

There follows a list of all node formats that can occur in connection with value queries.

In the attributes `const_col_s`, `inserted_col_s` and `updated_col_s`:

```
column_set [
  column :
  expr :
]
```

Expressions

The nodes **plus**, **minus**, **times**, **divide**, **concat**, **negation** and **cast** stand for +, binary -, *, /, ||, unary - and the cast function.

Example

```

plus [
  left :
  right :
]
negation [
  arg :
]

cast [
  arg :
]

```

The nodes **position**, **char_length**, **substring**, **upper**, **lower**, **trim_both**, **trim_leading** and **trim_trailing** stand for the text functions POSITION, CHAR_LENGTH, SUBSTRING, UPPER, LOWER, TRIM(BOTH...), TRIM(LEADING...) and TRIM(TRAILING...).

Example

```

position [
  arg :
  part :
]

substring [
  arg :
  from :
  for :
]

```

In the same way, conditional expressions can in certain circumstances be output in node format in connection with value queries.

The nodes **nullif** and **coalesce** stand for the corresponding keywords.

Example

```

nullif [
  left :
  right :
]

```

and

```

coalesce [
  expr_s : <
                                     -- comma-delimited list of WHEN/THEN expressions
>
]

```

The node **case** stands for the CASE clause.

Example

```

case [
  arg :      -- Optional CASE - argument
  default :  -- Value of the ELSE clause
  case_s : <
                                     -- comma-delimited list of WHEN/THEN expressions
>
]

```

Depending on their complexity, the WHEN/THEN expressions can occur as SQL equivalents or as the **when_then** node format.

Example

```

when_then [
  when :
  then :
]

```



Functions such as ABS(), MOD(), etc. are normally output as in the SQL text. However, depending on the complexity of the argument, the output can also occur as a node with the name "specific_value_call".

Function calls

Example

The simple function call `select fff(c001+c002) from t10` supplies:

```
dynamic_select_stmt [  
  output: <  
    (column #1)  
  >  
  query: eproj [  
    from: table_scan [  
      base: THUCAT.JOINSCHEMA.T10  
      column_s: <  
        C001,  
        C002  
      >  
      isolation level: set at runtime  
      lock mode: shared  
      cost_ratio: 6.000000E+00/9.000000E+00  
    ]  
    column_s: <  
      (column #1) ::= THUCAT.JOINSCHEMA.FFF((C001 + C002))  
    >  
  ]  
  is_for_update: False  
  read_only: True
```

The function value is thus, as expected, displayed as a simple function call with a fully qualified function name: `THUCAT.JOINSCHEMA.FFF((C001 + C002))`

If the function's parameters contain a complex expression which explodes the syntax of a simple expression (see [section "Value queries" on page 194](#)), the function call may possibly be output in node format.

For the function call in the `select fff((select max(c001) from t2 where c001 < t1.c001)) from t1` statement, the value is output as a function_value as follows:

```
function_value [
  function: THUCAT.JOINSCHEMA.FFF
  value_s: <
    cast_rel_to_row [
      outer_reference_s: <
        (expr #2) ::= C001
      >
      from: eproj [
        from: sorted_group [
          from: table_scan [
            outer_reference_s: <
              (expr #3) ::= (expr #2)
            >
            base: THUCAT.JOINSCHEMA.T2
            column_s: <
              C001 #4
            >
            used index: <Primary Key, Interval>
            isolation level: set at runtime
            lock mode: shared
            cost_ratio: 1.000000E+00/1.000000E+00
            pred:
              ((C001 #4 < (expr #3)))
          ]
          aggregate_s: <
            (column #5) ::= all_max(C001 #4)
          >
        ]
        column_s: <
          (column #6) ::= (column #5)
        >
      ]
    ]
  >
]
```


Predicates

The nodes **eq**, **ne**, **lt**, **gt**, **le** and **ge** stand for =, <>, <, >, <= and >=.

Example

```
eq [
  left :
  right :
```

The nodes **is_null** and **is_not_null** stand for IS NULL and IS NOT NULL.

Example

```
is_null [
  arg :
```

The nodes **is_like** and **is_not_like** stand for IS LIKE and IS NOT LIKE.

Example

```
is_like [
  match_value :      - String to be checked
  pattern :          - Pattern
  escape :           - Wildcard (joker character)
```

The nodes **between**, **not_between** stand for BETWEEN, NOT BETWEEN.

Example

```
not_between [
  mid :
  left :
  right :
```

The node **reg_exp_like_pred** stands for LIKE_REGEX.

Example

```
reg_exp_like_pred [  
  match_value :  
  pattern_automaton :  
  op :  
]
```

The node **castable_pred** stands for IS CASTABLE.

Example

```
castable_pred [  
  arg :  
  op :  
]
```

There are no node formats for AND and OR, even if the operands have node formats.

7.7 Intermediate files

Intermediate files are created when a table needs to be read more than once. Other reasons for buffering are technical in nature, e.g. sorting a table (with a sort operation).

As far as possible, intermediate files are avoided. Instead, an attempt is made to exploit SESAM features and directly read the base table as required. On the other hand, it may be practical to evaluate a frequently required and cost-intensive intermediate result table only once. The optimizer generates the operation `store_temp` in such cases.

Due to the architecture of the SQL handler, there is one case in which the optimizer cannot decide whether or not an intermediate file is required. Since only the queries contained in `DECLARE CURSOR` statements are passed to the optimizer and not the statements themselves, it is not known at optimization if the cursor is scrollable. Consequently, all tables must be essentially prepared for scrolling. To avoid creating intermediate files unnecessarily, the interpreter decides on the basis of the parameters passed to it whether or not this is really essential.

Buffering in intermediate files is performed by the interpreter by two methods. When executing `store_temp` and sort operations at the time of an `Open`, the entire table is written to the intermediate file (and sorted). The table is then read from the intermediate file. If a cursor is opened in scroll mode, the intermediate file is created on opening the table, but not filled. During a sequential read, the table is written to the intermediate file one row at a time. However, the first deviation in positioning from the sequential read causes the entire table to be read until the end and written to the intermediate file. Thereafter, the intermediate file is positioned and read from that point on.

The creation of scrollable intermediate files is, however, not performed blindly. If the underlying table has already been buffered or is a base table, it is possible to scroll without an intermediate file.

8 Performance-related aspects of utility functions and DDL statements.

This chapter describes how to optimize use of the utility functions and of DDL statements.

8.1 LOAD - Load user data into a base table

In addition to the utility statement `LOAD OFFLINE`, the utility statement `LOAD ONLINE` is also available.

The differences between these two utility statements are as follows:

- Their locking behaviors are different, in other words, the way in which they are synchronized with other statements accessing the same spaces, tables and indexes is different
- Their internal execution is different.

This means that the performance behavior of the two statements is also different depending on the prevalent conditions.

The following sections provide hints on those situations where you should use the clause `OFFLINE` and those situations where you should use the parameter `ONLINE` in order to obtain the fastest and most efficient processing of the `LOAD` statement.

You are able to continue an aborted `LOAD` statement conveniently and with high performance. You will find performance-related information concerning this subject in the [section “Continuing an aborted LOAD statement” on page 211](#).

8.1.1 Processing LOAD in SESAM/SQL

This section gives a short description of the differences in processing between LOAD OFFLINE and LOAD ONLINE in SESAM/SQL.

LOAD OFFLINE

Like most of the utility functions, LOAD OFFLINE locks the current table and its indexes exclusively. Parallel access to other tables and indexes of the spaces involved is read-only. Most data accessing takes place in a service task.

To process a table with a primary key, SESAM needs to sort the rows to be loaded internally according to the primary key value. If the load file has already been sorted (i.e. if the parameter SORTED has been specified), the rows to be loaded will be edited in this order. If the load file has not been sorted, SESAM/SQL will create a work file where it stores all the rows to be loaded in the edited (and possibly expanded) form and also creates a SORT buffer where it enters all the primary key values (with a reference to the assigned row in the work file). Once the load file has been fully processed, the SORT buffer will be sorted. The edited rows will be inserted in the table (which in general is not empty) at the matching positions. To do this, the table primary data is read in full and, where necessary, a row is added or modified at the position corresponding to the primary key. At the same time, the access paths are recreated in full.

In a table without a primary key, the rows are edited in the order they occur in the load file and are inserted in the table in front of the existing primary data. Only the access paths for the newly inserted area are updated.

Index maintenance can be suppressed (parameter NO INDEX, default setting); in this case, all table indexes will be marked as “invalid”. If the parameter GENERATE INDEX is specified, all the indexes will be recreated after all the rows to be loaded have been inserted in the primary data of the table. In effect, the statement RECOVER INDEX runs internally; during this time the index spaces are locked exclusively.

The integrity constraint check can be suppressed (parameter NO CONSTRAINT CHECK, default setting). The table will then be placed in the state “check pending”. If the parameter CONSTRAINT CHECK is specified, all the integrity constraints will be checked after all the rows to be loaded have been inserted in the primary data of the table. In effect, the statement CHECK CONSTRAINTS runs internally.

If the user space containing the table is included in the logging, the logging will be interrupted. The space will be placed in the state “copy pending”.

If the statement LOAD OFFLINE is aborted during processing in the service task (for example, by a resource bottleneck), the table will be inconsistent and will be placed in the state “load running”. After this, all other tables and indexes on the same space can only be accessed in read mode. The user space must be repaired so that a RECOVER statement resets it to its state before the LOAD.

LOAD ONLINE

LOAD ONLINE locks the current table and its indexes in shared mode and behaves like a DML update statement. During LOAD ONLINE, the table involved has read and write access. Data accesses take place entirely in a DBH task.

The rows are inserted in the table one at a time in the order in which they occur in the load file. The access paths are updated for each row.

Index maintenance cannot be suppressed. The table indexes are updated for the inserted and modified rows. Invalid indexes are ignored.

The integrity constraint check can be suppressed (parameter NO CONSTRAINT CHECK). The table will then be placed in the state "check pending". If this parameter is not specified, the integrity constraints for the inserted and modified rows will be checked. Only if the table has already been in the state "check pending" all integrity constraints for all the rows will be checked.

Logging is not interrupted.

Even if the statement is aborted during processing (for example, by a resource bottleneck) the table will continue to have a formally correct structure. However, it will be placed in the state "check pending" if the integrity constraints cannot be fully checked.

8.1.2 Performance-related aspects of LOAD

The differences in processing LOAD statements as described in the previous section can influence the performance of a LOAD statement.

