# Getting The Best From The Cost Based Optimizer


Jože Senegačnik

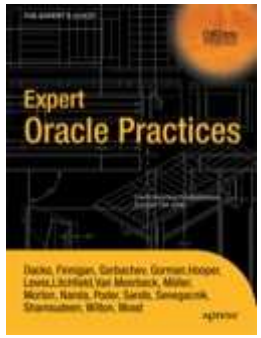
joze.senegacnik@dbprof.com

# About the Speaker

Jože Senegačnik

- Owner of Dbprof d.o.o.
- First experience with Oracle Version 4.1 in 1988
- 21+ years of experience with Oracle RDBMS.
- Proud member of the OakTable Network www.oaktable.net
- Oracle ACE Director
- Co-author of the OakTable book “Expert Oracle Practices” by Apress (Jan 2010)
- VP of Slovenian OUG (SIOUG) board
- CISA – Certified IS auditor
- Blog about Oracle: http://joze-senegacnik.blogspot.com




- PPL(A) – private pilot license
- Blog about flying: http://jsenegacnik.blogspot.com
- Blog about Building Ovens, Baking and Cooking: http://senegacnik.blogspot.com

# Agenda

1. Misused Initialization Parameters & System Statistics
2. Extended Statistics
3. Cost of execution of PL/SQL functions
4. Constraints
5. SQL Plan Management
6. SQL Monitoring
7. Automatic Cardinality Feedback Tuning

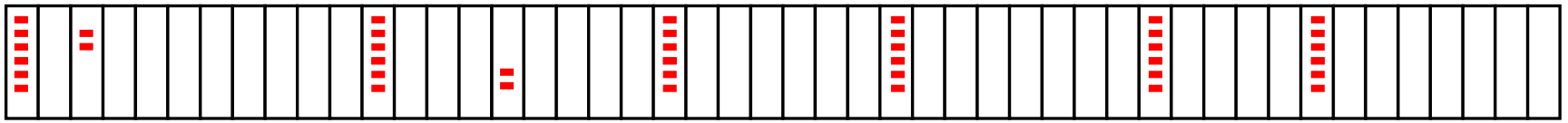
# Misused Initialization Parameters

- There are parameters for which people think that they can do the MAGIC:
  - OPTIMIZER_INDEX_COST_ADJ
- Or are never set and left at default value:
  - OPTIMIZER_INDEX_CACHING

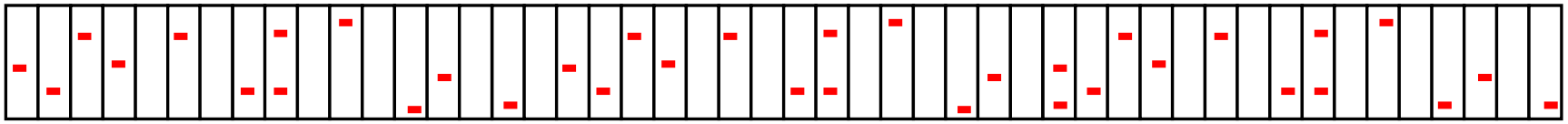
- OPTIMIZER_INDEX_COST_ADJ was introduced in 8i to help CBO using index access paths versus FTS
- People still think one should set it in versions >=9i
- Let us look some details

# Problem of Data Distribution

## Rows are clustered



## Rows are spread across the table



When CBO detects that it would perform more IO using an index then by performing a FTS it decides to use full table scan (FTS)

# Index Selectivity

- Row selectivity
  - Defines how many rows are retrieved.
  - From number of retrieved rows one cannot conclude how many blocks we will have to visit.
- Block selectivity – actually used by CBO
  - How many blocks are visited to retrieve the rows that pass our where condition.
  - We can use here the ROWID pseudo column to determine the file and block.

```
select count(distinct substr(rowid,7,9)) blocks
from &table where &query_condition;
```

# OPTIMIZER_INDEX_COST_ADJ

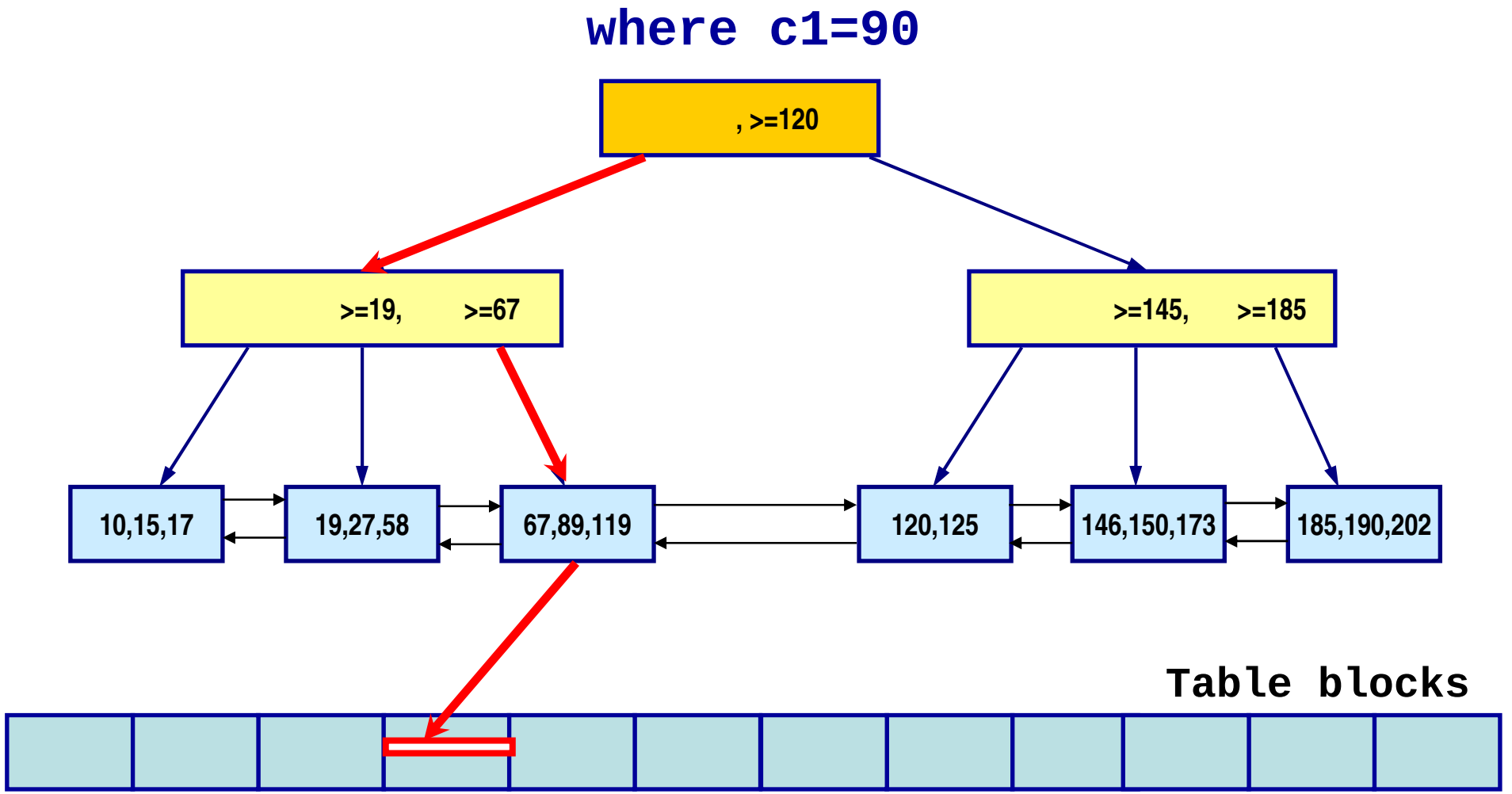
- Initialization parameter OPTIMIZER_INDEX_COST_ADJ virtually lowers the cost of index access.
- In Oracle 8i it was used to distinguish between the cost of a single block read versus multi block read.
- Lowering the cost of index access makes index access path the more attractive.
- Since introduction of the System Statistics CBO is able do distinguish between single and multi block I/O.

