# Improve data access in SAP environments with the Oracle Cost Based Optimizer (CBO)

by Martin Frauendorfer

**Martin Frauendorfer**
Platinum Support Consultant,
Active Global Support,
SAP AG

*Martin Frauendorfer studied computer science at the University of Erlangen, Germany. He joined SAP Support in 1999 and has focused on Oracle-related problems since then. Today he works as an Expert Technical Support Consultant in SAP Solution Operation Support. His main tasks involve supporting large, critical SAP customers who use Oracle databases. Key focus areas are Oracle performance tuning and Oracle administration. You may reach him via email at martin.frauendorfer@sap.com.*

The Oracle Cost Based Optimizer (CBO) can be a powerful tool for determining the most efficient way to retrieve data from an Oracle database. The CBO identifies ways to improve data-access time by considering resource costs such as I/O, by accounting for statistical values and parameters, and by selecting appropriate indexes. Compared to the Rule Based Optimizer[1] (RBO), the CBO is more sophisticated and complex, but it also has a few drawbacks.

Over the past few years, I've investigated dozens of performance complaints related to CBO decisions to retrieve table data using access paths that are not the fastest or most efficient. Often, I find that the statistics and parameter settings available for the CBO (whether default or administrator-specified settings) are not sufficient for making the best decisions. A poor CBO decision can significantly affect the performance of the database in particular and the system overall. For example, if the CBO decides to scan a full table instead of using a selective index, retrieving data can take a thousand times longer. To optimize data access and improve database performance, you should understand how the CBO works and what you can do to resolve performance problems caused by CBO decisions.

This article is intended for Oracle database administrators who troubleshoot long-running Structured Query Language (SQL) statements and for SAP developers who want to avoid potential SQL performance problems with their programs. I begin by explaining the process and configuration settings the CBO uses to determine its data access approach. Then I describe the factors involved in making a CBO decision, and trace the decision using sample data, identifying why a CBO decision sometimes

[1] The RBO uses fixed rules to determine the access path, and does not take information such as data clustering, selectivity estimations, or data distribution into account. Although a quick and straightforward approach, the results of an RBO analysis are not as reliable as a CBO analysis.

provides unexpected results. This article also describes essential troubleshooting techniques to diagnose and solve performance problems, with a focus on problems particular to SAP environments.

> ***Note!***
>
> This article is designed for people who have experience analyzing Oracle SQL statements and are familiar with the concepts of the Oracle Cost Based Optimizer (CBO).

# Basics of CBO cost calculation

The following sections define the term "cost" as it relates to database optimization, and explain the factors that influence the cost calculation of the CBO. They conclude with a concrete SQL example of a CBO cost calculation.

## Defining cost

Database administrators and users often measure the success of a database in terms of *access time* — how quickly they can retrieve data using a certain SQL statement. Access time is closely related to the system resources an SQL statement uses, including I/O, memory, CPU, and locks. (Locks can be different Oracle internal lock mechanisms such as enqueues, latches, mutexes, or library cache locks.) Additionally, the number of retrieved records influences the network load between SAP and Oracle, which can affect access time. You can consider these system resources as costs in a database access transaction because your system expends these resources when retrieving data. The Oracle CBO enhances database performance by improving data access time and by identifying and reducing system costs.

The CBO calculates costs mainly in terms of I/O. This means it first determines how many blocks of data Oracle must read from disk, and then estimates the costs as the number of blocks read. The typical size of a block is 8,192 bytes, which is the default Oracle block size in SAP environments.

> ***Note!***
>
> Starting with Oracle 10g, SAP recommends taking advantage of *system statistics*, which reflect CPU use and the time it takes Oracle to read a single block of data and a series of consecutive blocks during cost calculation for a particular system. As a consequence, CPU consumption is also incorporated in the overall costs. However, because CPU costs are typically responsible for only a minor fraction of the total costs, this article does not consider them in the CBO cost calculation for simplification purposes. For more details on system statistics, see SAP Note 927295.

## Costs for typical database accesses

Consider the typical example of retrieving one record using a fully qualified primary B*TREE index access.[2] The Oracle runtime engine performs the following steps to retrieve a record:

1. Oracle enters the B*TREE index at the root block.
2. Oracle scans the branch blocks. Depending on the height of the index, the branch block can have 0, 1, 2, or more layers. In each branch block

[2] A B*TREE index, which is organized like a tree, is the typical index type in SAP environments. A primary index is a dedicated, unique index that is defined in SAP systems for almost every table. In a fully qualified index, all index columns are specified with an "=" (equal to) condition. As a consequence of the uniqueness and full qualification, a B*TREE primary index returns a maximum of one record.

 

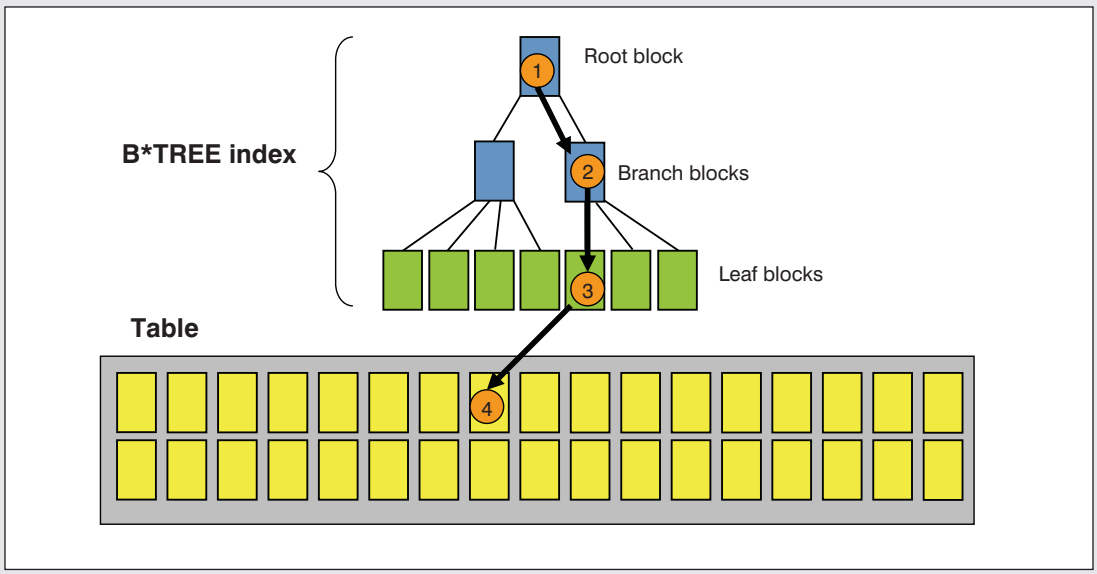


**Figure 1** Retrieving a record using a fully qualified primary B*TREE index access

layer, Oracle must read exactly one block. The branch blocks contain navigational information and pointers to the blocks at the next lower level.

3. Oracle reads the leaf block with the requested primary key record.

4. The leaf block also contains a pointer (called the ROWID or row ID) that points to the related record in the table. Using this pointer, Oracle accesses the table block that contains the complete data record.

***Note!***

This article focuses on the most common segment types in SAP environments: B*TREE indexes and normal tables. It does not cover other segment types such as bitmap indexes and index-organized tables.

**Figure 1** shows a B*TREE index access for an index consisting of three layers: root block, a branch block layer, and a leaf block layer.

As you can see, the Oracle runtime engine must read four blocks for a fully qualified primary index access: the root block, one branch block, one leaf block, and one table block. **Figure 2** (on the next page) shows the *explain plan* for this kind of access.

