Top 7 Plan Stability Pitfalls & How to Avoid Them

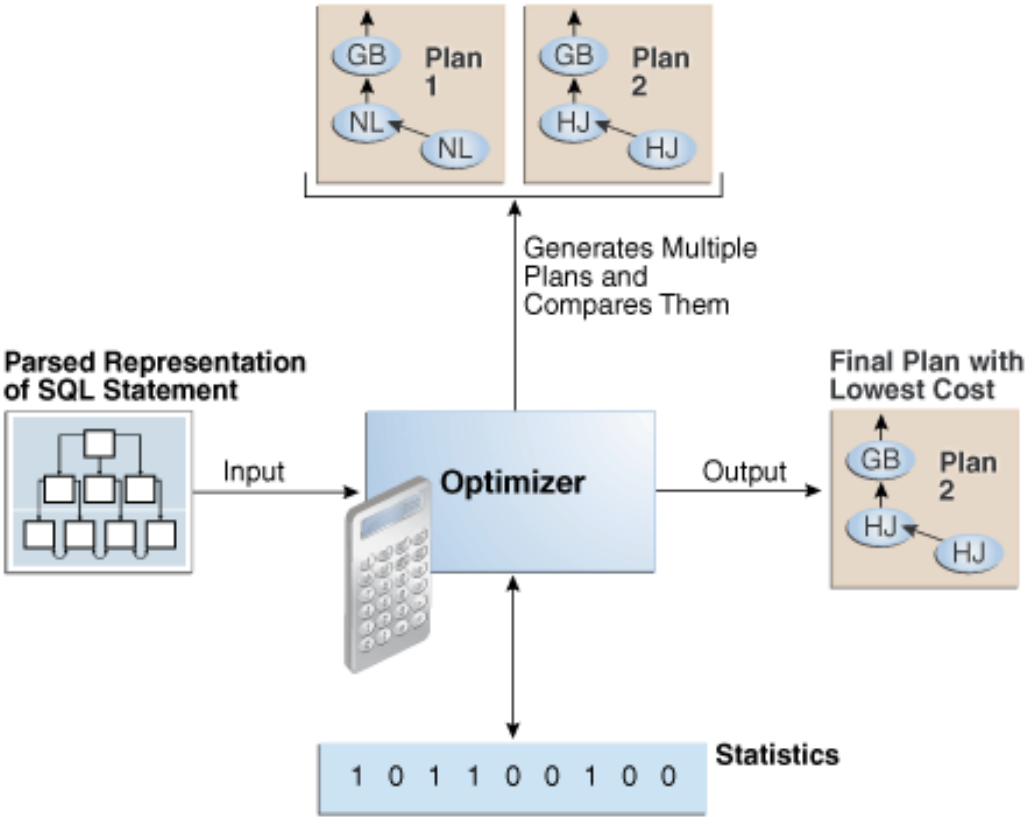
Neil Chandler
Chandler Systems Ltd
UK

Keywords:

SQL Optimizer Plan Change Stability Outlines Baselines Plan Directives

Introduction

When you write some SQL, Oracle runs it through the optimizer to determine the best access path. This involves taking the parsed representation of the SQL Statement, taking statistics about the data (which have been gathered in advance), generating multiple plans, comparing the plans and finally selecting the plan with the lowest "cost" and using it to access the data.



Oracle does not always get it right. In fact, if frequently gets the plans wrong by trying to find a reasonable execution plans in a reasonable amount of time. If the optimizer gets the plan wrong, then the next time the SQL is parsed, it will get it wrong again. and again. This presentation describes why it gets it wrong and what we can reasonably do to get the optimizer to keep picking the same plan consistently.

The Optimizer Always Gets It Wrong

Oracle weighs up a lot of information about your query to determine the optimal method of access. That takes time, and for complex SQL with many joins it may take longer to work out the best plan than it would take to just go and get the data with a sub-optimal plan. Oracle takes a pragmatic approach and, once it reaches its (by default) 2000$^{th\text{-}ish}$ "best guess" it just goes and gets the data.