- Therefore the parameter should be left at default of 100.

# OPTIMIZER_INDEX_CACHING

- Defines the amount of index blocks that are expected to be present in the buffer cache – no physical read is required. It adjusts the behavior of optimization to favor nested loops joins and IN-list iterators.
- Indexes for nested loops join probes are frequently cached and we should tell this to CBO!
- Default value of this init parameter is **0**.
- Set it to the value of > 80 or to the value of your Buffer Cache Hit Ratio (BCHR) .

# Index Unique Scan



# Index Unique Scan Cost

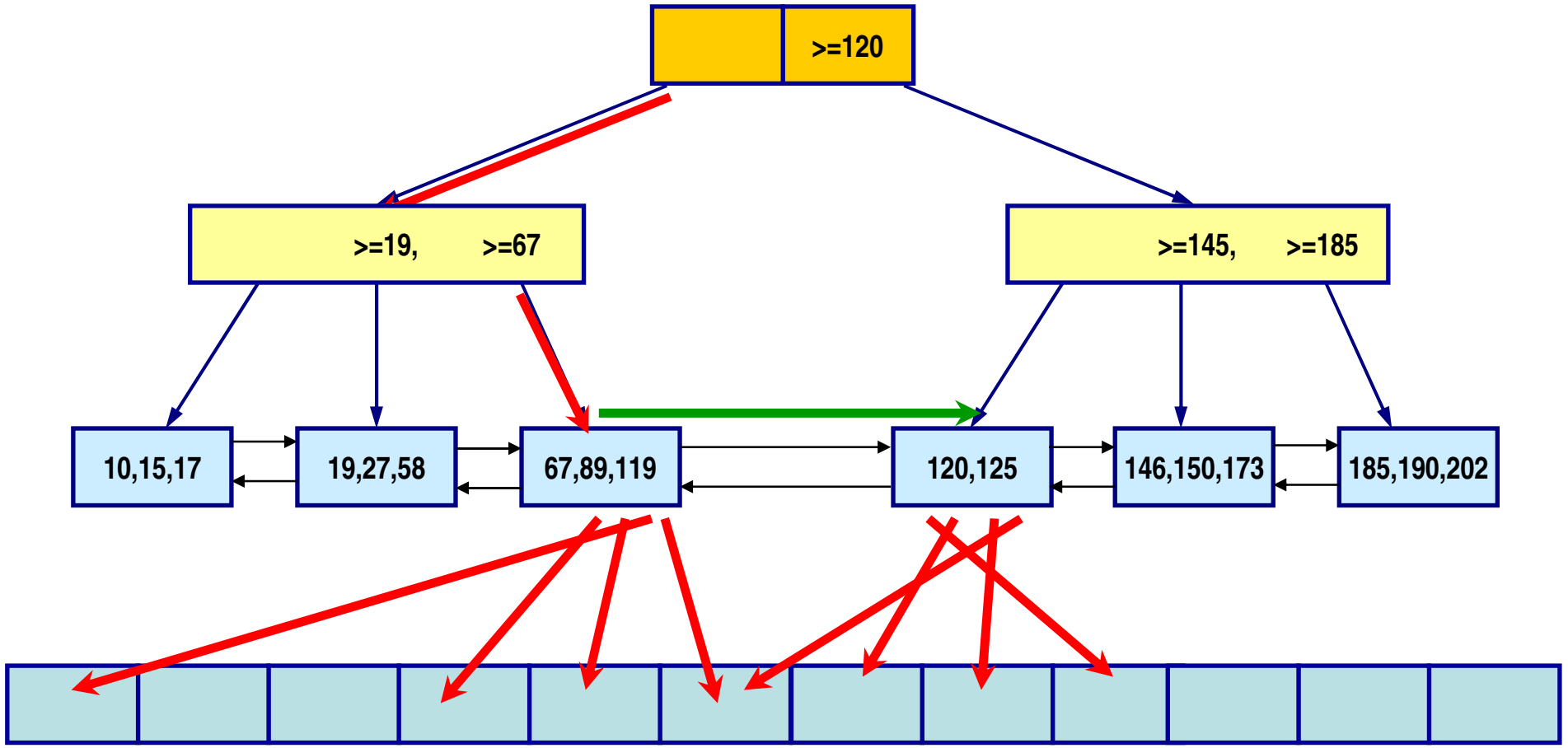
INDEX UNIQUE SCAN COST = (BLEVEL (1-(OIC/100)) + 1) * (OICA/100)

** formula does not include the CPU cost*

- BLEVEL = number of branch levels in index
- add +1 for leaf block
- FF = filtering factor - selectivity
- LFBL = number of leaf blocks
- CLUF = index clustering facto
- OIC = optimizer_index_caching
- OICA = optimizer_index_cost_adj parameter (default=100)

# Index Range Scan

**`where c1 between 90 and 123`**



# Index Range Scan Cost

INDEX RANGE SCAN COST = (BLEVEL + FF*LFBL)*(1-(OIC/100))
+ FF*CLUF)
* (OICA/100)

- BLEVEL = number of branch levels in index
- FF = filtering factor - selectivity
- LFBL = number of leaf blocks
- CLUF = index clustering factor
- OIC = optimizer_index_caching
- OICA = optimizer_index_cost_adj parameter (default=100)

# System Statistics

# System Statistics Usage Quiz

- How many of you are using WORKLOAD system statistics?

# System Statistics

- System statistics is the only actual information about the HW properties that can be used by the CBO for optimization of SQL statements.
- Available since version Oracle 9i.
- Enables CPU Costing.
- Can be used for tuning.
- Workload/Noworkload Statistics
- **My recommendation is that the Workload Statistics is used which is gathered during typical workload.**
- See Randolf Geist blog about system statistics.