The explain plan shows total costs of 4 (line 1), index access costs of 3 (line 7), and table access costs of 1 (line 4). Although line 4 specifies “Estim. Costs = 4,” this includes the index access costs of 3; the pure table access costs are only 1. Furthermore, line 6 indicates that Oracle uses an INDEX UNIQUE SCAN on index TAB1~0. An INDEX UNIQUE SCAN can return a maximum of one record. You can use this type of scan if you specify all columns of a unique index with an equal sign (=) in the WHERE clause, which means that the index is fully qualified. For more information on explain plans, see the sidebar on the next page.

```
1 SELECT STATEMENT ( Estimated Costs = 4 , Estimated #Rows = 1 )
2   |
3   ---   2 TABLE ACCESS BY INDEX ROWID TAB1
4       |   ( Estim. Costs = 4 , Estim. #Rows = 1 )
5       |
6       ------1 INDEX UNIQUE SCAN TAB1~0
7               ( Estim. Costs = 3 , Estim. #Rows = 1 )
```

**Figure 2** Explain plan for a fully qualified primary index access

## Explain plans

An explain plan for an SQL statement is the result of the work of the CBO. It shows you the best access path according to the CBO, and it provides you with the estimated costs and the estimated rows for each step of the execution plan.

You can display an explain plan in SAP and Oracle in several ways:

- In SAP transaction ST05, select Enter SQL Statement, enter the SQL statement, and then click on the Explain button.
- In SAP transaction ST04, explain currently running SQL statements in the Oracle Sessions overview (ST04 → Detailed Analysis Menu → Oracle Session → click on a particular session → click on the Explain button) or explain SQL statements from the shared cursor cache (ST04 → Detailed Analysis Menu → SQL request → Confirm popup → click on one particular SQL statement → click on the Explain button).
- Explain an SQL statement with Oracle tools such as SQLPLUS using the following commands:

```
EXPLAIN PLAN FOR <sql_statement_text>;

SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);
```

- In Oracle, the Oracle view V$SQL_PLAN contains the execution plans of previously executed SQL statements.

Sometimes the output of an explain plan differs from the content in V$SQL_PLAN. (See the typical deviation scenarios in SAP Note 723879.) In general, V$SQL_PLAN contains the execution plan that Oracle uses, while explain functionalities can show incorrect accesses. Current versions of SAP Basis transaction ST04 retrieve the information from V$SQL_PLAN whenever possible. (This is also indicated in the output.)

 

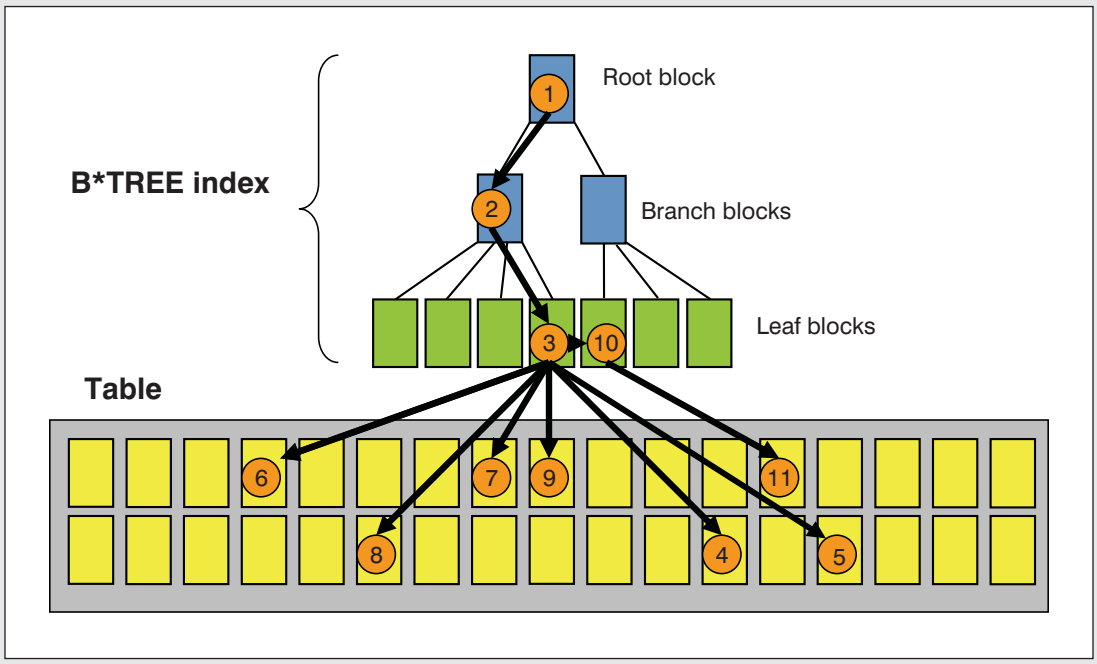


**Figure 3** Retrieving several records using a B*TREE index

```
1 SELECT STATEMENT ( Estimated Costs = 11 , Estimated #Rows = 7 )
2   |
3   ---   2 TABLE ACCESS BY INDEX ROWID TAB1
4       |   ( Estim. Costs = 11 , Estim. #Rows = 7 )
5       |
6       ------1 INDEX RANGE SCAN TAB1~0
7               ( Estim. Costs = 4 , Estim. #Rows = 7 )
```

**Figure 4** Explain plan for a B*TREE index range scan

Consider another similar example (see **Figure 3**) that does not use a fully qualified primary index access, and therefore retrieves several records.

In this case, the Oracle runtime engine finds seven records in the index. It locates six records in the first accessed leaf block (3). It finds the last record in the next leaf block (10). In total, Oracle must read 11 blocks. The explain plan for this kind of access looks similar to **Figure 4**.

You can see that the estimated costs are now 11 (line 4). Furthermore, line 6 specifies an INDEX RANGE SCAN, not an INDEX UNIQUE SCAN. For more information on index access types, see the sidebar on the next page.

## Index access types

B*TREE indexes can be accessed in the following ways:

- INDEX UNIQUE SCAN: Accesses a fully qualified primary index; it returns a maximum of one record.
- INDEX RANGE SCAN: Includes all index accesses where Oracle reads a contiguous range of one or more index leaf blocks; this is the most typical access type.
- INDEX FULL SCAN: Reads all index leaf blocks in the order of appearance in the index tree structure.
- INDEX FAST FULL SCAN: Reads all index blocks in physical order (as they are stored on disk); the advantage is that this index access type can read multiple consecutive blocks with one I/O request, while the disadvantage is that it does not sort the result set like the index.
- INDEX SKIP SCAN: Includes index accesses that skip columns specified within a range or not at all; a skip scan is a set of INDEX RANGE SCANs (one range scan for each value of the skipped columns).

## Factors influencing the CBO cost calculation

Factors that influence the CBO cost calculation include CBO statistics and Oracle parameters, patches, and hints. Each of these factors is discussed in the following sections.

### *CBO statistics*

To estimate precise costs, the CBO must gather details, called *CBO statistics*, such as the number of distinct column values, the height of the index, and the number of leaf blocks and table blocks. You can find CBO statistics in the Oracle data dictionary and can retrieve them from views including DBA_TABLES, DBA_INDEXES, and DBA_TAB_COLUMNS. **Figure 5** outlines the CBO statistics that have the most significant effect on calculating data access costs.

In SAP Business Intelligence (BI) environments and for a few other SAP tables, the CBO also creates *histogram statistics* in DBA_TAB_HISTOGRAMS views. Histogram statistics provide information on how the column values are distributed. These statistics are particularly useful with unevenly distributed column values, but they don't provide additional benefits if a system parses statements with bind variables rather than literals, which is standard for SAP Online Transaction Processing (OLTP) systems.

