Sanity during Change

Arup Nanda

The $zillion Question

- *If it ain't broken, don't fix it!*
- *Must* Have Answers
  - What will happen – will the database at least perform as much as right now, or it might be worse?
  - How do we know?
  - How certain are we?

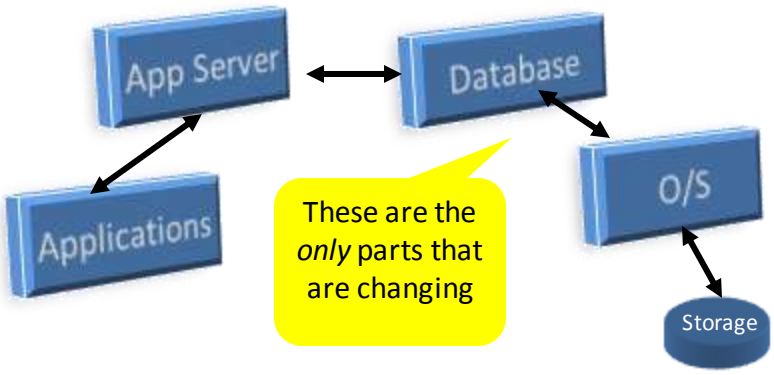
Risks during Migration

- Optimizer Plans could be change for better (or, *worse*) – performance related
- Functionality may have changed, producing unexpected results
- New bugs may be encountered for which there will be no patches, at least not immediately
- Some new functionality may require further attention

What we will cover

- Executing a stress-free migration
- Ensuring Efficient Plans
- Tools at your disposal
  - Database Replay
  - SQL Performance Analyzer
  - SQL Plan Baselines
  - Pending Stats
  - Intelligent Stats

Parts of a System



Database Replay

- This is where Database Replay really shines
- It captures the actual transactions from the production system, in the same order, with the same breaks in between
- It's as if the users are redoing the same activities in front of the test system
- Even sequence numbers are fetched the same way they occurred in production
- No primary key violation

Workload Capture

- The package dbms_workload_capture captures workload from current production
- The package exists in 11g, so what about 10g?
- In 10.2.0.4 it exists
- For earlier versions, a patch needs to be applied
  - Refer to MetaLink Note 560977.1 for details
- The easiest is to use Enterprise Manager Grid Control
- Grid Control 10.2.0.5 has the toolkit

Steps

- Capture Workload
  - It produces a set of files with extension *.rec
- Move them to the 11g system
- Use Replay feature in command line or EM to replay the activities
- Both these activities take AWR snapshots before and after events.
  - Use AWR Compare Period Report to compare the performance.

Capture from 10g

- Create a directory to hold the rec files

```
create directory DBREPLAY as 'c:\dbreplay'
```

- Add a Filter

```
BEGIN
   dbms_workload_capture.add_filter(
      fname      => 'myapp_filter',
      fattribute => 'USER',
      fvalue     => 'MYAPP');
END;
```

- Allows you to capture only those for the user called MYAPP

- Start the Capture Process

```
BEGIN
   DBMS_WORKLOAD_CAPTURE.START_CAPTURE (
      name              => 'capture1',
      dir               => 'DBREPLAY',
      duration          => 3600,
      default_action    => 'INCLUDE',
      auto_unrestrict   => TRUE);
END;
```

- It will generate a lot of files in the format wcr_*.rec in the c:\DBReplay directory.

- Get the capture ID

```
select ID from dba_workload_captures
where status = 'COMPLETED'
```

- Export the AWR

```
begin
  dbms_workload_capture.export_awr
    (capture_id => <captureid>);
end;
```

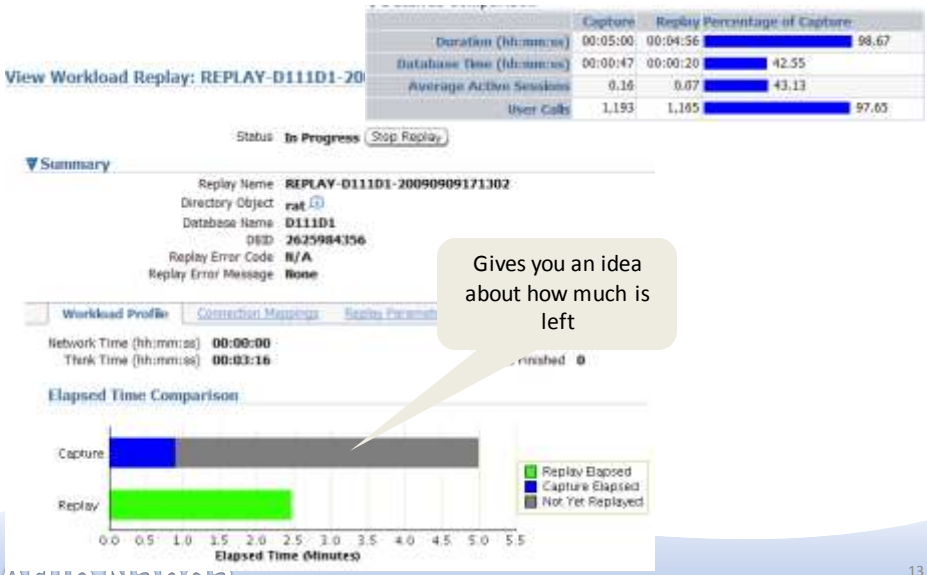
- AWR will also be exported as a dumpfile in the DBReplay directory.
- Copy all the files in that directory to the target system

Replay Steps

1. Create directory on the target
2. Pre-process the captured workload
3. Replay the workload
4. From the command line

```
$ wrc system/manager
   replaydir=/u01/oracle/rat
```

