SAP® MaxDB™
Introduction to Query Optimization
Version 7.7

Werner Thesing
Optimization Overview

Optimization
Explain
Strategy Examples
Query Rewrite
Update Statistics
Goals of Optimization

Goal: Minimizing resource-consumption like
- CPU-time
- I/O-load
- Memory
- Disk space

SQL Commands affected by optimization
- SELECT (mass select, single select)
- Update
- Delete
- Insert
There are two types of optimizers for relational database systems: rule-based and cost-based optimizers.

The rule-based optimizer works according to certain rules. For example, if an index is available, this index will be used for access - independent of the values in the WHERE condition. With the rule-based optimizer, the strategy for processing SQL statements is decided at the time of parsing.

Cost-based optimizers determine the best search strategy with the help of statistical information about the size of the table and values within the table columns.

A cost-benefit plan is created for the various access options. The best strategy is chosen to execute the command depending on the values defined in the WHERE condition. Therefore, the eventual search strategy can only be determined at the time of execution.

MaxDB supports cost-based optimizers.

Before the optimization Query Rewrite checks if the statement can be reformulated in a reasonable way. This check and conversion is done rule-based.
First, an SQL statement is processed by the parser. This performs a syntactic and semantic analysis. In the semantic analysis, tables and their column data are checked.

The optimizer determines which primary and secondary keys are available for the table and checks whether a corresponding key can be used to search for values.

For secondary keys, the number of differing values plays an important role. Example: it does not make sense to search using an index if there is only one secondary key value, it is precisely this value that will be searched for, and additional table fields will be queried.

The number of pages that have to be read in the secondary index is determined by generating a start and a stop key. Depending on the number of pages of the table, it is decided whether it is worthwhile to search using the index. The number of pages of the entire table is located in the statistics.

At the end, the strategy with which the SQL statement will be executed is determined.
For a JOIN, the optimizer seeks out the most suitable access path for each table.

Then it has to be decided in which order the tables will be processed and connected with each other. The resulting result sets should be as small as possible. For the join columns, the values are unknown before the execution. Therefore, the join optimizer can only work with the statistical values for columns.
The search conditions that the optimizer can use to determine the optimal search strategy are the following:

- Equality conditions
- Range conditions
- IN conditions
- IN conditions

Here, the search conditions are displayed in the order of their valency. In other words, with the same preconditions an equality condition is evaluated as being better than an IN condition.

The SQL Optimizer also converts conditions under certain circumstances. If a single value is specified in an IN condition multiple times, the condition is converted into an equality condition.
In the following examples, we use the table ZZTELE with approx. 115,000 records.

For joins and subqueries, the examples also refer to the table ZZSTADTTEIL with approx. 20000 records.
The data of the base tables and the indexes are stored in B*Tree format.

When creating a table, the root page is created. A root page can contain a maximum of 8 KB.

If data records are entered in the tables, the root page is filled with what are known as separators. A separator is made up of the primary key of the data record. However, due to space limitations, the entire key is not saved as a separator in the root page, but rather only the part of the key up to the first significant digit in the key. The more significant a key is, the smaller the separators are and the more separators can be managed in the root page.

For very small tables, all data records are already stored in the root page. If the root page is filled, entering additional records in the table will automatically generate an additional tree level, or what is known as the Level 1 pages level. The root page will then consist only of separators and pointers to the corresponding lower level containing the information with a distinguishing separator.
Primary and Secondary keys (indexes)

Primary key
- The primary key is kept on the data tree (clustered)
- No separate tree for primary key!
- The primary key is used as separator in B*trees
- The records are stored in primary key order

Secondary key (index)
- Create a separate B*tree for the secondary key
- A secondary key does not contain physical addresses pointing to the base data but logical addresses in terms of primary keys
In the ABAP-based SAP application server, EXPLAIN is available in transactions ST05 and DB50 (in the command monitor).

In the SQL editor of the Database Studio you can send an EXPLAIN via context menu (right mouse click) to the database. The output is shown in a separate window.

You can display the search strategy for INSERT, DELETE and UPDATE commands by transforming the command into a SELECT. The additional option FOR REUSE ensures that the result table is stored.