For more information on the clustering factor of indexes, see the sidebar on the following page.

***Note!***

You need accurate, reliable CBO statistics to make effective CBO decisions. You should therefore make sure that you are using a proper statistics update procedure. SAP recommends you schedule a BRCONNECT statistics run at least once a week using the following command:

```
brconnect –u / -c –f stats –t all
```

See SAP Note 588668 for more information related to CBO statistics in SAP environments.

 

| View | Type of statistics | Details |
|---|---|---|
| DBA_TABLES | Table statistics | • NUM_ROWS: Number of records in the table<br>• BLOCKS: Blocks that the table allocated (up to the last block ever filled, known as a "high-water mark") |
| DBA_INDEXES | Index statistics | • BLEVEL: Number of index root and branch block levels (e.g., 2 as shown in Figure 1 and Figure 3)<br>• LEAF_BLOCKS: Number of index leaf blocks<br>• CLUSTERING_FACTOR: Clustering of the table data compared to the index |
| DBA_TAB_COLUMNS | Column statistics | • NUM_DISTINCT: Number of distinct values of the column<br>• DENSITY: 1 / NUM_DISTINCT (if no histograms exist), or filter factor for unpopular values (if histograms exist)<br>• NUM_NULLS: Number of NULL values |

**Figure 5** CBO statistics

## The mystery of the clustering factor

The clustering factor of indexes is a statistic that influences CBO cost calculation and real execution times and is calculated when Oracle creates statistics based on an internal function. The clustering factor indicates how many table blocks have to be accessed if you scan all index leaf blocks and access the related table blocks for each record. If two or more consecutive index records point to the same table block, these block accesses only count as one for the clustering factor. When you have to switch to a different table block, the clustering factor is increased by one. This means the clustering factor of an index can vary in the following scenarios:

- If the table is sorted in the same way as the index, subsequent index entries usually point to the same table block and the database must read each table block only once. In this best-case scenario, the clustering factor is close to the number of table blocks.
- If the index records are spread across the table randomly, subsequent index entries usually point to different table blocks. In this worst-case scenario, the clustering factor is close to the number of table rows.

The following diagram summarizes how to interpret the clustering factor statistic:


To optimize the clustering factor of an index, you have to change the sort order of the table records. See SAP Note 832343 for more details.

| Parameter | Description | SAP Recommendation | Background |
|---|---|---|---|
| OPTIMIZER_MODE | Controls which optimizer mode Oracle uses | CHOOSE (Oracle≤ 9i)<br>ALL_ROWS (Oracle 10g) | With the setting of CHOOSE, Oracle (versions 9i and earlier) can choose to use the RBO or CBO. Usually, Oracle uses the RBO to process tables without statistics, while it uses the CBO to process tables with statistics. Because Oracle 10g no longer supports the RBO, in general it uses the CBO. The target of the CBO should be to return all rows of the result set as quickly as possible (OPTIMIZER_MODE = ALL_ROWS). |
| OPTIMIZER_FEATURES_ENABLE | Defines the CBO features that Oracle can use | Keep the default (current Oracle patchset) | Because you should generally use all CBO features, avoid setting the parameter to a previous Oracle release or patchset. |
| OPTIMIZER_INDEX_COST_ADJ | Percentage to which Oracle adapts the initially calculated index costs | 100 (OLAP)<br>10 (OLTP on Oracle ≤ 9i)<br>20 (OLTP on Oracle 10g) | Without reducing the index costs to 10% or 20% of the originally calculated index costs, Oracle performs too many table scans in OLTP environments. |
| OPTIMIZER_INDEX_CACHING | Percentage to which Oracle adapts the initially calculated costs for IN LIST iterations and nested loop joins (0 → default, 1 → no or minimum reduction, 100 → maximum reduction) | 0 (OLAP; OLTP on Oracle ≤ 9i)<br>50 (OLTP on Oracle 10g) | Oracle 10g does not always use indexes supporting columns with long IN LISTs or nested loop joins in OLTP environments if the costs for these accesses are not adapted. |

**Figure 6** Oracle parameters

### *Oracle parameters*

Some Oracle parameters affect the cost calculation of the CBO. The most important ones are outlined in **Figure 6**.

Online Analytical Processing (OLAP) defines systems that use a significant amount of BI functionality, e.g., BI and Advanced Planner and Optimizer (APO) with Demand Planning. OLTP

 

| Parameter | Description | SAP recommendation | Background |
|---|---|---|---|
| DB_FILE_MULTIBLOCK_READ_COUNT | Number of contiguous blocks that Oracle can read with one I/O request; relevant for full table scans and index fast full scans | 8 (OLTP on Oracle ≤ 9i)<br>32 (OLAP on Oracle ≤ 9i)<br>default of 128 (Oracle ≥ 10g) | With Oracle ≤ 9i, higher values for this parameter resulted in reduced costs for full table scans. To avoid inappropriate full table scans, the parameter value had to remain low.<br>With Oracle 10g, DB_FILE_MULTIBLOCK_READ_COUNT no longer affects the CBO as long as it is not explicitly set. |
| _OPTIM_PEEK_USER_BINDS | If set to TRUE (the default), Oracle reveals the values behind the bind variables during parsing | FALSE | Poor CBO decisions may result as long as bind value peeking is active. |

**Figure 6** (continued)

represents all systems with minor or no BI functionality.

In addition to these important parameters, a number of other parameters can also influence the CBO, such as PGA_AGGREGATE_TARGET and several underscore and event parameters. See SAP Note 750631 for additional details.

### *Oracle patches*

Release upgrades, patchset applications, and bug fixes can significantly affect CBO decisions. Particularly critical are CBO-related merge fixes:

- Oracle 8.1.7.4: SAP Note 601668
- Oracle 9.2.0.3: SAP Note 610445
- Oracle 9.2.0.4: SAP Note 695080
- Oracle 9.2.0.5: SAP Note 755629
- Oracle 9.2.0.6: SAP Note 834100
- Oracle 9.2.0.7: SAP Note 896903
- Oracle 9.2.0.8: SAP Note 992261
- Oracle 10.2.0.2: SAP Note 981875

Be sure to test the performance of the Oracle software after making minor and major changes before applying the changes in production environments.

### *Oracle hints*

Specifying hints[3] can influence the CBO cost calculation or the resulting access path. Some hints such as INDEX or FULL affect the type of access, while other hints such as FIRST_ROWS(1) or ALL_ROWS influence the cost calculation.

## CBO cost calculation example

A detailed example in an Oracle 10g OLTP environment can show how the CBO works. The following Requests table contains information about 100,000 requests, as shown in **Figure 7** (on the next page):

- MANDT: Client
- REQNO: Unique request number
- QUEUE: Queue that processes the request

[3] See SAP Note 772497 for more details regarding Oracle hints in SAP environments.

```
CREATE TABLE REQUESTS
( MANDT VARCHAR2(3),
  REQNO VARCHAR2(10),
  QUEUE VARCHAR2(10),
  STATE VARCHAR2(10),
  DEPREQ VARCHAR2(10)
);

BEGIN
FOR I IN 1..100000 LOOP
  INSERT INTO REQUESTS VALUES
    ( '100',
      I,
      MOD(I, 11),
      DECODE(MOD(I, 33), 0, 'NEW',
                         1, 'IN PROCESS',
                         2, 'ERROR',
                         'PROCESSED'),
      DECODE(TRUNC(I/10), 0, I, 0)
    );
END LOOP;
COMMIT;
   END;
   /
```

**Figure 7** Requests table containing information about 100,000 requests

- STATE: Current state of the process
- DEPREQ: Dependent request number (0 if no dependent request exists)

We can add the following indexes to this table:

```
CREATE UNIQUE INDEX "REQUESTS~0"
   ON REQUESTS (MANDT, REQNO);
CREATE INDEX "REQUESTS~1"
   ON REQUESTS (MANDT, QUEUE, STATE);
CREATE INDEX "REQUESTS~2"
   ON REQUESTS (DEPREQ, QUEUE, STATE);
```

Now statistics are gathered for table, indexes, and columns:

```
EXEC DBMS_STATS.GATHER_TABLE_STATS -
( USER, -
  'REQUESTS', -
  ESTIMATE_PERCENT=>100, -
  CASCADE=>TRUE, -
  METHOD_OPT=>'FOR ALL COLUMNS SIZE 1' -
);
```

**Figure 8** shows how the important statistics values now look. Be aware that the statistics values can vary depending on factors such as the Oracle release (Oracle 9i or 10g) and the tablespace type.