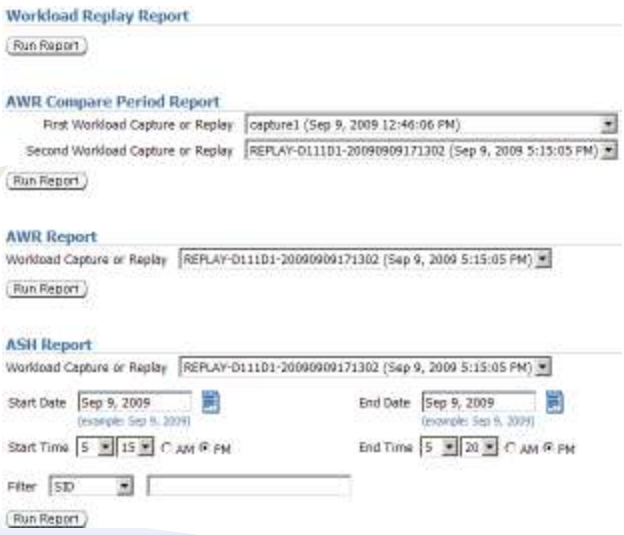
During Replay



Gives you an idea about how much is left

Get the Reports

This "compare" report, aka "Diff-diff Report" is the most important. It shows the system stats on the target and the source when the *same* activities were occurred there.



SQL Performance Analyzer

- Some SQLs showed regression, i.e. they underperformed compared to 10g
- You need to know *why*
  - optimizer environment, bind variables, etc?
- SPA allows you to *run* captured SQLs in differing environments
  - In the same database but
    - Different optimizer parameters
    - Different ways of collecting stats,
    - With pending stats in 11g, can validate on PROD during maintenance windows/non-peak
    - Different indexes, or MVs

Source of SQLs

- Shared Pool
- Captured from Production during a workload
- Stored in a SQL Tuning Set (STS)
- Continuous Capture functionality to capture all SQLs

Capture from 10g

- The following captures the SQL Statements into a SQL Tuning Set (STS) in 10g.

```
BEGIN dbms_sqltune.capture_cursor_cache_sqlset(
      sqlset_name    =>'10GSTS',
      time_limit     => '3600',
      repeat_interval=>'300',
      sqlset_owner   =>'SYS');
END;
```

  This incrementally captures the SQL statements every 5 mins for 10 hours.

- You can export this STS and import into 11g.

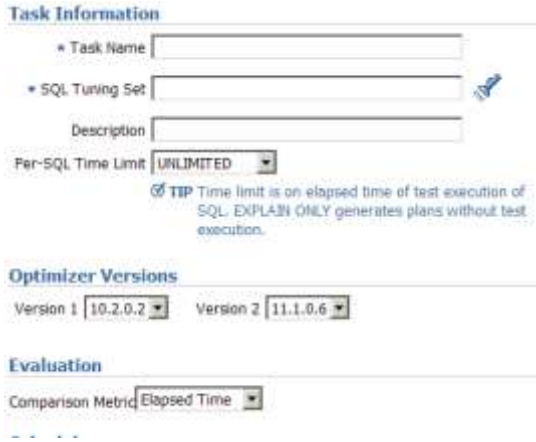
SPA Tasks

Step	Description
1	Create SQL Performance Analyzer Task based on SQL Tuning Set
2	Replay SQL Tuning Set in Initial Environment
3	Replay SQL Tuning Set in Changed Environment
4	Compare Step 2 and Step 3
5	View Trial Comparison Report

- Create an SPA Task on the STS imported
- Replay with Optimizer = 10.2.0.4
- Replay with Optimizer = 11.1.0.7
- Compare and make adjustments
- Repeat 2 through 4 as needed
- `http://www.oracle.com/technology/oramag/oracle/08-mar/o28sqlperf.html`

SPA Optimizer Change

Create an SPA Task on the STS imported



Compare

Elapsed time significantly reduced

Majority of SQLs didn't see their plan changed!

Compare ...

Shows the SQL_IDs, we can find from v$sql

Top 10 SQL Statements Based on Impact on Workload

SQL ID	Net Impact on Workload (%)	Elapsed Time initial_sql_trial	Elapsed Time second_sql_trial	Net Impact on SQL (%)	% of Workload initial_sql_trial	% of Workload second_sql_trial	Plan Changed
[illegible]	58.390	0.031	0.000	100.000	58.390	0.000	N
[illegible]	13.770	0.004	0.000	100.000	13.770	0.000	N
[illegible]	5.540	0.011	0.000	100.000	5.540	0.000	N
[illegible]	5.540	0.011	0.000	100.000	5.540	0.000	N
1rswbxwhbpmr7	2.560	0.070	0.011	84.290	3.030	21.410	Y
[illegible]	2.470	0.015	0.000	100.000	2.470	0.000	N

Plan changed for this SQL, Using SQL_ID, check from v$sql

You can call upon SQL Tuning Advisor to suggest possible tuning options on this SQL

SQL Details: 1rswbxwhbpmr7

Parsing Schema **SYS** Execution Frequency **276** (Schedule SQL Tuning Advisor)

▶ SQL Text

Single Execution Statistics

Execution Statistic Name	Net Impact on Workload (%)	Execution Statistic Collected initial_sql_trial	Execution Statistic Collected second_sql_trial	Net Impact on SQL (%)	% of Workload initial_sql_trial	% of Workload second_sql_trial
Elapsed Time	2.560	0.070	0.011	84.290	3.030	21.410
Parse Time	-5.400	0.013	0.051	-292.310	1.850	8.010
CPU Time	37.100	0.070	0.011	84.290	44.020	14.920
Buffer Gets	0.100	22.000	20.000	9.090	1.090	0.990
Optimizer Cost	0.140	14.000	12.000	14.290	0.970	0.840
Disk Reads	0.000	0.000	0.000	0.000	0.000	0.000
Direct Writes	0.000	0.000	0.000	0.000	0.000	0.000
Rows Processed	0.000	1.000	1.000	0.000	0.000	0.000

Symptom Findings

The structure of the SQL execution plan has changed.

