

SQL Profiles

Christian Antognini
Trivadis AG, Zürich (CH)
22 June 2006

Abstract

The cost-based optimizer, because of a lack of information or internal flaws, is not always able to produce accurate estimates. In such situations poorly performing execution plans could be generated. Usually, to work around such problems, hints are added to the SQL statements. As of 10g there is a new, and in some ways better, possibility: SQL profiles. While explaining what SQL profiles are and how to create and use them, this presentation provides internals regarding their storage.

Table of Contents

1	Automatic Tuning Optimizer	3
2	Why SQL Profiles?	10
3	Management and Use	12
4	Requirements	21
5	Undocumented Stuff	22
6	Sample Cases	30
7	SQL Profiles – Core messages...	32

Who Am I



- Senior consultant and trainer at Trivadis AG in Zurich, Switzerland
 - christian.antognini@trivadis.com
 - www.trivadis.com
- Focus: get the most out of Oracle
 - Logical and physical database design
 - Query optimizer
 - Application performance management and tuning
 - Integration of databases with Java applications
- Proud member of
 - Trivadis Performance Team
 - OakTable Network



Since 1995, Christian Antognini has been focusing on understanding how the Oracle database engine works. His main interests range from logical and physical database design, the integration of databases with Java applications, to the cost-based optimizer and basically everything else related to performance management and tuning. He is currently working as a senior consultant and trainer at Trivadis AG (<http://www.trivadis.com>) in Zürich, Switzerland. If he is not helping one of his customers to get the most out of Oracle, he is somewhere lecturing on optimization or new Oracle database features for developers. He is member of the Trivadis Performance Team and of the OakTable Network.

1 Automatic Tuning Optimizer

Automatic Tuning Optimizer (1)



- The cost-based optimizer has been strongly enhanced in the last few years



- Presently, if correctly configured, it provides good execution plans for most SQL statements
- Also for this reason the rule-based optimizer, as of 10g, is officially no longer supported (finally!!!)

For more information about the configuration of the cost-based optimizer have a look at my paper "CBO - A Configuration Roadmap"

(http://www.trivadis.com/Images/CBOConfigurationRoadmap_tcm17-14317.pdf).

Automatic Tuning Optimizer (2)



- Note that sub-optimal execution plans can be generated even if
 - The cost-based optimizer is correctly configured
 - Object statistics correctly describe the data
 - Histograms are available for skewed data
 - System statistics correctly describe the system where Oracle runs
- Why? Because the cost-based optimizer is based on a mathematical model and, as with any model, it is not perfect, it is only a model!
- In addition information provided by object statistics is limited, e.g. they do not provide information about correlated columns
- Therefore, in some situations, SQL tuning is still needed...

The script **correlated_columns.sql** shows a case, because of correlated columns, where the cost-based optimizer is completely wrong. In my honest opinion the inability to create a single histogram on multiple columns is a major flaw in the current cost-based optimizer. I hope that Oracle will fix it in the near future (other databases have supported them for a long time).

Automatic Tuning Optimizer (3)



- As of 10g SQL tuning can be delegated to an extension of the cost-based optimizer called *automatic tuning optimizer*
- It might seem strange to delegate this task to the same component that is not able to find an optimal execution plan in the first place; in reality the two situations are very different...
- In normal circumstances the cost-based optimizer is constrained to generate an execution plan very quickly
- Otherwise much more time can be given to the automatic tuning optimizer to carry out the optimal execution plan and, in addition, it is also able to perform what-if analysis and make strong utilization of dynamic sampling techniques to verify its estimations

Integrating such a tuning tool in the optimizer has been a neat move. Other commercial tools that do something similar are missing important information. In fact they can only guess why the optimizer cannot provide an optimal execution plan. In addition for them it is difficult, sometimes impossible, to exactly know what the optimizer will do with, for example, new access structures in place.

Automatic Tuning Optimizer (4)



- The automatic tuning optimizer is exposed through the SQL Tuning Advisor
- As usual the core interface is available through a PL/SQL package, DBMS_SQLTUNE, while a graphical user interface is integrated in Enterprise Manager
- The following advice is provided
 - Gather missing or stale statistics
 - Create new indexes
 - Restructure SQL statement
 - SQL profiles

The **input** statements can be specified in the following forms:

- SQL statement text.
- Reference to SQL statement in the shared pool.
- SQL tuning set, which is a set of SQL statements and the associated execution context and statistics.
- Range of Automatic Workload Repository snapshots.