LOAD OFFLINE

In a table with a primary key, the performance of LOAD OFFLINE is strongly influenced by the size of the table (after the LOAD) but only slightly influenced by the amount of rows to be loaded.

The basic load is always high even when only a single row is being loaded because the operation entails reading all the existing primary data, recreating the access paths and, where necessary, cases recreating the indexes and checking the integrity constraints.

In a table without a primary key, on the other hand, the performance of LOAD OFFLINE is not so strongly influenced by table size because, in this case, the reading in of existing data and the complete recreation of the access paths are not necessary. However, if indexes or integrity constraints already exist, there is a certain basic load here, too.

There are two other cases, which were not mentioned in the previous section for reasons of clarity, where SESAM/SQL optimizes tables with a primary key during a LOAD OFFLINE:

- Where all the rows of the load file have a primary key value that is higher than the previous highest primary key value of the table.
- Where the first part of a combined primary key is used as an integer field.

In these cases, all the rows to be inserted will be added at the end of the existing primary data. Here also, existing primary data is not read in and the access paths are not recreated from scratch.

LOAD ONLINE

LOAD ONLINE behaves in very much the same way as a DML update statement. Here, performance is mainly influenced by the number of rows to be loaded. The basic load in LOAD ONLINE is virtually negligible.

Processing time per row is longer in LOAD ONLINE than it is in LOAD OFFLINE because the access paths and the indexes have to be maintained row by row and a certain amount of time is also required for synchronizing and logging.

8.1.3 Advice on using LOAD

This section provides advice on how to improve the performance of LOAD. In particular, we provide advice on how to decide when to use LOAD OFFLINE or LOAD ONLINE in order to achieve optimum performance.

LOAD OFFLINE

LOAD OFFLINE performs better than LOAD ONLINE when loading into an empty table.

Above all when working with large data amounts in LOAD OFFLINE you should always sort the load file by primary key value and call the LOAD statement with the parameter SORTED. Apart from speeding up processing, no work file is created which can be substantially larger than the load file.

If several LOAD OFFLINE statements are to be run one after another, the parameters GENERATE INDEX and CONSTRAINT CHECK should only be specified in the last LOAD OFFLINE call. This will save unnecessary, time-consuming actions.

LOAD OFFLINE interrupts the logging of the user space containing the table and also of the related index spaces. It is therefore subsequently necessary to create a SESAM backup or a foreign copy for these spaces using COPY. The copying time and the storage space for the backup files can be saved by using LOAD ONLINE. You should note, however, that LOAD ONLINE requires additional storage space for the logging information.

LOAD ONLINE

LOAD ONLINE performs better than LOAD OFFLINE when loading a relatively small numbers of rows. This is particularly the case for a table with a primary key and many indexes.

A “relatively small number” here should be understood as meaning that the number of rows in the load file is low in relation to the number of existing rows in the table.

Precise figures cannot be given because the number of integrity constraints and, above all, the number of indexes play a crucial role. As a rule of thumb, we can say that LOAD ONLINE is faster when the number of rows in the load file is less than 10% of the number of rows in the table.

LOAD ONLINE does not explicitly utilize sorting of the load file. However, this feature improves buffer performance and can be used to increase efficiency. If the load file, for example, contains a large number of rows whose primary key values lie in the same value range or are even adjacent, then sorting can have considerable advantages.

With LOAD ONLINE the integrity constraints can be checked more efficiently than with LOAD OFFLINE. Because of this, specifying the NO CONSTRAINT CHECK option in LOAD ONLINE in general does not save much time. You should also note that a further check carried out later (using LOAD or CHECK CONSTRAINTS) will entail a repeat check

of the integrity of the entire table. It makes sense to suppress the checks if you can foresee that LOAD ONLINE will place the table in the state “check pending” (e.g. in cases where not all the columns can be loaded at once).

In the event of an error where the statement is aborted, the table will always have a formally correct structure when LOAD ONLINE is used, while with LOAD OFFLINE it can be placed in the state “load running”. In the latter case it will be necessary to repair the table spaces using RECOVER; with LOAD ONLINE, on the other hand, this is not necessary.

Generally speaking, in a table with a primary key an aborted LOAD ONLINE can be repeated using the parameter OVERWRITE. If required, the rows already loaded should be removed from the load file before starting the repeat.

8.1.4 Continuing an aborted LOAD statement

The SKIP FIRST n RECORDS parameter enables you to continue an aborted LOAD statement conveniently and with high performance.

For this purpose, after a LOAD statement has been aborted the number of records which had been read up to the abortion time and already entered in the table (" n RECORDS PROCESSED" entry) is recorded in the error file.

The SKIP FIRST n RECORDS parameter ensures that the first n records of the input file are skipped in the next LOAD statement.

It can then make sense to continue an aborted LOAD statement (in particular LOAD ONLINE) if, for example, the abortion was caused by a resource bottleneck and this bottleneck has now been removed.

LOAD ONLINE

The following options are available for continuing after a LOAD ONLINE statement aborts:

- The LOAD statement is repeated with the additional parameter SKIP FIRST n RECORDS; it is possible to ascertain the value n directly from the error file.
This variant is the most user-friendly and high-performance means of continuing an aborted LOAD ONLINE statement.
- The LOAD statement can be executed again using the available input file if it is guaranteed that a repetition will lead to the same result as the first attempt. A prerequisite is therefore that records which were created with the first LOAD statement are overwritten when the LOAD statement is repeated. The base table must consequently have a primary key, and no count field may have been specified in the first LOAD statement. Furthermore, when multiple fields are used it must be ensured that a continuous instance range beginning with 1 is available, and that only significant values are loaded into the multiple fields.
- Before the LOAD statement is repeated the first n records of the input file which have already been processed are deleted.
The number of records that has already been processed can be derived from the " n RECORDS PROCESSED" entry of the error file. However, the processing of files is time-consuming (especially when the records are over 256 characters long).

An aborted LOAD ONLINE statement can also be continued by a LOAD OFFLINE statement with the SKIP FIRST n RECORDS parameter.

LOAD OFFLINE

In the case of LOAD OFFLINE user data is loaded either in full or not at all.

Here, too, the number of records that has already been entered in the table can be derived from the “*n* RECORDS PROCESSED” entry of the error file.

The “0 RECORDS PROCESSED” entry can be caused by the following:

- The LOAD statement was aborted before the table itself was modified. In this case the LOAD OFFLINE statement can simply be repeated.
- The LOAD statement was aborted after the table had already been modified. In this case the user space is in the “load running” status and must be reset to a status it had before the LOAD statement using a RECOVER statement. The LOAD OFFLINE statement can then be repeated.

If according to the error file records had already been processed ($n > 0$), this means that loading of the primary data records was completed successfully. The abortion thus took place when creating the secondary indexes (GENERATE INDEX parameter) or when checking the integrity constraints (CONSTRAINT CHECK parameter).

Basically the LOAD statement with the SKIP FIRST *n* RECORDS parameter can also be repeated here. However, since in this case the entire input file is (unnecessarily) read again, the performance is better if the RECOVER INDEX ON TABLE or CHECK CONSTRAINTS ON TABLE statement is executed explicitly.

8.2 UNLOAD - Unload user data from a table

The utility statement UNLOAD ONLINE is available in addition to the utility statement UNLOAD OFFLINE.

The differences between these two utility statements are as follows:

- Their locking behaviors are different, in other words, the way in which they are synchronized with other statements accessing the same spaces, tables and indexes is different
- Their internal execution is different.
- Their functional scope is different.

This means that the performance behavior of the two statements is also different depending on the prevalent conditions. The sections below indicate the situations in which you should use the OFFLINE or ONLINE clause.

UNLOAD OFFLINE

Like most of the utility functions, UNLOAD OFFLINE locks the table concerned exclusively. Indexes are not evaluated and are therefore not locked. When other users access other tables and indexes in the space concerned in parallel, they can only do this in read-only mode.

Most data accesses take place in a service task.

Only base tables can be addressed.

UNLOAD ONLINE

UNLOAD ONLINE locks the records and index values concerned in shared mode only and behaves like a DML read statement. Parallel reading and writing is therefore possible in the table concerned.

Data accesses take place entirely in a DBH task.

Both base tables and views can be addressed. The set of hits can be sorted and restricted using a search condition.

Performance-related aspects of UNLOAD

UNLOAD ONLINE behaves similarly to a database query using DML. The difference is that with UNLOAD ONLINE the results are not stored in user variables but in an output file. In particular when the main effort is involved in determining the set of hits, for example in the case of complex WHERE clauses or views, the performance of UNLOAD ONLINE is comparable to that of the corresponding database query.

UNLOAD OFFLINE locks the table exclusively, which means that no effort is required for synchronization. The entire base table is read. It is then always processed sequentially, which permits very efficient algorithms to be used. UNLOAD OFFLINE therefore performs significantly better than UNLOAD ONLINE.

Advice on using UNLOAD

When a complete base table is to be output and restrictive locking behavior is justifiable, UNLOAD OFFLINE should be used.

UNLOAD ONLINE, on the other hand, can be used more flexibly (see WHERE clause, ORDER-BY clause, views) and hinders parallel application at most by imposing read locks on individual records.

The output formats of UNLOAD ONLINE and UNLOAD OFFLINE are identical. If the layout of the output file is to be tested first (in particular in the case of a self-defined format) when very large base tables are output, UNLOAD ONLINE and UNLOAD OFFLINE can be combined effectively. A sensible approach is, for example, first of all to try out various variants with UNLOAD ONLINE (with a small set of hits when an appropriate WHERE clause is used). The more powerful variant UNLOAD OFFLINE can then be used when the complete output file is ultimately generated.

8.3 Creating SESAM backup copies (COPY) and foreign copies

You can create backup copies of your database with the following functions:

- with the utility statement COPY
- with the BS2000 command COPY-FILE
- with the replication functions of the disk storage systems

You can use the utility statement COPY to make backup copies to disk. You can also use COPY to carry out archival functions with the help of ARCHIVE or HSMS. Archiving here is understood as the long-term storage on magnetic tape cartridges or other storage systems (e.g. CentricStor).

You can also choose to make the backup ONLINE or OFFLINE. Use COPY OFFLINE to lock the database being backed up and protect it against write access during the backup process.

With COPY ONLINE, after a synchronizing phase, read access and updating DML statements on the database are permitted.

An ONLINE backup represents the state of the database at the start of the backup. All the updates completed during the backup process are not included in the backup but are only included in the database.