 

```
TABLE_NAME                     NUM_ROWS     BLOCKS
------------------------------ ---------- ----------
REQUESTS                           100000        496

INDEX_NAME                       BLEVEL LEAF_BLOCKS CLUSTERING_FACTOR
------------------------------ ---------- ----------- -----------------
REQUESTS~0                              1         278             19292
REQUESTS~1                              2         415              7093
REQUESTS~2                              2         346              5545

COLUMN_NAME                   NUM_DISTINCT    DENSITY  NUM_NULLS
------------------------------ ------------ ---------- ----------
MANDT                                     1          1          0
REQNO                                100000     .00001          0
QUEUE                                    11 .090909091          0
STATE                                     4        .25          0
DEPREQ                                   10         .1          0
```

**Figure 8** CBO statistics for the table, its indexes, and its columns

```
SELECT
  *
FROM
  REQUESTS
WHERE
  MANDT = :A0 AND
  QUEUE = :A1 AND
  STATE = :A2;

-------------------------------------------------------------------------------
| Id  | Operation                    | Name       | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |            |  2273 | 59098 |    35   (0)|
|   1 |  TABLE ACCESS BY INDEX ROWID| REQUESTS   |  2273 | 59098 |    35   (0)|
|   2 |   INDEX RANGE SCAN           | REQUESTS~1 |  2273 |       |     2   (0)|
-------------------------------------------------------------------------------
```

**Figure 9** Explain plan for a SELECT statement

Now let's look at a sample query. A typical SAP request could be to select all records belonging to a certain queue with a certain state. **Figure 9** shows an explain plan for this SELECT statement.

The CBO favors an access via index REQUESTS~1 and calculates costs of 35 for this access. To check whether these costs are reasonable for the existing CBO statistics, consider the following factors:

- **Root and branch block access costs:** The costs for root and branch block accesses should match the BLEVEL because from each root and branch block level, Oracle has to read only one block while navigating to the leaf blocks. The root and branch block access costs are two.
- **Leaf block access costs:** Calculating the number of leaf blocks that Oracle must read is more complicated. **Figure 8** shows a total of 415 leaf blocks, but the SELECT statement only has to read the range of leaf blocks that fulfills the WHERE clause conditions. Here the CBO assumes that the values are equally distributed and that values in one column are independent of values in another column. The 415 leaf blocks are divided by the distinct values of the different columns specified in the WHERE clause. The costs for leaf block accesses are therefore 10 (rounded to ceiling):

```
LEAF_BLOCKS / NUM_DISTINCT(MANDT) /
   NUM_DISTINCT(QUEUE) /
   NUM_DISTINCT(STATE) =
415 / 1 / 11 / 4 =
10
```

- **Table block access costs:** Calculating the number of table blocks that Oracle must read is similar to calculating the leaf blocks. Without any restriction, the clustering factor of the index would indicate the number of table block accesses, but in this case, the clustering factor must be divided by the distinct values of the specified columns, so that the result is 161:

```
CLUSTERING_FACTOR /
   NUM_DISTINCT(MANDT) /
   NUM_DISTINCT(QUEUE) /
   NUM_DISTINCT(STATE) =
7093 / 1 / 11 / 4 =
161
```

The total costs should be 2 + 10 + 161 = 173 according to this calculation. This is different from the CBO costs of 35. The reason for the deviation is that the CBO must account for the setting of OPTIMIZER_INDEX_COST_ADJ = 20, which Oracle 10g uses in OLTP environments. Using this setting, 20% of 173 is 35, which reflects the CBO costs.

# Typical CBO problems in SAP environments

This section examines CBO problems that are particularly prevalent in SAP environments. It identifies the circumstances in which these problems appear and explains how you can resolve them.

## The initial-value problem

SAP usually initializes every column with a default value. Typically these default values are as follows:

- ' ' (one blank) for character columns
- 0 for number columns
- '00…' for character columns containing numbers

When making CBO calculations, the CBO treats default values in the same way as any other value. If it uses bind variables during parsing, the CBO does not know whether the data includes an SAP default value. Default values can affect the CBO decision in two ways:

- **Underestimating[4] selectivity and omitting an index:** Records often have a default value in one column. Only a few records contain other values for that column. If you search for all records with a certain nondefault value, this condition is very selective: It returns few records, if any. However, the CBO assumes an equal distribution of column values and that the cardinality of the default value

[4] "Underestimating" in this context means that compared to the reality, a higher fraction of records is expected. This means that the estimated selectivity value is higher than in reality.

 