# Conversion Formula

- The new cost model in 9i/10g requires system statistics.
- Conversion from CPU cost units  to I/O units:

```
COST = CPU-RSC / (1000*CPUSPEED*SREADTIM)
```

CPU-RSC  = CPU cost

CPUSPEED = CPU speed from system statistics

SREADTIM = single block read time from system statistics

# Cost of a Full Table Scan (2)

- If you set system statistics:
  - cpuspeed and sreadtim must be set for cpu_costing to become effective.
  - mreadtim, mbrc and sreadtim must be set for access path costing of multiblock paths (table scan, fast full scan) to use the system statistics.

# Cost of a Full Table Scan – NOWORKLOAD (3)

From a CBO trace file (event 10053):

```
*******************************
SYSTEM STATISTICS INFORMATION
*******************************
  Using NOWORKLOAD Stats
  CPUSPEED: 1042 millions instruction/sec
  IOTFRSPEED: 4096 bytes per millisecond (default is 4096)
  IOSEEKTIM: 10 milliseconds (default is 10)
***************************************
BASE STATISTICAL INFORMATION
***********************
Table Stats::
  Table: T3  Alias: T3  (Using composite stats)
    #Rows: 918843  #Blks:  15056  AvgRowLen:  114.00
Index Stats::
  Index: T3_I1  Col#: 1
    LVLS: 2  #LB: 2430  #DK: 918843  LB/K: 1.00  DB/K: 1.00  CLUF: 810972.00
***************************************
SINGLE TABLE ACCESS PATH
  Table: T3  Alias: T3
    Card: Original: 918843  Rounded: 918843  Computed: 918843.00  Non Adjusted: 918843.00
  Access Path: TableScan
    Cost:  4101.54  Resp: 4101.54  Degree: 0
    Cost_io: 4079.00  Cost_cpu: 281800571
```

**Default NOWORKLOAD statistics is USED**

# Cost of a Full Table Scan (4)

From a CBO trace file (event 10053):

```
*******************************
SYSTEM STATISTICS INFORMATION
*******************************
  Using WORKLOAD Stats
  CPUSPEED: 1000 millions instructions/sec
  SREADTIM: 2 milliseconds
  MREADTIM: 4 millisecons
  MBRC: 8.000000 blocks
  MAXTHR: 1000000 bytes/sec
  SLAVETHR: -1 bytes/sec
***************************************
BASE STATISTICAL INFORMATION
***********************
Table Stats::
  Table: T3  Alias: T3  (Using composite stats)
    #Rows: 918843  #Blks:  15056  AvgRowLen:  114.00
Index Stats::
  Index: T3_I1  Col#: 1
    LVLS: 2  #LB: 2430  #DK: 918843  LB/K: 1.00  DB/K: 1.00  CLUF: 810972.00
***************************************
SINGLE TABLE ACCESS PATH
  Table: T3  Alias: T3
    Card: Original: 918843  Rounded: 918843  Computed: 918843.00  Non Adjusted: 918843.00
  Access Path: TableScan
    Cost:  3905.90  Resp: 3905.90  Degree: 0
 Cost_io: 3765.00  Cost_cpu: 281800571
```

**WORKLOAD statistics is USED**

# Extended statistics

# Object Statistics and Histograms

- If histogram is gathered on a column, where there should be no histogram, then don't let the system to gather it.
- If the number of buckets in histogram is not the appropriate one and the statistics should be gathered with a different number of histograms, then this rule should be applied for all future gatherings.
- Use extended statistics on several columns in all cases when the columns are dependent on each other.

# Extended statistics

- Let us create a table with two columns where both columns have the same values in all rows – high column dependency

```
SQL> create table t2 as
     select trunc(rownum/300)-1 as c1,
            trunc(rownum/300)-1 as c2
     from dual connect by level <= 100000;

Table created.
```

# Execution Without Extended Statistics

```
SQL> explain plan for select * from t2 where c1=5 and c2=5;

Explained.

SQL> select * from table(dbms_xplan.display);

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 1513984157

-----------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |   876 | 22776 |    52   (2)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T2   |   876 | 22776 |    52   (2)| 00:00:01 |
-----------------------------------------------------------------------------

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

   1 - filter("C1"=5 AND "C2"=5)

Note
   - dynamic sampling used for this statement (level=2)
```

# Gather Statistics

```
SQL> exec dbms_stats.gather_table_stats(
      ownname=>user,
      tabname=>'T2',
      method_opt=>'for all columns size skewonly');

PL/SQL procedure successfully completed.

SQL> select column_name,num_distinct,histogram
     from user_tab_col_statistics
     where table_name='T2' order by 1;
```

```
COLUMN_NAME                    NUM_DISTINCT HISTOGRAM
------------------------------ ------------ ---------------
C1                                      334 HEIGHT BALANCED
C2                                      334 HEIGHT BALANCED
```

```
SQL> select count(*) from t2 where c1=5 and c2=5;
```

```
  COUNT(*)
----------
       300
```

# Execution With Gathered Statistics

```
SQL> explain plan for select * from t2 where c1=5 and c2=5;

Explained.

SQL> select * from table(dbms_xplan.display);

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 1513984157

-----------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |     8 |    52   (2)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T2   |     1 |     8 |    52   (2)| 00:00:01 |
-----------------------------------------------------------------------------

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

   1 - filter("C1"=5 AND "C2"=5)
```

# Creating Extended Statistics

- Creating extended statistics

```
SQL> select
     dbms_stats.create_extended_stats(user,'T2','(c1,c2)')
     from dual;

DBMS_STATS.CREATE_EXTENDED_STATS(USER,'T2','(C1,C2)')
-----------------------------------------------------
SYS_STUF3GLKIOP5F4B0BTTCFTMX0W
```

# Gathering Extended Statistics

```
SQL> exec dbms_stats.gather_table_stats(
          ownname=>user,tabname=>'T2',
          method_opt=>'for all columns size skewonly',
          estimate_percent=>null);

PL/SQL procedure successfully completed.

SQL> select column_name,num_distinct,histogram
     from user_tab_col_statistics
     where table_name='T2' order by 1;

COLUMN_NAME                    NUM_DISTINCT HISTOGRAM
------------------------------ ------------ ---------------
C1                                      334 HEIGHT BALANCED
C2                                      334 HEIGHT BALANCED
SYS_STUF3GLKIOP5F4B0BTTCFTMX0W          334 HEIGHT BALANCED
```

# Execution With Extended Statistics

```
SQL> explain plan for select * from t2 where c1=5 and c2=5;

Explained.

SQL> select * from table(dbms_xplan.display);

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 1513984157

------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |   299 |  2392 |    52   (2)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T2   |   299 |  2392 |    52   (2)| 00:00:01 |
------------------------------------------------------------------------------

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

   1 - filter("C1"=5 AND "C2"=5)
```

# Defining Selectivity And Cost For PL/SQL Functions

# CBO's Defaults For Functions

```
select * from t
where c2 between 20 and 49
and my_func(c2)=0;
```

Default values:

| | |
|---|---|
| Selectivity | 1% (0.01) |
| CPU cost | 3000 |
| IO cost | 0 |
| Network cost | 0 |

# Demo Function Used in Where Clause

```
create or replace function my_func
   (p1 number)
   return number
is
begin
     return 0;
end;
/
```

# Default Cost Definition

- The cost for a *single execution* is defined by: CPU, I/O and NETWORK cost.
  - **CPU cost** value is represented with the number of machine cycles executed by the function or domain index implementation.
  - One can estimate the number of machine cycles with the package function **DBMS_ODCI. ESTIMATE_CPU_UNITS**.
  - **I/O cost** value is the number of data blocks read by the function or domain index implementation.
  - **NETWORK cost** – this value is currently not used. It represents the number of data blocks transmitted to the network.