The next sections describe the performance-related aspects of the utility statement COPY ONLINE. These aspects also apply to COPY OFFLINE except the notes on PBI files and HSMS work files.

8.3.1 Processing the COPY statement in SESAM/SQL

In all SESAM backup procedures, the space file names are sorted by file size starting with the file names with the largest spaces. The spaces names are transferred in this order to the component which will carry out the backup.

This procedure ensures that, best case, the backup process will only last as long as the backup of the largest space.

8.3.2 Backup to disk

During a disk backup, up to 10 spaces will be backed up simultaneously using the “wrap around” procedure. Each space is assigned a 960 KB swap buffer. Units of up to 480 KB are read asynchronously from the original by this swap buffer and are written to the backup copy. The size of the read/write unit depends on the disk type. SESAM/SQL uses the maximum possible I/O length for each space that is to be copied. The I/O length is between 32 and 480 KB.

You can obtain information on the maximum I/O length in half pages (a half page is 2 KB in size) using the BS2000 command:

```
/SHOW-PUBSET-CONFIGURATION PUBSET=<catid>,INFORMATION=*PUBSET-FEATURES)
```

Next, the process “waits” for the end of the write operation on each space. When the database files have been distributed to the various disks and the disks have been connected to the host via different channels, the wait situations are minimized. When the backup process for one space has finished, processing of the next space on the list starts immediately. In the best case, ten spaces will always be backed up together at the same time.

If updating DML statements to the database are executed during the backup procedure, the physical before images of the updated database blocks are written to a PBI file. At the end of the backup this PBI file is included in the backup copy.

8.3.3 Backup with ARCHIVE

In order to back up spaces in parallel you will require redundant peripherals. The ideal situation is to have one tape device per space. In this cases the entire backup time is determined by the backup time of the largest space. SESAM packages spaces to be saved if the parameter DRIVE (which specifies the number of parallel runs) in the ARCHIVE parameter file is set to a value greater than 1.

The least ideal situation is one where only one tape device is available. In this case the spaces are then backed up in sequence.

If updating DML statements to the database are executed during the backup procedure, the physical before images of the updated database blocks are written to a PBI file. When backup of the spaces has been completed, this PBI file will also be backed up to tape. The next time the backup is read in, the PBI file will also be read in and will be included in the backup copy.

In this case, the time needed to back up the PBI file must be added to that needed to back up the space.

8.3.4 Backup with HSMS

There are two procedures for making backups with HSMS: backup of spaces in an HSMS archive using a work file; backup of spaces located on a mirror unit of a local or remote disk storage system in an HSMS archive.

Backup of spaces in an HSMS archive using a work file

To make a backup using a work file, SESAM/SQL uses the function “Concurrent Copy” (CCOPY) from HSMS. If updating DML statements to the database are executed during the backup procedure, HSMS writes the physical before images of the updated database blocks in a work file. When backing up database files in the HSMS archive, HSMS reads the blocks to be backed up from the database file or from the work file. In this procedure SESAM does not have to write the PBI file and does not have to back up the PBI file to tape. This means also that there is no need to read-in the PBI file or include the PBI file in the backup copy when the backup is next read in.

Backup of spaces located on a mirror unit in an HSMS archive

To make a backup from a mirror unit in an HSMS archive, SESAM/SQL implicitly uses, via HSMS, the function TimeFinder/Mirror of the Symmetrix systems. First, the mirror unit is split and then the database files from the mirror unit are backed up in the HSMS archive. After this, the mirror unit is synchronized once again with the original disk.

8.3.5 Backup with a foreign copy

You can back up your database without using SESAM/SQL. To do this, you must first use the administration statement (LOCK-SEQUENCE, PREPARE-FOREIGN-COPY) to create a transaction free state on the database and then you must write the modified database blocks to disk. After you have done this, you can create a copy of your database.

If the database is located on a mirror unit, you can split the mirror unit; before you do this you must create a transaction free state and the database blocks must be written to disk. After the split you will be able to access the database again on the original disk. The database on the mirror unit can now be formally checked (CHECK FORMAL) with SESAM functions and backed up as a SESAM backup or a foreign copy. After the backup has been completed, the mirror unit is synchronized once again with the normal disk, see “[Core manual](#)”, section “Creating a foreign copy”.

The disadvantage of creating a foreign copy with the BS2000 command COPY-FILE or with a mirror unit is that no parallel (read or write) DML access is possible while the copy is being made. Here, the foreign copy behaves very much like the utility statement COPY OFFLINE. The foreign copy is advantageous in repair operations (see [section “Shortening read in time” on page 220](#)).

Note on PREPARE-FOREIGN-COPY

The administration statement PREPARE-FOREIGN-COPY ..., CLOSE=*NO closes the database logically but not physically. It is suitable for backup operations which require a logically closed database, e.g. the replication function TimeFinder/Mirror.

The administration statement PREPARE-FOREIGN-COPY ..., CLOSE=*YES can be used for backup operations which require a physically closed database, e.g. the replication function TimeFinder/Snap or the BS2000 command COPY-FILE. The database is then also closed physically.

8.3.6 Criteria for selecting backup procedures

Disk backups require storage space of the same size as the database. The duration of the backup depends on the distribution of the files to the disks and on the peripherals used.

Backups made with ARCHIVE or HSMS only require storage space in the archive being used. The duration of the backup in this case greatly depends on the archive media used and on the level of parallelism specified. With ARCHIVE, in addition to the database files you must also back up the PBI file.

The actual backing up of the database takes place in a service task. This means that the backup places no load on the DBH. With disk and ARCHIVE backups, writing of the PBI file takes place in the DBH.

A SESAM backup in an HSMS archive can also be read in with HSMS functions. This represents a consistent state and can be used as a foreign copy. This is not possible when making backups with ARCHIVE.

In a backup from a mirror unit in an HSMS archive, the mirror unit after splitting will display the state of the database at the time the split took place. A backup in an HSMS archive can be repeated as many times as required if, for example, a backup ends in a defective state. This procedure makes it possible to create a “revision proof” backup, i.e. a backup which always represents the state at the time when the COPY statement was started. In situations where a “revision proof” backup is an absolute necessity, you can use the statement COPY ONLINE.

If you intend to back up a database with the BS2000 command COPY-FILE or with the replication functions of the disk storage systems (i.e. without SESAM functions), you should note the following:

- The user is directly responsible for the creation of a consistent backup.
- No parallel access is possible when the copy is in progress.

Backups must be administered by the user. Backups for recovery must be read in by the user.

Foreign copies can be used for making quick repairs by renaming.

8.4 RECOVER - Repairing and resetting

The utility statement RECOVER executes the following processing steps:

- Read in of the backup of the spaces to be repaired.
- Application of the modifications logged in the CAT-LOG or DA-LOG files to the spaces.
- Rebuild of secondary indexes where necessary.

The time required for repair depends on the following factors:

- Size of the backup files to be read in.
- Number of updates (including insertions and deletions) made since the backup was executed.
- Size and number of the tables where the secondary indexes require rebuilding.

The sections below describe the processing steps and provide instructions on how you can shorten the duration of each processing step. The instructions refer to a repair of the database or of the individual spaces. The information given in the sections below also applies in general to resetting, although not all the steps always need to be executed for resetting.

8.4.1 Shortening read in time

At the start of a repair, the backups are copied onto the spaces to be repaired. This procedure is always used in SESAM backups. For replications, the database administrator can specify if the files are to be copied or renamed (parameter COPY or RENAME). For foreign copies, the database administrator is responsible for the read in, i.e. he/she must copy or rename the files him/herself. You can considerably shorten the read in time by renaming the files.

RECOVER using replications and parameter RENAME

Before the RECOVER statement can be executed, the database administrator must delete the spaces to be repaired.

During processing of the utility statement RECOVER ... USING REPLICATION ... RENAME, the specified replication spaces (or the entire replication catalog) are renamed to the database files. After this, the modifications logged in the logging files are applied to the spaces.

In this case, the read in time required is reduced to that needed to delete the spaces and rename the replication spaces. You should note that if a space has been created via the CREATE SPACE statement using the parameter DESTROY, it will be deleted by overwriting with binary zeros. We recommend that you create the spaces with the parameter NO DESTROY.

Given that the files have been renamed, the repaired spaces are now located on the replication media.

After this statement has been executed, the specified replication spaces are no longer present. The replication must be recreated or the replication must be extended with the spaces.

RECOVER using foreign copies on disk

The procedure for a foreign copy is similar to that for a replication. The database administrator deletes the spaces to be repaired and renames the corresponding spaces of the foreign copy to the database spaces. After this, the utility statement RECOVER ... USING FOREIGN COPY uses the read in backup as the basis for the repair.

Given that the files have been renamed, the repaired spaces are now located on the foreign copy media.

After this statement has been executed, the specified foreign copy spaces are no longer present. This foreign copy cannot be recreated.

We therefore recommend that if you use this procedure you should create a foreign copy of the database at regular intervals and then back this up to a long-term archive. The most recently created foreign copy remains on disk until the next backup is made.

RECOVER using foreign copies from a mirror unit

If you employ disk storage systems with replication functions, you can use the replication functions of these disk storage systems to create foreign copies under SESAM/SQL. The replication functions are operated via the host component SHC-OSD, see the „[Storage Management for BS2000](#)“ manual.

The example below describes a procedure using two mirror units. We will only describe the special features of this procedure. For information about the serialization and preparation necessary, see the section “Using foreign copies of a database” in the “[Core manual](#)“.

Using two mirror units you can build the following configuration. A prerequisite in this case is that the database files must be located on the original unit. The logging files (CAT-LOG and DA-LOG files) and the CAT-REC file must not be mirrored with this mirror unit.