The **output** is available through views, the package DBMS_SQLTUNE and Enterprise Manager.

Note that the SQL Tuning Advisor is not always able to provide good advice. An example is given by the script **count.sql** where the automatic tuning optimizer is not able to provide any advice for a simple query. This does not mean it is bad... It simply means that it is not able to find a solution for all bad-performing SQL statements.

SQL Tuning Advisor – Sample Analysis



```
SQL> CREATE TABLE t (id CONSTRAINT t_pk PRIMARY KEY, pad) AS
  2  SELECT rownum, lpad('*',4000, '*')
  3  FROM all_objects
  4  WHERE rownum <= 10000;

SQL> VARIABLE tn VARCHAR2(30)

SQL> DECLARE
  2  l_sqltext CLOB := 'SELECT COUNT(*) FROM t WHERE id+42 = 126';
  3  BEGIN
  4  :tn := dbms_sqltune.create_tuning_task(sql_text=>l_sqltext);
  5  dbms_sqltune.execute_tuning_task(:tn);
  6  END;
  7  /

SQL> SELECT dbms_sqltune.report_tuning_task(:tn) FROM dual;
```

The script **fbi_or_restructure.sql** shows the automatic tuning optimizer advising to either create a function-based index or to restructure the SQL statement. In addition it also advised to gather missing object statistics.

SQL Tuning Advisor – Sample Output



Type	Findings	Recommendations	Rationale
Statistics	Table "OPS\$CHA"."T" was not analyzed.	Consider collecting optimizer statistics for this table.	The optimizer requires up-to-date statistics for the table in order to select a good execution plan.
Index	The execution plan of this statement can be improved by creating one or more indices.	Consider running the Access Advisor to improve the physical schema design or creating the recommended index. OPS\$CHA.T("ID"+42)	Creating the recommended indices significantly improves the execution plan of this statement. However, it might be preferable to run "Access Advisor" using a representative SQL workload as opposed to a single statement. This will allow to get comprehensive index recommendations which takes into account index maintenance overhead and additional space consumption.
Restructure SQL	<u>The predicate "T"."ID"+42=126 used at line ID 2 of the execution plan contains an expression on indexed column "ID". This expression prevents the optimizer from selecting indices on table "OPS\$CHA"."T".</u>	Rewrite the predicate into an equivalent form to take advantage of indices. Alternatively, create a function-based index on the expression.	The optimizer is unable to use an index if the predicate is an inequality condition or if there is an expression or an implicit data type conversion on the indexed column.

The report can be directly generated through data dictionary views as well.

```
SQL> SELECT a.command AS type,
2         f.message AS findings,
3         a.message AS recommendations,
4         t.message AS rationale
5 FROM dba_advisor_actions a,
6      dba_advisor_recommendations r,
7      dba_advisor_findings f,
8      dba_advisor_rationale t
9 WHERE a.task_id = 7985
10 AND a.task_id = r.task_id
11 AND a.rec_id = r.rec_id
12 AND a.task_id = t.task_id
13 AND a.rec_id = t.rec_id
14 AND f.task_id = r.task_id
15 AND f.finding_id = r.finding_id;
```

TYPE	FINDINGS	RECOMMENDATIONS	RATIONALE
GATHER TABLE STATISTICS	Table "OPSS\$CHA"."T" was not analyzed.	Consider collecting optimizer statistics for this table.	The optimizer requires up-to-date statistics for the table in order to select a good execution plan.
CREATE INDEX	The execution plan of this statement can be improved by creating one or more indexes.	Consider running the Access Advisor to improve the physical schema design or creating the recommended index.	Creating the recommended indexes significantly improves the execution plan of this statement. However, it might be preferable to run "Access Advisor" using a representative SQL workload as opposed to a single statement. This will allow to get comprehensive index recommendations which takes into account index maintenance overhead and additional space consumption.
REWRITE QUERY	The predicate "T"."ID">42=126 used at line 2 of the execution plan contains an expression on indexed column "ID". This expression prevents the optimizer from selecting indices on table "OPSS\$CHA"."T".	Rewrite the predicate into an equivalent form to take advantage of indices. Alternatively, create a function-based index on the expression.	The optimizer is unable to use an index if the predicate is an inequality condition or if there is an expression or an implicit data type conversion on the indexed column.