# Using My_func in Where Clause

```
SQL> explain plan for
select * from t
where c2 between 20 and 49
and my_func(c2)=0;
```

```
SQL> select * from table(dbms_xplan.display);

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 1601196873
------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |  2996 |   447K|  3068   (2)| 00:00:37 |
|*  1 |  TABLE ACCESS FULL| T    |  2996 |   447K|  3068   (2)| 00:00:37 |
------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("C2">=20 AND "MY_FUNC"("C2")=0 AND "C2"<=49)
```

# Defining Default Selectivity and Cost

```
SQL> associate statistics with functions
       my_func
       default selectivity 10,
       default cost (1000000 /* CPU */,
                     1000 /* I/O */,
                     0  /* network */);

Statistics associated.
```

# Changed Execution Plan

```
SQL> explain plan for
     select * from t
     where c2 between 20 and 49
     and my_func(c2)=0;

Explained.
```

```
SQL> select * from table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 729576041
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             | 29961 |  4476K|    30M  (1)|100:22:27 |
|   1 |  TABLE ACCESS BY INDEX ROWID| T           | 29961 |  4476K|    30M  (1)|100:22:27 |
|*  2 |   INDEX SKIP SCAN            | T_I_COMBINE | 29961 |       |    30M  (1)|100:22:19 |
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("C2">=20 AND "C2"<=49)
       filter("C2">=20 AND "MY_FUNC"("C2")=0 AND "C2"<=49)
```

# CPU Cost Estimation

```
SQL> begin
  2    :a := 1000*dbms_odci.estimate_cpu_units(0.002);
  3  end;
  4  /

PL/SQL procedure successfully completed.
```

```
SQL>print a

         A
----------
3908814,85
```

# Optimization Facts

- It is important to remember the following **three rules**:
  1. The execution of functions which are very costly in terms of CPU usage or perform a lot of I/O should be postponed as much as possible in order to be executed on the smallest possible set of rows.
  2. More selective functions – those which will filter out more rows – will be executed first because they will filter out many rows in the very first steps of execution.
  3. If neither default selectivity nor default cost are defined then the functions will be executed in the order as they appear in the text of the SQL statement.
- One can disassociate statistics from with the command `DISASSOCIATE STATISTICS FROM`.

# Constraints

# Test Table T1

```
CREATE TABLE JOC.T1
(
  ID  NUMBER,
  C1  VARCHAR2(30 BYTE),
  C2  VARCHAR2(1 BYTE),
  C3  DATE,
  C4  NUMBER,
  C5  NUMBER
)
TABLESPACE USERS;
```

```
ALTER TABLE JOC.T1 ADD ( CONSTRAINT C4_CHECK CHECK (c4 in (10,20,30))
   DISABLE);
```

```
...
populate the table
```

# Constraint Disabled / No histogram

```
SQL> select c4,count(*) from t1 group by c4;

        C4   COUNT(*)
---------- ----------
        30         20
        20         80
        10        900

SQL> explain plan for select * from t1 where c4 > 30;
```

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |   333 |  7659 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T1   |   333 |  7659 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - filter(C4>30)
```

# Enabled Constraint

```
SQL>ALTER TABLE JOC.T1 ENABLE CONSTRAINT C4_CHECK;

Table altered.

SQL>explain plan for select * from t1 where c4 >= 35 and c4 <= 36;

Explained.
```

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    28 |     0   (0)|          |
|*  1 |  FILTER            |      |       |       |            |          |
|*  2 |   TABLE ACCESS FULL| T1   |    10 |   280 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - filter(NULL IS NOT NULL)
   2 - filter(C4>=35 AND C4<=36)
```

# Cardinality Misestimates (1)

- Let us look how good the CBO is in “understanding” the constraints. If we write a simple equality condition as c4 = 30 we get the exact answer.
- We have check constraint on column “c4 in (10,20,30)” and the frequency histogram

```
explain plan for select * from t1 where c4 =  30;
```

```
----------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |    20 |   640 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T1   |    20 |   640 |     3   (0)| 00:00:01 |
----------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("C4"=30)
```

# Cardinality Misestimates (2)

```
explain plan for select * from t1 where c4 >=  30;
```

```
----------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |    19 |   532 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T1   |    19 |   532 |     3   (0)| 00:00:01 |
----------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("C4">=30)
```

# Cardinality Misestimates (3)

```
explain plan for select * from t1 where c4 = 26;
```

```
----------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    32 |     0   (0)|          |
|*  1 |  FILTER            |      |       |       |            |          |
|*  2 |   TABLE ACCESS FULL| T1   |    10 |   320 |     3   (0)| 00:00:01 |
----------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(NULL IS NOT NULL)
   2 - filter("C4"=26)
```

