Stats with Confidence

Arup Nanda

Starwood Hotels

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*

cr_sales.sql
sales_stats.sql
count.sql
explct.sql
explny.sql
upd.sql
prefs_false.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
```

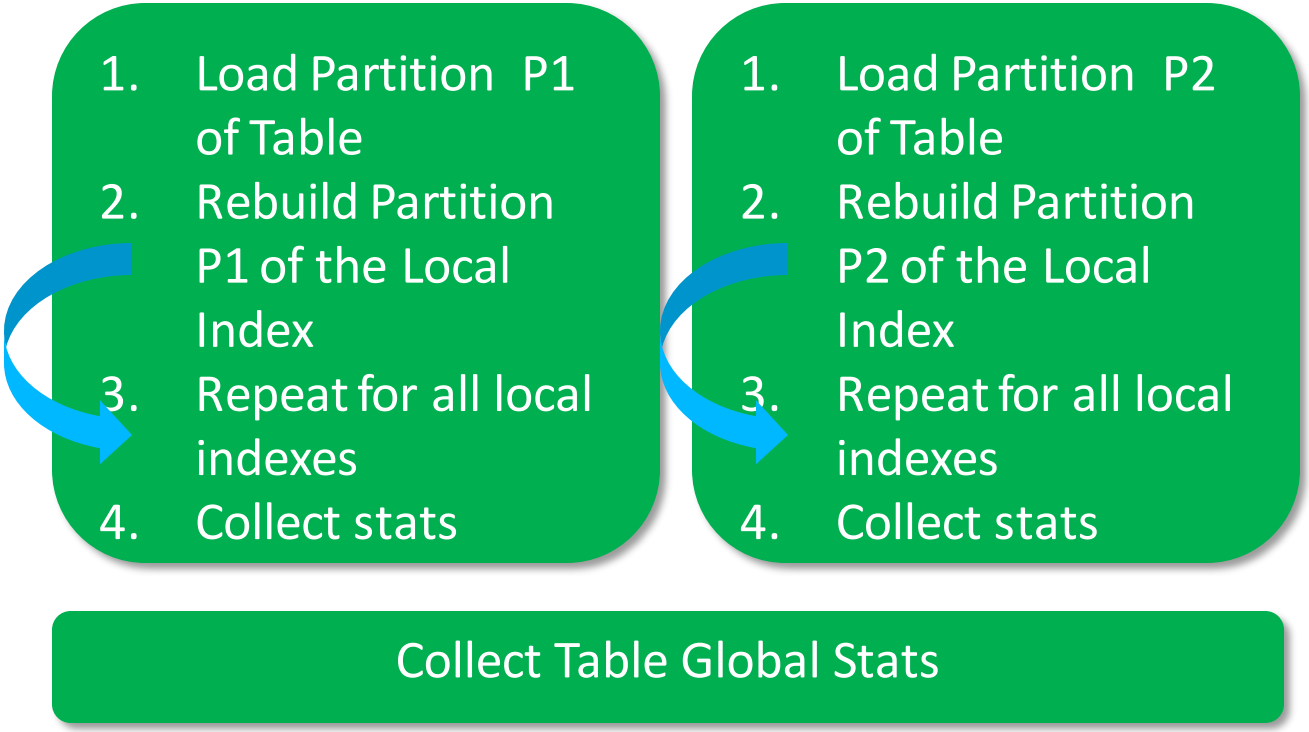
Other Preferences