The report continues with the plans before and after the upgrade, so you can compare them

Pending Stats

Lowdown on Stats

- Optimizer Statistics on tables and indexes are vital for the optimizer to compute optimal execution plans
- In many cases you gather stats with estimate
- Without accurate stats, the optimizer may decide on a sub-optimal execution plan
- When stats change, the optimizer may change the plan
- Truth: stats affect the plan, but not necessarily positively

Meet John the DBA

- John the DBA at Acme Bank
- Hard working, knowledgeable, politically not very savvy
- Collects statistics every day via an automated job

Data: Value *vs* Pattern

State	Customers	%age
CT	1,000	10%
NY	5,000	50%
CA	4,000	40%

After some days

State	Customers	%age
CT	2,000	10%
NY	10,000	50%
CA	8,000	40%

Important

The data itself changed; but the pattern did not. The new stats will not change the execution path, and therefore probably not needed

Case 2

State	Customers	%age
CT	1,000	10%
NY	5,000	50%
CA	4,000	40%

After some days

State	Customers	%age
CT	2,500	12.5%
NY	10,500	52.5%
CA	7,000	35.0%

Important

The pattern is different; but still close to the original pattern. *Most* queries should perform well with the original execution plan.

Naked Truth

- Stats can actually create performance issues
- Example
  - A query plan had nested loop as a path
  - Data changed in the underlying tables
  - But the pattern did not change much
  - So, NL was still the best path
  - Stats were collected
  - Optimizer detected the subtle change in data pattern and changed to hash joins
  - Disaster!

The problem with new stats

- The CBO does not now what is close *enough*
  - For it, 50.0% and 52.5% are *different* values
- The internal logic of the CBO may determine a different plan due to this *subtle* change
- This new plan may be better, or worse
  - This is why many experts recommend not collecting stats when database performance is acceptable

John followed the advice

- John followed the advice
- He stopped collecting stats
- The database performance was acceptable
- But one day – disaster struck!

Data Pattern Changed

State	Customers	%age
CT	1,000	10%
NY	5,000	50%
CA	4,000	40%

After some days

State	Customers	%age
CT	10,500	52.5%
NY	2,500	12.5%
CA	7,000	35.0%

CT was 12.5% but now it is 52.5%

- Optimal Plan is Different
  - Queries against CT used to have index scan; but now a full table scan would be more appropriate
- Since the stats were not collected, CBO did not know about the change
  - Queries against CT still used index scan
  - And NY still used full table scan
- Disaster!
- John was blamed

What's the Solution?

- If only you could predict the effect of new stats before the CBO uses them
  - and make CBO use them if there are no untoward issues
- Other Option
  - You can collect stats in a different database
  - Test in that database
  - If everything looks ok, you can export the stats from there and import into production database
- The other option is not a very good one
  - The test database may not have the same distribution
  - It may not have the same workload
  - Worst – you don't have time to test all queries

Pending Stats

- In Oracle 11g R1, John can use a new feature – **Pending Statistics**
- In short
  - John collects stats as usual
  - But the CBO does not see these new stats
  - John examines the effects of the stats on queries of a session where these new stats are active
  - If all look well, he can "publish" these stats
  - Otherwise, he discards them

How to Make Stats "Pending"

- It's the property of the table (or index)
- Set it by a packaged procedure
  `DBMS_STATS.SET_TABLE_PREFS`
- Example:

```
begin
  dbms_stats.set_table_prefs (
    ownname => 'ARUP',
    tabname => 'SALES',
    pname   => 'PUBLISH',
    pvalue  => 'FALSE'
  );
end;
```

- After this, the stats collected will be *pending*

prefs_false.sql
sales_stats.sql_

Table Preferences

- The procedure is not new. Used before to set the default properties for stats collection on a table.
  - e.g. to set the default degree of stats collection on the table to 4:

```
dbms_stats.set_table_prefs (
   ownname => 'ARUP',
   tabname => 'SALES',
   pname   => 'DEGREE',
   pvalue  => 4
 );
```

Stats after "Pending"

- When the table property of stats "PUBLISH" is set to ""FALSE"
- The stats are not visible to the Optimizer
- The stats will not be updated on USER_TABLES view either:

```
select to_char(last_analyzed,'mm/dd/yy hh24:mi:ss')
from user_tables
where table_name = 'SALES';

TO_CHAR(LAST_ANAL
-----------------
09/10/07 22:04:37
```

la.sql_

Visibility of Pending Stats

- The stats will be visible on a new view USER_TAB_PENDING_STATS

```
select to_char(last_analyzed,'mm/dd/yy hh24:mi:ss')
from user_tab_pending_stats
where table_name = 'SALES';

TO_CHAR(LAST_ANAL
-----------------
09/21/07 11:03:35
```

pending.sql_

Checking the Effect of Pending Stats

- Set a special parameter in the session

```
alter session set
  optimizer_use_pending_statistics = true;
```

- After this setting, the CBO will consider the new stats *in that session only*
- You can even create and index and collect the pending stats on the presence of the index
- To check if the index would make any sense

alter_true.sql_

Publishing Stats

- Once satisfied, you can make the stats visible to optimizer

```
begin
   dbms_stats.publish_pending_stats
      ('ARUP', 'SALES');
end;
```

- Now the USER_TABLES will show the correct stats
- Optimizer will use the newly collected stats
- Pending Stats will be deleted

publish.sql_

What if the New Stats make it Worse?