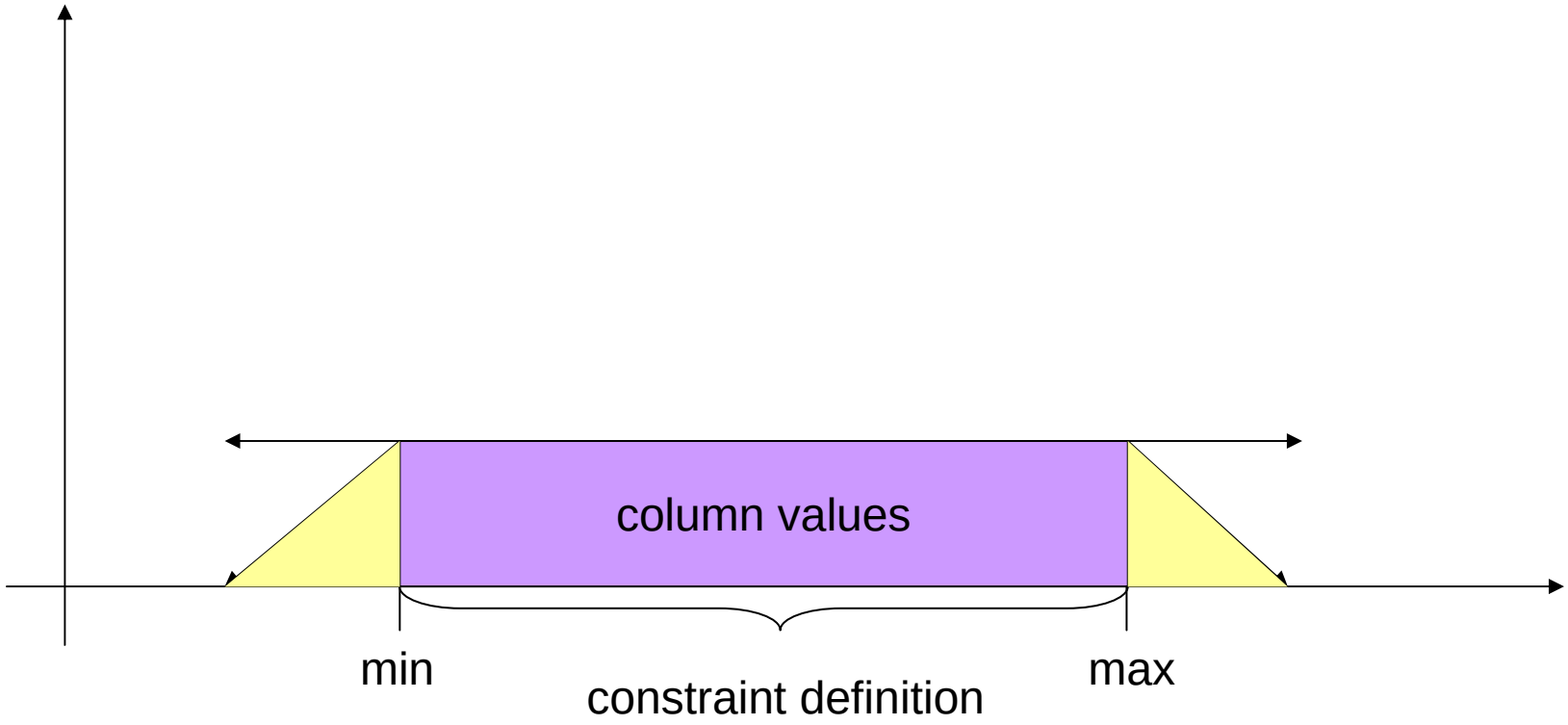
# Cardinality Misestimates (4)

```
explain plan for select * from t1 where c4 > 25 and c4 < 27;
```

```
-----------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |    10 |   320 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T1   |    10 |   320 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("C4">25 AND "C4"<27)
```

- complex expression – not evaluated

# Cardinality Misestimates (5)



# Join Elimination (1)

- Eliminate unnecessary joins if there are constraints defined on join columns. If join has no impact on query results it can be eliminated.
  - e.departmens_id is foreign key and joined to primary key d.department_id
- Eliminate unnecessary outer joins – doesn't even require primary key – foreign key relationship to be defined.

```
SQL> select e.first_name, e.last_name, e.salary
     from employees e,
          departments d
     where e.department_id = d.department_id;
```

```
-------------------------------------+-----------------------------------+
| Id  | Operation          | Name     | Rows  | Bytes | Cost  | Time      |
-------------------------------------+-----------------------------------+
| 0   | SELECT STATEMENT   |          |       |       |     3 |           |
| 1   |  TABLE ACCESS FULL | EMPLOYEES|   106 |  2332 |     3 |  00:00:01 |
-------------------------------------+-----------------------------------+
Predicate Information:
----------------------
1 - filter("E"."DEPARTMENT_ID" IS NOT NULL)
```

# Join Elimination (2)

- **Join elimination is one of possible query transformations.**

- **Purpose of join elimination**
  - Usually people don't write such "stupid" statements directly
  - Such situations are very common when a view is used which contains a join and only a subset of columns is used.

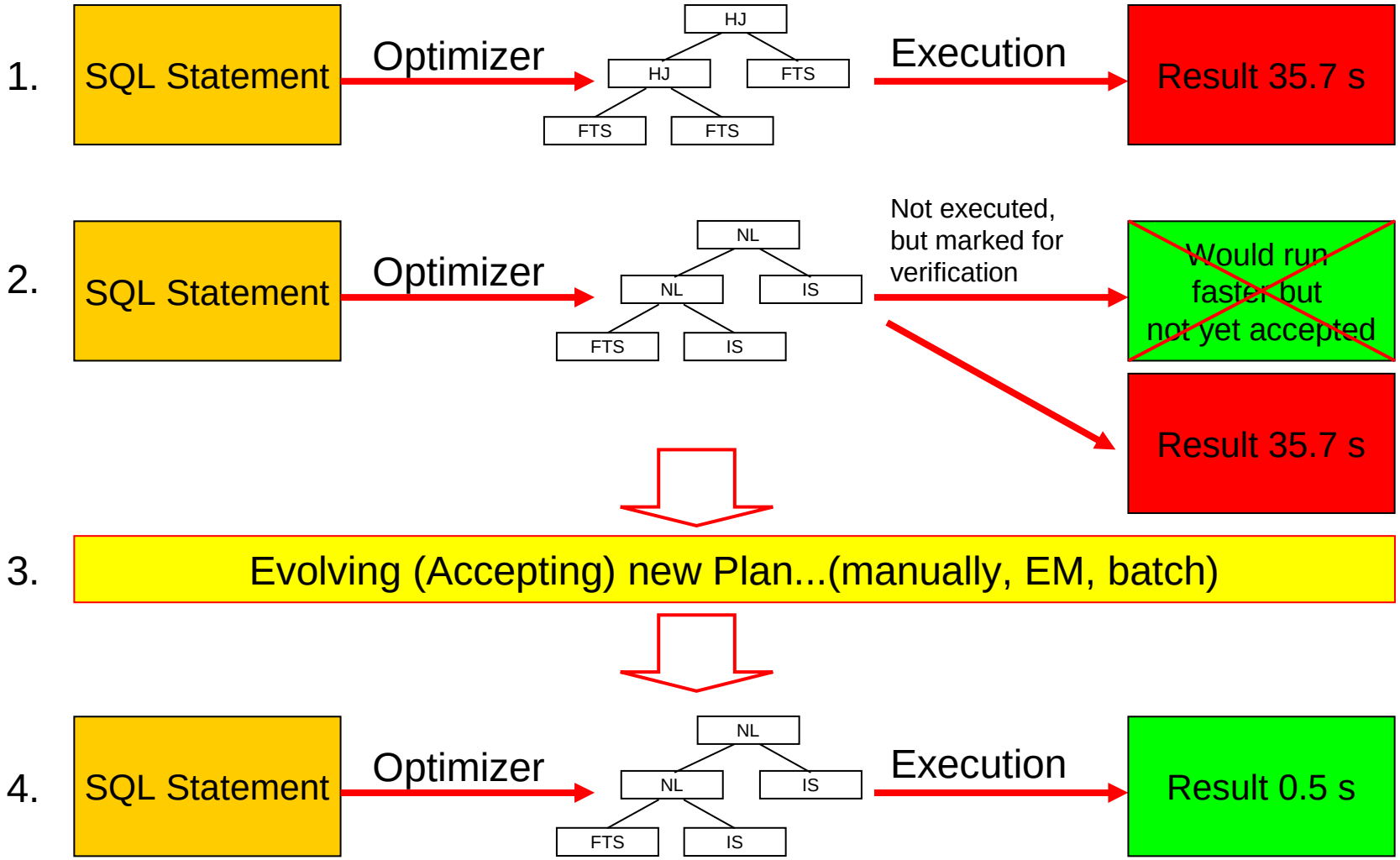
- **Known Limitations** (Source: Optimizer group blog)
  - Multi-column primary key-foreign key constraints are not supported.
  - Referring to the join key elsewhere in the query will prevent table elimination. For an inner join, the join keys on each side of the join are equivalent, but if the query contains other references to the join key from the table that could otherwise be eliminated, this prevents elimination. A workaround is to rewrite the query to refer to the join key from the other table.