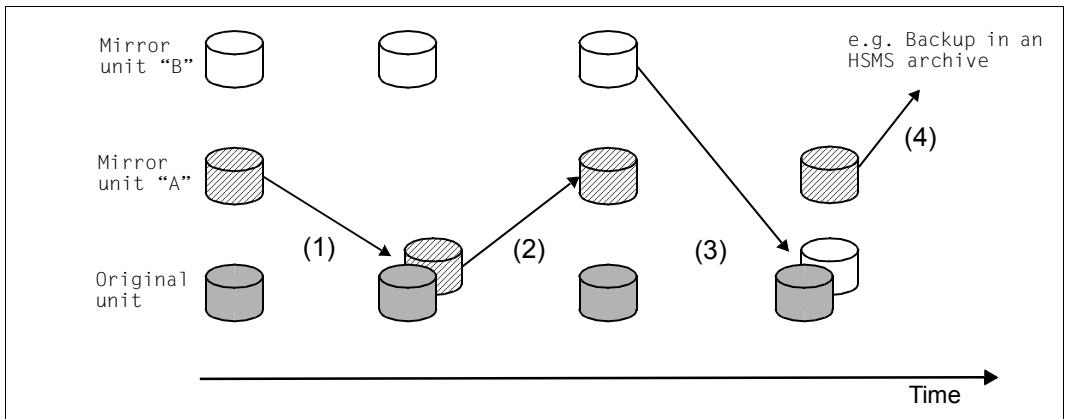


Figure 2: Process up to TimeFinder foreign copy

- (1) The original unit containing the database is paired with the mirror unit “A”. The updates to the database from this point onwards will be mirrored on mirror unit “A”.
- (2) At a later time, in order to make the backup, the mirror unit “A” is split.
- (3) The original unit is paired with the mirror unit “B” for further operation. The updates to the database from this point onwards will be mirrored on mirror unit “B”.
- (4) Next, the split mirror unit “A” is imported as a separate subset. The database can now be processed with SESAM functions. It can, for example, be checked for formal consistency with CHECK FORMAL and then backed up with HSMS in a long-term archive. The database remains available on mirror unit “A” ready for use as a rapid repair.

It is now possible to carry out a repair using the last backup created on the mirror unit "A"; proceed as follows:

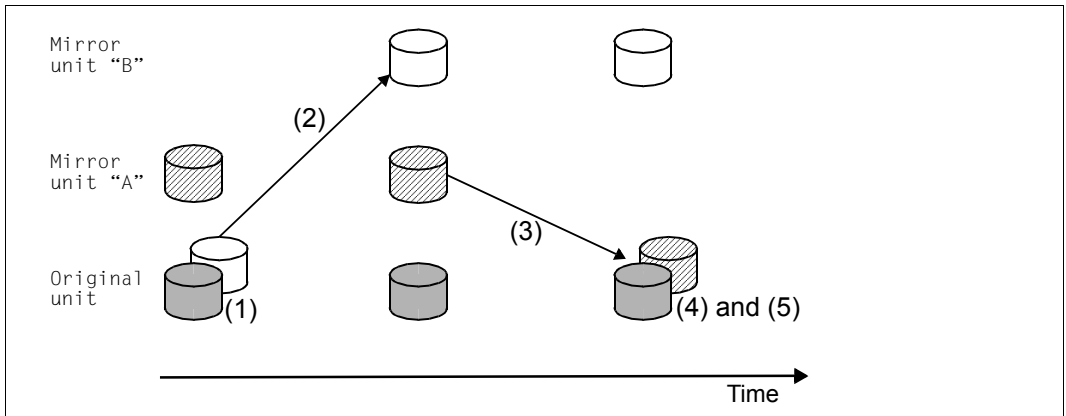


Figure 3: Rapid Recovery process

- (1) Close the database via the administration statement
`SET-SQL-DB-CATALOG-STATUS STATUS=*FREE,SELECT=...`
- (2) Split the active mirror unit "B"
- (3) Pair the normal unit with the mirror unit "A" (with the backup)
 This command copies the updated data on mirror unit "A" onto the original unit. From this point onwards, the database updates will be mirrored, once again, on mirror unit "A".
- (4) Open the database via the administration statement
`SET-SQL-DB-CATALOG-STATUS STATUS=*ACTIVE,SELECT=...`
- (5) Repair the database and apply the modifications logged in the logging files to the database via the utility statement
`RECOVER ... USING FOREIGN COPY`

RECOVER using the parameter SCOPE PENDING

The duration of a repair can be shortened by limiting the repair to those spaces which SESAM/SQL identifies as defective. A space is identified as defective when at least one of the following conditions is present:

- During a DBH session a consistency check with the error weight 37 (defective space) or 38 (defective database catalog space) occurs.
- During a DBH session a consistency check concerning a base table on the current space with the error weight 36 (defective table) occurs.
- The utility function CHECK FORMAL detects an error (CHECK FORMAL without parameter NO ACTION).
- The space is set to defective (space state “load running” or “recover pending”) by an aborted utility statement (LOAD OFFLINE or RECOVER).
- The space does not match the current state of the metadata in the catalog space.
- A DMS error occurs when a space is opened.

Note

A space with one or more defective indexes is not considered to be defective in this case.

The utility statement RECOVER CATALOG consists of the following processing steps:

1. Read in of the catalog space backup.
2. Application of the modifications logged in the logging files to the catalog space.
3. Determination and read in of the space backups.
4. Application of the modifications logged in the logging files to the spaces.
5. Rebuild of indexes where necessary.

If the parameter SCOPE PENDING is specified in RECOVER CATALOG, the processing steps 1 and 2 will always be executed. Processing step 3 also checks which space is defective. Steps 3 and 4 will only be executed for those spaces identified as defective. After this, processing step 5 will be executed where necessary.

Processing steps 1 and 2 will not be executed in RECOVER SPACE and RECOVER SPACESET with or without the parameter SCOPE PENDING. Processing steps 3 and 4 (and where necessary, step 5) will be executed as described above.

8.4.2 Speeding up the application of the modifications logged in the log files

All modifications to the tables and indexes of a space will be logged in DA-LOG files as long as the logical data saving for that space is enabled. A DA-LOG file is divided into DA-LOG units. These DA-LOG units are administered in the DA_LOGS table.

The DA-LOG unit is changed in the following circumstances:

- When a DA-LOG file is full, a change to the next DA-LOG file will take place implicitly.
- When the administration statement CHANGE-DALOG is used to change the DA-LOG file explicitly.
- When the DA-LOG unit is changed implicitly in the statements CREATE SPACE and COPY SPACE.
- When the DA-LOG file is changed implicitly in the statements COPY CATALOG, RECOVER CATALOG and RECOVER SPACE.
- When the first change in a space implicitly changes the DA-LOG unit.
- When the DA-LOG unit is implicitly changed in the restart of a DBH.

For each DA-LOG unit, the DA_LOGS table indicates the spaces for which changes have been logged in this DA-LOG unit.

When the modifications logged in the logging files are to be applied to a space, the DA-LOG units required for this are determined from the DA_LOGS table. Only the necessary DA-LOG units will be read when the modifications logged in the logging files are applied.

The time needed to apply the modifications logged in the logging files is mainly influenced by the number of rows for which modifications have been logged. The number of columns to be changed in a row plays a minor role here.

Parallel application of modifications logged in the logging files to individual spaces

Several RECOVER SPACE statements on different spaces of the same database can be executed by different programs at the same time. These RECOVER SPACE statements are processed in parallel in the DBH. In this case, the DA-LOG files must be located on disk because parallel accessing of tape files is not possible.

Parallel processing can be advantageous when the modifications to be applied to the spaces are logged in very disjointed DA-LOG units.

A further advantage is that a service task with all the resources for this task is available for each space to which modifications are to be applied. The SYSTEM-DATA and USER-DATA buffers are now available once for each space to which modifications are to be applied. This considerably reduces the displacement of datablocks in the buffer.

Parallel RECOVER SPACE statements should only be used when the space backups are located on disks. For backups located on magnetic tape cartridges you can use the corresponding parameters in the HSMS or ARCHIVE parameter files which control the parallel read in of files on different tape devices.

If partitions of a partitioned table lie on the spaces, these spaces should be repaired in **one** RECOVER Space Set / Space List statement so as to avoid possible deadlock situations.

Applying modifications using the most current replication

The functions CREATE REPLICATION and REFRESH REPLICATION in SESAM/SQL can be used to create a shadow database. This shadow database can be relatively current. Whenever a changeover is made to a new logging file in the original session, the modifications logged in the previous logging file can be applied to the replication by using REFRESH REPLICATION. In a repair using the statement RECOVER ... USING REPLICATION RENAME only the modifications logged in the current logging file must be applied.

8.4.3 Repairing index spaces

Some database administrators organize their databases so that some spaces only contain indexes but no tables. These spaces are known as index spaces and can be repaired very quickly. Index spaces can also be removed from the logical data saving because the indexes can be rebuilt at any time from the data of the corresponding table.

In tables with very few rows where a large number of changes have been made we advise the following procedure:

- Remove the index space from the logical data backup. To recreate the indexes, specify the parameter `GENERATE INDEX ON NO LOG INDEX SPACE` in the utility statement `RECOVER CATALOG`. To recreate the indexes of a table, the data for the entire table will be read and sorted and the indexes will then be recreated.
- If you want to repair an individual index space, it does not matter if the index space is included in the logical data saving or not. Just use the utility statement `RECOVER SPACE index_space TO COPY_FILE '*DUMMY'`. The index space and the indexes will be recreated.

If the table contains a relatively high number of rows, it is better to include the index space in the logical data saving and to then apply the modifications logged in the logging files. You can do this via the statement `RECOVER CATALOG` without specifying the parameter `GENERATE INDEX ON NO LOG INDEX SPACE`. You can repair index spaces individually with the utility statement `RECOVER SPACE index_space`.

8.5 REORG - Reorganizing spaces and base tables

The utility statement REORG ONLINE TABLE is available in addition to the utility statement REORG SPACE.

The differences between these two utility statements are as follows:

- Their internal execution is different.
- Their lock behavior is different.

This means that the performance behavior of the two statements is also different depending on the prevalent conditions.

In the case of REORG SPACE and REORG ONLINE TABLE, SESAM/SQL also takes into account free space reservation which was specified beforehand in the CREATE SPACE or ALTER SPACE statement with the PCTFREE clause.

REORG SPACE

SESAM/SQL reorganizes a user space in two phases using a work file on disk:

1. The work file is created using the reorganized blocks of the space. Each base table and each index of the space is reorganized in the work file. Logically contiguous blocks of a table or of an index are then also physically contiguous. In this phase users can access the tables and indexes of the original space in read mode using DML statements.
2. The work file is renamed (RENAME) or copied (COPY). In this phase the space involved cannot be accessed. The user can keep the lock time short by using RENAME instead of COPY.

If the REORG SPACE statement is aborted in the first phase, the space will be in a consistent status. The statement can be repeated.

REORG ONLINE TABLE

SESAM/SQL reorganizes a base table by modifying and copying the blocks within the user space during ongoing operation (“online”). The base table is reorganized piece by piece. Free blocks which are contiguous in a segment are used for this purpose. The blocks which are released as a result of the reorganization are reused. If not enough contiguous free blocks are available when reorganization of an extremely fragmented table begins, the space is extended.

