

■ ■ ■ Query Optimizer 11g – What's new?



Christian Antognini
Principal Consultant

December 2, 2008

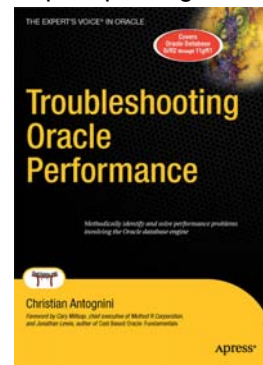
trivadis
makes IT easier. ■ ■ ■

Basel · Baden · Bern · Lausanne · Zurich · Düsseldorf · Frankfurt/M. · Freiburg i. Br. · Hamburg · Munich · Stuttgart · Vienna

Who Am I

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

<http://top.antognini.ch>



trivadis
performance team

OakTable.net

Introduction



- With every new release the query optimizer is enhanced, Oracle Database 11g isn't an exception to the rule
- This presentation provides an overview of central improvements and, for each of them, the problem in Oracle Database 10g Release 2 (10.2.0.4) and the solution in Oracle Database 11g Release 1 (11.1.0.6) are described

Agenda



- Indexing
- Optimization Techniques
- Object Statistics
- Plan Stability



Invisible Indexes (1)



- Problem
 - We want to hide an index from the query optimizer without dropping it
- Solution
 - Make the index invisible

```
ALTER INDEX i INVISIBLE
```

- Annotations
 - Hints are not honored
 - Invisible indexes are regularly maintained and can be made visible at any time

```
ALTER INDEX i VISIBLE
```

Invisible Indexes (2)



- Problem
 - We want to add a new index without making it available per default
- Solution
 - Create a new index by specifying the option INVISIBLE

```
CREATE INDEX i ON t (col1) INVISIBLE
```

- Make it available to specific sessions only

```
ALTER SESSION SET optimizer_use_invisible_indexes = TRUE
```

- Annotations
 - The default value of the initialization parameter OPTIMIZER_USE_INVISIBLE_INDEXES is FALSE

Index Support for Linguistic LIKE

■ ■ ■

- Problem
 - Linguistic sorting works fine with function-based indexes
 - Unfortunately, such indexes are not used for comparisons with LIKE
- Solution
 - It works ;-)

```
CREATE INDEX fbi ON emp (nlssort(ename,'nls_sort=binary_ci'));
ALTER SESSION SET nls_sort=binary_ci;
ALTER SESSION SET nls_comp=linguistic;
SELECT * FROM emp WHERE ename LIKE 'sco%';
```

Operation	Name
SELECT STATEMENT	
TABLE ACCESS BY INDEX ROWID	EMP
INDEX RANGE SCAN	FBI

Agenda

■ ■ ■

- Indexing
- Optimization Techniques
- Object Statistics
- Plan Stability



Full Outer Join



- Problem
 - Full outer join syntax is supported, but the query optimizer rewrites it to a UNION ALL query, so joined tables are accessed twice ☹
- Solution
 - With equi joins it works ;-)

```
SELECT *
FROM t1 FULL OUTER JOIN t2 ON t1.id = t2.id
```

Operation	Name
SELECT STATEMENT	
VIEW	VW_FOJ_0
HASH JOIN FULL OUTER	
TABLE ACCESS FULL	T1
TABLE ACCESS FULL	T2

Join-Filter Pruning



- Problem
 - Subquery pruning allows the utilization of hash joins instead of nested loops to perform partition pruning on join conditions
 - However, part of the SQL statement is executed twice
- Solution
 - Double execution is avoided thanks to join-filter pruning

```
SELECT * FROM t t1, t t2 WHERE t1.n = t2.n AND t1.id = 1
```

Operation	Name	Pstart	Pstop
HASH JOIN			
PART JOIN FILTER CREATE	:BF0000		
TABLE ACCESS BY GLOBAL INDEX ROWID	T	ROWID	ROWID
INDEX UNIQUE SCAN	T_PK		
PARTITION RANGE JOIN-FILTER		:BF0000	:BF0000
TABLE ACCESS FULL	T	:BF0000	:BF0000

Partition Pruning



- Does partition pruning work with all new types of composite partitioning?
 - Range – Range
 - List – Range
 - List – List
 - List – Hash
- Yes
 - No exceptions have been found yet ☺