2 Why SQL Profiles?

Why SQL Profiles? (1)



- To fix sub-optimal execution plans the following techniques are available
 - Creating, modifying or dropping access structures
 - Adding hints
 - Adding or editing stored outlines
 - Rewriting SQL statement
 - Tweaking INIT.ORA parameters
 - Gathering or refreshing, possibly even faking, object or system statistics
 - Dynamic sampling



Except for stored outlines all these techniques either modify an already available component (database object or application module) or could have an impact on many SQL statements running in the same database.

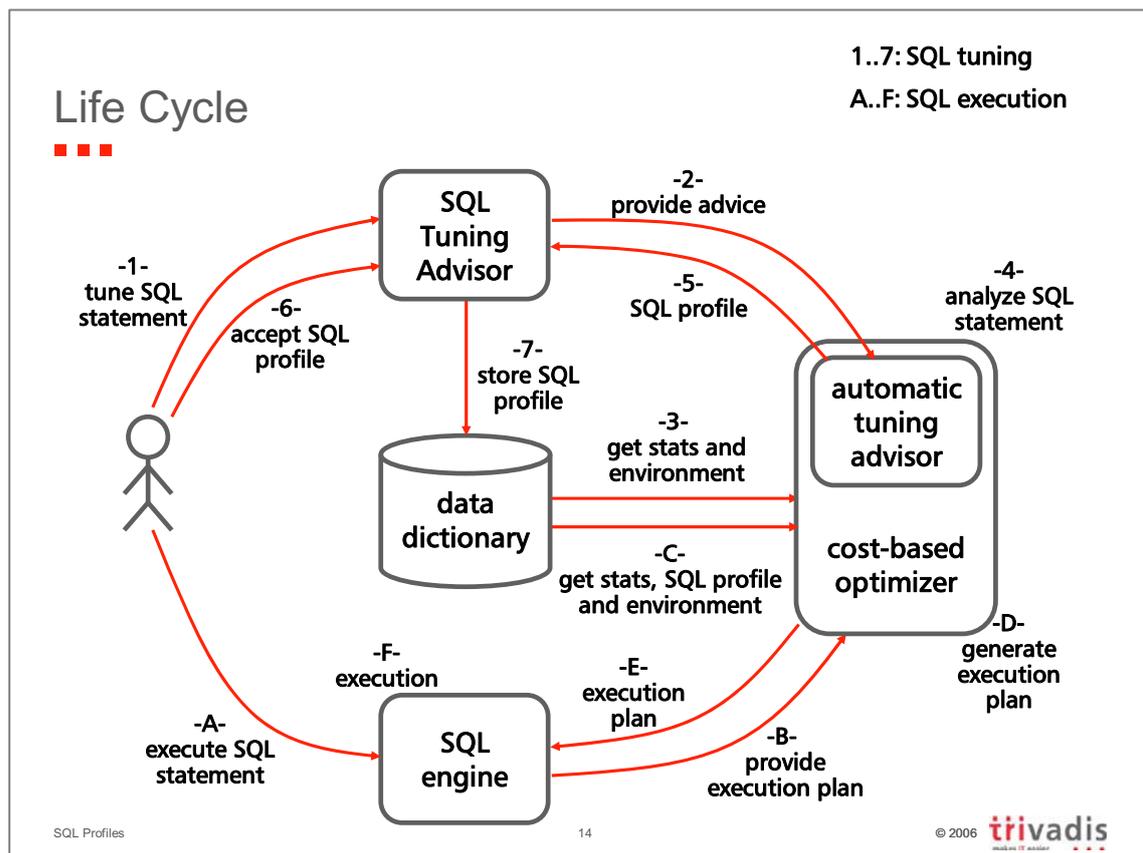
Why SQL Profiles? (2)



- Some SQL tuning techniques are not always applicable
 - Application code is not available or cannot be modified
 - INIT.ORA parameters, object statistics and access structures cannot be modified
- SQL profiles, without any modification to the application code, the referenced objects and their statistics, provide the cost-based optimizer adjustments for
 - INIT.ORA parameters
 - Estimated cardinalities
 - Object statistics
- The validity of a SQL profile is limited to a single SQL statement

It is a good thing that SQL profiles provide a solution at statement level. In fact, with a correctly configured optimizer, only a few SQL statements have to be tuned and, therefore, it is a must to reduce the impact of the tuning measure to a given SQL statement.