- 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

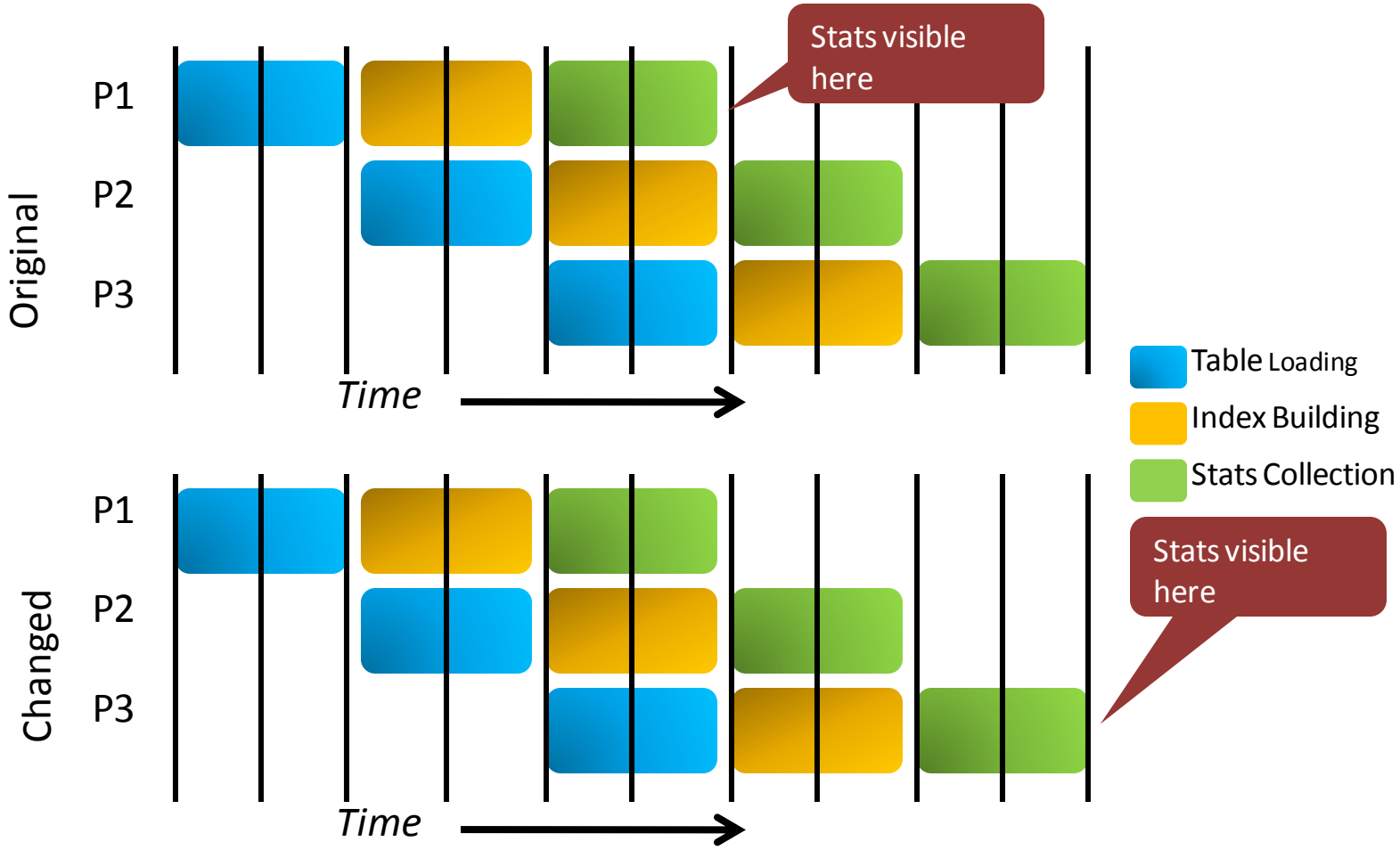


1. You may want to make sure that the final table global stats are collected after all partition stats are gathered
2. And all are visible to CBO at the same time

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