Example:

```sql
UPDATE ZZTELE
SET ADDINFO = 'ledig'
WHERE NAME = 'Mueller'
AND VORNAME = 'Egon'
AND STR = 'Wexstraße'

SELECT * FROM ZZTELE
WHERE NAME = 'Mueller'
AND VORNAME = 'Egon'
AND STR = 'Wexstraße'
FOR REUSE
```
**EXPLAIN** shows:

- one block for each table from the SELECT-FROM list
- the order of the strategies reflects the order of execution
- the order of the strategies reflects the order of execution
- COPIED / NOT COPIED --> Results set is generated/not generated
- "Estimated costs" provides an estimate about the number of disk accesses (logical I/Os).
- Applied Query Rewrite rules and the frequency of their use
An index contains the data of the secondary key as well as the respective primary key.

Using the primary key, the data can be found in the base table. For each index, a B*-tree is created, which is sorted according to the values of the secondary key.

There is no record ID or anything similar. The unique ID of a record is the primary key (or for multiple keys, the combination of primary key fields).

If no primary key was specified when the table was generated, the database generates the internal field `SYSKEY` of the type `CHAR(8) BYTE`. This field is filled with unique values.

Searching via an index is relatively costly. The access is only worthwhile if less than approx. 30% of the records can be determined from the index and no result set is generated.

On the following page you will find examples of search strategies. The list of strategies is not complete. A complete list of search strategies can be found in the documentation.

Basic Information -> Background Knowledge -> SQL Optimizer -> Search Strategy -> List of all search strategies
EQUAL CONDITION FOR KEY provides an efficient access path through "direct access" to the base table.

The decision in favor of this strategy will already have been made at the time of parsing because, independent of the data in the search conditions, no better search strategy is possible.
If a portion of the start of the primary key is specified in the WHERE condition, the strategy RANGE CONDITION FOR KEY will be executed.

If the index and primary key cannot be used, the base table will be searched completely (TABLE SCAN).

An intermediate result set is not generated.
The IN condition can be placed on each field of a primary key.

Only one IN condition is taken into account.

The primary key fields that precede the field with the IN condition may only be specified in an EQUAL condition.

An intermediate result set is generated. The result set is sorted according to the primary key.

As of version 7.4, the optimizer checks whether the RANGE CONDITION FOR KEY is advantageous. This happens if the values in the IN condition are close to each other.

Example:

```
SELECT *
FROM zztele
WHERE Name IN ('Schaefer', 'Schmidt')
```

There are additional names in the table that are located between the values 'Schaefer' and 'Schmidt'. There are additional names in the table that are located between the values 'Schaefer' and 'Schmidt'. Thus, using this search condition, records are also included that
do not belong to the results set. However, the strategy is more favorable since only one start and stop key have to be determined.
If a subquery returns primary key values, EQUAL CONDITION FOR KEY or RANGE CONDITION FOR KEY is used on the base table. The result set is sorted according to primary key values.

An intermediate result set is generated.
If a SELECT statement only addresses columns that are also contained in an index (SELECT list, WHERE clause), then only this index will be accessed for the execution of the command.

**Advantage:**
- In some cases, significantly fewer pages that have to be searched
- Optimal usage of sorting of secondary and primary keys in the index
- No additional access to the base table
- No determination of access costs (only for the join)

**Exceptions:**
- No index SCAN if the index is larger than the base table
- Efficient primary key strategy via the base table

The old name for parameter IndexlistsMergeThreshold was OPTIM_INV_ONLY.
Efficient access path for fields with greater selectivity

When determining the strategy, additional costs (index_overhead) for accessing the base data via the index are taken into account.

The optimizer also opts for the strategy EQUAL CONDITION FOR INDEX, if all fields of a multiple index in the WHERE condition are specified with an equality condition.

An intermediate result set is not generated.
CREATE INDEX "ZZTELE-3" ON ZZTELE(PLZ)
SELECT * FROM zztele WHERE plz IN
  ( SELECT plz FROM zzstadtteil WHERE plz = '12047' )

The result set is sorted according to the secondary key sequence. If only values from the index are queried, the Only Index strategy is used.