- Simply delete them

```
begin
  dbms_stats.delete_pending_stats ('ARUP','SALES');
end;
```

- The pending stats will be deleted
- You will not be able to publish them

Checking for Preferences

- You can check for the preference for publishing stats on the table SALES:

```
select dbms_stats.get_prefs ('PUBLISH','ARUP','SALES') from dual;

DBMS_STATS.GET_PREFS('PUBLISH','ARUP','SALES')
----------------------------------------------
FALSE
```

- Or, here is another way, with the change time:

```
select pname, valchar, valnum, chgtime
from optstat_user_prefs$
where obj# = (select object_id from dba_objects
where object_name = 'SALES' and owner = 'ARUP')

PNAME      VALCHAR CHGTIME
---------- ------- ---------------------------------
PUBLISH    TRUE    02-MAR-10 01.38.56.362783 PM -05:00
```

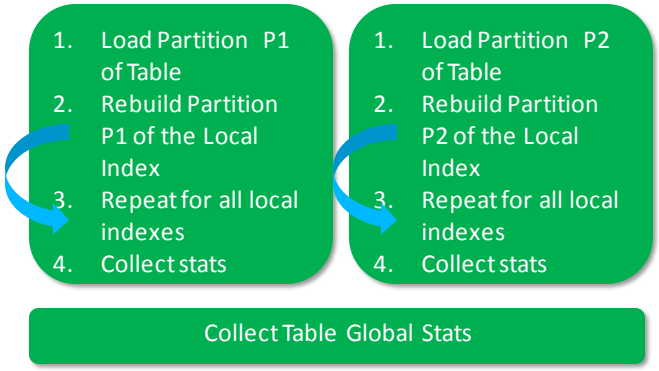
Other Preferences

- The table property is now set to FALSE
- You can set the default stats gathering of a whole schema to pending

```
begin
    dbms_stats.set_schema_prefs (
        ownname => 'ARUP',
        pname   => 'PUBLISH',
        pvalue  => 'FALSE');
end;
```

- You can set it for the whole database as well
  - `dbms_stats.set_database_prefs`

Loading of Partitioned Tables

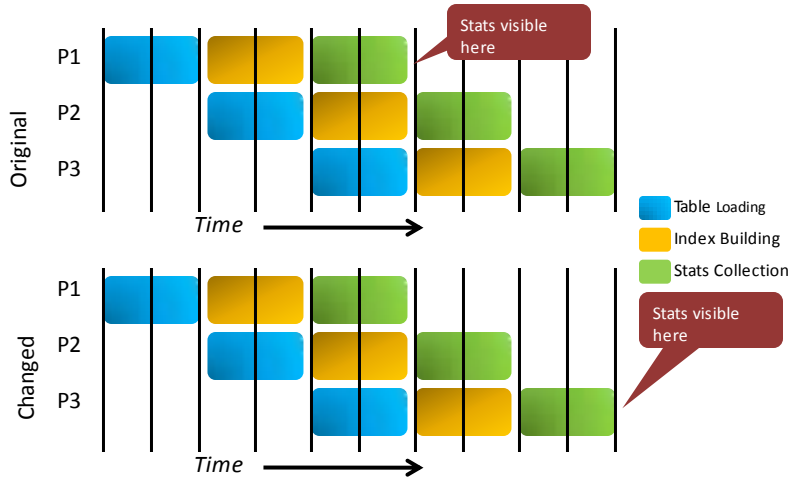


1. You may want to make sure that the final table global stats are collected after all partition stats are gathered
2. And all are visible to CBO at the same time

Options

- You can postpone the stats collection of the partitions to the very end
- But that means you will lose the processing window that was available after the partition was loaded
- Better option: set the table's stats PUBLISH preference to FALSE
- Once the partition is loaded, collect the stat; but defer the publication to the very end

Defer Partition Table Stats



Stats History

- When new stats are collected, they are maintained in a history as well
- In the table `SYS.WRI$_OPTSTAT_TAB_HISTORY`
- Exposed through `*_TAB_STATS_HISTORY`

```
select to_char(stats_update_time,'mm/dd/yy hh24:mi:ss')
from user_tab_stats_history
where table_name = 'SALES';

    TO_CHAR(STATS_UPD
    -----------------
    03/01/10 21:32:57
    03/01/10 21:40:38
```

hist.sql

Reinstate the Stats

- Suppose things go wrong
- You wish the older stats were present rather than the newly collected ones
- You want to restore the old stats

```
begin
   dbms_stats.restore_table_stats (
       ownname         => 'ARUP',
       tabname         => 'SALES',
       as_of_timestamp => '14-SEP-07 11:59:00 AM'
   );
end;
```

-

reinstate.sql

Exporting the Pending Stats

- First create a table to hold the stats

```
begin
   dbms_stats.create_stat_table (
      ownname  => 'ARUP',
      stattab  => 'STAT_TABLE'
   );
end;
```

- This will create a table called `STAT_TABLE`
- This table will hold the pending stats

cr_stattab.sql_

Export the stats

- Now export the pending stats to the newly created stats table

```
begin
   dbms_stats.export_pending_stats (
     tabname    => 'SALES',
     stattab    => 'STAT_TABLE'
   );
end;
```