You may think that 2,000 guesses is a lot, but you need to consider the sheer number of possibilities that the optimizer must consider to determine the best plan. For a 5 table join, the maths goes something like this:

Get a ***Join Order*** – Should we join table A-to-B-to-C-to-D-to-E, or A-to-B-to-C-to-E-to-D, or… well, you get the idea. That works out at 5! (factorial), or **120** possibilities.
Get a ***Join Method*** – There's 3 options; Nested Loop, Hash Join and Merge Join. So that's A-to-B x 3 + B-to-C x 3 + C-to-D x 3 and D-to-E x 3. 3 x 3 x 3 x 3 is **81** possibilities.
Get a ***Data Access*** path for each table. There are many of these. Full Table Scan, Index Unique Scan, Index Range Scan, Index Skip Scan, Index Fast Full Scan, Index Join Scan, Bitmap Conversion, Star Schema Transformation, Bloom Filters (which can only work with HJ), Bitmap Index Single Value, Bitmap Index Range Scans, Bitmap Index Merge, etc… but basically there's 5 common ones which we use all of the time so for simplicity, so 5 x 5 x 5 x 5 x 5 is **3,000** possibilities.

(125 x 81) = 10,125 x 3,000 = over **30,000,000** possible execution plans. And the optimizer has 2,000 guesses and stops.

So the optimizer always gets complex SQL wrong. Sometimes it changes its mind, and sometimes that's bad. Very very bad.

Let's explore why the plans change.

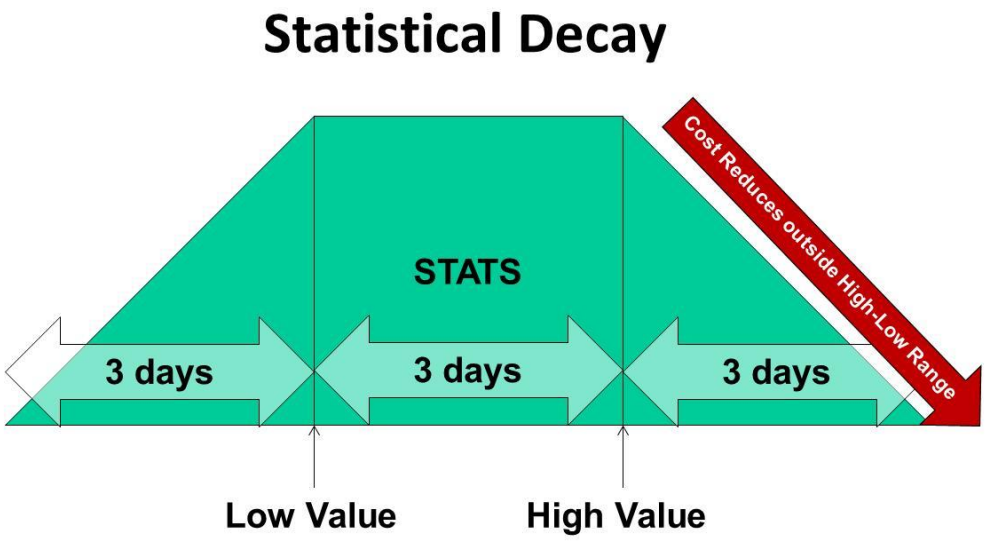
STATS

(1) STATS Changed : The plan may change because you changed the data in your database. Then you gathered statistics. The new statistics cause Oracle to change its mind and you get a new plan.

(2) STATS Did Not Change : The plan may change because you changed the data in you database. You didn't gather new statistics. You encountered statistical decay and that caused Oracle to change its mind.

[Demo showing these 2 plan changes happen]

Statistical Decay is the optimizer being told to select data outside of the low-to-high values it believes exists in the database.



Statistical Decay: If you query outside of the Low-to-High Values (held in view DBA_TAB_COLUMNS), Oracle "reduces" the cost of access the further away you get from those values, assuming there's an increasingly unlikely chance it'll encounter any data.

NO STATS

So we can get around the stats problem by having NO STATS! This is actually quite hard to as Oracle will automatically gather stats when running commands such as CTAS or CREATE INDEX. However, if you have no stats, the Optimizer will use Dynamic Sampling to get a set of stats at runtime. By default, this will involve reading 64 random blocks, assuming this is representative of the data and inferring stats based upon those blocks.