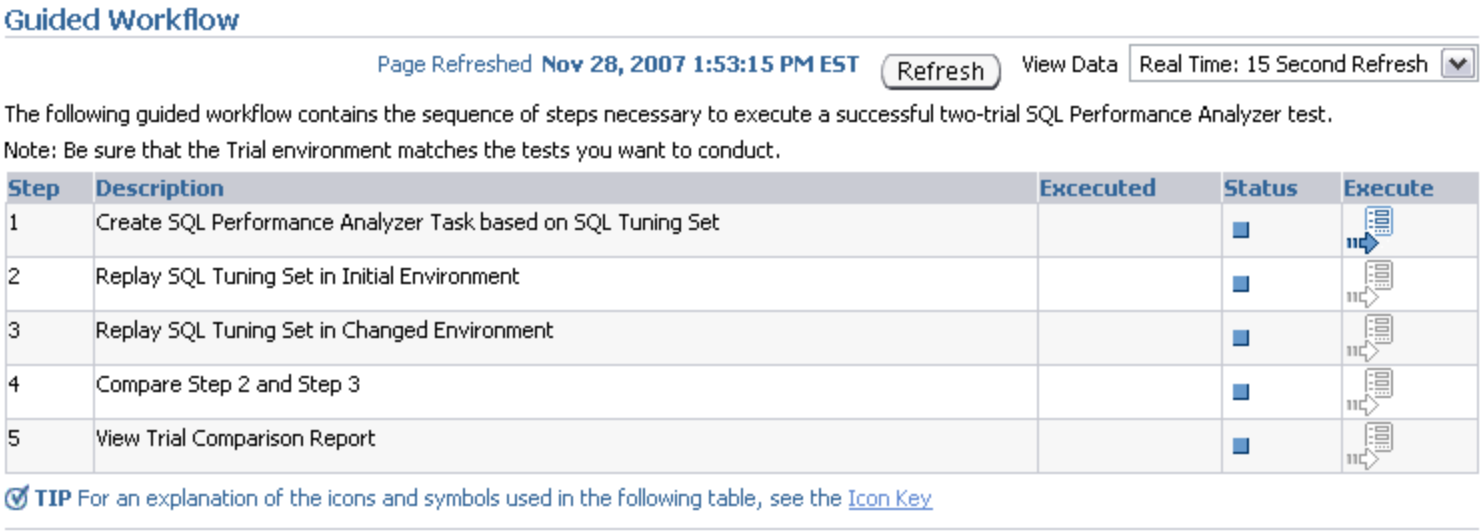
```
begin
   dbms_stats.export_pending_stats (
     tabname    => 'SALES',
     stattab    => 'STAT_TABLE'
   );
end;
```

- 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

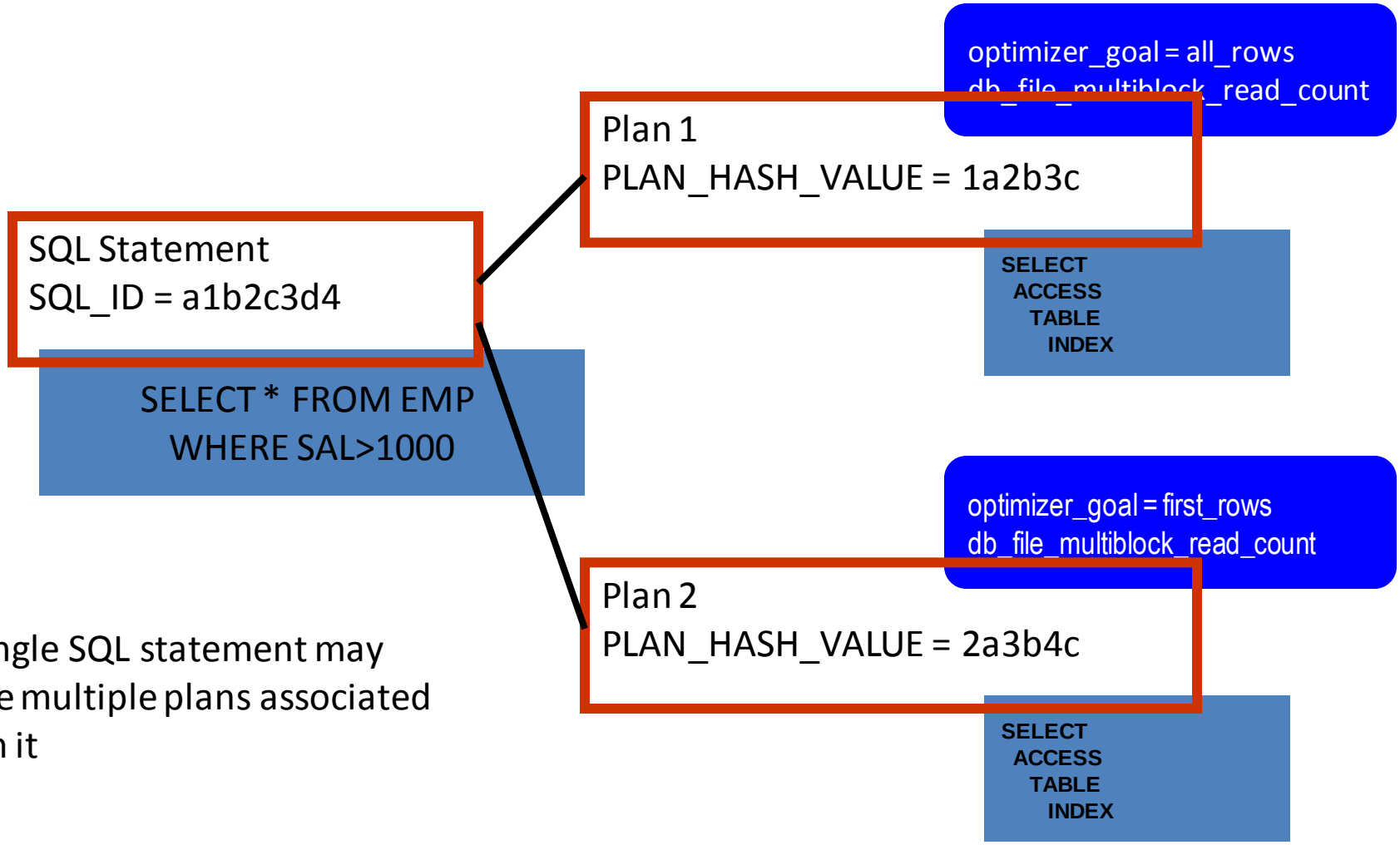
- You can use Database Replay and SQL Performance Analyzer to recreate the production workload