3 Management and Use



SQL Tuning

1. The user provides the poorly performing SQL statement to the SQL Tuning Advisor.
2. The SQL Tuning Advisor asks the automatic tuning advisor to tune the SQL statement.
3. The optimizer gets the statistics related to the objects referenced by the SQL statements and the INIT.ORA parameters that set up the execution environment.
4. The SQL statement is analyzed. During this phase the automatic tuning optimizer performs what-if analysis and partially executes the SQL statement to confirm its guesses.
5. The automatic tuning advisor returns the SQL profile to the SQL Tuning Advisor.
6. The user accepts the SQL profile.
7. The SQL profile is stored in the data dictionary.

SQL Execution

- A. The user executes a SQL statement.
- B. The SQL engine asks the optimizer to provide an execution plan.
- C. The optimizer gets the statistics related to the objects referenced by the SQL statement, the SQL profile and the INIT.ORA parameters that set up the execution environment.
- D. The optimizer analyzes the SQL statement and generates the execution plan.
- E. The execution plan is given to the SQL engine.
- F. The SQL engine executes the SQL statement.

If a SQL statement has at the same time a SQL profile and a stored outline, instead of using the SQL profile the cost-based optimizer gives precedence to the stored outline. Of course to do so the usage of stored outlines has to be active, i.e. `USE_STORED_OUTLINES` has to be set to `TRUE` or to the category containing the stored outline.

Categories



- SQL profiles are grouped
 - Groups are called *categories*
 - The initial category is specified when they are accepted
 - They can be moved in another category at any time
- The active category is set through the INIT.ORA parameter SQLTUNE_CATEGORY
 - It can be changed with ALTER SYSTEM or ALTER SESSION

```
ALTER SESSION SET sqltune_category = TEST
```

- The default category is DEFAULT

SQL Text Normalization



- To map a SQL profile to a SQL statement a hash value (a.k.a. signature) of the SQL statement is used
- Before computing the hash value the SQL statement is normalized, i.e. blank spaces are removed and non-literal strings are uppercased
- If the SQL statement contains literals that are likely to change, it is probable that the hash value changes as well, therefore the SQL profile may be useless
- As of Oracle Database 10g Release 2 it is possible to allow substitute literals with bind variables during the normalization phase. The substitution is not allowed if other bind variables are present in the SQL statement

Important notes:

- Two queries with the same text have the same hash value, even if they reference objects in different schemas.
- The computation of the hash value is not case-insensitive queries aware (e.g. when `NLS_SORT=BINARY_CI` and `NLS_COMP=ANSI`).
- It is not possible to specify literal substitution if Enterprise Manager is used to create the SQL profile.

The function **SQLTEXT_TO_SIGNATURE** is used to compute the hash value of a given SQL statement.

```
FUNCTION SQLTEXT_TO_SIGNATURE RETURNS NUMBER
Argument Name                Type                In/Out Default?
-----
SQL_TEXT                     CLOB                IN
FORCE_MATCH                  BINARY_INTEGER     IN
```

The parameter **FORCE_MATCH**, which specifies if literals have to be substituted, only exists in Oracle Database 10g Release 2.

The script **sqltext_to_signature.sql** shows the impact of the parameter **FORCE_MATCH**. Here an excerpt of its output:

- **FORCE_MATCH = 0**

```
SQL_TEXT                                SIGNATURE
-----
SELECT * FROM dual WHERE dummy = 'A'    17152174882085893964
select * from dual where dummy='A'      17152174882085893964
SELECT * FROM dual WHERE dummy = 'a'    17650142110400474019
SELECT * FROM dual WHERE dummy = 'B'    4901325341701120494
```

- **FORCE_MATCH = 1**

```
SQL_TEXT                                SIGNATURE
-----
SELECT * FROM dual WHERE dummy = 'A'    10668153635715970930
select * from dual where dummy='A'      10668153635715970930
SELECT * FROM dual WHERE dummy = 'a'    10668153635715970930
SELECT * FROM dual WHERE dummy = 'B'    10668153635715970930
```

DBMS_SQLTUNE



- To manage SQL profiles the package DBMS_SQLTUNE contains the following procedures/functions
 - Basics
 - ACCEPT_SQL_PROFILE
 - ALTER_SQL_PROFILE
 - DROP_SQL_PROFILE
 - Copy SQL profiles into/from data dictionary (10g Release 2 only)
 - CREATE_STGTAB_SQLPROF
 - PACK_STGTAB_SQLPROF
 - REMAP_STGTAB_SQLPROF
 - UNPACK_STGTAB_SQLPROF

The procedure **ACCEPT_SQL_PROFILE** is used to accept a SQL profile advised by the SQL Tuning Advisor.