Agenda



- Indexing
- Optimization Techniques
- Object Statistics
- Plan Stability



Predicates Based on Expressions



- Problem
 - Whenever a predicate is based on an expression (e.g. `trunc(col)=val`) the query optimizer is not able, based on regular object statistics, to correctly estimate the selectivity
 - As a result, sub-optimal execution plans might be generated
- Solution
 - By defining so-called *extended statistics* based on expressions used in predicates, it is possible to instruct the package `DBMS_STATS` to gather additional statistics

```
dbms_stats.create_extended_stats(ownname => user,
                                tabname  => 'T',
                                extension => '(trunc(col))')
```

- Thanks to these additional statistics the query optimizer is able to correctly estimate the selectivity of such predicates

Correlated Data



- Problem
 - With regular object statistics no information is available about the correlation of data stored in several columns
 - Whenever the query optimizer has to estimate the selectivity of a `WHERE` clause based on such correlated columns, sub-optimal execution plans are usually generated

- Solution
 - Define extended statistics based on a group of columns

```
dbms_stats.create_extended_stats(ownname => user,
                                tabname  => 'T',
                                extension => '(col1,col2)')
```

- Annotations
 - Extended statistics only works with predicates based on equality

Default Preferences for DBMS_STATS



- Problem
 - Default values for parameters can only be set database wide
 - The procedure SET_PARAM is available to do this

- Solution
 - Default values for parameters can be set at table level as well

```
dbms_stats.set_table_prefs(ownname => user,  
                           tabname => 't',  
                           pname   => 'cascade',  
                           pvalue  => 'true')
```

- Annotations
 - The procedure SET_PARAM is deprecated; SET_GLOBAL_PREFS should be used instead
 - The new procedures SET_SCHEMA_PREFS and SET_DATABASE_PREFS are also available, but they set the preferences only at table level!!!

ESTIMATE_PERCENT – Auto Sample Size



- Problem
 - Low sample sizes cannot always be used to gather representative object statistics
 - High sample sizes cannot be considered for big tables

- Solution
 - Implementation of auto sample size was improved
 - As a result, representative object statistics can be gathered in a much shorter timeframe

Statistics Collection for Partitioned Objects



- Problem
 - Object statistics for partitioned objects are gathered independently for each level (global, partition, sub-partition)
 - All segments must be scanned for global statistics. This might take a long time
- Solution
 - Incremental gathering can be enabled to avoid a scan of all segments
 - To compute the global statistics, additional information (so called synopsis) is stored in the data dictionary

```
dbms_stats.set_table_prefs(ownname => user,
                           tabname => 't',
                           pname   => 'incremental',
                           pvalue  => 'true')
```

Pending Statistics



- Problem
 - It is not possible to gather object statistics without replacing the statistics already available in the data dictionary
 - So, it is not possible to verify them before making them available
- Solution
 - Object statistics might be gathered in a private area

```
dbms_stats.set_table_prefs(ownname => user,
                           tabname => 't',
                           pname   => 'publish',
                           pvalue  => 'false')
```

- Statistics in the private area are called *pending statistics*
- The query optimizer uses pending statistics only when the initialization parameter OPTIMIZER_USE_PENDING_STATISTICS is set to TRUE
- Pending statistics can be published, deleted and exported

Comparing Statistics



- Problem
 - When (pending) object statistics are gathered it is sometimes useful to compare them with the previous ones
 - Writing SQL statements that compare different sets of object statistics is bothersome
- Solution **Backported to 10.2.0.4 as well!**
 - Use the new functions provided by the package DBMS_STATS
 - DIFF_TABLE_STATS_IN_HISTORY
 - DIFF_TABLE_STATS_IN_PENDING
 - DIFF_TABLE_STATS_IN_STATTAB

```
dbms_stats.diff_table_stats_in_pending(ownname => user,
                                       tabname  => 'T',
                                       time_stamp => NULL,
                                       pctthreshold => 10))
```

Comparing Statistics – Sample Output



```
STATISTICS DIFFERENCE REPORT FOR:
.....

TABLE          : T
OWNER          : OPS$CHA
SOURCE A      : Statistics as of 09-NOV-07 01.40.11.794565 AM +01:00
SOURCE B      : Current Statistics in dictionary
PCTTHRESHOLD  : 10
-----

NO DIFFERENCE IN TABLE / (SUB)PARTITION STATISTICS
-----

COLUMN STATISTICS DIFFERENCE:
.....

COLUMN_NAME  SRC NDV   DENSITY  HIST NULLS  LEN  MIN  MAX  SAMPSIZ
-----
N            A    1      1          NO    0    2   80   80   1000
            B    1      .0005      YES    0    2   80   80   1000
-----

NO DIFFERENCE IN INDEX / (SUB)PARTITION STATISTICS
```

Agenda



Data is always
part of the game.