As no exclusive transaction locks are requested for reorganization, other DML applications can access the base table in read and write mode. Only the base table is reorganized, not the indexes which belong to it.

When a REORG ONLINE TABLE statement is aborted, the space will be in a consistent status. The statement can be repeated.

Advice on using REORG

In a user space containing a single base table the runtime of REORG SPACE is shorter than that of REORG ONLINE TABLE. However, during a REORG SPACE no modifying DML statements are permitted which are possible during a REORG ONLINE TABLE. The runtime of REORG ONLINE TABLE is longer than that of REORG SPACE.

8.6 Optimized index creation for partitioned tables

The CREATE INDEX statement is processed in three phases:

1. Reading the values from the primary file.
2. Sorting the values.
3. Creating the secondary indexes.

In the case of partitioned tables the first phase can be parallelized if the indexes to be created are not to reside on one of the spaces which contain the partitions of the table.

Parallel reading is performed in the first phase provided sufficient service tasks are available. The data read is buffered in a memory pool. Parallel to this another service task outputs the data from the memory pool to the BS2000 sort with SORT. The sort starts when all data has been read and transferred. When the sort has ended, the secondary indexes are constructed from the sorted data. This process cannot be parallelized. It is performed sequentially by a service task.

To achieve optimum performance it is necessary to take into account dependencies on the DBH options. Maximum parallelism is achieved when the number of service tasks available is at least one higher than the number of partitions. However, a larger number of service tasks should be selected for further parallel requests. The value of the DBH option THREADS must be at least one higher than the number of partitions. The value of the INITIAL parameter of the DBH option SERVICE-TASKS must be sufficiently large.

If not enough service tasks are available, multiple work steps are processed by a service task. The possible parallelism is reduced.

The optimized CREATE INDEX is not available in the linked-in DBH.

8.7 DROP TABLE DEFERRED / DROP INDEX DEFERRED

The physical structure of a table or an index has two areas:

- The contiguous part
- The relocations

When a table is created using the SQL statement CREATE TABLE, the records which are to be inserted are initially stored in the contiguous part. If this part is no longer large enough, a relocation is created; a free block is requested from the free space administration. Logically this block belongs to the table, but physically it is no longer located in the contiguous part.

When a user space is reorganized using the REORG SPACE statement, the table is then created completely in a contiguous area.

When a table is deleted using the SQL statement DROP TABLE, the contiguous part and the relocations must be deleted. The contiguous part is deleted very quickly because which blocks are to be deleted is known here. To delete the relocations all these blocks must be determined and deleted individually. The time required for release depends on the size of the table. In the case of very large tables whose user space has not be reorganized for a long time, deletion can take several hours.

The behavior described applies analogously for indexes.

The DEFERRED parameter is available for the DROP TABLE and DROP INDEX statements.

When the DEFERRED parameter is specified, only the contiguous part of a table or index is deleted. The relocations are retained. Consequently the DROP TABLE DEFERRED and DROP INDEX DEFERRED statements run significantly quicker than the statements without the DEFERRED parameter in the case of large tables which are rarely reorganized. The next time the user space is reorganized using REORG SPACE all the existing tables and indexes in the user space are reorganized. The “undeleted” relocations then disappear. Only then can the blocks which have become free be reused.

8.8 ALTER TABLE ... ADD COLUMN with index definition

The SQL statement ALTER TABLE ... ADD COLUMN has an ADD INDEX clause (optional). This enables columns to be added with an SQL statement and the associated indexes to be defined.

The indexes are constructed very quickly if no DEFAULT clause with a value unequal to NULL is specified for any of the new columns. In this case the associated columns contain no data and the table need not be read to construct indexes.

However, if the indexes are defined in a separate CREATE INDEX statement, the entire table must be read for this purpose. How long reading takes depends on the number of rows in the table. This time can be saved by using ALTER TABLE ... ADD COLUMN ... ADD INDEX.

In the ADD INDEX clause only indexes for columns can be defined which are also specified in the ADD COLUMN list.

This enhancement provides overall savings on when writing and on runtime.

8.9 Media definition for DDL-TA-LOG files

When DDL and SSL statements are executed, the physical before images of updated blocks are logged in the DDL-TA-LOG file (DDLTA file). This information is needed to reset the DDL and SSL statements. The DDLTA file is space-specific with the name:

`<catalog_name>.<space_name>.DDLTA`

A media definition can be specified for this file. It makes sense to adapt the media definition when DDL statements are to be executed which log a large amount of data in the DDLTA file, for example

- ALTER TABLE ADD COLUMN with DEFAULT clause
- ALTER TABLE ALTER COLUMN when the column length is shortened
- ALTER TABLE ... DROP COLUMN

With these statements the volume of data to be logged depends on the number of records in the table.

With the other DDL and SSL statements the volume of data to be logged is low.

9 Performance-related aspects of administration statements

The runtime of some administration commands can be reduced by favorable settings for certain DBH options or by particular measures.

9.1 RECONFIGURE-DBH-SESSION and RELOAD-DBH-SESSION

A large number of changed blocks for which writing to the spaces of the databases is delayed possibly prolongs the runtime of the actions which are required for these administration commands.

This number can be minimized by selecting low values for the BUFFER-LIMIT and TALOG-LIMIT parameters of the DBH option RESTART-CONTROL.

The administration statement MODIFY-RESTART-CONTROL enables these values to be modified during the DBH session, in other words before RECONFIGURE-DBH-SESSION or RELOAD-DBH-SESSION is executed.

For a description of the DBH option RESTART-CONTROL and the administration statement MODIFY-RESTART-CONTROL, please refer to the „[Database Operation](#)“ manual.

The runtime of the administration commands is also reduced if commands which force the execution of the delayed write requests (e.g. PREPARE-FOREIGN-COPY) were not issued long ago.

A large number of open transactions extends the time before the administration commands are executed,

9.2 SET-SQL-DB-CATALOG-STATUS (STATUS=FREE)

Rows which are deleted in a transaction are initially only deleted logically. The rows are physically deleted asynchronously at a later time, at the latest when the space is closed.

When a large number of rows is deleted not in one transaction but in multiple transactions, some of the rows are also deleted physically in this phase.

Each change on the space triggers physical write requests which are executed asynchronously. When a space is closed, all write requests which are still open must be executed.

The number of open write requests can be minimized by selecting low values for the BUFFER-LIMIT and TALOG-LIMIT parameters of the DBH option RESTART-CONTROL. The administration statement MODIFY-RESTART-CONTROL enables these values to be modified during the DBH session, in other words before RECONFIGURE-DBH-SESSION or RELOAD-DBH-SESSION is executed.

For a description of the DBH option RESTART-CONTROL and the administration statement MODIFY-RESTART-CONTROL, please refer to the „[Database Operation](#)“ manual.

The runtime of the administration commands is also reduced if commands which force the execution of the delayed write requests (e.g. PREPARE-FOREIGN-COPY) were not issued long ago.

A large number of open transactions extends the time before the administration commands are executed,

9.3 STOP-DBH

In the context of terminating the DBH, initially open transactions are reset. A large number of open transactions or also open long-running transactions extend the execution time for this administration command.

As all databases are also closed properly, the information in the [section “SET-SQL-DB-CATALOG-STATUS \(STATUS=FREE\)”](#) above also applies.

Related publications

You will find the manuals on the internet at <http://manuals.ts.fujitsu.com>. You can order printed versions of manuals which are displayed with the order number.

SESAM/SQL-Server (BS2000)
SQL Reference Manual Part 1: SQL Statements
User Guide

SESAM/SQL-Server (BS2000)
SQL Reference Manual Part 2: Utilities
User Guide

SESAM/SQL-Server (BS2000)
CALL-DM Applications
User Guide

SESAM/SQL-Server (BS2000)
Core manual
User Guide

SESAM/SQL-Server (BS2000)
Database Operation
User Guide

SESAM/SQL-Server (BS2000)
Utility Monitor
User Guide

SESAM/SQL-Server (BS2000)
Messages
User Guide

ESQL-COBOL (BS2000)
ESQL-COBOL for SESAM/SQL-Server
User Guide

SESAM-DBAccess

Server-Installation, Administration (available on the manual server only)

openUTM

Concepts and Functions

User Guide

openUTM (BS2000, UNIX Systems, Windows NT)

Programming Applications with KDCS for COBOL, C and C++

Core manual

openUTM (BS2000)

Generating and Handling Applications

User Guide

BS2000 OSD/BC

Introduction to System Administration

User Guide

BS2000 OSD/BC

Performance Handbook

User Guide

SHC-OSD (BS2000)

Storage Management for BS2000

User Guide

openSM2 (BS2000)

Software Monitor

User Guide

SORT (BS2000)

SDF Format

User Guide

SPACEOPT (BS2000)

Disk Optimization and Reorganization

User Guide

Index

In the index, **bold** page numbers refer to the main sources of the index entries, while *italicized* page numbers refer to examples. The collation sequence is as follows: symbols come before digits which come before letters. A punctuation mark is a symbol.