PROCEDURE ACCEPT_SQL_PROFILE

Argument Name	Type	In/Out	Default?
TASK_NAME	VARCHAR2	IN	
OBJECT_ID	NUMBER	IN	DEFAULT
NAME	VARCHAR2	IN	DEFAULT
DESCRIPTION	VARCHAR2	IN	DEFAULT
CATEGORY	VARCHAR2	IN	DEFAULT
TASK_OWNER	VARCHAR2	IN	DEFAULT
REPLACE	BOOLEAN	IN	DEFAULT
FORCE_MATCH	BOOLEAN	IN	DEFAULT

The parameters REPLACE and FORCE_MATCH (which specifies whether literals have to be substituted) only exist in Oracle Database 10g Release 2.

A function with the same signature and the same name as the procedure exists. The only difference is that the function returns the name of the SQL profile. This is useful if the name is not specified as an input parameter, i.e. if the system has to generate it.

The procedure **ALTER_SQL_PROFILE** is used to alter the status, the name, the description and the category of a SQL profile.

PROCEDURE ALTER_SQL_PROFILE

Argument Name	Type	In/Out	Default?
NAME	VARCHAR2	IN	
ATTRIBUTE_NAME	VARCHAR2	IN	
VALUE	VARCHAR2	IN	

Through the parameter ATTRIBUTE_NAME it is possible to specify the following attributes:

- NAME
- DESCRIPTION
- CATEGORY
- STATUS (it can be set to either DISABLED or ENABLED)

The procedure **DROP_SQL_PROFILE** is used to drop a SQL profile.

PROCEDURE DROP_SQL_PROFILE

Argument Name	Type	In/Out	Default?
NAME	VARCHAR2	IN	
IGNORE	BOOLEAN	IN	DEFAULT

The procedure **CREATE_STGTAB_SQLPROF** is used to create a staging table to import/export SQL profiles.

PROCEDURE CREATE_STGTAB_SQLPROF

Argument Name	Type	In/Out	Default?
TABLE_NAME	VARCHAR2	IN	
SCHEMA_NAME	VARCHAR2	IN	DEFAULT
TABLESPACE_NAME	VARCHAR2	IN	DEFAULT

The procedure **PACK_STGTAB_SQLPROF** is used to copy a SQL profile from the data dictionary to the staging table.

```
PROCEDURE PACK_STGTAB_SQLPROF
Argument Name                Type                In/Out  Default?
-----
PROFILE_NAME                 VARCHAR2           IN      DEFAULT
PROFILE_CATEGORY            VARCHAR2           IN      DEFAULT
STAGING_TABLE_NAME          VARCHAR2           IN
STAGING_SCHEMA_OWNER        VARCHAR2           IN      DEFAULT
```

The procedure **REMAP_STGTAB_SQLPROF** is used to change the name and/or the category of a SQL profile stored in the staging table.

```
PROCEDURE REMAP_STGTAB_SQLPROF
Argument Name                Type                In/Out  Default?
-----
OLD_PROFILE_NAME            VARCHAR2           IN
NEW_PROFILE_NAME            VARCHAR2           IN      DEFAULT
NEW_PROFILE_CATEGORY        VARCHAR2           IN      DEFAULT
STAGING_TABLE_NAME          VARCHAR2           IN
STAGING_SCHEMA_OWNER        VARCHAR2           IN      DEFAULT
```

The procedure **UNPACK_STGTAB_SQLPROF** is used to copy a SQL profile from the staging table to the data dictionary.

```
PROCEDURE UNPACK_STGTAB_SQLPROF
Argument Name                Type                In/Out  Default?
-----
PROFILE_NAME                 VARCHAR2           IN      DEFAULT
PROFILE_CATEGORY            VARCHAR2           IN      DEFAULT
REPLACE                      BOOLEAN           IN
STAGING_TABLE_NAME          VARCHAR2           IN
STAGING_SCHEMA_OWNER        VARCHAR2           IN      DEFAULT
```