- Now you can export the table and plug in these stats in a test database
  - `dbms_stats.import_pending_stats`

export.sql
del_stats.sql
import.sql_

Real Application Testing

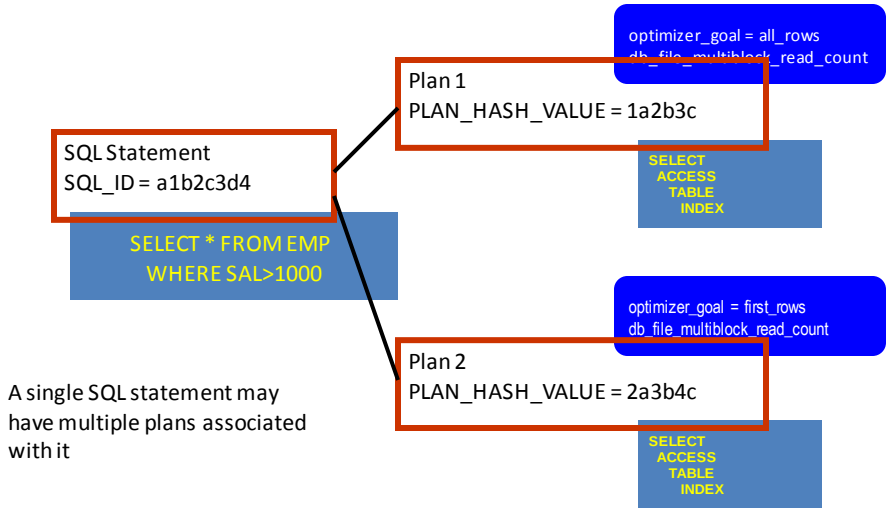
- You can use Database Replay and SQL Performance Analyzer to recreate the production workload
- But under the *pending* stats, to see the impact
- In SPA use `alter session set optimizer_use_pending_statistics = true;`
- That way you can predict the impact of the new stats with your specific workload

Some additional uses

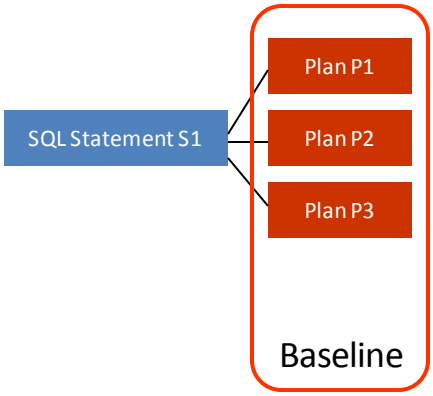
- You can create a SQL Profile in your session
  - With private stats
- Then this profile can be applied to the other queries
- You can create SQL Plan Management Baselines based on these private stats
- Later you can apply these baselines to other sessions

SPM

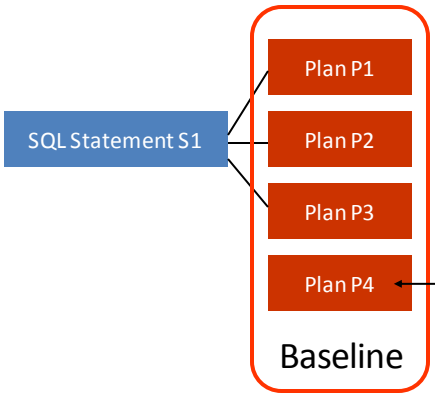
SQL Plan Management



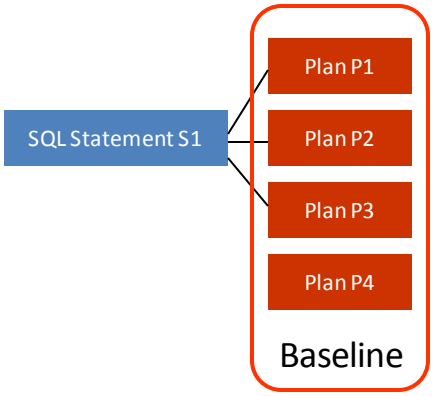
A single SQL statement may have multiple plans associated with it



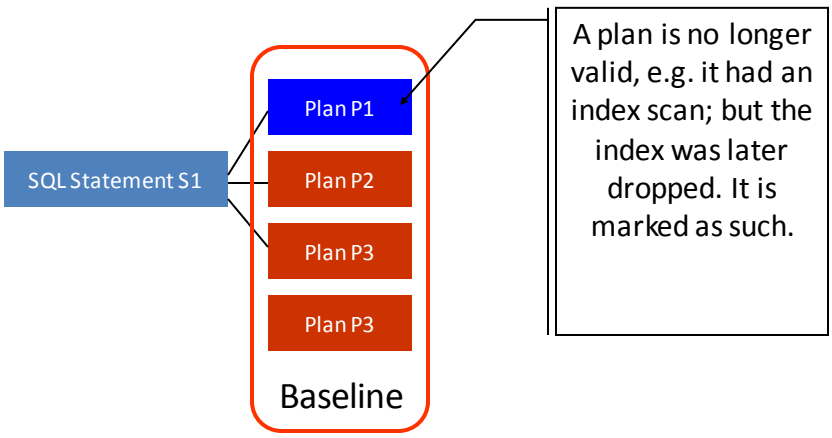
A baseline is a collection of plans for a specific SQL statement



A new plan was generated as a result of some change, e.g. the optimizer parameters were changed. This plan is added to the baseline

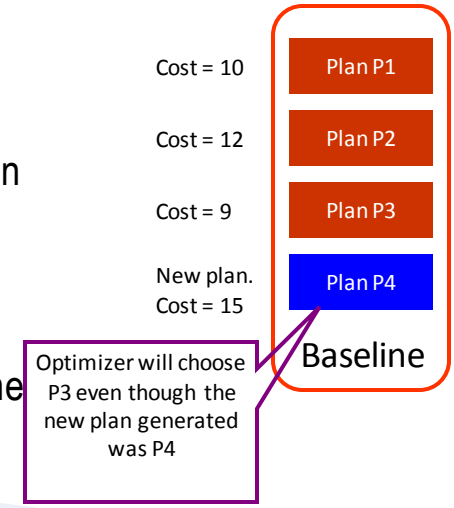


When a SQL is reparsed, the optimizer compares the plan to the list of plans in the baseline, but **not the newly generated plan** as it is not "accepted".