- 197
* 197
/ 197
(column reference with the same name) 195
+ 197
< 201
<= 201
<> 201
= 201
> 201
>= 201
|| 197

A

access
 base table 36
access data 73
access path selection 30
 time 37
access plan, see plan
ADD INDEX 231
ADDRESS-SPACE-LIMIT 102
administration block 73
administration data 73
after image 74
aggregate function 126, 127, 136, 162
algebraic optimization 25
ALL 192
allocation, indexes 66
ALTER TABLE 231
analysis tool 14
AND 202
annotation 42
ANY 192

application
 relevance to performance 23
 SQL 114
application task 115
APPLICATIONS mask (SESMON) 16
applications, information 16
assignment
 plan to SQL statement 82
asynchronous 109
attribute graph 132
automatic
 optimization 31
 simplification deactivation 27
AVG 126

B

backup 67
backup concept 66
backup of messages 109
backup operations
 selection criteria 219
backup unit 67
base table 124
 access 36
batch application 90
batch program 91
before image 74
Begin of Transaction 128
BETWEEN 201
 elimination 25
between 201
block 73, 74, 92
 memory area 92, 93
Block checkinfo 59

- block header 59
- block mode 54, 70, 92, 115, 116, 128
- block size 59
- block utilization 76
- blocking factor 116
- bottleneck analysis 14
- bottleneck prevention
 - buffer 73
 - data type 52, 53
 - prepared statement 53
 - SELECT list 52
 - SESDCN 109
 - statement type 52
 - VGM 53
- bracket
 - Explain component 136
- BS2000 sorting 99
- buffer
 - setting 66
- buffer bottleneck 73
- buffer size 73, 76, 116
- C**
- calculating
 - number of blocks in user data buffer 74
 - number of plans 91
 - number of user data blocks 76
 - plan buffer size 86
 - secondary data blocks 76
- CALL DML
 - join processing 130
- CALL statement 188
- CAPACITY mask (SESMON) 16, 108, 109
- Cartesian product 122
- CASE 198
- case 198
- cast 197
- cast_rel_to_value 194
- cast_rows_to_rel 140
- castable_pred 202
- CAT-LOG file 16, 66
 - allocation 66
- CAT-REC file 16, 66
 - allocation 66
- catalog contents, optimization 18
- change of thread 14
- CHAR_LENGTH 197
- char_length 197
- check constraint 64
- CHECK CONSTRAINTS
 - plan 50
- CHECK FORMAL
 - lock 50
- check_after_all 124, 172, 174, 177, 180
- check_on_the_fly 124, 172, 174, 177, 178, 180
- CLOSE 82
- CNF 25
- CO-LOG file 20
- coalesce 197
- collecting
 - search conditions 26
- column 115
- column references
 - with the same name 195
- COLUMNS, see DBH option
- combining
 - search range 34
- comment
 - avoiding 52
- COMMIT WORK 55, 82
- common
 - subexpressions 195, 195
 - subpredicates 195, 195
- communication buffer 52
 - space requirements 52
 - statement type 52
 - upper limit 52
- communication overhead 55, 100
- communication path length
 - economic use of 128
- comparison 127
- complex, predicate 41
- complexity
 - processing step 20
- concat 197
- concatenation 128
- configuration 91, 109
- configuration file 116

conjunctive normal form 25, 41
consistency level 46
constant expression
 simplification 26
containers 108
contingency task 109
continuing, LOAD 211
contradiction, elimination 26, 26
conversation memory, see VGM
conversation-specific
 memory area 53
 space requirements 53
COPY 215
 ARCHIVE 216
 disk backup 216
 foreign copy 218
 HSMS 217
COPY OFFLINE 215
 lock 50
COPY ONLINE 215
 lock 50
CORE (SORT parameter) 102
correlated subquery 31
correlation name 52
cost
 processing step 19
 proportion 14
 subquery 31
cost_ratio 119, 120, 121
COUNT 126
count field
 automatic 67
COUNT(*) 126
CPU time 55
CPU timeout 97
CREATE INDEX 229, 231
CREATE SCHEMA statement
 splitting 52
cross 122, 142, 142
cross product 142
current
 lock situation 14
 operational data 15
current row, cursor 143

Cursor 82
cursor 55
 block mode 54, 92
 declaration 181
 dynamic 82, 84
 number 15
 scrollable 203
 static 82, 84
 VGM utilization 53
cursor buffer 78
cursor file
 accesses 16
 internal 78
cursor plan 82
cursor statement 82
cursor_scan 143, 143
cursor_select_stmt 181

D

DA-LOG file 16
 allocation 66
data allocation 65
data distribution 65
data type 74
 bottleneck prevention 52, 53
database
 closing logically 218
 closing physically 218
database contents, optimization 18
database design
 aspects relevant to performance 57
database key 78
database operation 69
DBH option 14
 COLUMNS 82, 86, 91
 RESTART-CONTROL 77
 SERVICE-TASKS 97, 99
 SQL-SUPPORT 70, 82, 84, 88, 89
 SYSTEM-DATA-BUFFER 73, 76
 THREADS 79
 TRANSFER-CONTAINER 72
 USER-DATA-BUFFER 73, 76
 USERS 82, 87
 WORK-CONTAINER 72

- DBH parameter
 - INITIAL [97](#), [102](#)
 - JOBCLASS [97](#)
 - MAXIMUM [97](#)
 - PLANS [82](#), [84](#), [86](#), [88](#), [89](#)
 - RECORDS-PER-CYCLE [99](#)
 - WORK-FILES [99](#)
- DCN operation
 - information [16](#)
- DDL [82](#)
 - parallel [71](#)
 - plan creation [24](#)
 - synchronization against DML [48](#)
- DDL-TA-LOG file [231](#)
- deadlock
 - DDL [49](#)
 - DML [49](#)
 - utility [50](#)
- decision support application [89](#)
- declaration
 - cursor [181](#), [181](#)
- DECLARE statement [181](#)
- DELETE statement [82](#), [124](#), [180](#), [180](#)
 - WHERE CURRENT OF [115](#)
- delete_stmt [124](#), [180](#)
- deleting
 - plans [88](#)
 - rows [180](#)
- deleting, plan [85](#)
- derived table
 - Explain component [140](#)
- descriptor [114](#)
- diagnosis [70](#)
 - lock conflict [110](#)
- dimensioning
 - plan buffer [88](#)
 - prefetch buffer [94](#), [95](#)
 - SESDCN memory pool [108](#)
 - system data buffer [75](#)
 - user and system data buffers [76](#)
 - user data buffer [74](#)
- dirty read [45](#)
- disk device [71](#)
- disk distribution, openSM2 [14](#)
- disk storage requirements [61](#)
 - BLOBs [60](#)
 - metadata [61](#)
 - primary data [59](#)
- displacement mechanism [73](#)
- DISTINCT
 - elimination [29](#)
- distributed
 - environment [115](#)
 - processing [108](#)
- distribution, index values [37](#)
- divide [197](#)
- DML [82](#)
- DML statements [172](#), [188](#)
- DROP INDEX
 - DEFERRED [230](#)
 - pragma [120](#)
- DROP TABLE
 - DEFERRED [230](#)
- dynamic [82](#), [84](#), [85](#), [88](#)
- dynamic SELECT statement [183](#)

E

editing format
 IO-STATISTICS 20
 STEP-COMPLEXITY 20
 STEP-IO-STATISTICS 20
 STRING-FORMAT 20
 efficient I/O 65
 elapsed time 109
 element query 127
 elimination 25
 contradiction 26, 26
 costly evaluation steps 28
 DISTINCT 29
 join 26
 predicate 26
 sort operation 35
 tautology 26
 Empty 144, 144
 empty
 row 157
 table 144, 144
 eproj 145, 145
 eproj node 145
 eq 201
 equi-join 33, 148, 153
 join order 32
 equi-join predicate 125
 error file, LOAD 211
 estimating, number of plans 89
 evaluated in advance, predicate 29
 evaluation
 for tuning measures 15
 example
 calculating buffer sizes 76
 calculating minimum number of plans 89
 Explain component 133
 optimizing the prefetch buffer 95
 processing strategy 134
 exclusive lock 47
 CALL DML 47
 utility 50
 EXECUTE 55
 EXECUTE IMMEDIATE 82, 85
 EXISTS predicate 125

EXPLAIN 17, 118, 133
 Explain 131
 Explain component 17, 118, 131
 bracket 136
 changes 131
 example 133
 EXPORT DATA statement 184
 expression 195
 node format 196, 197, 197, 198
 precalculated 28
 extents 99
 external format 129
 external reference 31

F

FETCH statement 82, 115, 116
 file
 logical 15
 TA-LOG 65
 TA-LOG file 103
 WA-LOG file 103
 file Close 129
 follow-up statement 128
 FOR UPDATE clause 70
 FOR UPDATE cursor 47
 forcing
 simplification 27, 145
 sort-merge join 33
 foreign copy 215
 free pool elements 108
 FROM clause 123
 front masking 71, 129
 full outer join 146, 148
 full_outer_join 146, 146
 full_outer_mjoin 148, 148

G

ge 201
 grouping column 162
 grouping operation 136, 162
 groups 162
 gt 201

H

heuristic techniques 40
hit rate 38, 73, 75, 78

I

I/O
 SESMON mask 71, 78
 WA-LOG file 74, 75
 waiting period 71
I/O accesses 16
I/O behavior
 processing step 20
 statement 20
I/O mask (SESMON) 16
identical statement texts 83
identity 164
IGNORE INDEX
 pragma 37, 119, 120, 121
IGNORE/USE INDEX
 effect 39
 pragma 18
IGNORE/USE SORT_INDEX
 effect 39
 pragma 18
ignoring, index 39
IN
 elimination 25
Index
 low selectivity 120
index 137
 access path selection 30
 allocation 66
 ALTER TABLE 231
 compound 119
 creating 119
 evaluating search condition 127
 ignoring 39
 join algorithm 33
 sort minimization 36
 statistics 18
 unnecessary 71
 update 66
index block 73
index evaluation 130
index processing 37
index scan, sort 40
index use 28
 prerequisites 36
 sort 36
indexed scan, sort 36
indexes 71, 101, 122, 129
individual steps, SQL-DML statement 17
influencing
 simplification 27
influencing, optimization overhead 32
INFORMATION_SCHEMA 71
inheriting, sort order 35
INITIAL 97, 102
input/output behavior, statement 20
INSERT 67
INSERT statement 174, 175, 176
insert_stmt 124, 176
inserting
 rows 174, 177
insignificant condition 129
integrity constraint 62, 124
 relevance to performance 62
 testing 172
intermediate file 181, 203
internal
 access plan 17
 cursor file 78
 format 129
internal EXPORT DATA statement 184
inverted length 101
IO-STATISTICS 20
IS CASTABLE 202
IS LIKE 201
IS NOT LIKE 201
IS NOT NULL 201
IS NULL 201
is_like 201
is_not_like 201
is_not_null 201
is_null 201
isolation level 45
 lock 47

J

job class 97
JOBCLASS
 DBH parameter 97
join 124
 elimination 26
 full outer 146
 join orders 32
 left outer 150, 151
 optimization 32, 32
 simplification 27
join algorithm 33, 42
 selection 33, 33
join components 130
join condition 121, 123
JOIN entry 97, 100, 102
JOIN method 39
join operand 122
join operands, switching 40
Join optimization 42
join optimization 40
 OPTIMIZATION LEVEL 40
join order 33, 40, 123
 OPTIMIZATION LEVEL 40
join processing, CALL DML 130

K

KDCDEF parameter VGMSIZE 54
KEEP JOIN ORDER
 pragma 18
KEEP KOIN ORDER 123
key values 90