- But under the *pending* stats, to see the impact
- In SPA use `alter session set optimizer_use_pending_statistics = true;`
- That way you can predict the impact of the new stats with your specific workload



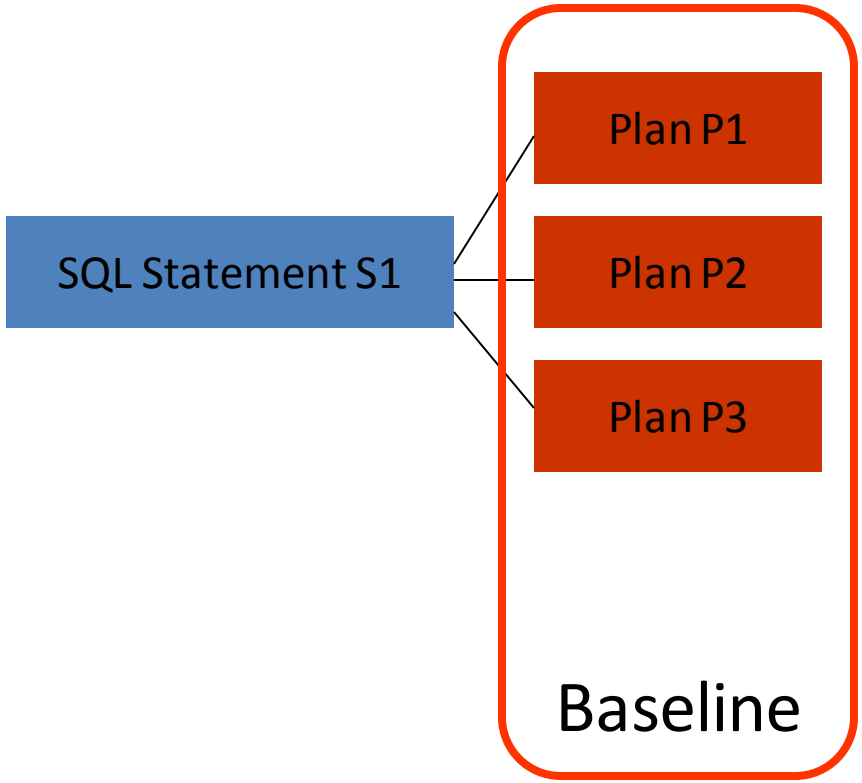
Some additional uses