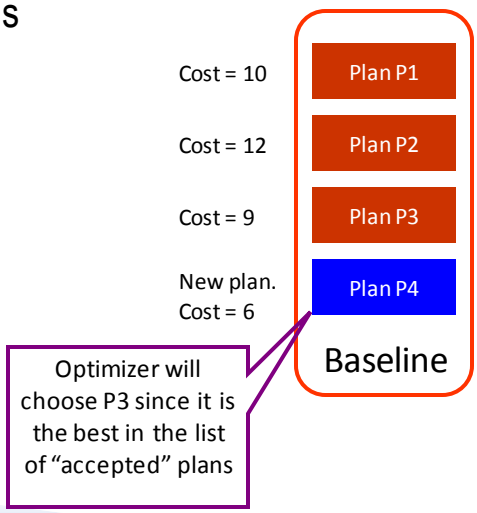
New Plan is Worse

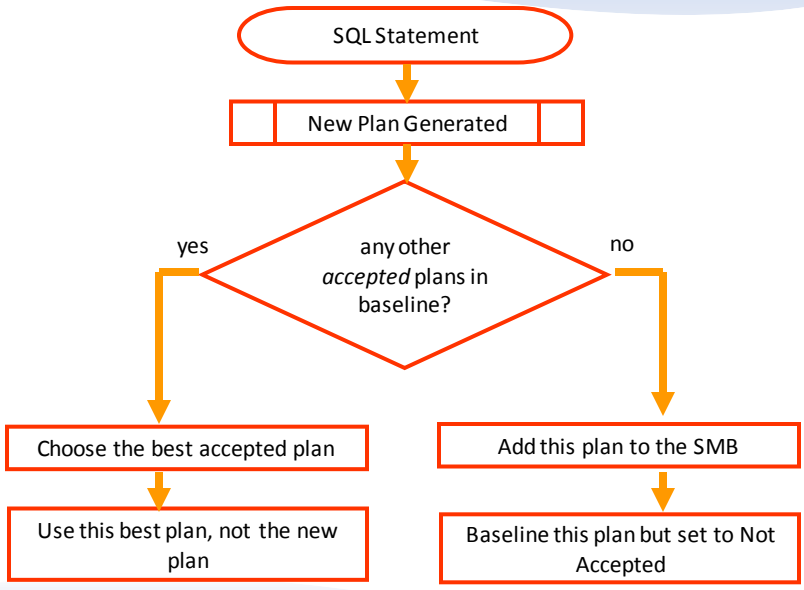
- Baselines contain the history of plans for an SQL statement
- If there was a good plan ever, it will be there in the baseline
- So the optimizer can choose the plan with the lowest cost



New Plan is the Best

- Even if the new plan is the best, it will be not be immediately used
- The DBA can later made the plan fit for consideration by "evolving" it!





SQL Management Base

- A repository where the following are stored
  - Statements
  - Plan histories
  - Baselines
  - SQL profiles
- Stored in SYSAUX tablespace

Configuring SMB

To Check

```
select parameter_name, parameter_value
from dba_sql_management_config;
PARAMETER_NAME          PAMETER_VALUE
----------------------- -------------
SPACE_BUDGET_PERCENT               10
PLAN_RETENTION_WEEKS               53
```

To Change:

```
BEGIN
  DBMS_SPM.CONFIGURE(
    'PLAN_RETENTION_WEEKS',100);
END;
```


Adding Baselined Plans

- To capture baselines

```
alter session set
  optimizer_capture_sql_plan_baselines = true
/
```

- … execute the queries at least 2 times each
- Or run the application as usual

```
alter session set
  optimizer_capture_sql_plan_baselines = false
/
```

- A plan is baselined when a SQL is executed more than once

Adding more plans

- Change the optimizer parameter to use pending stats

```
alter session set
  optimizer_use_pending_statistics = true;
```

- a new plan is generated
- Capture the plans for the baseline

```
alter session set
  optimizer_capture_sql_plan_baselines = true;
```

- Now all the plans will use pending stats in the session
- The new plan is stored in baseline but not "accepted"; so it will not be used by the optimizer

To check for Plans in the baseline

```
select SQL_HANDLE, PLAN_NAME
from dba_sql_plan_baselines
where SQL_TEXT like '%SPM_TEST%'
/

SQL_HANDLE                     PLAN_NAME
------------------------------ ------------------------------
SYS_SQL_4602aed1563f4540       SYS_SQL_PLAN_563f454011df68d0
SYS_SQL_4602aed1563f4540       SYS_SQL_PLAN_563f454054bc8843
```

SQL Handle is the same since it's the same SQL; but there are two plans

To See Plan Steps in Baseline

- Package DBMS_XPLAN has a new function called display_sql_plan_baseline:

```
select * from table (
    dbms_xplan.display_sql_plan_baseline (
        sql_handle=>'SYS_SQL_4602aed1563f4540',
        format=>'basic note')
    )
```


Checking Plans Being Used

```
Execution Plan
----------------------------------------------------------
Plan hash value: 2329019749

-------------------------------------------------------------------------------------------
| Id  | Operation                   | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |               | 17139 |  1690K|   588   (1)| 00:00:08 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| ACCOUNTS      | 17139 |  1690K|   588   (1)| 00:00:08 |
|*  2 |   INDEX RANGE SCAN          | IN_ACCOUNTS_01 | 34278 |       |    82   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("TEMPORARY"='Y')
   2 - access("STATUS"='INVALID')

Note
-----
   - SQL plan baseline "SYS_SQL_PLAN_51f8575d04eca402" used for this statement
```

This shows that a SQL Plan Baseline is being used.