L

large files 57
large space 57
latitude
 join order 32
le 201
left outer join 150, 151
left_outer_join 150, 150
left_outer_mjoin 126, 151, 151
length
 data elements 59

life-span
 plans 84
LIKE_REGEX 202
LOAD 205
 continuing 211
 lock 50
LOAD OFFLINE 205
 advice on using 209
 continuing 212
 performance aspects 208
 processing 206
LOAD ONLINE 50, 67, 205
 advice on using 209
 continuing 211
 performance aspects 208
 processing 207
load running (space state) 212
load, DBH 14, 15, 20
lock 47
lock conflict 71, 110
lock granularity 66
lock object 46
 hierarchy 46
lock situation, current 14
locking mechanism 46
logging
 statistical data 15
logical file 128
 number 15
logical logging
 deactivating 66
LOWER 197
lower 197
lt 201

M

mask, see SESMON mask 16
master DCN 108
MAX 126
Max. Occ. (SESMON) 94, 95, 96
MAXIMUM 97, 102
measure 70, 102
measurement tool 14
MEDIA DESCRIPTION
 DDLTA file 231
 synchronization 49
memory area
 conversation-specific 53
memory requirements
 block mode 93, 94, 115
memory utilization 94
merge join 130, 139
MERGE statement 177, 178
merge_stmt 178
merging, indexes 38
message buffer 116
message length 70, 116
message volume 16
metadata 61
MIGRATE
 lock 50
MIN 126
minimization, sort operation 35
minus 197
mirror unit 222
mjoin 139, 153, 153
MODIFY
 synchronization 49
MODIFY-RESTART-CONTROL 233, 234
MODIFY-RECOVER-OPTIONS 103
MODIFY-SERVICE-TASKS 97, 99
MODIFY-STORAGE-SIZE 70, 72
multi-task sort 99
multiple attribute 128
multiple column
 optimization 34
multiprocessor 99

N

n Open statement 128
ne 201
negation 197
nested request 31
nested-loop join 32, 33, 33, 120, 121, 130, 136, 155
 OPTIMIZATION LEVEL 40
nesting 28, 28
nesting level 130
nljoin 121, 136, 155, 155
NO-CPU-LIMIT 102
node 132
node format 196
non-repeatable read 45
normalization 25, 25
 complex predicate 41
 elimination 25
 predicate 25
 view substitution 25
normalized representation 25
NOT
 elimination 25
NOT BETWEEN 201
NOT EXISTS predicate 125
NOT NULL constraint 64
not_between 201
notational conventions 11
NULL constraint
 index use 37
null set 144
nullif 197
number
 columns in SELECT list 115
 containers 108
 plan alternatives 40
 plans 84
 service tasks 97
 sort criteria 98
 suborders 81
 threads 79
number for column reference 195
number of rows found, estimated 18

O

object-locking mechanism 46
occurrence 128
OLTP application 89
once-only evaluation, expression 28
One 157
one-relation request
 optimization 34
OPEN 82
open phase 140
openSM2
 BS2000 monitor 14, 71
optimization 24
 automatic 31
 CALL DML 128
 join 32, 32
 multiple column 34
 one-relation request 34
 query block 32
 sort operation 35
 SQL 114
 subquery 31
 without pragma 32
OPTIMIZATION LEVEL 31
 effect 40
 join order 32, 123
 pragma 18, 119
 sort minimization 35
optimization overhead
 influencing 32
optimization value, join processing 130
optimizer 24, 119
 annotation 42
OPTIONS mask (SESMON) 15
OR 202
order backlog 97
ORDER BY clause 98, 120
 sort 35
order of processing 130
order type T1 97
Orders-Not-Processed 97
outer join 125
 full 146
outer reference 136

outer values, sort 37
output descriptor 114
output to the screen
 SESMON 15
overall cost 14
overload, openSM2 14
OVERVIEW mask (SESMON) 16

P

parameter file, SESMON 16
partition 67
partitioned table 67
path length
 economic use of 128
pattern comparison 127
performance 23
 administration statements 233
 analysis 118
 COPY 215
 CREATE INDEX 229
 database operation 69
 foreign copy 215
 LOAD 205
 problems 70, 118, 119, 124, 125
 programming recommendations 114
 RECOVER 220
 REORG 228
 UNLOAD 213
performance analysis 13
performance monitor 15, 72, 78, 79, 81, 87, 97,
 108, 109, 110
performance service tasks 102
phantom 45
phase
 access path selection 30
 algebraic optimization 25
 front-end 25
 plan creation 24
phenomenon 45
physical
 before image 74
 sort 40

- plan [24](#), [41](#), [71](#), [82](#), [82](#), [118](#), [119](#)
 - internal [17](#)
 - maximum number [84](#)
 - selection [30](#)
 - SQL-DML [48](#)
 - standard plan [25](#)
 - total number [84](#)
- plan access [87](#), [88](#)
- plan alternatives, number [40](#)
- plan buffer [70](#), [82](#), [84](#), [87](#), [91](#), [114](#)
 - dimensioning [86](#), [88](#)
- plan creation [24](#), [82](#), [87](#), [88](#)
- plan variants, selection [30](#)
- PLANS [82](#), [84](#), [86](#), [88](#), [89](#)
- plus [197](#)
- Polish list [130](#)
- pool elements [108](#)
- pool utilization [16](#)
- POSITION (clause) [197](#)
- postfix notation [130](#)
- pragma [39](#), [119](#), [120](#)
 - advice on using [41](#)
 - EXPLAIN [17](#), [133](#)
 - IGNORE INDEX [37](#)
 - IGNORE/USE INDEX [18](#), [39](#)
 - IGNORE/USE SORT_INDEX [18](#), [39](#)
 - JOIN [18](#), [39](#), [119](#)
 - JOIN NESTED LOOP [121](#)
 - JOIN SORT MERGE [121](#)
 - KEEP JOIN ORDER [18](#), [33](#), [40](#), [119](#)
 - OPTIMIZATION LEVEL [18](#), [31](#), [32](#), [35](#), [40](#)
 - optimization without [32](#)
 - PREFETCH [70](#), [115](#), [116](#)
 - SIMPLIFICATION [39](#), [133](#)
 - SIMPLIFICATION OFF [27](#), [33](#)
 - SIMPLIFICATION ON [145](#)
 - SIMPLIFICATION ON/OFF [18](#), [39](#)
- pre_rest [158](#), [158](#)
- precalculated, expression [28](#)
- precompilation [53](#)
- precompilation time, VGM utilization [53](#)
- precomputable [28](#)
 - uncorrelated subquery [31](#)
- precondition [29](#), [29](#)
- predicate [125](#), [136](#), [195](#)
 - complex [41](#)
 - elimination [26](#)
 - evaluating [28](#)
 - index use [37](#)
 - node format [201](#)
 - normalization [25](#), [41](#)
 - subquery [124](#)
 - that can be evaluated in advance [29](#)
- preempting, restrictions [28](#)
- prefetch buffer [54](#), [70](#), [92](#), [116](#)
 - dimensioning [94](#), [95](#)
 - occupancy [54](#), [92](#)
- prefetch cursor [115](#), [116](#)
- PREFETCH-BUFFER [54](#), [70](#), [92](#), [94](#)
- PREFETCH-BUFFERS mask (SESMON) [16](#), [70](#)
- preparable statement, VGM utilization [53](#)
- PREPARE [82](#), [88](#)
- PREPARE-FOREIGN-COPY [218](#), [233](#), [234](#)
- prepared statement, bottleneck prevention [53](#)
- prerequisites
 - index use [36](#)
- primary buffer [86](#), [87](#)
- primary data
 - disk storage requirements [59](#)
- primary key [37](#), [67](#), [119](#)
- primary key constraint [62](#), [64](#)
- primary key index [75](#)
- primary key range
 - sequential read [36](#)
- PRIMARY-ALLOCATION [99](#), [102](#)
- printable
 - statement output [20](#)
- PRIVATE-DISK [99](#)
- problems
 - causes [70](#)
 - corrective measures [70](#)
 - diagnostic options [70](#)
 - performance [70](#), [124](#), [125](#)
- procedure [188](#)
- procedure statements [188](#)
- processing
 - distributed [108](#)

- processing step
 - cost 19
 - I/O behavior 20
 - processing strategy, example 134
 - programming recommendation
 - CALL-DML 129
 - SQL 114
 - propagation of constants
 - simplification 26
 - pseudo update 129
 - PUBLIC-DISK 99
 - pubset, service tasks 99
- Q**
- quantifier 124, 192, 192
 - query block, optimization 32
- R**
- range construction
 - OPTIMIZATION LEVEL 41
 - range query 127
 - read
 - sequential 36, 119
 - read no lock 46
 - read no write 46
 - read phase 140
 - RECONFIGURE-DBH-SESSION 233
 - record header 59
 - record lock, isolation level 47
 - RECORDS-PER-CYCLE 99, 102
 - RECOVER 220
 - shortening read in times 220
 - speeding up modifications logged in log files 225
 - using foreign copy 221
 - using replication 221
 - utility statement 78
 - with SCOPE-PENDING 224
 - RECOVER INDEX
 - utility statement 71
 - RECOVER-OPTIONS 103
 - reducing, statement text 52
 - referential constraint 63, 124
 - when updating 63
 - REFRESH 78
 - reg_exp_like_pred 202
 - RELATION 140
 - relational condition 129
 - relevance to performance
 - application 23
 - RELOAD-DBH-SESSION 233
 - REORG 228
 - advice on using 229
 - REORG ONLINE TABLE 228
 - REORG SPACE 37, 228
 - REORG STATISTICS 120, 121
 - REORG STATISTICS FOR INDEX 18, 30
 - updating 37
 - repair
 - user space 67
 - request
 - entry 14
 - exit 14
 - request-specific, space requirements 52
 - resident
 - indexes 75
 - tables 74
 - resource consumptionLH> 51
 - resource requirements 114
 - resources
 - bottlenecks 16
 - use 19, 20
 - rest 159, 159
 - restart log file see WA-LOG file
 - RESTART-CONTROL 77, 233, 234
 - restriction predicate 158, 159
 - restrictions
 - preempting 28
 - retrieval statement 128
 - reusing, plan 85
 - RNL 46
 - RNW 46
 - ROLLBACK 55
 - ROLLBACK WORK 82