DBMS_SQLTUNE – Examples



- Accepting a SQL profile

```
dbms_sqltune.accept_sql_profile(  
  task_name   => 'TASK_8416',  
  name        => 'first_rows.sql',  
  category    => 'DEFAULT',  
  force_match => TRUE  
)
```

- Modifying the category of a SQL profile

```
dbms_sqltune.alter_sql_profile(  
  name           => 'first_rows.sql',  
  attribute_name => 'CATEGORY',  
  value          => 'TEST'  
)
```

- Dropping a SQL profile

```
dbms_sqltune.drop_sql_profile(name => 'first_rows.sql')
```

4 Requirements

Requirements

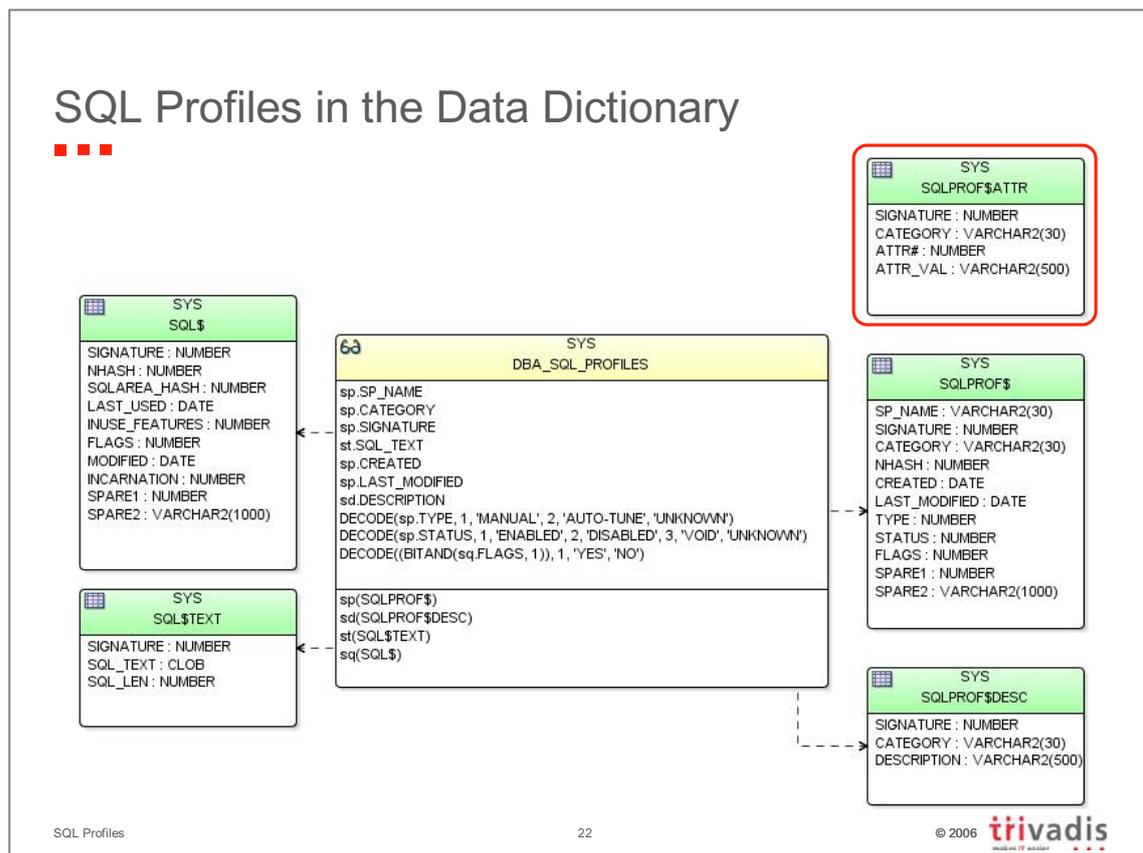


- SQL Tuning Advisor
 - System privilege ADVISOR is needed
- SQL profiles
 - To create, drop and modify a SQL profile the following system privileges are required (object privileges do not exist)
 - CREATE ANY SQL PROFILE
 - DROP ANY SQL PROFILE
 - ALTER ANY SQL PROFILE
- For both
 - Tuning Pack must be licensed
 - The Diagnostic Pack is a prerequisite to the Tuning Pack



Detailed information about Oracle licensing agreement is available in manual *Oracle Database Licensing Information* (http://download-uk.oracle.com/docs/cd/B19306_01/license.102/b14199.pdf).