Evolve a Plan

- Make a plan as acceptable (only if it is better)

```
variable rep CLOB
begin
 :rep :=
   dbms_spm.evolve_sql_plan_baseline (
    sql_handle => 'SYS_SQL_5a8b6da051f8575d',
    verify  => 'YES'
   );
end;
/
```

- Variable REP shows the analysis.

Fixing a Plan

spm_test6.sql

- A plan can be fixed by:

```
dbms_spm.alter_sql_plan_baseline (
     sql_handle => 'SYS_SQL_5a8b6da051f8575d',
     plan_name => 'SYS_SQL_PLAN_51f8575d04eca402',
     attribute_name => 'fixed',
     attribute_value => 'YES'
)
```

- Once fixed, the plan will be given priority
- More than one plan can be fixed
- In that case optimizer chooses the best from them
- To "unfix", use `attribute_value => 'NO'`

Use of Baselines

- Checking the plan before accepting new stats
- Fixing Plan for Third Party Applications
- Database Upgrades
  - Both within 11g and 10g->11g
  - Capture SQLs into STS then move the STS to 11g
- Database Changes
  - Parameters, Tablespace layout, etc.
  - Fix first; then gradually unfix them

Stored Outlines

- Outlines make a plan for a query *fixed*
  - The optimizer will pick up the fixed plan every time
- Problem:
  - Based on the bind variable value, data distribution, etc. specific plan may change
  - A fixed plan may actually be worse

Summary

- You can modify the property of a table so that new stats are not immediately visible to the optimizer
- In a session, you can use a special parameter to make the optimizer see these pending stats, so that you can test the effect of these stats.
- If you are happy with the stats collected, you can make them visible to optimizer
- Otherwise, you can discard the stats
- You can see the history of stats collected on tables
- You can restore a previously collected set of stats
- You can export the pending stats to a test database
- You can test the effect of the pending stats with your specific workload by SQL Performance Analyzer and Database Replay.
- You can create baselines by using the pending stats

Compound Stats

Effect of Stats on Two Columns

- Optimizer Statistics on tables and indexes are vital for the optimizer to compute optimal execution plans
- If there are stats on two different columns used in the query, how does the optimizer decide?
- It takes the selectivity of each column, and multiplies that to get the selectivity for the query.

Example

- Two columns
  - Month of Birth: selectivity = 1/12
  - Zodiac Sign: selectivity = 1/12
- What will be the selectivity of a query
  - Where zodiac sign = 'Pisces'
  - And month of birth = 'January'
- Problem:
  - According to the optimizer it will be 1/12 × 1/12 = 1/144
  - In reality, it will be 0, size the combination is not possible
- What will be the selectivity of a query
  - Where zodiac sign = 'Capricorn'
  - And month of birth = 'January'

Multi-column Intelligence

- If the Optimizer knew about these combinations, it would have been able to choose the proper path
- How would you let the optimizer learn about these?
- In Oracle 10g, we saw a good approach – SQL Profiles
  - which allowed data to be considered for execution plans
  - but was not a complete approach
  - it still lacked a dynamism – applicability in all circumstances
- In 11g, there is an ability to provide this information to the optimizer
  - Multi-column stats

Arup Nanda

An Example

- Table BOOKINGS
- Index on (HOTEL_ID, RATE_CODE)
- What will be plan for the following?

```
select min(book_txn)
from bookings
where hotel_id = 10
and rate_code = 23
```

```
  HOTEL_ID  RATE_CODE   COUNT(1)
---------- ---------- ----------
        10         11     444578
        10         12      50308
        20         22     100635
        20         23     404479
```

vals.sql

Arup Nanda

The Plan

Here is the plan

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |          |     1 |    10 |   769   (3)| 00:00:10 |
|   1 |  SORT AGGREGATE    |          |     1 |    10 |            |          |
|*  2 |   TABLE ACCESS FULL| BOOKINGS |  199K| 1951K|   769   (3)| 00:00:10 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):

PLAN_TABLE_OUTPUT
------------------------------------------------
   2 - filter("RATE_CODE"=23 AND "HOTEL_ID"=10))
```

- It didn't choose index scan
- The estimated number of rows are 199K, or about 20%; so full table scan was favored over index scan

expl1.sql

Solution

- Create Extended Stats in the related columns – HOTEL_ID and RATE_CODE

```
var ret varchar2(2000)
begin
  :ret := dbms_stats.create_extended_stats(
    'ARUP', 'BOOKINGS','(HOTEL_ID, RATE_CODE)'
  );
end;
/
print ret
```

- The variable "ret" shows the name of the extended statistics

xstats.sql

Then Collect Stats Normally

```
begin
  dbms_stats.gather_table_stats (
    ownname          => 'ARUP',
    tabname          => 'BOOKINGS',
    estimate_percent=> 100,
    method_opt       => 'FOR ALL COLUMNS SIZE SKEWONLY',
    cascade          => true
  );
end;
/
```

stats.sql

The Plan Now

- After extended stats, the plan looks like this:

```
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                |     1 |    10 |   325   (1)| 00:00:04 |
|   1 |  SORT AGGREGATE              |                |     1 |    10 |            |          |
|   2 |   TABLE ACCESS BY INDEX ROWID| BOOKINGS       | 23997 |  234K|   325   (1)| 00:00:04 |
|*  3 |    INDEX RANGE SCAN          | IN_BOOKINGS_01 | 23997 |      |    59   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