- 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

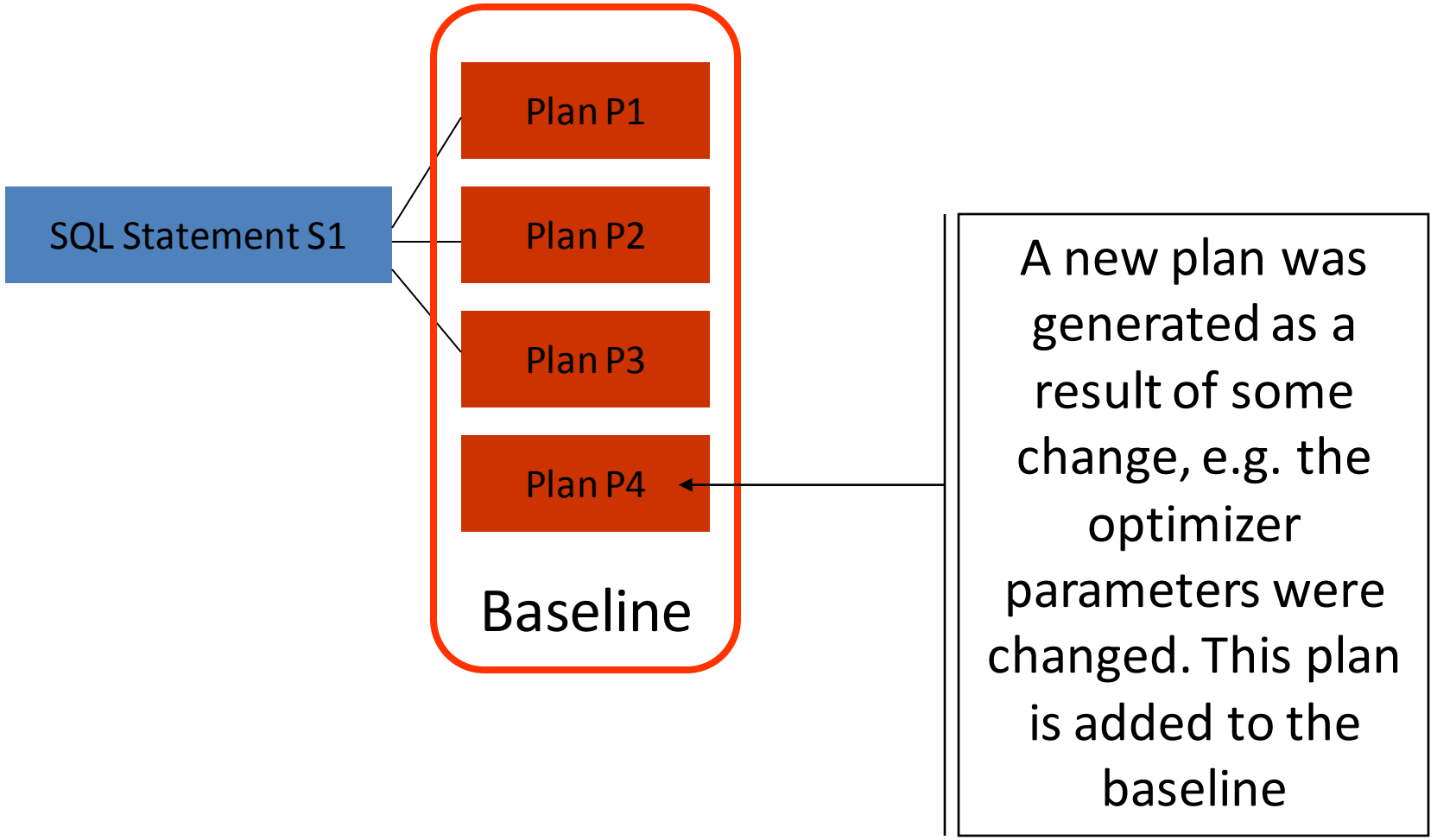
SQL Plan Management

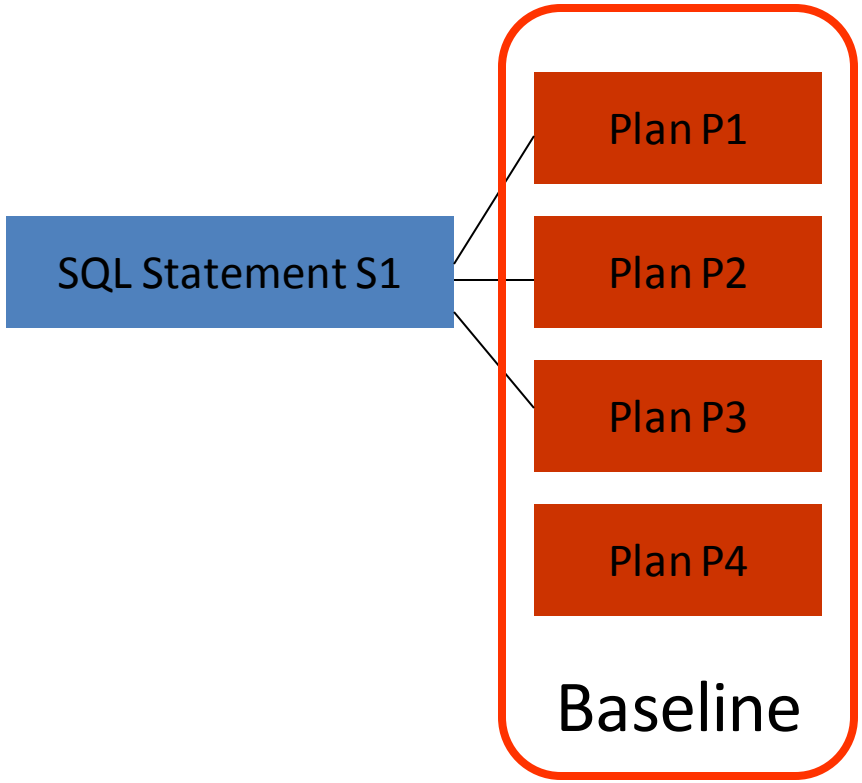


A single SQL statement may have multiple plans associated with it

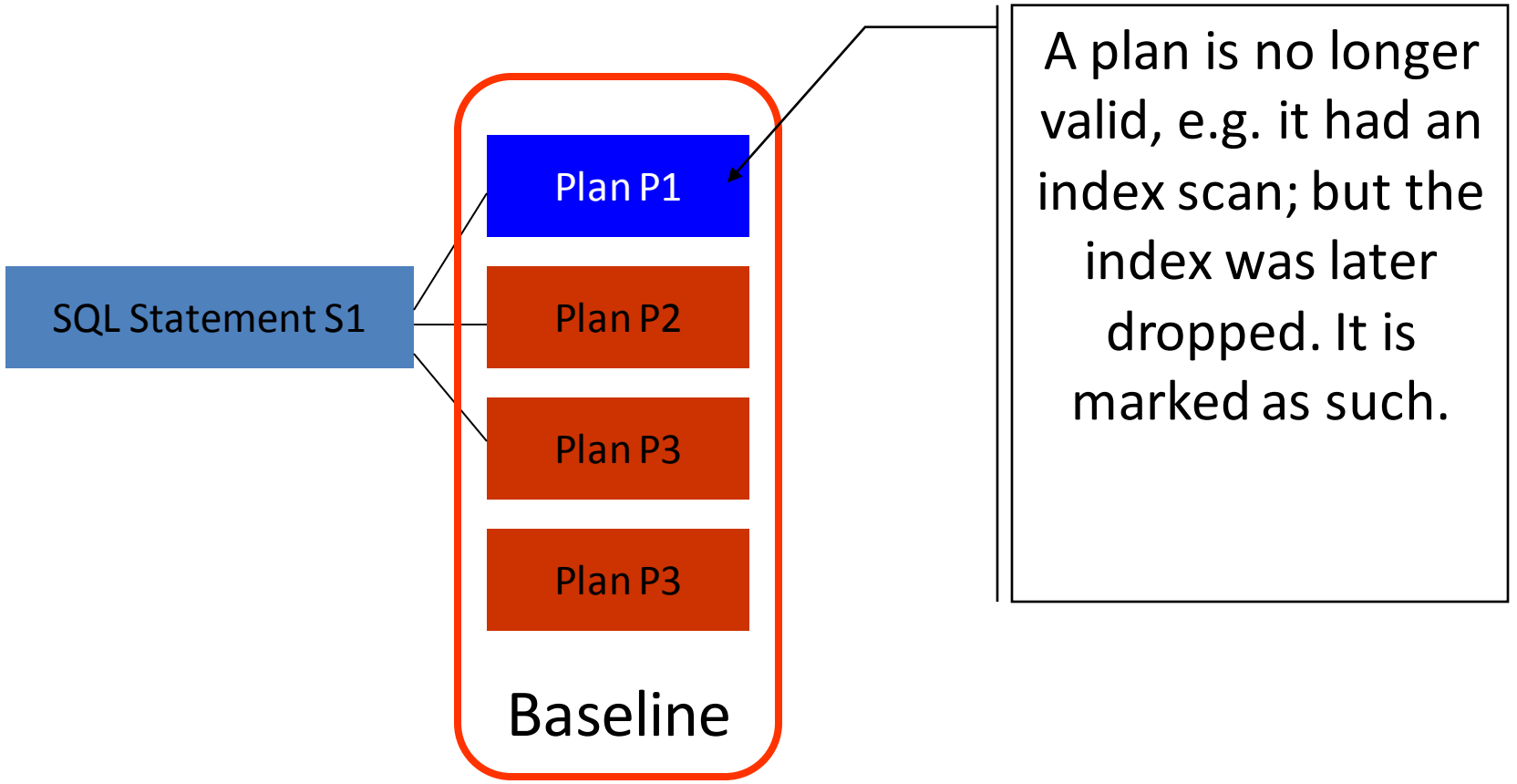


A baseline is a collection of plans for a specific SQL statement