The blocks are random. The bigger the table, the less representative they will be. **(3)** Your plan is going to be random too.

The amount of blocks sampled can be increased, and the decision by Oracle of when to do dynamic sampling can be modified for it to happen when you have conjunctive/disjunctive predicates or complex expressions in the SQL, and this can be useful for Data Warehouse applications but is generally counterproductive in OLTP databases.

(4) Cardinality/Statistics Feedback

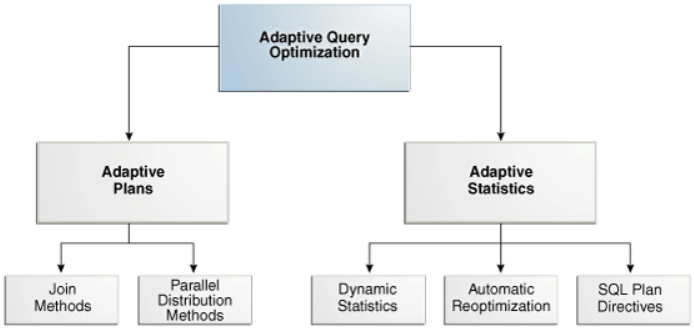
Statistics feedback (formally known as cardinality feedback) is one form of reoptimization that automatically improves plans for repeated queries that have cardinality mis-estimates. During the first execution of a SQL statement, the optimizer generates an execution plan and decides if it should enable statistics feedback monitoring for the cursor. Statistics feedback is enabled in the following cases: tables with no statistics, multiple conjunctive or disjunctive filter predicates on a table, and predicates containing complex operators for which the optimizer cannot accurately compute cardinality estimates. At the end of the execution, the optimizer compares its original cardinality estimates to the actual cardinalities observed during execution and, if estimates differ significantly from actual cardinalities, it stores the correct estimates for subsequent use.

It will also create a **(4.5)** SQL plan directive so other SQL statements can benefit from the information learnt during this initial execution. If the query executes again, then the optimizer uses the corrected cardinality estimates instead of its original estimates to determine the execution plan. If the initial estimates are found to be accurate no additional steps are taken. After the first execution, the optimizer disables monitoring for statistics feedback. NOTE! SQL Plan Directives have (at 12.1) only 1 option to improve your statistic at runtime *[shown in in Demo 4]* . ***(3)*** Dynamic Sampling!

(5) Bind Variable Peeking

When you have an OLTP system, you should generally be using Bind Variables to minimise the amount of hard parsing associated with executing SQL, with the value of the bind variable being the only thing which changes. However, the side effect of this was to lock in the plan for which the first variable was optimized. *[Shown in Demo 3]* This largely sucked, so Oracle 11 introduced…

(6) Adaptive Cursor Sharing



Adaptive Cursor Sharing noted that bind variables input values were being used on columns with skewed data and re-optimized getting a different plan for the same SQL. However, you still needed to get a BAD plan, and a BAD execution which was suboptimal before Oracle realised this, marked the cursor as GV$SQL.“IS_BIND_SENSITIVE” & “IS_BIND_AWARE”, and stating if the cursor was “SHARABLE” with different BIND variable inputs on subsequent runs.

```
SELECT sql_id, plan_hash_value, child_number, is_bind_sensitive, is_bind_aware,
is_shareable FROM   v$sql where sql_id = 'a7kb4nqnuvbt6'

SQL_ID        PLAN_HASH  CHILD_NUMBER IS_BIND_SENSITIVE IS_BIND_AWARE IS_SHAREABLE
------        ---------- ------------ ----------------- ------------- ------------
a7kb4nqnuvbt6 1745491793            0 Y                 N             N
a7kb4nqnuvbt6 4133469942            1 Y                 Y             Y
```

(7) Adaptive Execution Plans

Oracle attempted to cure the single BAD run by introducing Adaptive Execution Plans in Oracle 12C *[Shown in Demo 4]*. Join Method "Nested Loop" works better with small data sets. If the *Actual* data being selected from the table is significantly larger than the *Estimate* from the statistics, Oracle may decide to change the execution plan to a Hash Join whilst in flight. Likewise, it may switch from Hash Join to Nested Loop if there is significantly less data than expected. If you start with a Merge Join, you keep your Merge Join.