- Note:
  - No of Rows is now more accurate
  - As a result, the index scan was chosen

expl1.sql

Extended Stats

- Extended stats store the correlation of data among the columns
  - The correlation helps optimizer decide on an execution path that takes into account the data
  - Execution plans are more accurate
- Under the covers,
  - extended stats create an invisible virtual column
  - Stats on the columns collects stats on this virtual column as well

10053 Trace

```
Single Table Cardinality Estimation for BOOKINGS[BOOKINGS]
  Column (#2):
    NewDensity:0.247422, OldDensity:0.000000 BktCnt:1000000,
PopBktCnt:1000000, PopValCnt:2, NDV:2
  Column (#3):
    NewDensity:0.025295, OldDensity:0.000000 BktCnt:1000000,
PopBktCnt:1000000, PopValCnt:4, NDV:4
  Column (#5):
    NewDensity:0.025295, OldDensity:0.000000 BktCnt:1000000,
PopBktCnt:1000000, PopValCnt:4, NDV:4
  ColGroup (#1, VC) SYS_STU4JHE7J4YQ3ZLDXSW5L1O8KX
    Col#: 2 3    CorStrength: 2.00
  ColGroup Usage:: PredCnt: 2  Matches Full:   Using density:
0.025295 of col #5 as selectivity of unpopular value pred
```

Extended Stats

- This hidden virtual column shows up in column statistics

```
select column_name, density, num_distinct
from user_tab_col_statistics
where table_name = 'BOOKINGS'

COLUMN_NAME                    DENSITY NUM_DISTINCT
------------------------------ ---------- ------------
BOOKING_ID                      .000001      1000000
HOTEL_ID                       .0000005            2
RATE_CODE                      .0000005            4
BOOK_TXN                      .002047465         2200
SYS_STU4JHE7J4YQ3ZLDXSW5L1O8KX  .0000005            4
```


Checking for Extended Stats

- To check the presence of extended stats, check the view dba_stat_extensions.

```
select extension_name, extension
from dba_stat_extensions
where table_name='BOOKINGS';
```

Output:

```
EXTENSION_NAME                 EXTENSION
------------------------------ ------------------------
SYS_STU4JHE7J4YQ3ZLDXSW5L1O8KX ("HOTEL_ID","RATE_CODE")
```

check.sql

Deleting Extended Stats

- If you want, you can drop the extended stats, you can use the dbms_stats package, specifically the procedure drop_exteneded_stats

```
begin
  dbms_stats.drop_extended_stats (
    ownname => 'ARUP',
    tabname => 'BOOKINGS',
    extension => '("HOTEL_ID","RATE_CODE")'
   );
end;
```

drop.sql

Another way

- You can collect the extended stats using the normal dbms_stats as well:

```
begin
    dbms_stats.gather_table_stats (
        ownname          => 'ARUP',
        tabname          => 'BOOKINGS',
        estimate_percent => 100,
        method_opt       =>
'FOR ALL COLUMNS SIZE SKEWONLY FOR COLUMNS
(HOTEL_ID,RATE_CODE)',
        cascade          => true
    );
end;
/
```

startx.sql

The Case on Case Sensitivity

- A table of CUSTOMERS with 1 million rows
- LAST_NAME field has the values
  - McDonald – 20%
  - MCDONALD – 10%
  - McDONALD – 10%
  - mcdonald – 10%
- They make up 50% of the rows, with the variation of the same name.
- When you issue a query like this:

```
select * from customers where upper(last_name) = 'MCDONALD'
```

Cust_cnt.sql

Normal Plan

- The plan looks like this:

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           | 10000 |   498K|  2140   (2)| 00:00:26 |
|*  1 |  TABLE ACCESS FULL| CUSTOMERS | 10000 |   498K|  2140   (2)| 00:00:26 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------
   1 - filter(UPPER("LAST_NAME")='MCDONALD')
```

No of rows wrongly estimated

expl2.sql

Extended Stats

- You collect the stats for the UPPER() function

```
begin
  dbms_stats.gather_table_stats (
     ownname    => 'ARUP',
     tabname    => 'CUSTOMERS',
     method_opt => 'for all columns size
skewonly for columns (upper(last_name))'
  );
end;
```

statsx_cust.sql

With Extended Stats

- The plan is now:

No of rows correctly estimated

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |   500K|    33M|  2140   (2)| 00:00:26 |
|*  1 |  TABLE ACCESS FULL| CUSTOMERS |   500K|    33M|  2140   (2)| 00:00:26 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------
   1 - filter("CUSTOMERS"."SYS_STUJ6BPFDTE396EPTURAB2DBI5"='MCDONALD')
```

Extended stats name come here

expl2.sql

Alternatives

- Remember, the extended stats create a virtual column – hidden from you
- You can have the same functionality as extended stats by defining virtual columns
- Advantage
  - You can have a column name of your choice
  - You can index it, if needed
  - You can partition it
  - You can create Foreign Key constraints on it

Restrictions

- Has to be 11.0 or higher
- Not for SYS owned tables
- Not on IOT, clustered tables, GTT or external tables
- Can't be on a virtual column
- An Expression
  - can't contain a subquery
  - must have ≥1 columns
- A Column Group
  - no of columns should be ≤32 and ≥2
  - can't contain expressions
  - can't have the same column repeated

Summary

- Normally the optimizer does not know the correlation between the columns
  - e.g. no one born in January can have a sun sign of Pisces
  - Therefore, perform an index scan if that combination is passed as predicate
- Extended statistics enable the optimizer to know that relationship
- More information to the optimizer
- … results in better plans