An intermediate result set is generated.
The result set is sorted according to the primary key. Using the additional strategy TEMPORARY INDEX CREATED, the primary keys are sorted in a merge list. The optimum cache usage is guaranteed using access to the base data in the order of the primary keys.

The maximum size of the merge lists that are generated can be configured using the parameter IndexlistsMergeThreshold (OPTIM_MAX_MERGE).

An intermediate result set is not generated.
A secondary key can be taken into account for an IN condition. Only one IN condition is taken into account.

The secondary key fields that precede the field with the IN condition may only be specified in an EQUAL condition.

The result set is sorted according to the secondary key.

The Only Index strategy can be used.

An intermediate result set is generated.
During an INDEX SCAN, all entries are read via the index in the order of the secondary key. An intermediate results set is not generated.

As of version 7.4, NULL values are also included in single indexes. Thus, this strategy can be used on all indexes.

If a Table Scan is to be carried out for an ORDER BY because no index can be used, an intermediate result set is generated.
Nested OR terms are analyzed down to the third level.

The strategy search is only carried out if there is no adequate strategy on the highest level.

If the costs of the strategy search exceed the costs determined for the highest level, the strategy search is discontinued.

An intermediate result set is generated.

Within the SAP environment, similar statements are also generated by SELECTS with RANGES.
NO STRATEGY NOW (ONLY AT EXECUTION TIME)

Strategy will be determined first during execution of the command

Is displayed for queries if the access path will be determined first when they are executed

Is displayed for queries containing sub-queries or correlated sub-queries: strategy will first be determined when interim results become available.
The costs for a join are based on information about the value distribution.

In general, the costs of a join decrease as the number of joined columns increases.

For joins, an intermediate result set is always generated.
Join Strategies (2)

```
SELECT * FROM scantab, jointab
WHERE scantab.A = jointab.Col1
AND scantab.B = jointab.Col2
```

<table>
<thead>
<tr>
<th>Join Strategy</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOIN VIA KEY COLUMN</td>
<td>col1 is the sole primary key column</td>
</tr>
<tr>
<td></td>
<td>col2 is a standard column</td>
</tr>
<tr>
<td>JOIN VIA KEY RANGE</td>
<td>col1 is the first primary key column</td>
</tr>
<tr>
<td></td>
<td>col2 is a standard column</td>
</tr>
<tr>
<td>JOIN VIA MULTIPLE KEY COLUMNS</td>
<td>col1 is the first primary key column</td>
</tr>
<tr>
<td></td>
<td>col2 is the last primary key column</td>
</tr>
<tr>
<td>JOIN VIA RANGE OF MULTIPLE KEY COLUMNS</td>
<td>col1 is the first primary key column</td>
</tr>
<tr>
<td></td>
<td>col2 is the second primary key column</td>
</tr>
</tbody>
</table>
SELECT * FROM scantab, jointab
WHERE scantab.A = jointab.Col1
AND   scantab.B = jointab.Col2

<table>
<thead>
<tr>
<th>Join Strategy</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| JOIN VIA INDEXED COLUMN                           | col1 is a single index column
|                                                   | col2 is a standard column                         |
| JOIN VIA MULTIPLE INDEXED COLUMNS                 | col1 is the first column of a multiple index
|                                                   | col2 is the last column of a multiple index       |
| JOIN VIA RANGE OF MULTIPLE INDEXED COLUMNS        | col1 is the first column of a multiple index
|                                                   | col2 is the second column of a multiple index     |
Joins are mostly executed with the Nested Loop method. In doing so for the single join transitions no result sets are built. Only the final result is fully created before the first row is delivered.

As of version 7.7 there is no more possibility to choose between Sorted Merge or Nested Loop by a parameter setting (JOIN_OPERATOR_IMPLEMENTATION). There are only marginal disadvantages concerning CPU usage for Nested Loop with the current algorithms. Therewith the Nested Loop can deliver the result faster and with the use of less resources.
The hash join strategy is employed when a join transition to a small table is done and it is probable that a large number of records needs to be read from the small table.

In this case it would be faster to import the small table once and generate a temporary hash table. Searching for the keys in a hash table is faster than searching via the B* tree of the table.

The strategy "TABLE HASHED" identifies the join via a hash table.

