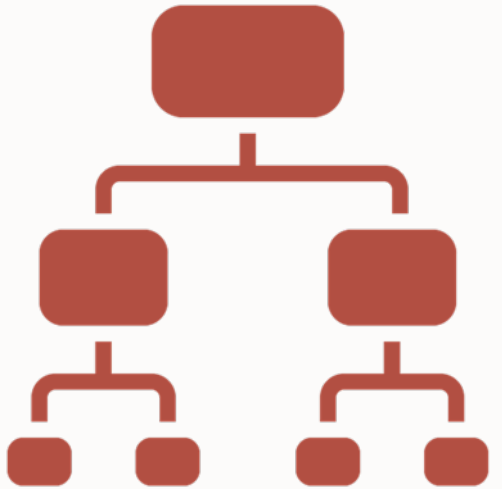
ORACLE

Explain the Explain Plan

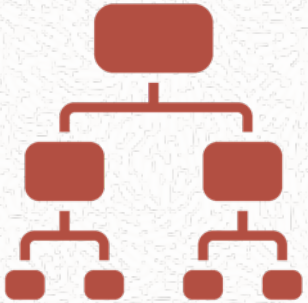
PART 1 INTERPRETING EXECUTION PLANS FOR SQL STATEMENTS



Maria Colgan
Distinguished Product Manager
Oracle Database
August 2020
@SQLMaria

SQLMaria

Program Agenda



Part 1

1. What is an execution plan
2. How to generate a plan
3. Understanding execution plans

Part 2

4. Partitioning
5. Parallel Execution
6. Execution Plan Example

What is an execution plan?

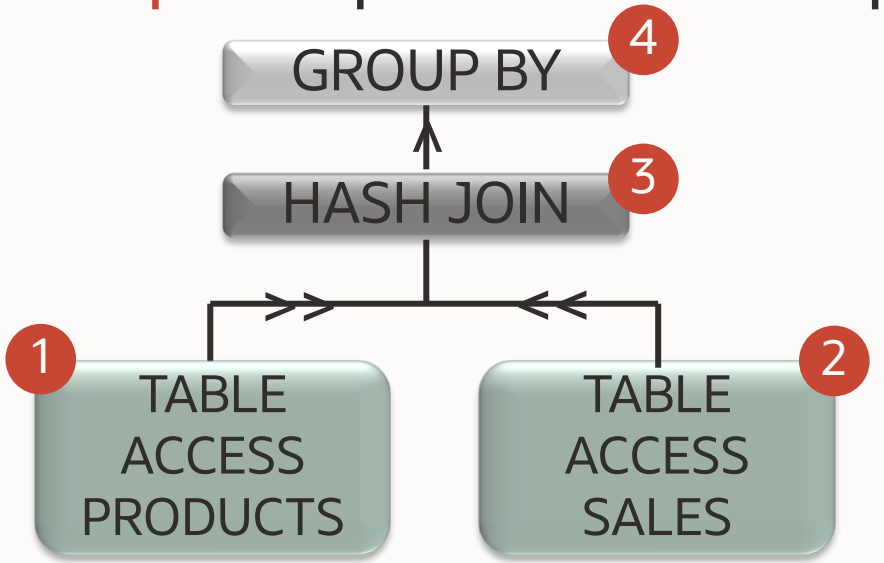
Query:

```
SELECT prod_category, avg(amount_sold)
  FROM sales s, products p
 WHERE p.prod_id = s.prod_id
GROUP BY prod_category;
```

Tabular representation of plan

```
-------------------------------------------
| Id  | Operation           | Name        |
-------------------------------------------
|   0 | SELECT STATEMENT    |             |
|   1 |  HASH GROUP BY      |             |
|*  2 |   HASH JOIN         |             |
|   3 |    TABLE ACCESS FULL| PRODUCTS    |
|   4 |    TABLE ACCESS FULL| SALES       |
-------------------------------------------
```