5 Undocumented Stuff



A SQL profile stores a set of hints, which represents the adjustments, in the table `SQLPROF$ATTR`. Some of them are documented, some of them are undocumented and only available in 10g, i.e. they have probably been implemented for this purpose. All of them are regular hints and, therefore, can be directly added to a SQL statement (the script `hints.sql` shows some examples).

Note that the owner of the objects referenced by the SQL statement is not stored anywhere. This means that a single SQL profile could be used for two tables with the same name but located in a different schema!

Hints Used in SQL Profiles (1)



- Setting the optimizer mode

```
ALL_ROWS
```

- Disabling the hints present in the SQL statement

```
IGNORE_OPTIM_EMBEDDED_HINTS
```

- Setting OPTIMIZER_FEATURES_ENABLE to its default value, i.e. activate all the available features

```
OPTIMIZER_FEATURES_ENABLE (default)
```

The script **all_rows.sql** shows the automatic tuning optimizer advising to switch from RULE (provided with a hint) to ALL_ROWS. In it also interesting to note that to disable the hints specified at query level the hint IGNORE_OPTIM_EMBEDDED_HINTS is added to the SQL profile.

Hints Used in SQL Profiles (2)



- Adjusting the number of rows returned from a table
 - E.g. 10 times as many rows as expected are returned from table C

```
OPT_ESTIMATE(@SEL$1, TABLE, C@SEL$1, SCALE_ROWS=10)
```

- Adjusting the number of rows returned through an index scan
 - E.g. 10 times fewer rows as expected are returned from table C through index CUSTOMER_PK

```
OPT_ESTIMATE(@SEL$1, INDEX_SCAN, C@SEL$1, CUSTOMERS_PK, SCALE_ROWS=.1)
```

- Adjusting the number of rows returned from a join
 - E.g. 2.8 times as many rows as expected are returned when C1 and C2 are joined

```
OPT_ESTIMATE(@SEL$1, JOIN, (C1@SEL$1, C2@SEL$1), SCALE_ROWS=2.8)
```

The script **opt_estimate.sql** shows the automatic tuning optimizer advising OPT_ESTIMATE hints to adjust the cost-based optimizer estimations.

Hints Used in SQL Profiles (3)



- Providing missing or inaccurate statistics

- For a table

```
TABLE_STATS ("SH"."AMOUNT_SOLD_MV", scale, blocks=7 rows=1460)
```

- For columns in a detailed or partial way

```
COLUMN_STATS ("SH"."AMOUNT_SOLD_MV", "TIME_ID", scale, length=7  
distinct=1460 nulls=0 min=2450815 max=2452275)
```

```
COLUMN_STATS ("SH"."AMOUNT_SOLD_MV", "SUMSLD", scale, length=5)
```

- For an index

```
INDEX_STATS ("SH"."AMOUNT_SOLD_MV", "I_SNAP$_AMOUNT_SOLD_MV",  
scale, blocks=8 index_rows=1460)
```

In some situations, when statistics are missing or inaccurate, the automatic query optimizer suggests hints containing all the relevant statistics. The script **object_stats.sql** provides such an example.

Manually Create a SQL Profile



- It is possible to manually create SQL profiles through an undocumented procedure (possibly unsupported?)

```
dbms_sqtune.import_sql_profile(  
  name      => 'import_sql_profile.sql',  
  category => 'DEFAULT',  
  sql_text => 'SELECT * FROM t ORDER BY id',  
  profile  => sqlprof_attr('first_rows(42)')  
);
```

- This procedure is used from the SQL Tuning Advisor to create a profile when a tuning task is accepted

The procedure **IMPORT_SQL_PROFILE** is used to store a SQL profile in the data dictionary.

```
PROCEDURE IMPORT_SQL_PROFILE
```

Argument Name	Type	In/Out	Default?
SQL_TEXT	CLOB	IN	
PROFILE	SQLPROF_ATTR	IN	
NAME	VARCHAR2	IN	DEFAULT
DESCRIPTION	VARCHAR2	IN	DEFAULT
CATEGORY	VARCHAR2	IN	DEFAULT
VALIDATE	BOOLEAN	IN	DEFAULT
REPLACE	BOOLEAN	IN	DEFAULT
FORCE_MATCH	BOOLEAN	IN	DEFAULT

The parameters **FORCE_MATCH** (which specifies whether literals have to be substituted) and **REPLACE** only exist in Oracle Database 10g Release 2.