When a plan adapts, it may switch to a completely different plan. This also means that EXPLAIN PLAN and AUTOTRACE EXPLAIN ONLY are even less reliable than they were previously and shouldn't be used. The only way to know what the plan executed was is to look at it after the execution is complete. The plan below switches from 2 nested loops and index access to 2 Full Table Scans with a Hash Join. The rows marked "-" never happened, even though they were the original plan. Note the significant difference between the (starts * estimated-rows) and the (actual rows).

```
select * from table(dbms_xplan.display_cursor(format=>'+adaptive'));

PLAN_TABLE_OUTPUT from Demo 4 showing disabled plan rows
------------------------------------------------------------------------------------
SQL_ID fu3wvs2gsncx1, child number 0
-------------------------------------
select /*+ gather_plan_statistics */ --NEC     a.col1, a.col2, b.col1,
b.col2 from test_tab2 a, test_tab3 b where a.col1 = 0 and a.col2 = 1000
and b.id = a.id and b.col1 = 0 and b.col2 = 200

------------------------------------------------------------------------------------
|   Id | Operation                      | Name         | Starts | E-Rows | A-Rows
------------------------------------------------------------------------------------
|     0 | SELECT STATEMENT              |              |      1 |        |   2550
|  *  1 |  HASH JOIN                    |              |      1 |     25 |   2550
|-    2 |   NESTED LOOPS                |              |      1 |     25 |   5050
|-    3 |    NESTED LOOPS               |              |      1 |     25 |   5050
|-    4 |     STATISTICS COLLECTOR      |              |      1 |        |   5050
|  *  5 |      TABLE ACCESS FULL        | TEST_TAB2    |      1 |     25 |   5050
|- *  6 |     INDEX UNIQUE SCAN         | TEST_TAB3_PK |      0 |      1 | 0
|- *  7 |    TABLE ACCESS BY INDEX ROWID| TEST_TAB3    |      0 |      1 | 0
|  *  8 |   TABLE ACCESS FULL           | TEST_TAB3    |      1 |      1 |   7500
------------------------------------------------------------------------------------

Note
-----
   - this is an adaptive plan (rows marked '-' are inactive)
```

The plan changes to:

```
Plan hash value: 1241168087
-----------------------------------------------------------------
| Id  | Operation          | Name      | Starts | E-Rows | A-Rows
```

```
-------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |      1 |        |   2550
|*  1 |  HASH JOIN         |           |      1 |   2550 |   2550
|*  2 |   TABLE ACCESS FULL| TEST_TAB2 |      1 |   5050 |   5050
|*  3 |   TABLE ACCESS FULL| TEST_TAB3 |      1 |   7012 |   7500
-------------------------------------------------------------------

Note
-----
   - dynamic statistics used: dynamic sampling (level=2)
   - statistics feedback used for this statement
   - 1 Sql Plan Directive used for this statement
```

Some Bonus Reasons for Plan Changes

- Initialisation Parameter changes
- You're using AMM and your memory balance changes
- Changed an Optimizer-specific parameter
- Dropped/Added/Changed an index where Oracle was using it's stats
- Patching (except CPU patching, which does not change the optimizer)
- Hardware change.
- More or Less RAC nodes in the cluster
- Etc…

How to Control This?

Stored Outlines : Stored Outlines were introduced long long ago, and they *try* to convince the optimizer to use a particular path. You associate them with SQL Statements and they are basically a whole lot of SQL HINTS used to try to describe the Optimizer Access Path. Unlike when a Developer/DBA uses one or two hints, trying to persuade a plan out of the optimizer, Outlines use lots. A dozen. Twenty. More. I have found they generally worked pretty well, but are inflexible.
Once you put them in, they stick. Management (and DBA's) are scared to change or remove them, which you *really* should at upgrade time to look to use the new Optimizer features.