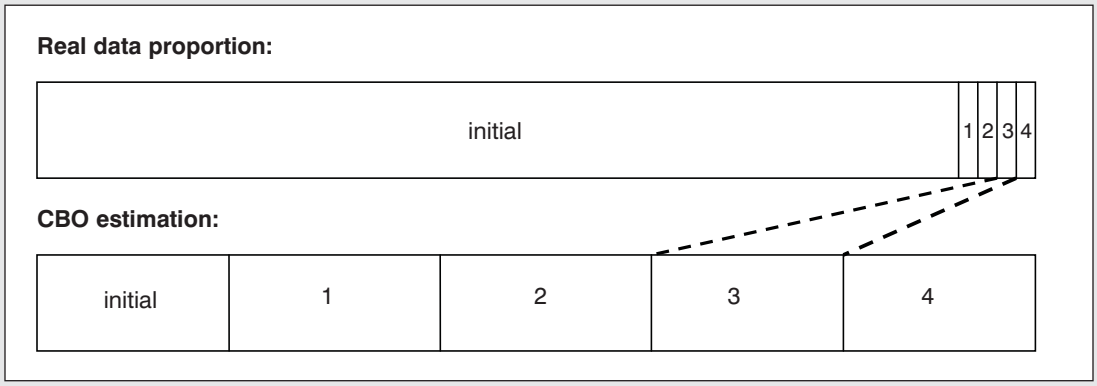


**Figure 10** Underestimating selectivity

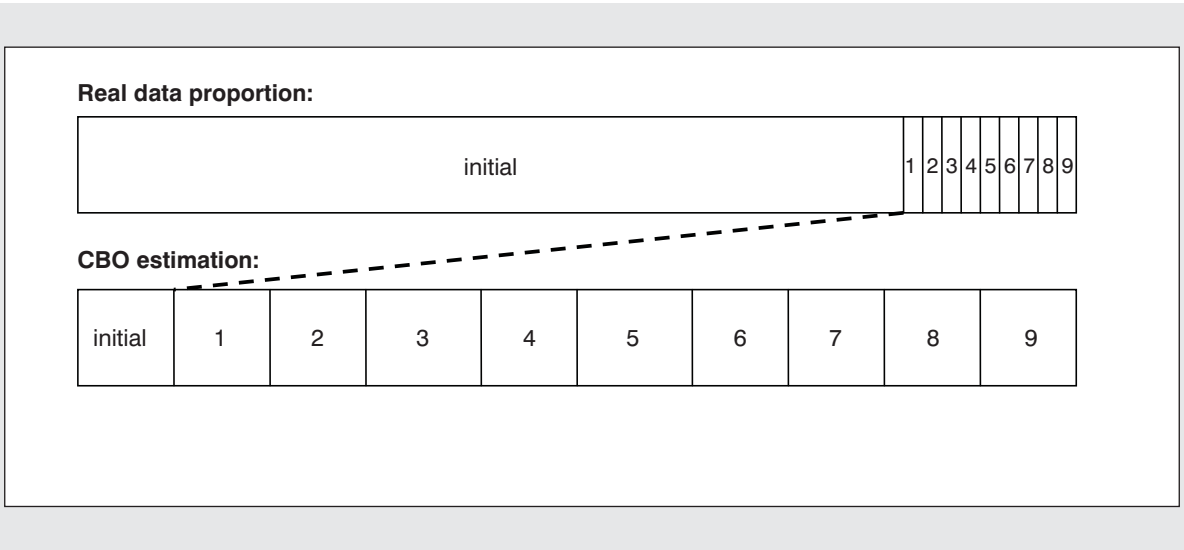


**Figure 11** Overestimating selectivity

is identical to all other existing values. As a consequence, it underestimates the selectivity of the condition and might skip an existing index on that column. See **Figure 10**. For more information on cardinality and selectivity, see the sidebar on the next page.

- **Overestimating[5] selectivity and using an index:** The opposite problem is also possible: One column has many nondefault values, but some records still contain the default value. If you select all columns with the default value, Oracle overestimates the selectivity and might use an existing index on that column although other columns might have better indexes (see **Figure 11**).

Consider the previous example of the Requests table. **Figure 12** (on the next page) compares the real and estimated cardinality for a series of values. The dependent

[5] "Overestimating" in this context means that compared to the reality, a lower fraction of records is expected. This means the estimated selectivity value is lower than it is in reality.

## Cardinality and selectivity

The terms "cardinality" and "selectivity" are often used in the area of SQL tuning and CBO.

The *cardinality* defines the number of rows that are returned. For example, if 300 out of 5,000 records are returned, the cardinality is 300. The "estimated rows" in the CBO explain plan contains the estimated cardinality.

The *selectivity* defines the fraction of data that is returned. For example, if 300 out of 5,000 records are returned, the selectivity is 6%. The lower the selectivity value, the fewer records are returned. Divide the "estimated rows" of the explain plan by the total rows to determine the estimated selectivity.

You can check whether the CBO decision changes assuming a different cardinality or selectivity. To do so, use the CARDINALITY or SELECTIVITY hint:

CARDINALITY(<*table_name*> <*rows*>): Set the estimated rows for <table_name> to <rows>

SELECTIVITY(<*table_name*> <*selectivity*>): Set the estimated selectivity for <table_name> to <selectivity>

See SAP Note 772497 for more information on Oracle hints.

| Value | Real cardinality | Estimated cardinality |
|---|---|---|
| Default (0) | 99.991 | 10.000 |
| 1 | 1 | 10.000 |
| 2 | 1 | 10.000 |
| 3 | 1 | 10.000 |
| 4 | 1 | 10.000 |
| 5 | 1 | 10.000 |
| 6 | 1 | 10.000 |
| 7 | 1 | 10.000 |
| 8 | 1 | 10.000 |
| 9 | 1 | 10.000 |
| Any other value | 0 | 10.000 |

**Figure 12** Cardinalities for DEPREQ condition on REQUESTS table

request column DEPREQ is almost always initial (0), but the CBO doesn't know if you are searching for the initial value or any other value. It assumes that the SQL statement returns 10% of records for a certain value because there are 10 distinct values.

If SAP now checks for requests with the dependent request number 1234, the CBO assumes that 10,000 records meet the WHERE conditions, so index REQUESTS~2 is not used, as shown in **Figure 13**.

A full table scan is shown, but often Oracle uses other bad indexes instead because in OLTP environments, the CBO is configured to use these indexes with OPTIMIZER_INDEX_COST_ADJ, DB_FILE_MULTIBLOCK_READ_COUNT, and OPTIMIZER_INDEX_CACHING commands. With the full table scan, Oracle must read all 496 table blocks. However, an index range scan on REQUESTS~2 would finish after the third block (BLEVEL + 1 leaf block) because Oracle recognizes that no record fulfills the WHERE clause.

 

```
SELECT
  *
FROM
  REQUESTS
WHERE
  MANDT = :A0 AND
  DEPREQ = :A1;

-------------------------------------------------------------------------------
| Id  | Operation         | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |          | 10000 |   253K|   105   (2)| 00:00:03 |
|   1 |  TABLE ACCESS FULL| REQUESTS | 10000 |   253K|   105   (2)| 00:00:03 |
-------------------------------------------------------------------------------
```

**Figure 13** CBO decision of full table scan for DEPREQ selection

## About good and bad indexes

In general, a good index for a certain set of column conditions is an index that can filter a high percentage of the table records. This means the index is selective for that set of column conditions. You cannot easily compare an index to a full table scan because several other factors affect the accesses in a table (e.g., clustering factor, range and IN LIST conditions, multiblock read count, system-specific CPU, and I/O value or degree of caching). Nevertheless, if an index has a selectivity of ≤ 1% (meaning at least 99 out of 100 records are already excluded in the index), the index is usually better than a full table scan.

See the sidebar above for more on good and bad indexes.

In Oracle, the best solution for this kind of problem is to use NULL as the default value. Oracle treats NULL in a special way: It is not comparable to any other value, so operators such as less than (<) or equal to (=) do not work with NULL values. Instead, you must use the IS operator to check whether a value is NULL. In addition, the CBO statistics contain information about the number of NULL values in each column (NUM_NULLS). As a result, the CBO can exclude NULL values from cardinality and selectivity considerations and estimate values that are much closer to reality. See **Figure 14** on the next page.

To work around this problem in our concrete example, you can set all DEPREQ values to NULL rather than 0. Afterwards, create new statistics so that the CBO is aware of this change, as shown in **Figure 15** on the next page.

After this change, the explain plan for the previous example provides the best data-access path. As shown in **Figure 16** (on the next page), Oracle expects only one row (the absolute minimum), decreases the costs to 1, and uses the good index REQUESTS~2.