# SQL Plan Management

# SQL Plan Management

- SQL Plan Baselines and Adaptive Cursor Sharing (ACS) - new in 11g.
- The execution plan should not change without being previously tested and approved.
- Possible conversion from “Stored Outlines”
- Due to Adaptive Cursor Sharing each cursor (SQL statement) can have multiple children with potentially different execution plans.
- ACS resolved problem about peeking at the values of bind variables because it allows the optimizer to generate a set of plans that are optimal for different sets of bind values.
- The bind sets may sometimes share the same execution plan.

# Scenario With SQL Plan Management



# Manual Loading of Execution Plans

```
set serveroutput on
declare b binary_integer;
begin
  for a in
       (select sql_id,sql_text
        from v$sql
        where sql_text like '%BYPASS%') loop
    b := dbms_spm.LOAD_PLANS_FROM_CURSOR_CACHE(a.sql_id);
    dbms_output.put_line( to_char(b)||' sql_text: '||a.sql_text);
  end loop;
end;
/
```

# Evolving New Plan Added

```
SQL> SET SERVEROUTPUT ON
SQL> SET LONG 10000
SQL> DECLARE
  2      report clob;
  3  BEGIN
  4      report := DBMS_SPM.EVOLVE_SQL_PLAN_BASELINE(
                   plan_name=>'SYS_SQL_PLAN_904f9eda94ecae5c');
  5      DBMS_OUTPUT.PUT_LINE(report);
  6  END;
  7  /
```

# Evolving New Plan Report

```
                  Evolve SQL Plan Baseline Report
--------------------------------------------------------------------------------
Inputs:
-------
  SQL_HANDLE = SYS_SQL_8a54f32d904f9eda
  PLAN_NAME  =
  TIME_LIMIT = DBMS_SPM.AUTO_LIMIT
  VERIFY     = YES
  COMMIT     = YES

Plan: SYS_SQL_PLAN_904f9eda10f5b979
-----------------------------------
  Plan was verified: Time used ,062 seconds.
  Passed performance criterion: Compound improvement ratio >= 28.
  Plan was changed to an accepted plan.
                       Baseline Plan      Test Plan     Improv. Ratio
                       -------------      ---------     -------------
  Execution Status:         COMPLETE       COMPLETE
  Rows Processed:                  1              1
  Elapsed Time(ms):                0              0
  CPU Time(ms):                    0              0
  Buffer Gets:                    84              3              28
  Disk Reads:                      0              0
  Direct Writes:                   0              0
  Fetches:                         0              0
  Executions:                      1              1
--------------------------------------------------------------------------------
Report Summary
--------------------------------------------------------------------------------
Number of SQL plan baselines verified: 1.
Number of SQL plan baselines evolved: 1.
```

# DBA_SQL_PLAN_BASELINES

```
SQL> select sql_handle,plan_name,enabled,accepted
  2  from DBA_SQL_PLAN_BASELINES
  3  where sql_text like '%id=3%';

SQL_HANDLE -> SQL_BASELINE     PLAN_NAME                      ENA ACC
------------------------------ ------------------------------ --- ---
SYS_SQL_8a54f32d904f9eda        SYS_SQL_PLAN_904f9eda10f5b979  YES NO
SYS_SQL_8a54f32d904f9eda        SYS_SQL_PLAN_904f9eda94ecae5c  YES YES
```

- DBA_SQL_PLAN_BASELINE contains information about all baselines in the database
- The SQL handle is a unique identifier for each SQL statement used in SPM for managing plan history.
- V$SQL contains information if the baseline was used (shown SQL_PLAN_BASELINE column)

# Not Every Baseline Can Be Evolved

```
Enter value for sql_plan: SQL_PLAN_d3qcgtcrxfy3s7c6cc440
-------------------------------------------------------------------------------
   Evolve SQL Plan Baseline Report
-------------------------------------------------------------------------------
Inputs:
-------
  SQL_HANDLE =
  PLAN_NAME  = SQL_PLAN_d3qcgtcrxfy3s7c6cc440
  TIME_LIMIT = DBMS_SPM.AUTO_LIMIT
  VERIFY     = YES
  COMMIT     = YES
Plan: SQL_PLAN_d3qcgtcrxfy3s7c6cc440
------------------------------------
  Plan was verified: Time used 9,85 seconds.
  Error encountered during plan verification (ORA-1732).
    ORA-01732: data manipulation operation not legal on this view
```

```
16960, 00000, "SQL Analyze could not reproduce the desired plan."
// *Cause:  SQL Analyze failed to reproduce a particular plan using an
//          outline.
// *Action: Check the outline data.
```

# SQL Plan Management Parameters

- SQL Plan Management is controlled by two init.ora parameter
  - **optimizer_capture_sql_plan_baselines**
    - Controls auto-capture of SQL plan baselines for repeatable statements
    - Set to false by default in 11gR1 (system and session modifiable)
  - **optimizer_use_sql_plan_baselines**
    - Controls the use of existing SQL plan baselines by the optimizer
    - Set to true by default in 11gR1 (system and session modifiable)
- SQL Plan Baselines are visible in view **DBA_SQL_PLAN_BASELINE**
- Package DBMS_SPM is used in background for managing SQL Plans

# Displaying SQL Plan Baselines

- To view the plans stored in the SQL plan baseline for a given statement, use the DISPLAY_SQL_PLAN_BASELINE function of the DBMS_XPLAN package:

```
select * from table(
   dbms_xplan.display_sql_plan_baseline(
                  sql_handle=>'SYS_SQL_8a54f32d904f9eda',
                  format=>'basic')
 );
```

# Anatomy of SQL Plan Baseline (Outline)

- Every query block is uniquely named in >=10g
- Query block names are system-generated or hinted (using new QB_NAME hint)
- System-generated names contain two parts:
  - fixed prefix based on query block type: DEL$, INS$, MRG$, SEL$, UPD$, CRI$, SET$, MISC$
  - followed by alphanumeric string (up to 8 characters long) e.g. SEL$1, SEL$A5FF74C1, etc.
- Global hints can be specified in any query block, not just the one they target.