- routine 117
 - COMMIT WORK 55
 - communication overhead 55
 - CPU time 55
 - cursor 55
 - EXECUTE privilege 55
 - ROLLBACK 55
 - runtime 55
- row
 - empty 157
 - space requirements 74
- runtime 55
 - VGM utilization 53
- S**
- S subquery 98
- same name
 - column references with 195
- SAN 129
- schema information 86
- schema reduction 34
- scratch file 99
- SCROLL statement 115
- search
 - sequential 73, 130
- search condition 119, 127, 129
- search range
 - combining 34
- secondary buffer 86, 87
- secondary index 37, 71, 75, 129
 - compound 38
 - sequential read 36
- secondary index block, lock 47
- secondary index use, base table 36
- SELECT list 115
 - bottleneck prevention 52
- SELECT statement 182
 - dynamic 183
- selection, join algorithm 33
- selectivity 38, 71
 - subpredicate 37
- semantic analysis, SQL statement 25
- separation, subpredicate 34
- sequence 132
- sequential
 - read 36, 38, 124, 127
 - search 73, 130
- SERVICE ORDERS mask (SESMON) 16
- service task 78, 97
 - information 15, 16
- SERVICE TASKS mask (SESMON) 15, 97
- SERVICE-TASKS see DBH option
- SESAM/SQL-DBH
 - information 15
- SESAM/SQL-DCN 108
 - information 16
- SESAM/SQL-Server 9
- SESCOS 14, 19, 71, 118
 - recommended approach 20
 - steps 20
- SESCOSP 19, 71
- SESDCN 109
 - control statement SET-DCN-OPTIONS 108
 - parameter USERS 108
- SESDCN mask
 - CAPACITY 16
- SESDCN memory pool 108
- SESMON 14, 15, 70, 72, 78, 79, 81, 87, 110
 - in interactive mode 15
- SESMON mask
 - APPLICATIONS 16
 - I/O 16, 77, 78
 - OPTIONS 15
 - OVERVIEW 16
 - PREFETCH-BUFFERS 16
 - SERVICE ORDERS 16
 - SERVICE TASKS 15, 97
 - SQL INFORMATION 15
 - STATEMENTS 16
 - SYSTEM INFORMATION 15, 70, 72, 81
 - SYSTEM THREADS 16
 - TASKS 16
 - TRANSACTIONS 16, 71, 110
- SESMON mask "PREFETCH-BUFFERS" 54, 94, 95
- session-control statement 82
- SET CATALOG statement 82

- SET clause 115
- SET SCHEMA statement 82
- SET-SQL-DB-CATALOG-STATUS 234
- shared lock 47
 - utility 50
- shared plans 91
- SI base level 75
- SI index 75
- SIMPLIFICATION
 - effect 39
 - pragma 18, 39, 119, 133
- simplification 26, 26
 - automatic deactivation 27
 - constant expression 26
 - forcing 27, 145
 - influencing 27
 - join 27
 - propagation of constants 26
 - search conditions 26
- SIMPLIFICATION OFF 27, 123, 123
 - join order 33, 123
- SIMPLIFICATION ON 145
- Single SELECT statement 182
- single_select_stmt 182
- single-task sort 99
- size
 - plan buffer 82, 86, 86, 87, 88
 - prefetch buffer 92, 94
 - primary key index 75
 - secondary index 75
 - SESDCN memory pool 108
 - system data buffer 75, 76
 - user data buffer 74, 76
 - work container 72
- SOME 192
- SORT 98, 99, 100
- sort 120, 161, 161
 - index scan 35
 - multi-task 99
 - outer values 31
 - physical 40
 - single-task 99
- sort criteria 98, 102
- sort index 120
- sort minimization
 - index 36
 - influencing 35
 - OPTIMIZATION LEVEL 40
- sort operation
 - elimination 35
 - optimization 35
- sort operation minimization 35
- sort order
 - inheriting 35
- sort row 101
- sort sequence 102
- sort subtask 99
- sort-merge join 33, 33, 35, 120, 121
 - forcing 33
- sorted table 161
- sorted_group 126, 127, 162
- sorting 78, 98, 120, 126
 - intermediate file 203
- sortorder 120
- source text
 - Explain component 132
- space
 - access 16
 - distribution 65
 - lock granularity 66
 - size 57
- space requirements
 - base table 57
 - communication buffer 52
 - conversation-specific 53
 - row 74
 - secondary index 58
- SPACEOPT 106
- splitting
 - into query blocks 31
- splitting, CREATE SCHEMA statement 52
- SQL
 - dynamic 114
 - static 114
- SQL access plan, see plan
- SQL applications 114
- SQL descriptor area
 - VGM utilization 53

- SQL INFORMATION form (SESMON) [70](#), [118](#)
- SQL INFORMATION mask (SESMON) [15](#)
- SQL optimizer [17](#)
- SQL plan, see plan
- SQL statement [82](#), [83](#), [84](#), [85](#), [88](#)
 - ALTER TABLE [231](#)
 - CREATE INDEX [231](#)
 - for session control [82](#)
 - for transaction control [82](#)
 - optimization [24](#)
 - performance-critical [119](#)
 - prepared [114](#)
 - semantic analysis [25](#)
 - syntactic analysis [25](#)
 - tuning [118](#)
 - without plan creation [24](#)
- SQL-DML plan [48](#)
- SQL-DML statement
 - Explain component [17](#)
- SQL-SUPPORT see DBH option
- SSL [82](#)
- SSL statement [24](#)
- standard plan [25](#)
- START-IMMED [102](#)
- starting
 - multiple SESDCNs [108](#)
- statement
 - ALTER TABLE [231](#)
 - CALL [188](#)
 - CALL-DML [128](#)
 - I/O behavior [20](#)
 - measuring duration [19](#)
 - number [16](#)
 - printable output [20](#)
- statement name [114](#)
 - reusing [53](#)
- statement statistics
 - SESCOS [20](#)
- statement text
 - identical [83](#)
 - reducing [52](#)
- statement type
 - bottleneck prevention [52](#)
 - communication buffer [52](#)
- STATEMENTS mask (SESMON) [16](#)
 - static [82](#), [83](#), [84](#), [85](#)
 - statistics [18](#), [30](#), [37](#), [138](#)
 - data [14](#)
 - resources [20](#)
 - updating [18](#), [121](#)
 - statistics counters [15](#)
- STEP-COMPLEXITY [20](#)
- STEP-IO-STATISTICS [20](#)
- storage requirements
 - index [101](#)
- STORE statement [115](#)
- store_temp [164](#)
- Stored Procedure [188](#)
- strategy
 - join order [32](#)
- STRING-FORMAT [20](#)
- subdividing
 - into subrequests [30](#)
- suborders [81](#)
- subpredicate
 - selectivity [37](#)
 - separation [34](#)
- subquery [125](#)
 - correlated [31](#)
 - non-correlated [124](#)
 - optimization [31](#), [31](#)
 - uncorrelated [31](#)
- subquery optimization
 - OPTIMIZATION LEVEL [41](#)
- subrequest [30](#)
- SUBSTRING [197](#)
- substring [197](#)
- SUM [126](#)
- suppressing
 - subquery optimization [31](#)
- switching, join operands [40](#)
- synchronization [45](#)
 - utilities [49](#)
- syntactic analysis, SQL statement [25](#)
- syntax analysis [128](#)
- SYS_DBC_ENTRIES [110](#)
- SYS_DML_RESOURCES [21](#), [71](#), [118](#)
- SYS_INFO_SCHEMA [71](#)

SYS_LOCK_CONFLICTS 110
 system data buffer 73, 75
 SYSTEM INFORMATION form (SESMON) 72, 79
 SYSTEM INFORMATION mask (SESMON) 15, 70, 72, 81
 system table 61
 SYSTEM THREADS mask (SESMON) 16
 SYSTEM-DATA-BUFFER see DBH option

T

TA nesting 71
 TA-LOG file 16, 65, 103
 table
 buffering 74
 empty 144
 table_function_scan 166
 table_scan 119, 120, 121, 122, 124, 136, 167, 168
 table-value operation 140
 TASKS mask (SESMON) 16
 tautology, elimination 26
 temporary
 cursor file 66
 work file 66
 The 37
 theta-join 33
 join order 32
 thread
 change of 14
 threads 79
 TIAM application
 prefetch buffer 94
 time
 access path selection 37
 times 197
 tools, performance analysis 13
 transaction 233, 234
 measuring duration 19
 number 16
 SESMON 16
 transaction lock
 space 48
 transaction log file, see TA-LOG file

transaction states 16
 transaction statistics
 SESCOS 20
 transaction-control statement 82
 TRANSACTIONS mask (SESMON) 16, 71
 transfer container 15, 70, 72
 TRANSFER-CONTAINER see DBH option
 trim_both 197
 trim_leading 197
 trim_trailing 197
 TRIM(BOTH ..) 197
 TRIM(LEADING ..) 197
 TRIM(TRAILING ..) 197
 tuning 14
 of CALL DML applications 128
 of SQL statements 114, 118

U

uncorrelated subquery 31
 union 169, 169, 171
 UNIQUE constraint 64
 uniqueness constraint
 DISTINCT elimination 29
 index 136
 UNLOAD 213
 advice on using 214
 lock 50
 UNLOAD OFFLINE 213
 UNLOAD ONLINE 50, 213
 UPDATE statement 82, 115, 124, 128, 129, 172, 173
 WHERE CURRENT OF 115
 update_stmt 124, 173
 updating
 base table 172
 REORG SPACE 37
 REORG STATISTICS FOR INDEX 37
 rows 177
 statistics 18, 30
 UPPER 197
 upper 197
 used index 119, 120, 121
 used pool elements 108
 User Close 129

user data 73
user data block 73, 74
user data buffer 73, 74
user space 67
USER-DATA-BUFFER see DBH option
USERS (DCN parameter) 108
USERS, see DBH option 87
utilities
 parallel 71
 synchronization 49
Utility statement 82
utility statement
 plan creation 24
utilization
 disk access buffer 14
 service tasks 97
UTM application
 bottleneck prevention 54
 prefetch buffer 94, 95

V

value distribution 30
 current 37
 database 18
value query 194, 194, 196
VARCHAR 126, 127
VGM 53, 114
 utilization 53
VGMSIZE 54
view
 lock conflict 110
 permanent 52
 SYS_DBC_ENTRIES 110
 SYS_LOCK_CONFLICTS 110
view substitution 25
virtual_scan 171

W

WA-LOG file 74, 75, 103
when_then 198
WHERE CURRENT OF 82
work container 15, 72
work file 99, 101
WORK-CONTAINER see DBH option

WORK-FILES 99, 102
workflows
 analyzing 14