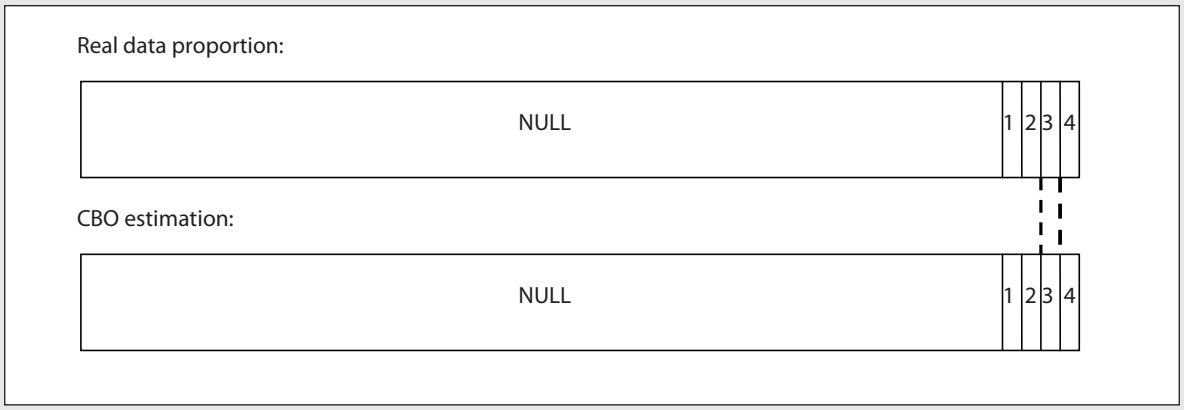


**Figure 14** Better CBO estimation if NULL values are used

```
UPDATE REQUESTS SET DEPREQ = NULL WHERE DEPREQ = '0';
COMMIT;
EXEC DBMS_STATS.GATHER_TABLE_STATS(USER, 'REQUESTS', ESTIMATE_PERCENT=>100, -
  CASCADE=>TRUE, METHOD_OPT=>'FOR ALL COLUMNS SIZE 1');
```

**Figure 15** Setting DEPREQ values to NULL

```
-------------------------------------------------------------------------------------
| Id | Operation                    | Name      | Rows  | Bytes | Cost (%CPU)| Time  |
-------------------------------------------------------------------------------------
|   0| SELECT STATEMENT             |           |     1 |    22 |     1  (0)|00:00:01|
|   1|  TABLE ACCESS BY INDEX ROWID| REQUESTS  |     1 |    22 |     1  (0)|00:00:01|
|   2|   INDEX RANGE SCAN           | REQUESTS~2|     1 |       |     1  (0)|00:00:01|
-------------------------------------------------------------------------------------
```

**Figure 16** Results after using NULL values

You can't switch from a default value to NULL in SAP environments for the following reasons:

- You must change every ABAP table access with WHERE … <column> = <default_value> to WHERE … <column> = <default_value> OR <column> IS NULL.
- Switching to NULL can affect other ABAP table

 

accesses (e.g., with NOT EQUAL conditions) so you need to add the IS NULL condition.

- SAP primary indexes rely on defining each indexed column with NOT NULL. If you specify indexed columns with NULL values, you lose the primary key property.
- A switch to NULL can be time consuming with large tables, because you must update each column individually.

Therefore, you need to use the following solutions in SAP environments:

- **Increase the statistic value (if a nondefault value is specified and the selectivity is underestimated):** For the distinct value in the critical column, increase the statistic value according to SAP Note 724545:

```
EXEC DBMS_STATS.SET_COLUMN_STATS -
( USER, -
  '"<table_name>"', -
  '"<column_name>"', -
  DISTCNT=><new_value>, -
  NO_INVALIDATE=> FALSE –
);
```

  The higher the distinct values for a column are, the more attractive the column becomes. Usually this change does not detract from other transactions if it is related to an initial-value problem. You should test the influence of the change before applying it in production environments.

  If you specify a default value in the WHERE clause, thereby overestimating the selectivity, you could theoretically reduce the statistic value for the distinct value to make the index less attractive. Nevertheless, in this case side effects are more likely because other transactions may query the same column with nondefault values.

***Note!***

In the following scenarios, increasing the distinct values does not have the desired effect:

- With Oracle 10g (and Oracle 9i ≥ 9.2.0.7 with EVENT 38060) the distinct keys of an index also limit the selectivity estimations; see SAP Note 176754 (36). In this case you must additionally increase the distinct keys using DBMS_STATS.SET_INDEX_STATS.
- If a column is specified with a range condition and already has a significant amount of distinct values, a further increase doesn't influence the cost calculation.

SAP already delivers "good" statistics for some critical tables. Check whether the problem table is mentioned in SAP Note 1020260 and apply the script attached to the note.

- **Specify a database hint:** Use a database hint to force the correct access. Depending on the data, you can use one of the following hints:
  - INDEX("<table>" "<index>"): This hint forces access via the specified index.
  - FIRST_ROWS(<n>): This hint instructs the CBO to return the first rows of the result set as quickly as possible. This is particularly useful when you want to switch from a full table scan to an index access.
  - &SUBSTITUTE VALUES& and &SUBSTITUTE LITERALS&: With these hints, you can instruct the SAP database interface to use literals rather than bind variables when sending an SQL statement to Oracle for parsing. &SUBSTITUTE VALUES& doesn't use any bind variable, while &SUBSTITUTE LITERALS& only has an effect on ABAP literal values (such as constants). These hints are SAP database interface hints, not Oracle hints. In addition, you need to create histogram

statistics on the columns because the CBO can take full advantage of knowing the values behind the bind variables only with histograms. See SAP Note 797629 for more information regarding histograms.

See SAP Note 772497 for more information regarding database hints.

- **Change the ABAP coding:** In some cases, changes to the ABAP coding can be considered, such as the following:
  - Reduce the number of elements in the IN list to make an index on the IN list column more attractive.
  - Add further indexed conditions so that the good index becomes more attractive for the CBO.

If your problem concerns SAP standard coding, SAP should provide a working solution. If your problem involves customer-specific coding or customer-specific selection variants, you will need to find your own solution.

## The timestamp problem

In SAP systems, dates and timestamps are often stored in VARCHAR2 fields. For example, the date 02.15.2008 is stored as string 20080215. As long as data uses bind variables, this representation doesn't cause problems for the CBO. Nevertheless, in some SAP environments, data does not use bind variables and so problems can occur, as with BI or Bank Analyzer.

In these environments, the system sends explicit date conditions to the database during parsing and creates histograms on the columns. To illustrate this problem, consider the following simplified example. The table ORDERDATE consists only of one column, ORDDAT. In this column, every calendar day of the years 2007 to 2010 is stored exactly once (not including February 29, 2008). In this example, ORDDAT has a NUMBER data type because the histogram information for NUMBER columns is stored in a format that is easier to read than VARCHAR2 histograms. The effects for NUMBER columns are comparable with those for VARCHAR2 columns. See **Figure 17**.

```
CREATE TABLE ORDERDATE (ORDDAT NUMBER);

BEGIN
  FOR I IN 1..31 LOOP
    INSERT INTO ORDERDATE VALUES (2007 * 10000 + 100 + I);
    INSERT INTO ORDERDATE VALUES (2007 * 10000 + 300 + I);
    INSERT INTO ORDERDATE VALUES (2007 * 10000 + 500 + I);
    INSERT INTO ORDERDATE VALUES (2007 * 10000 + 700 + I);
    INSERT INTO ORDERDATE VALUES (2007 * 10000 + 800 + I);
    INSERT INTO ORDERDATE VALUES (2007 * 10000 + 1000 + I);
    INSERT INTO ORDERDATE VALUES (2007 * 10000 + 1200 + I);
    INSERT INTO ORDERDATE VALUES (2008 * 10000 + 100 + I);
    INSERT INTO ORDERDATE VALUES (2008 * 10000 + 300 + I);
    INSERT INTO ORDERDATE VALUES (2008 * 10000 + 500 + I);
    INSERT INTO ORDERDATE VALUES (2008 * 10000 + 700 + I);
    INSERT INTO ORDERDATE VALUES (2008 * 10000 + 800 + I);
    INSERT INTO ORDERDATE VALUES (2008 * 10000 + 1000 + I);
    INSERT INTO ORDERDATE VALUES (2008 * 10000 + 1200 + I);
```