Tree-shaped representation of plan



Additional information under the execution plan

```
SELECT /*+ gather_plan_statistics */ count(*)  FROM    sales2 WHERE
prod_id=to_number('139')

Plan hash value: 1631620387

-------------------------------------------------------------------------------------------
| Id  | Operation         | Name        | Starts | E-Rows | Cost (%CPU)| A-Rows |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |             |      1 |        |    35 (100)|      1 |
|   1 |  SORT AGGREGATE   |             |      1 |      1 |            |      1 |
|*  2 |   INDEX RANGE SCAN| MY_PROD_IND |      1 |  12762 |    35   (0)|  11574 |
-------------------------------------------------------------------------------------------

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

   2 - access("PROD_ID"=139)
```

Access predicate

- Where clause predicate used for data retrieval
  - The start and stop keys for an index
  - If rowids are passed to a table scan

Additional information under the execution plan

```
SQL> SELECT username
  2  FROM   my_users
  3  WHERE  username LIKE 'MAR%';

USERNAME
------------------------------------------------------------------------------
MARIA

Plan hash value: 2982854235

-------------------------------------------------------------------------------
| Id  | Operation         | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |          |       |       |     2 (100)|          |
|*  1 |  TABLE ACCESS FULL| MY_USERS |     1 |    66 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - filter("USERNAME" LIKE 'MAR%')
```

Filter predicate

- Where clause predicate that is not used for data retrieval but to eliminate uninteresting row once the data is found

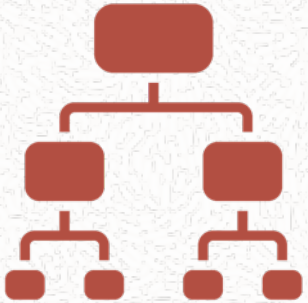
Additional information under the execution plan

```
SELECT      p.prod_name, sum(s.amount_sold) amt FROM       Sales s,
Products p  WHERE       s.prod_id=p.prod_id AND        p.supplier_id =
:sup_id  group by p.prod_name

Plan hash value: 107119040

-------------------------------------------------------------------------------------------
| Id  | Operation                | Name                | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |                     |       |       |   573 (100)|
|   1 |  HASH GROUP BY           |                     |    71 |  3550 |   573  (10)|
|*  2 |   HASH JOIN              |                     |    72 |  3600 |   572  (10)|
|   3 |    VIEW                  | VW_GBC_5            |    72 |  1224 |   570  (10)|
|   4 |     HASH GROUP BY        |                     |    72 |   648 |   570  (10)|
|   5 |      PARTITION RANGE ALL |                     |   918K|  8075K|   530   (3)|
|   6 |       TABLE ACCESS FULL  | SALES               |   918K|  8075K|   530   (3)|
|*  7 |    INDEX RANGE SCAN      | PROD_SUPP_ID_INDX   |    72 |  2376 |     1   (0)|
-------------------------------------------------------------------------------------------

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

   2 - access("ITEM_1"="P"."PROD_ID")
   7 - access("P"."SUPPLIER_ID"=:SUP_ID)

Note
-----
   - SQL plan baseline SQL_PLAN_11v0o04h0t3c1aa1ba510 used for this statement
```

Note Section

- Details on Optimizer features used such as:
  - Rule Based Optimizer (RBO)
  - Dynamic Sampling
  - Outlines
  - SQL Profiles or plan baselines
  - Adaptive Plans
  - Hints (Starting in 19c)