SQL Statement S1
Plan P1
Plan P2
Plan P3
Plan P4
Baseline
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

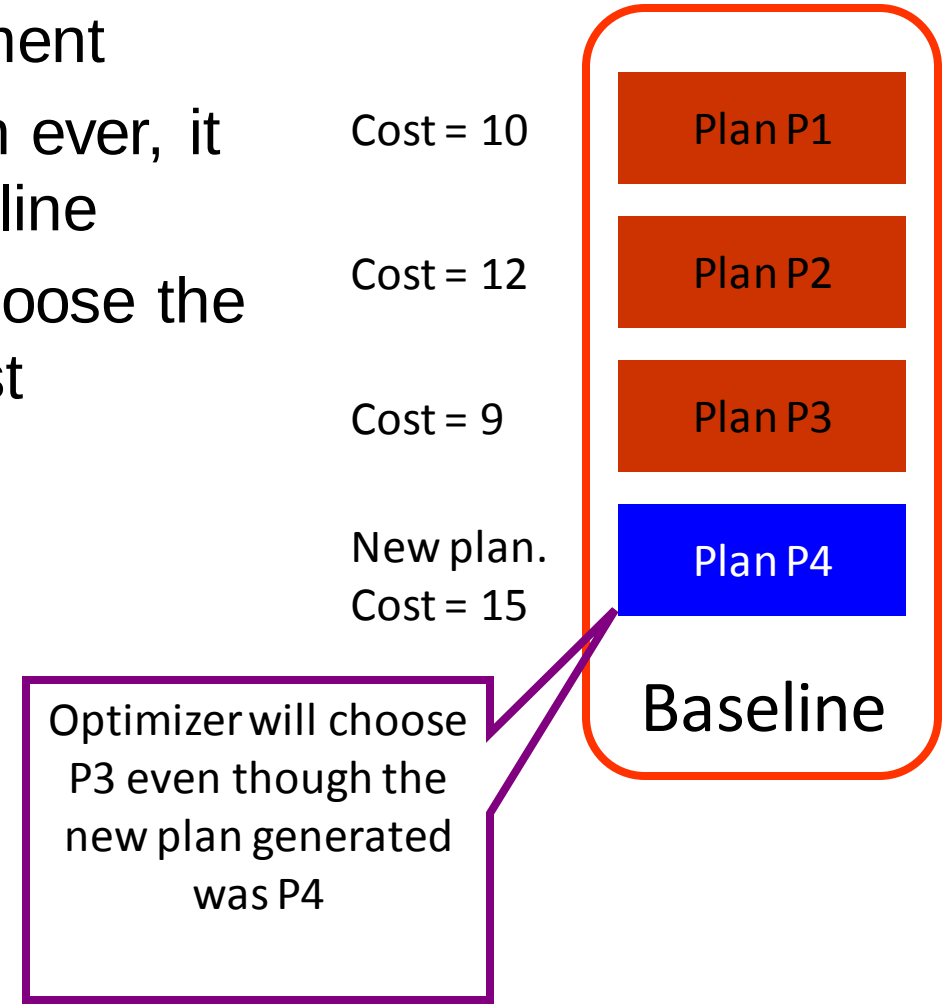


When a SQL is reparsed, the optimizer compares the plan to the list of plans in the baseline, but **not the newly generated plan** as it is not “accepted”.