- Indexing
- Optimization Techniques
- Object Statistics
- Plan Stability

Plan Stability



ex_execution_plan_stability.sql
ex_execution_plan_stability_10g.sql
ex_execution_plan_stability_11g.sql

- Problem
 - In theory, stored outlines provide plan stability
 - In reality, this is not always the case
- Solution
 - SQL plan baselines provide a new way to achieve plan stability
 - The query optimizer maintains a history of SQL statements
 - For each SQL statement, several execution plans (incl. bind variables and the environment leading to them) are stored
 - The query optimizer only uses verified (accepted) execution plans
- Annotations
 - Stored outlines are deprecated in Oracle Database 11g
 - SQL plan baselines are only available with Enterprise Edition

Managing SQL Plan Baselines – Capturing

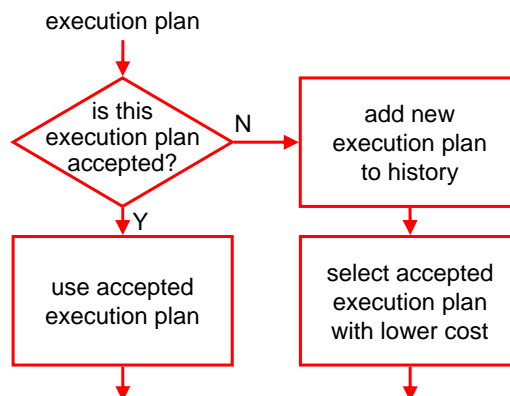


- Capturing SQL plan baselines can be automatic or manual
- Automatic capture is enabled when the parameter `OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES` is set to `TRUE`
 - Execution plans are automatically accepted only for SQL statements not yet stored in the history
- Manual capture (load) is performed through the following functions of the package `DBMS_SPM`
 - `LOAD_PLANS_FROM_SQLSET` loads execution plans from a SQL tuning set into the history
 - `LOAD_PLANS_FROM_CURSOR_CACHE` loads execution plans from the SGA into the history
 - Execution plans are automatically accepted

Managing SQL Plan Baselines – Selecting



- If the parameter `OPTIMIZER_USE_SQL_PLAN_BASELINES` is set to `TRUE` (by default it is `FALSE`), the query optimizer takes advantage of SQL plan baselines



Managing SQL Plan Baselines – Evolving



- The function `EVOLVE_SQL_PLAN_BASELINE` of the package `DBMS_SPM` verifies whether one of the non-accepted execution plans is better than the accepted ones. If this is the case, the execution plan is accepted

```
SELECT dbms_spm.evolve_sql_plan_baseline(  
        sql_handle => 'SYS_SQL_e56b12206dc9e29e'  
    )  
FROM dual
```

- During the maintenance window, the automatic SQL tuning task evolves baselines automatically
 - The Tuning Pack is required

ex_bind_peeking.sql

Bind Variable Peeking

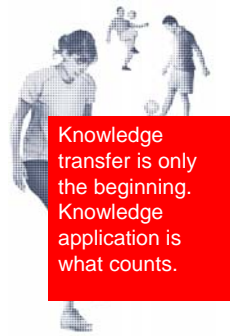


- Problem
 - Cursors based on SQL statements containing literals with different values, cannot be shared
 - Cursors based on SQL statements using bind variables might be shared even if the underlying execution plan is sub-optimal for some values
- Solution
 - Along with execution plans, information about the selectivity of predicates and their use is stored
 - During soft parses, the stored information and the selectivity of the current bind variable values is used to decide whether it is sensible to share a cursor or not

Core Messages



- Solutions for real problems are provided
- At last, DBMS_STATS is of age



■ ■ ■ Thank you!