**Figure 17** Sample ORDERDATE table

 

```
    INSERT INTO ORDERDATE VALUES (2009 * 10000 + 100 + I);
    INSERT INTO ORDERDATE VALUES (2009 * 10000 + 300 + I);
    INSERT INTO ORDERDATE VALUES (2009 * 10000 + 500 + I);
    INSERT INTO ORDERDATE VALUES (2009 * 10000 + 700 + I);
    INSERT INTO ORDERDATE VALUES (2009 * 10000 + 800 + I);
    INSERT INTO ORDERDATE VALUES (2009 * 10000 + 1000 + I);
    INSERT INTO ORDERDATE VALUES (2009 * 10000 + 1200 + I);
    INSERT INTO ORDERDATE VALUES (2010 * 10000 + 100 + I);
    INSERT INTO ORDERDATE VALUES (2010 * 10000 + 300 + I);
    INSERT INTO ORDERDATE VALUES (2010 * 10000 + 500 + I);
    INSERT INTO ORDERDATE VALUES (2010 * 10000 + 700 + I);
    INSERT INTO ORDERDATE VALUES (2010 * 10000 + 800 + I);
    INSERT INTO ORDERDATE VALUES (2010 * 10000 + 1000 + I);
    INSERT INTO ORDERDATE VALUES (2010 * 10000 + 1200 + I);
  END LOOP;
  FOR I IN 1..30 LOOP
    INSERT INTO ORDERDATE VALUES (2007 * 10000 + 400 + I);
    INSERT INTO ORDERDATE VALUES (2007 * 10000 + 600 + I);
    INSERT INTO ORDERDATE VALUES (2007 * 10000 + 900 + I);
    INSERT INTO ORDERDATE VALUES (2007 * 10000 + 1100 + I);
    INSERT INTO ORDERDATE VALUES (2008 * 10000 + 400 + I);
    INSERT INTO ORDERDATE VALUES (2008 * 10000 + 600 + I);
    INSERT INTO ORDERDATE VALUES (2008 * 10000 + 900 + I);
    INSERT INTO ORDERDATE VALUES (2008 * 10000 + 1100 + I);
    INSERT INTO ORDERDATE VALUES (2009 * 10000 + 400 + I);
    INSERT INTO ORDERDATE VALUES (2009 * 10000 + 600 + I);
    INSERT INTO ORDERDATE VALUES (2009 * 10000 + 900 + I);
    INSERT INTO ORDERDATE VALUES (2009 * 10000 + 1100 + I);
    INSERT INTO ORDERDATE VALUES (2010 * 10000 + 400 + I);
    INSERT INTO ORDERDATE VALUES (2010 * 10000 + 600 + I);
    INSERT INTO ORDERDATE VALUES (2010 * 10000 + 900 + I);
    INSERT INTO ORDERDATE VALUES (2010 * 10000 + 1100 + I);
  END LOOP;
  FOR I IN 1..28 LOOP
    INSERT INTO ORDERDATE VALUES (2007 * 10000 + 200 + I);
    INSERT INTO ORDERDATE VALUES (2008 * 10000 + 200 + I);
    INSERT INTO ORDERDATE VALUES (2009 * 10000 + 200 + I);
    INSERT INTO ORDERDATE VALUES (2010 * 10000 + 200 + I);
  END LOOP;
  COMMIT;
END;
   /
```

**Figure 17** (continued)

```
SELECT
  ENDPOINT_NUMBER,
  ENDPOINT_VALUE
FROM
  USER_TAB_HISTOGRAMS
WHERE
  TABLE_NAME = 'ORDERDATE' AND
  COLUMN_NAME = 'ORDDAT'
ORDER BY
ENDPOINT_NUMBER;

ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
              0       20070101
              1       20070526
              2       20071019
              3       20080314
              4       20080807
              5       20081231
              6       20090526
              7       20091019
              8       20100314
              9       20100807
             10       20101231
```

**Figure 18** Details retrieved from USER_TAB_HISTOGRAMS

The ORDERDATE table contains 1,460 records with the following values:

```
    ORDDAT
----------
  20070101
  20070102
  20070103
  20070104
  20070105
  20070106
  20070107
  20070108
  20070109
  20070110
...
```

Next, you can create statistics for this table including histograms with 10 buckets:

```
EXEC DBMS_STATS.GATHER_TABLE_STATS -
(USER, -
 'ORDERDATE', -
 ESTIMATE_PERCENT=>100, -
 CASCADE=>TRUE, -
 METHOD_OPT=>'FOR ALL COLUMNS SIZE 10' –
);
```

The histogram statistics divide the sorted column values into 10 distinct sets with the same number of records (1460 / 10 = 146). You can retrieve the details from USER_TAB_HISTOGRAMS, as shown in **Figure 18**.

 

```
SELECT
  *
FROM
  ORDERDATE
WHERE
  ORDDAT BETWEEN 20080601 AND 20080630;

-------------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |     9 |    54 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| ORDERDATE |     9 |    54 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

**Figure 19** Selecting all records in June 2008

This output indicates that every bucket contains records of four to five months. The first bucket consists of all dates between 01.01.2007 and 05.26.2007 (in dd.mm.yyyy format), the second bucket contains all dates between 05.26.2007 and 10.19.2007, and so on. Based on this information, the CBO should be aware of the data distribution and the number of estimated rows should be close to the real data proportion. You can write a test statement and select all records of June 2008 (see **Figure 19**); the expected cardinality is 30.

The CBO only expects nine rows from this selection, which is surprising because this is three times less than the reality although the dates are perfectly distributed (the table includes exactly one record for each calendar day). The cardinality estimation is incorrect because of the NUMBER data type. Because the CBO isn't aware that values like 20080683 or 20080699 are not valid dates, it assumes the table contains records with this type of value. The cardinality estimation of the CBO is simply based on the relation of histogram endpoint values and the BETWEEN range:

```
"Estimated Rows" =
<rows_per_bucket> * <between_range> /
   <bucket_range> =
(1460 / 10) * (20080630 – 20080601) /
   (20080807 – 20080314) =
146 * 29 / 493 =
9
```

The CBO calculates a high cardinality in other constellations for the same reason. The SELECT statement shown in **Figure 20** (on the next page) should return only two records (06.30.2008 and 07.01.2008), but the CBO assumes a cardinality of 21.

This estimation is a result of the following formula, which is similar to the preceding calculation:

```
"Estimated Rows" =
<rows_per_bucket> * <between_range> /
   <bucket_range> =
(1460 / 10) * (20080701 – 20080630) /
   (20080807 – 20080314) =
146 * 71 / 493 =
21
```

```
SELECT
  *
FROM
  ORDERDATE
WHERE
ORDDAT BETWEEN 20080630 AND 20080701;

-------------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |    21 |   126 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| ORDERDATE |    21 |   126 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

**Figure 20** SELECT statement that should retrieve two records

Ranges and buckets that contain a year or a decade change can exacerbate the problem. The following two selections return 31 records and the date range is almost identical. The rows the CBO estimated are different (1 vs. 140) because the CBO considers values like 20095678 or 20099999 as valid (see **Figure 21**).

Erroneous cardinality estimations don't matter for single table accesses, but they can critically affect the join order in multitable joins. Experience has shown that an incorrect cardinality estimation causes a significant number of BI and Bank Analyzer performance problems (due to many reasons). Therefore, the estimated cardinality must be close to the real cardinality.

Unfortunately, to avoid cardinality problems in this case, you cannot easily change the data type of the column from NUMBER to TIMESTAMP or DATE. Furthermore, it is often difficult to recognize this problem because of the complexity of BI and Bank Analyzer selections. Specifying hints, using literals, and changing ABAP coding are usually not options in these environments. Instead, consider the following solutions:

- **Create histograms with a higher bucket count:** By default, SAP creates histograms with 75 buckets. Starting with BRCONNECT 7.00, you can now configure a different number of buckets using the -b option or the STATS_BUCKET_COUNT BRCONNECT parameter. The maximum bucket number is 254.