The type **SQLPROF_ATTR** is defined as **VARRAY(2000) OF VARCHAR2(500)**.

The script **import_sql_profile.sql** shows how to manually create a SQL profile.

Parameter `_STN_TRACE` (1)



- With the undocumented parameter `_STN_TRACE` it is possible to trace the analysis performed by the automatic tuning optimizer
- Range of values: 0 to $2^{31}-1$
- Default value is 0 (tracing disabled)
- Values are handled as a bit mask
- Most of the first 10 bits enable tracing of a specific piece of information
- The output is written in a trace file and, for specific values, to the table `ORA_DEBUG_TABLE`

By default the table `ORA_DEBUG_TABLE` does not exist. It can be created with the following DDL statement:

```
CREATE TABLE ora_debug_table (  
  time DATE,  
  txt0 VARCHAR2(4000), txt1 VARCHAR2(4000),  
  txt2 VARCHAR2(4000), txt3 VARCHAR2(4000),  
  txt4 VARCHAR2(4000),  
  num0 NUMBER, num1 NUMBER, num2 NUMBER,  
  num3 NUMBER, num4 NUMBER, num5 NUMBER,  
  num6 NUMBER, num7 NUMBER, num8 NUMBER,  
  num9 NUMBER  
);
```

Parameter _STN_TRACE (2)



Value	Description
1	No output has been observed
2	No output has been observed
4	SQL text, findings, original execution plan and, optionally, INIT.ORA parameters
8	Same as 4 and, in addition, adjusted execution plan
16	SQL text and, optionally, INIT.ORA parameters
32	Almost same as 16
64	Verification of statistics
128	Almost same as 64
256	No output has been observed
512	List of recursive queries with execution statistics
1024	Same as 512 but output written into ORA_DEBUG_TABLE

6 Sample Cases

Case 1 – Change Optimizer Mode



- Situation
 - The database is running with `OPTIMIZER_MODE=ALL_ROWS`
 - The configuration is good for most SQL statements
 - A part of the application allows the end users to browse data, most of the time only a few rows are fetched from the cursors
- Problem
 - With `ALL_ROWS` the cost-based optimizer's goal is to optimize the retrieval of the last row
 - Since only a few rows are fetched, some sub-optimal execution plans are used
- Solution
 - Switch to `FIRST_ROWS_n`

The script **first_rows.sql** shows the automatic tuning optimizer advising to switch from `ALL_ROWS` to `FIRST_ROWS_n`. In this case it happens because the application fetches only a few rows each time the query is executed. It is also interesting to note that in this test case the application executes the query 10 times. The first time it fetches 1 row, the second time it fetches 2 rows, the third time it fetches 3 rows, ... and the tenth time 10 rows. In its advice the automatic tuning optimizer proposes the hint `FIRST_ROWS(6)`, i.e. the average number of rows that the application fetched.

Case 2 – Adjust Estimated Cardinalities

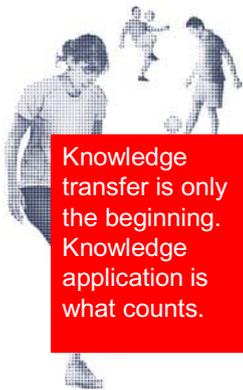


- **Situation**
 - A “simple” query, that returns a single row, takes too much time
 - The columns used for the restriction are already indexed
- **Problem**
 - Data from two columns is strongly correlated
 - Because of a lack of information the cost-based optimizer makes a wrong estimation, therefore data is accessed through a nested loop (which is not good in this case...)
- **Solution**
 - Adjust the cost-based optimizer estimates

The script **opt_estimate.sql** shows the automatic tuning optimizer advising OPT_ESTIMATE hints to adjust the cost-based optimizer estimations.

7 SQL Profiles – Core messages...

SQL Profiles – Core messages...



- Finally the optimizer is able, to some extent, to learn from its mistakes
- SQL profiles solve in a practical and efficient way common SQL tuning problems at statement level
- To use SQL profiles, the automatic tuning optimizer and the SQL Tuning Advisor, the Tuning and the Diagnostic pack have to be licensed!