The old parameters influencing this behaviour were MAX_HASHTABLE_MEMORY and MAX_SINGLE_HASHTABLE_SIZE.
MaxDB supports the following hints, the meaning of which can be extracted from SAP note 832544:
- KEYACCESS, KEYRANGE, INDEXACCESS[<INDEXNAME>], KEYSCAN,
  INDEXSCAN, INDEXRANGE, BUILDRESULT, FETCHRESULT, DISABLE_INVONLY,
  IN_STRATEGY, SUBQ_STRATEGY, TRACE, ORDERED, COORDINATOR_JOIN,
  OPERATOR_JOIN, PARALLEL_SERVER(<unsigned integer>), NOACCESSPATH,
  ACCESS=<access hint list>, BUFFERSIZE, QUERYREWRITE_OP,
  QUERYREWRITE_STMT, QUERYREWRITE_NO
Query Rewrite investigates the statement after the syntactical analysis.

Query Rewrite does a semantical analysis and rebuilds the statement if rules can be applied. Several rules can be applied.

Some rules (f.e. DistinctPushDownTo) do not change the statement itself but the internal Query Graph. This allows to apply other rules.

The execution of some rules does not rearrange the statement but provides some additional information. The rule DistinctPullUp deposits the information that all rows are unique. It is not necessary for the execution of the statement to create an internal result set to guarantee the uniqueness of the rows then.

The rearranged statement with the possible execution plans is stored in internal format within Shared SQL or the catalog cache, respectively. During the execution the optimizer determines the best execution plan for the rearranged statement.

Query Rewrite works rule-based. Statistical data is not taken into account. There is no evaluation of data.
You can influence the use of Query Rewrite by setting the parameter ENABLEQUERYREWRITE.

Furthermore you have the possibility to switch single rules on or off. Use an UPDATE statement on table QUERYREWRITERULES to set the attribute ACTIVE for the corresponding rule to YES or NO.

<table>
<thead>
<tr>
<th>RULENAME</th>
<th>ACTIVE</th>
<th>COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddLocalPredicates</td>
<td>YES</td>
<td>Add Local Predicates for Joins with OR-Predicates</td>
</tr>
<tr>
<td>ConvertExistentialSubquery</td>
<td>NO</td>
<td>Convert a correlated existential subquery to an IN clause</td>
</tr>
<tr>
<td>ConvertOrToIn</td>
<td>YES</td>
<td>Convert OR to IN</td>
</tr>
<tr>
<td>ConvertToExistentialSubquery</td>
<td>NO</td>
<td>Convert INTERSECT or EXCEPT to an existential subquery</td>
</tr>
<tr>
<td>DistinctForSubqueries</td>
<td>YES</td>
<td>Set Distinct for existential and all subqueries</td>
</tr>
<tr>
<td>DistinctPullUp</td>
<td>YES</td>
<td>Remove distinct elimination in a select if all fromselets are distinct</td>
</tr>
<tr>
<td>DistinctPushDownFrom</td>
<td>YES</td>
<td>Distinct push down from</td>
</tr>
<tr>
<td>DistinctPushDownTo</td>
<td>YES</td>
<td>Distinct push down to</td>
</tr>
<tr>
<td>EliminateGroupByOrDistinct</td>
<td>YES</td>
<td>Remove unnecessary GROUP BY or DISTINCT</td>
</tr>
<tr>
<td>EliminateOrderBy</td>
<td>YES</td>
<td>Remove unnecessary ORDER BY</td>
</tr>
<tr>
<td>EliminateSubqueries</td>
<td>YES</td>
<td>Eliminate Subqueries</td>
</tr>
<tr>
<td>MergeExistentialSubquery</td>
<td>YES</td>
<td>Merge a select with an existential subquery</td>
</tr>
<tr>
<td>MergeFromSelectOrView</td>
<td>YES</td>
<td>Merge a select with a fromselect or view</td>
</tr>
<tr>
<td>NormalizePredicates</td>
<td>NO</td>
<td>Normalize Predicates</td>
</tr>
<tr>
<td>OptimizeSubqueries</td>
<td>YES</td>
<td>Optimize subqueries</td>
</tr>
<tr>
<td>PushDownPredicates</td>
<td>YES</td>
<td>Push down predicates</td>
</tr>
<tr>
<td>PushDownProjection</td>
<td>YES</td>
<td>Push down projection</td>
</tr>
<tr>
<td>PushDownQuantifier</td>
<td>NO</td>
<td>Push down quantifier</td>
</tr>
<tr>
<td>RemoveConstFromGroupOrOrderBy</td>
<td>YES</td>
<td>Remove unnecessary constants from GROUP BY or ORDER BY</td>
</tr>
<tr>
<td>SimplifyPredicates</td>
<td>YES</td>
<td>Simplify Predicates</td>
</tr>
</tbody>
</table>
EXPLAIN QUERYREWRITE shows the result of Query Rewrite as SQL statement.