  Be aware that a higher number of buckets requires more storage space in the Oracle data dictionary. Therefore, you should only increase the bucket count for specific tables that suffer from the described problems.

- **Adapt CBO statistics:** To make the optimal access more attractive, adapt the CBO statistics as described in SAP Note 724545. Which changes are necessary to the statistics depend on the individual case.

## The CHAR vs. VARCHAR2 problem

The CHAR vs. VARCHAR2 problem causes very serious overall performance problems for SAP customers. In fact, it is currently the most critical Oracle performance issue in SAP environments. The root cause of the problem is harmless: the SAP database interface can use the CHAR data type internally, while Oracle uses the slightly different VARCHAR2 data type (with a variable length). This deviation should not affect the CBO cost calculation, but sometimes it does. The example in **Figure 22** (on page 26) simulates the problem.

 

```
SELECT
  *
FROM
  ORDERDATE
WHERE
  ORDDAT BETWEEN 20091201 AND 20091231;

-------------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |     1 |     6 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| ORDERDATE |     1 |     6 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------

SELECT
  *
FROM
  ORDERDATE
WHERE
  ORDDAT BETWEEN 20091202 AND 20100101;

-------------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |   140 |   840 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| ORDERDATE |   140 |   840 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

**Figure 21** Datestamp problems

The Orders table contains 30,000 order numbers with three positions each. Additionally, the table has one index on the MANDT column (which is not selective) and one index on the very selective BELNR column.

You can perform a selection with the conditions MANDT = '100' and BELNR = '22222' using a CHAR data type for the MANDT column in the SQL statement. See **Figure 23** on the next page. You would expect that the CBO would use index ORDERS_2 on BELNR and that the number of estimated rows is three (because all BELNRs contain three positions).

Looking into V$SQL_PLAN to retrieve the execution plan, the CBO surprisingly decides to use the unselective MANDT index ORDERS_1. Both estimated costs and estimated rows are 1. See **Figure 24** on page 27.

When executing the same selection with variable A0 defined as the VARCHAR2 rather than the CHAR data type, the CBO behaves as expected (see **Figure 25** on page 27).

Why does the CBO react incorrectly when converting a CHAR to VARCHAR2? In fact, this is Oracle bug 4752814. This bug is SAP-specific because the SAP database interface internally uses CHAR data types, so only a few other Oracle customers experience this problem. The good news

```
CREATE TABLE ORDERS (MANDT VARCHAR2(3), BELNR VARCHAR2(10), POSITION
VARCHAR2(10));

BEGIN
  FOR I IN 1..30000 LOOP
    INSERT INTO ORDERS VALUES('100',I, 1);
    INSERT INTO ORDERS VALUES('100',I, 2);
    INSERT INTO ORDERS VALUES('100',I, 3);
  END LOOP;
  COMMIT;
END;
/

EXEC DBMS_STATS.GATHER_TABLE_STATS(USER, 'ORDERS', ESTIMATE_PERCENT=>100, -
  CASCADE=>TRUE, METHOD_OPT=>'FOR ALL COLUMNS SIZE 1');

CREATE INDEX ORDERS_1 ON ORDERS (MANDT);
CREATE INDEX ORDERS_2 ON ORDERS (BELNR);
```

**Figure 22** Orders table with 30,000 order numbers

```
VAR A0 CHAR(3);
VAR A1 VARCHAR2(10);

EXEC :A0 := '100'; :A1 := '22222';

SELECT /* TEST1_MARKER */ * FROM ORDERS WHERE MANDT = :A0 AND BELNR = :A1;
```

**Figure 23** Selecting records using a CHAR data type

is that this bug applies only to Oracle 9.2.0.7 and 10.2.0.2 (it also applies to 10.2.0.3, which is not supported in SAP environments) and SAP kernels ≥ 6.10 (using the OCI8 call interface). Oracle 9.2.0.8 and 10.2.0.4 fix this problem. What can be done apart from upgrading to a later patchset? The best workaround is to set the Oracle parameter _OPTIM_PEEK_USER_BINDS to FALSE. The bug only occurs in combination with bind value peeking and so deactivating this feature resolves the issue. For more information, see the sidebar on page 28.

# Conclusion

Many SAP design decisions were made before the introduction of the CBO. SAP could not therefore take CBO factors into account in its software design. Some performance problems can be traced to SAP design details such as default values and data types. Because they are central components of the SAP system, you need to use workarounds and alternative optimizations to solve these problems.

This article highlighted the most important

 

```
COLUMN ACTION FORMAT A50
SELECT
  LPAD(' ', DEPTH) || OPERATION || ' ' || OPTIONS ||
    DECODE(OBJECT_NAME, NULL, NULL, ' (' || OBJECT_NAME || ')') ACTION,
  COST,
  CARDINALITY
FROM
  V$SQL_PLAN SP, V$SQL S
WHERE
  S.SQL_TEXT LIKE 'S%TEST1_MARKER%' AND
  S.ADDRESS = SP.ADDRESS AND
  S.HASH_VALUE = SP.HASH_VALUE;

ACTION                                                 COST CARDINALITY
-------------------------------------------------- ---------- -----------
SELECT STATEMENT                                            1           1
 TABLE ACCESS BY INDEX ROWID (ORDERS)                       0           1
  INDEX RANGE SCAN (ORDERS_1)                               0           1
```

**Figure 24** Costs and rows incorrectly estimated as 1

```
VAR A0 VARCHAR2(3);
EXEC :A0 := '100';

SELECT /* TEST2_MARKER */ * FROM ORDERS WHERE MANDT = :A0 AND BELNR = :A1;

COLUMN ACTION FORMAT A50
SELECT
  LPAD(' ', DEPTH) || OPERATION || ' ' || OPTIONS ||
    DECODE(OBJECT_NAME, NULL, NULL, ' (' || OBJECT_NAME || ')') ACTION,
  COST,
  CARDINALITY
FROM
  V$SQL_PLAN SP, V$SQL S
WHERE
  S.SQL_TEXT LIKE 'S%TEST2_MARKER%' AND
  S.ADDRESS = SP.ADDRESS AND
  S.HASH_VALUE = SP.HASH_VALUE;

ACTION                                                 COST CARDINALITY
-------------------------------------------------- ---------- -----------
SELECT STATEMENT                                            1           3
 TABLE ACCESS BY INDEX ROWID (ORDERS)                       0           3
  INDEX RANGE SCAN (ORDERS_2)                               0           3
```

**Figure 25** Correct cost estimation after defining the A0 variable as VARCHAR2

## Bind value peeking

If bind value peeking is active, the CBO can examine the concrete values behind the bind variables that pass to the database during parsing. Based on these values, the CBO can sometimes select a better access path instead of using only anonymous bind variables. This is particularly true if you are also using histogram statistics.

In theory, bind value peeking can improve performance, but in the past, many users found a significant number of problems and bugs when using this feature. As a consequence, you should deactivate the feature by setting the Oracle parameter _OPTIM_PEEK_USER_BINDS = FALSE.

Bind value peeking often causes different execution plans in V$SQL_PLAN and the standard explain functionality. For example, the normal explain functionality would have always shown an access via ORDERS_2 as shown in **Figure 25**.

CBO problems caused by the SAP design details, and provided possible solutions to these problems. It also demonstrated why CBO assumptions sometimes differ from the real data, which can help you select the appropriate optimization for a specific context.

Keep in mind that many unexpected CBO decisions are influenced by Oracle limitations, CBO features, and bugs; you should therefore understand these factors through further study.

 