# Global Hints Interpretation

```
Data from sys.sqlobj$data

<outline_data>
   <hint><![CDATA[INDEX_RS_ASC(@"SEL$1" "T"@"SEL$1" ("T"."ID"))]]></hint>
   <hint><![CDATA[OUTLINE_LEAF(@"SEL$1")]]></hint>
   <hint><![CDATA[ALL_ROWS]]></hint>
   <hint><![CDATA[DB_VERSION('11.1.0.6')]]></hint>
   <hint><![CDATA[OPTIMIZER_FEATURES_ENABLE('11.1.0.6')]]></hint>
   <hint><![CDATA[IGNORE_OPTIM_EMBEDDED_HINTS]]></hint>
</outline_data>
```

```
INDEX_RS_ASC(@"SEL$1" "T"@"SEL$1" ("T"."ID"))
```

- **INDEX_RS_ASC** - preform index range scan
- Hint refers to block named SEL$1 (system generated name)
- Hint refers to table T at block SEL$1
- Use Index which starts with ID column

# SQL Monitoring

# SQL Plan Monitoring

- New 11gR1 feature – requires Tuning pack licensing
- New views **V$SQL_MONITOR, V$SQL_PLAN_MONITOR**
- Captures statistics about SQL execution every second
- For parallel execution every process involved gets separate entries in V$SQL_MONITOR and V$SQL_PLAN_MONITOR
- Enabled by default for long running statements if parameter CONTROL_MANAGEMENT_PACK_ACCESS if it is set to "DIAGNOSTIC+TUNING" and STATISTICS_LEVEL=ALL|TYPICAL

# V$SQL_MONITOR, V$SQL_PLAN_MONITOR

```
SQL> SELECT status, KEY, SID, sql_id, elapsed_time, cpu_time, fetches, buffer_gets,
  2         disk_reads
  3    FROM v$sql_monitor;

STATUS            KEY SID SQL_ID        ELAPSED_TIME   CPU_TIME FETCHES BUFFER_GETS DISK_READS
--------- ----------- --- ------------- ------------ ---------- ------- ----------- ----------
EXECUTING 21474836481 170 b0zm3w4h1hbff    674281628  624578125       0           0          0

SQL> SELECT   plan_line_id, plan_operation || ' ' || plan_options operation,
  2           starts, output_rows
  3      FROM v$sql_plan_monitor
  4  ORDER BY plan_line_id;

PLAN_LINE_ID OPERATION                      STARTS OUTPUT_ROWS
------------ -------------------- ---------- -----------
           0 SELECT STATEMENT                1           0
           1 SORT AGGREGATE                  1           0
           2 MERGE JOIN CARTESIAN            1  4283731363
           3 MERGE JOIN CARTESIAN            1      156731
           4 INDEX FAST FULL SCAN            1           3
           5 BUFFER SORT                     3      156731
           6 INDEX FAST FULL SCAN            1       70088
           7 BUFFER SORT                156731  4283731363
           8 INDEX FAST FULL SCAN            1       70088
```

# SQL Monitoring Output (1)

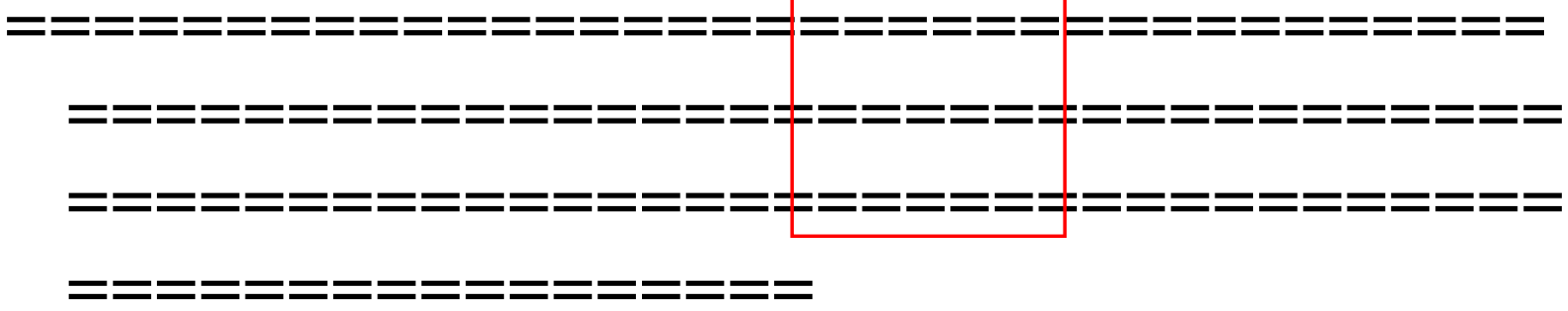
- dbms_sqltune.report_sql_monitor

SQL Plan Monitoring Details (Plan Hash Value=2056254005)