SQL Profiles : SQL Profiles are, well, a bunch of hints attached to a SQL Plan. Sound familiar? Their intention, over and above Outlines, is to provide the Optimizer with additional statistics about how the data is related, generally using the OPT_ESTIMATE hint to change the cardinality of the join predicate or column correlation, to make the COST more accurate
Basically, SQL Profiles are statistics, and need to be re-tuned regularly to ensure the scaling factor remains correct. They go stale, and they are not as easy to update as statistics.
And I try to avoid using them.

Baselines : Again, they are a hint-set used to manipulate a plan into the shape we want it, but with a couple of well-thought-out additions (and one poor one). Once we have baselines some SQL, which can be done in many ways – from AWR, from a current cursor in the cache or from a SQL Tuning set, Outline or SQL Profile, or from setting "optimizer_capture_sql_baselines=true" - we optimize the plan as usual.

If the COST of the plan is better than the cost of the Baseline, we keep the better plan but we are NOT allowed to use it yet. We then re-optimize with the hint-set and attempt to reproduce the Baseline plan. Hopefully this matches and we have our plan.

That night, when you are asleep, an auto-task job "SYS_AUTO_SPM_EVOLVE_TASK" runs, looks to see where you have better plans captured during the day, and "Evolves" them so you can get new plans tomorrow. This is exactly what I don't want to happen when I have spent a long time getting SQL Plans to behave exactly as I want them to. You can disable the autotask job as follows:

```
exec DBMS_SPM.SET_EVOLVE_TASK_PARAMETER
    ('SYS_AUTO_SPM_EVOLVE_TASK', 'ACCEPT_PLANS', 'false');
```

If you want to get new plans, for example if you have added a new index and would like to take advantage of it, you can evolve it yourself using dbms_spm.evolve_sql_plan_baseline:

```
set long 10000
set echo on
var evo_report clob
exec :evo_report := dbms_spm.evolve_sql_plan_baseline();
print :evo_report
EVO_REPORT
--------------------------------------------------------
GENERAL INFORMATION SECTION
--------------------------------------------------------
 Task Information:
 ---------------------------------------------
 Task Name            : TASK_61
 Task Owner           : NEIL
 Execution Name       : EXEC_151
 Execution Type       : SPM EVOLVE
 Scope                : COMPREHENSIVE
 Status               : COMPLETED
 Started              : 12/28/2016 00:00:00
 Finished             : 12/28/2016 00:00:00
 Last Updated         : 12/28/2016 00:00:00
 Global Time Limit    : 2147483646
 Per-Plan Time Limit  : UNUSED
 Number of Errors     : 0
--------------------------------------------------------
SUMMARY SECTION
--------------------------------------------------------
  Number of plans processed  : 1
  Number of findings         : 2
  Number of recommendations  : 1
  Number of errors           : 0
--------------------------------------------------------
DETAILS SECTION
--------------------------------------------------------
 Object ID     : 2
 Test Plan Name     : SQL_PLAN_3aanfnda31fhydc3ab811
 Base Plan Name     : SQL_PLAN_3aanfnda31fhy289a4f28
 SQL Handle     : SQL_352a8ea35430ba1e
 Parsing Schema     : NEIL
 Test Plan Creator  : NEIL
 SQL Text     : select count(pad_col) from test_tab1
                where text_col = :txt

Bind Variables:
------------------------------
 1  -  (VARCHAR2(128)):  INDEX_PLAN

Execution Statistics:
------------------------------
```

```
                    Base Plan   Test Plan
                    ---------    --------
 Elapsed Time (s):     .000038    .000002
 CPU Time (s):         .000033      0
 Buffer Gets:               24      0
 Optimizer Cost:            68      2
 Disk Reads:                 0      0
 Direct Writes:              0      0
 Rows Processed:             0      0
 Executions:                10     10
```

.
.
.

[Baseline capture and evolving shown in in Demo 5]

Happy Baselining!

Contact address:

Neil Chandler
Chandler Systems Ltd
London
UK

Phone: +44 (0)7802 686 528
Email: neil@chandler.uk.com
Twitter : @chandlerDBA
Internet: https://chandlerdba.wordpress.com