The view MONITOR indicates how often different rules were applied.

Mit Hilfe von EXPLAIN QUERYREWRITE können Sie das Statement ermitteln, wie es nach der Rewrite-Bearbeitung zur Ausführung kommt.

Die View SYSDBA.MONITOR zeigt an, welche Regel seit dem Start der Datenbank wie oft aufgerufen wurde.
As of version 7.5, MaxDB requires statistics data only for joins and selects with a restriction of the records in
the result, such as "WHERE ROWNUM <= n".

For the table itself, Update Statistics only determines data if the current size information is not already in the
file directory. This does not apply to table created with databases of versions < 7.6 and for which no size
information could yet be determined in the file directory.

Update Statistics determines statistics data for all columns that are primary key or index columns. It also
determines the statistics data for all columns outside of the primary key and the index, if statistics are
available.

When the Optimizer discovers tables with outdated statistics data, it enters them in the table
SYSUPDSTATWANTED. The DBM command sql_updatestat_per_systemtable executes Update
Statistics for all tables listed in SYSUPDSTATWANTED.

The DBM command executes Update Statistics for all tables in the database.

Update Statistics imports the data for a table from all data volumes in parallel. This makes it very speedy.

As of version 7.6, the sampling procedure in the standard uses a new algorithm for calculating the statistics
data. You can determine the algorithm to be used with the parameter UPDATESTAT_SAMPLE_ALGO.
The new algorithm generates more accurate statistics with fewer records read.

The programs "xpu" and "updcol" are no longer available as of version 7.6.
For tables that grow and shrink very quickly, such as spool tables, for example, it is a good idea to set the sampling rate to 0. This prevents Update Statistics from being requested and executed for these tables.

For tables that were created with versions < 7.6, the counters for size data in the file directory after upgrade to version 7.5 are not yet available. You can determine the counters with a CHECK DATA in the ADMIN state or with CHECK TABLE WITH SHARE LOCK. CHECK TABLE sets a share lock for the duration of the check.

After the upgrade from versions < 7.6 to versions >= 7.6, all table names are transferred to the table SYSUPDATECOUNTERWANTED. With every restart, the database attempts to determine the counters for all remaining tables in SYSUPDATECOUNTERWANTED for the file directory. A share lock is set on a table during processing. Determination of the counters is immediately terminated for a table if the share lock causes a lock collision.

With the following command dbmcli starts an Update Statistics with sampling for all tables
of one schema:

sql_updatestat SAP<SID>.* estimate
The Optimizer only uses the statistics data for tables only if the counters for size data are not in the file directory.
The values for TREENINDEXSIZE, TREELEAVESIZE and LOBSIZE are entered in KB.

For tables, ENTRYCOUNT shows the number of records per table. For indexes, ENTRYCOUNT shows the number of different values for the secondary key.
### Optimizer restrictions

<table>
<thead>
<tr>
<th>Restriction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of JOIN tables in SELECT Commands</td>
<td>256</td>
</tr>
<tr>
<td>Maximum number of JOIN connections</td>
<td>32767</td>
</tr>
<tr>
<td>Maximum number of ORDER columns</td>
<td>128</td>
</tr>
<tr>
<td>Maximum number of primary key columns within a strategy</td>
<td>20</td>
</tr>
</tbody>
</table>

An overview of general restrictions can be found in the reference handbook in the *Restrictions* chapter.
Thank you!