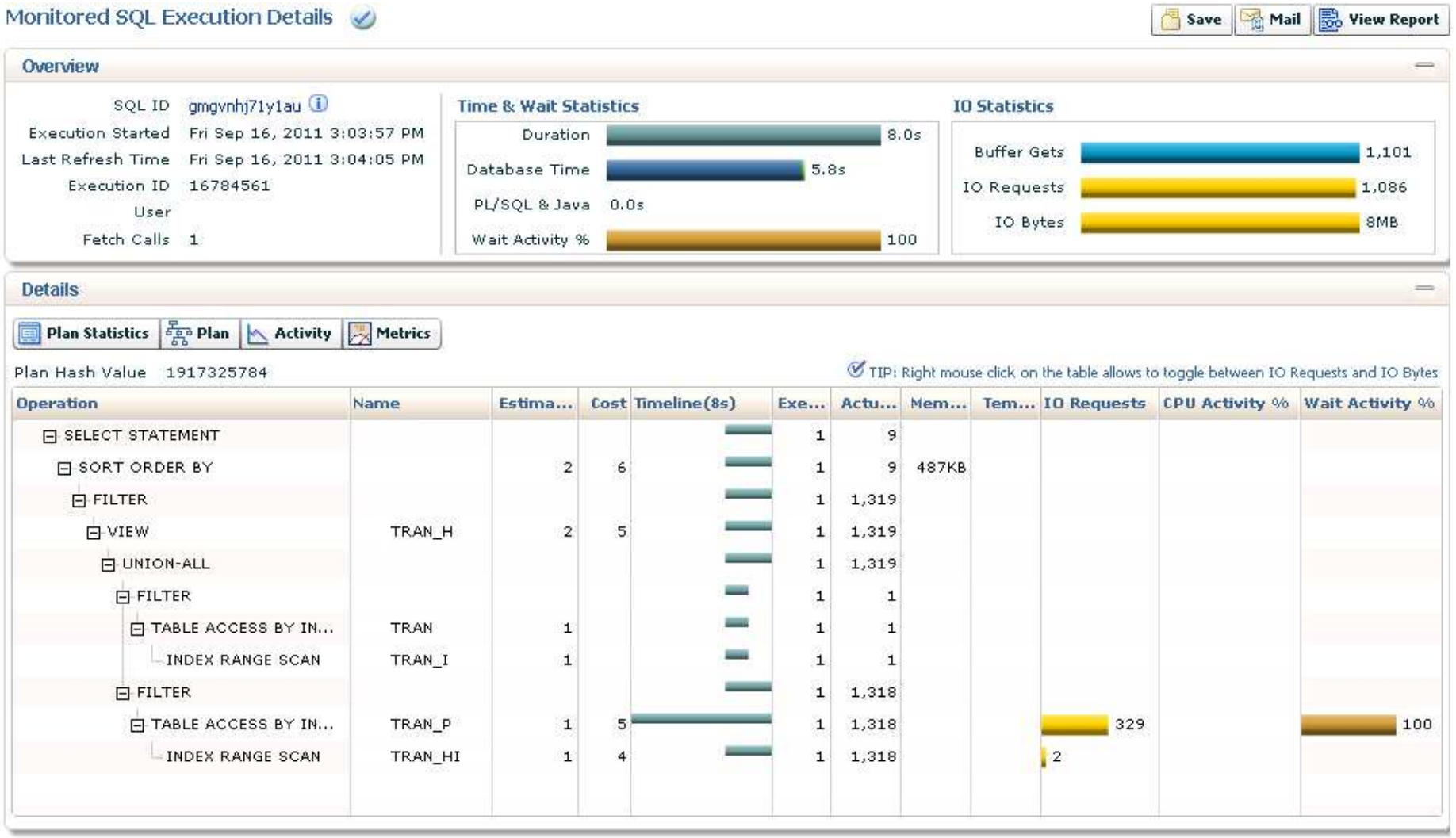
| Id | Operation | Name | Rows (Estim) | Cost | Time Active(s) | Start Active | Execs |
|---|---|---|---|---|---|---|---|
| 0 | SELECT STATEMENT | | | | 1 | +4 | 1 |
| 1 | SORT ORDER BY | | 24 | 39 | 1 | +4 | 1 |
| 2 | VIEW | AB_STMTEND | 24 | 38 | 1 | +4 | 1 |
| 3 | SORT UNIQUE | | 24 | 38 | 1 | +4 | 1 |
| 4 | UNION-ALL | | | | 1 | +4 | 1 |
| 5 | NESTED LOOPS | | | | 1 | +4 | 1 |
| 6 | NESTED LOOPS | | 1 | 6 | 1 | +4 | 1 |
| 7 | NESTED LOOPS | | 1 | 5 | 1 | +4 | 1 |
| 8 | TABLE ACCESS BY INDEX ROWID | AB_ACCOUNT_BAL | 1 | 3 | 1 | +4 | 1 |
| 9 | INDEX UNIQUE SCAN | AB_ACCT_BAL_UIDX | 1 | 2 | 1 | +4 | 1 |
| 10 | TABLE ACCESS BY INDEX ROWID | RB_STMT_MAST_SK | 1 | 2 | 1 | +4 | 1 |
| 11 | INDEX RANGE SCAN | RXM_INTERNAL_KEY_PK | 1 | 1 | 1 | +4 | 1 |
| 12 | INDEX UNIQUE SCAN | RSM_INTERNAL_KEY_PK | 1 | | 1 | +4 | 2 |
| 13 | TABLE ACCESS BY INDEX ROWID | RB_STMT | 1 | 1 | 1 | +4 | 2 |
| 14 | NESTED LOOPS | | | | 1 | +4 | 1 |
| 15 | NESTED LOOPS | | 23 | 30 | 1 | +4 | 1 |
| 16 | NESTED LOOPS | | 23 | 7 | 1 | +4 | 1 |
| 17 | TABLE ACCESS BY INDEX ROWID | AB_ACCOUNT_BAL | 1 | 3 | 1 | +4 | 1 |
| 18 | INDEX UNIQUE SCAN | AB_ACCT_BAL_UIDX | 1 | 2 | 1 | +4 | 1 |
| 19 | TABLE ACCESS BY INDEX ROWID | RB_STMT_MAST_HIST_SK | 23 | 4 | 4 | +1 | 1 |
| | | | | | | | |
| 20 | INDEX RANGE SCAN | RB_STMT_MAST_HIST_SK_I1 | 2 | 1 | 1 | +4 | 1 |
| 21 | INDEX UNIQUE SCAN | RSM_INTERNAL_KEY_PK | 1 | | 1 | +4 | 1133 |
| 22 | TABLE ACCESS BY INDEX ROWID | EB_STMT | 1 | 1 | 1 | +4 | 1133 |

# SQL Monitoring Output (2)

- dbms_sqltune.report_sql_monitor



# SQL Monitor Output In EM



# Automatic Cardinality Feedback Tuning

# Automatic Cardinality Feedback Tuning

```
SELECT sql_id, COUNT (*)
FROM v$sql_shared_cursor
WHERE use_feedback_stats = 'Y'
GROUP BY sql_id
ORDER BY 2 DESC


SQL_ID           COUNT(*) MAX(CHILD_NUMBER)
------------- ---------- -----------------
agfj9yqja74ka         10                10
91thqn3j4shfj          3                 4
gu7my5yzjm1ap          3                 2
68rvzaufbk6fr          3                 3
b7xhjjbmqdms4          3                 2
....
```

# SQL Statements Automatically Tuned

```
select sql_id,child_number
from v$sql_plan
where other_xml is not null
and other_xml like '%cardinality_feedback%'
order by child_number desc;

SQL_ID        CHILD_NUMBER
------------- ------------
2bpp4r8ajsuz3           41
a9s5xz5v4qw95           36
2bpp4r8ajsuz3           35
97h27ay3zhhar           14
97h27ay3zhhar           12
91thqn3j4shfj            9
7ta3c5mugdj8d            4
231q3js3fbn0r            4
axm5005kdufpd            4
gc6k70xc7kfwj            4
193nyt3gxjgmm            3
7ta3c5mugdj8d            3
```

# Potential Fallacies

- This new feature may somehow generate many child cursors with the identical execution plan.

```
select plan_hash_value,count(*)
from v$sql
where  sql_id='agfj9yqja74ka'
group by plan_hash_value;


PLAN_HASH_VALUE   COUNT(*)
--------------- ----------
     3602887883         10
```

# Disabling ACF Tuning

- One can even disable automatic cardinality feedback tuning by setting „_optimizer_use_feedback“ parameter at system or session level.

- With opt_param hint one can disable it at SQL statement level.

```
select /*+ opt_param('_optimizer_use_feedback','false') */
...
```

# Conclusions

- The root cause for sub-optimal plans is the lack of information available to the CBO.
- CBO requires good input to be able to produce optimal execution plans.
- When we tell "truth" to the optimizer we can expect that the prepared execution plan will most likely be an optimal one.
- Otherwise the "guess" made by the CBO will most likely turn into a sub-optimal plan.
- Give more attention to the estimated cardinality then to the cost of execution plan.

# Thank you for your interest!

# Q&A