SQL Statement S1
Plan P1
Plan P2
Plan P3
Plan P3
Baseline
A plan is no longer valid, e.g. it had an index scan; but the index was later dropped. It is marked as such.

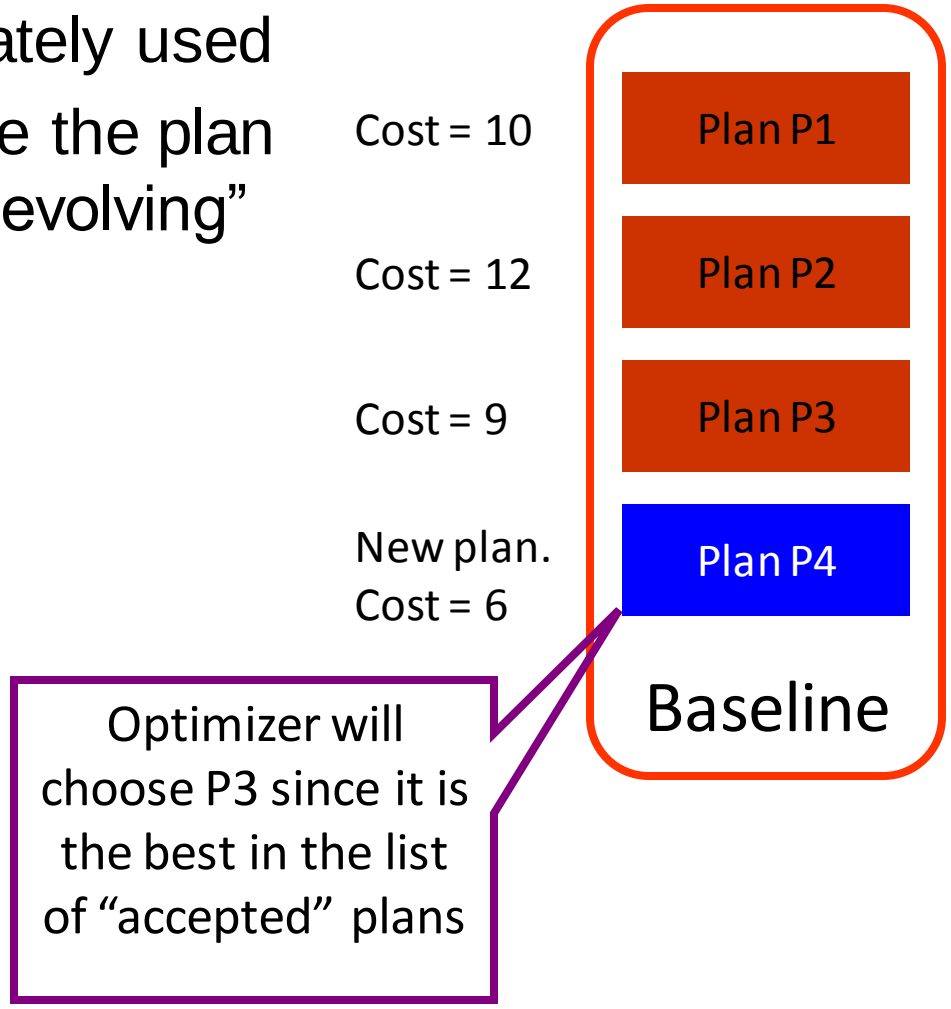
New Plan is Worse

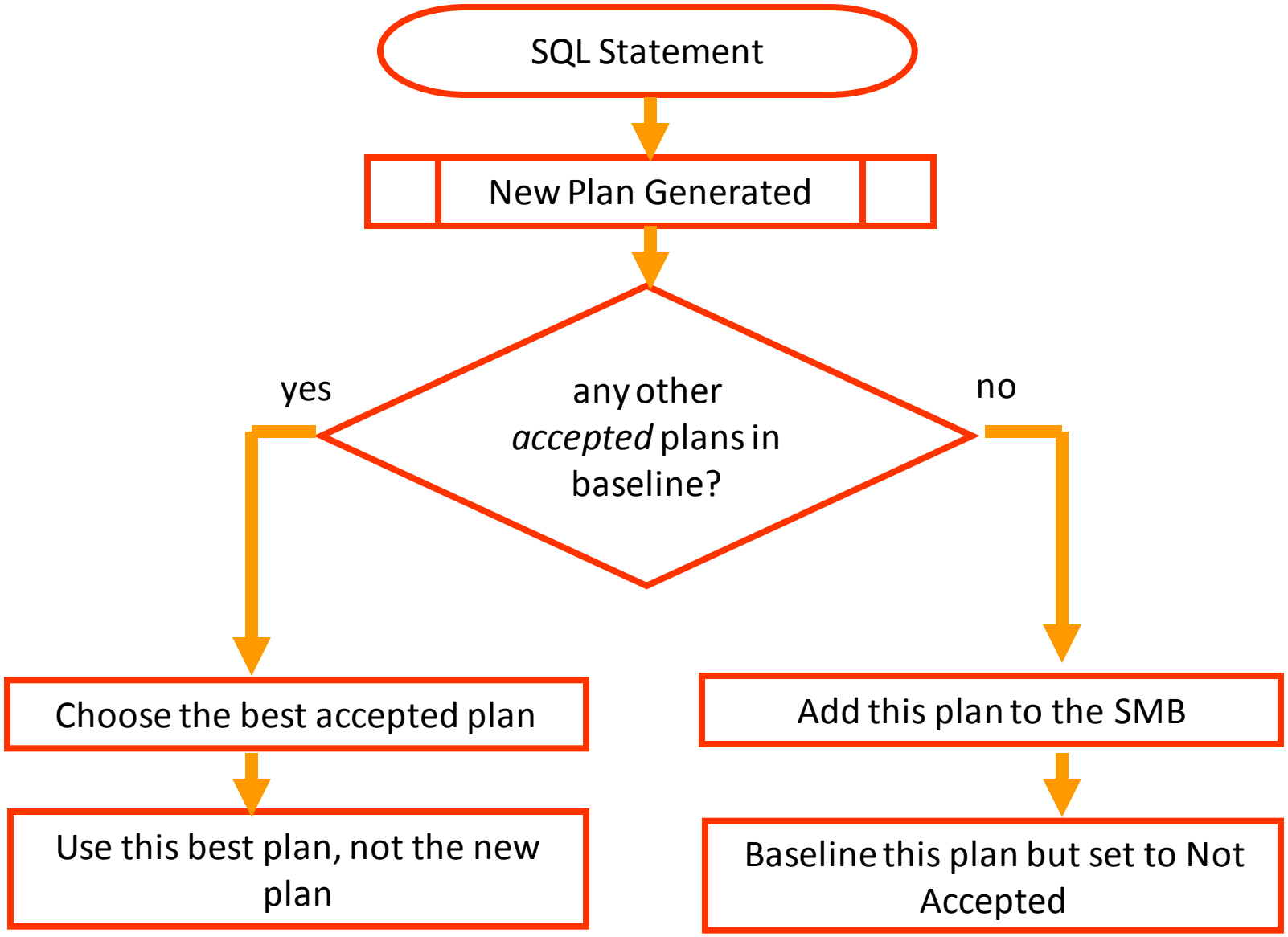
- Baselines contain the history of plans for an SQL statement
- If there was a good plan ever, it will be there in the baseline
- So the optimizer can choose the plan with the lowest cost



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 Statement
New Plan Generated
any other *accepted* plans in baseline?
yes
no
Choose the best accepted plan
Use this best plan, not the new plan
Add this plan to the SMB
Baseline this plan but set to Not Accepted

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

Cap_true.sql

Cap_false.sql_

Plans With Pending Stats

- 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 upper(SQL_TEXT) like '%AVG(SALES_AMT)%'
/

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

bl1.sql_

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')
    )
```

Handle1.sql_

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.

Evolve1.sql_

Fixing a Plan

- 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'`

Fix.sql_

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


감사합니다
Obrigado
Спасибо
धन्यवाद
Danke
תודה רבה
Thank you!
多謝
Grazie
Thank You
Merci
ขอบคุณ
நன்றி
شكرا
Gracias
ありがとうございました