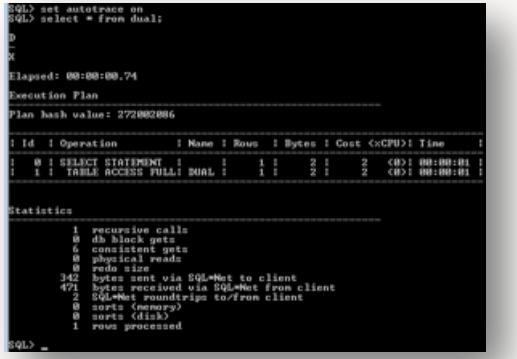
Program Agenda



1. What is an execution plan
2. How to generate a plan
3. Understanding execution plans

Many ways to view an execution plan

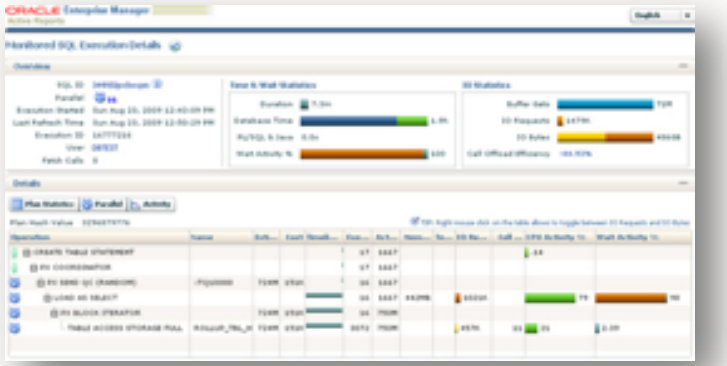
Autotrace



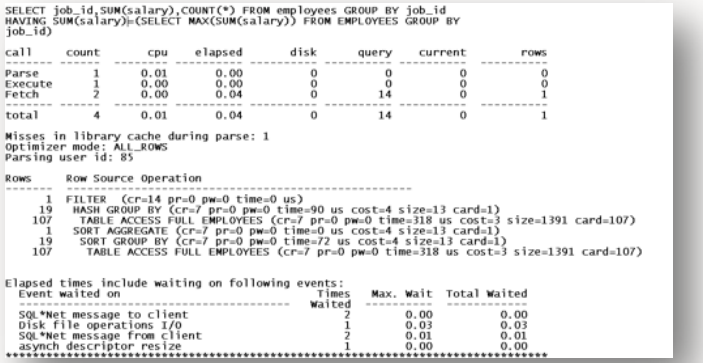
```
SQL> set autotrace on
SQL> select * from dual;

D
-
X

Elapsed: 00:00:00.74

Execution Plan
----------------------------------------------------------
Plan hash value: 272002086

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


Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
          6  consistent gets
          0  physical reads
          0  redo size
        342  bytes sent via SQL*Net to client
        471  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          1  rows processed

SQL> _
```

SQL Developer

SQL Monitor



TKPROF



```
SELECT job_id,SUM(salary),COUNT(*) FROM employees GROUP BY job_id
HAVING SUM(salary)=(SELECT MAX(SUM(salary)) FROM EMPLOYEES GROUP BY
job_id)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ----------  ----------  ----------  ----------  ----------
Parse        1      0.01       0.00           0           0           0           0
Execute      1      0.00       0.00           0           0           0           0
Fetch        2      0.00       0.04           0          14           0           1
------- ------  -------- ----------  ----------  ----------  ----------  ----------
total        4      0.01       0.04           0          14           0           1

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 85

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  FILTER  (cr=14 pr=0 pw=0 time=0 us)
     19   HASH GROUP BY (cr=7 pr=0 pw=0 time=90 us cost=4 size=13 card=1)
    107    TABLE ACCESS FULL EMPLOYEES (cr=7 pr=0 pw=0 time=318 us cost=3 size=1391 card=107)
      1   SORT AGGREGATE (cr=7 pr=0 pw=0 time=0 us cost=4 size=13 card=1)
     19    SORT GROUP BY (cr=7 pr=0 pw=0 time=72 us cost=4 size=13 card=1)
    107     TABLE ACCESS FULL EMPLOYEES (cr=7 pr=0 pw=0 time=318 us cost=3 size=1391 card=107)


Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------    Waited  ----------  ------------
  SQL*Net message to client                        2        0.00          0.00
  Disk file operations I/O                         1        0.03          0.03
  SQL*Net message from client                      2        0.01          0.01
  asynch descriptor resize                         1        0.00          0.00
********************************************************************************
```

.....But there are actually only 2 ways to generate one

How to generate an execution plan

Two methods for looking at the execution plan

1. EXPLAIN PLAN command

- Displays an execution plan for a SQL statement without actually executing the statement

2. V$SQL_PLAN

- A dictionary view introduced in Oracle 9i that shows the execution plan for a SQL statement that has been compiled into a cursor in the cursor cache

Under certain conditions the plan shown with EXPLAIN PLAN can be different from the plan shown using V$SQL_PLAN

How to generate an execution plan

EXPLAIN PLAN command & dbms_xplan.display function

```
SQL>  EXPLAIN PLAN FOR
      SELECT  p.prod_name, avg(s.amount_sold)
              FROM    sales s, products p
              WHERE   p.prod_id = s.prod_id
              GROUP BY p.prod_name;

SQL>  SELECT * FROM
        table(dbms_xplan.display('plan_table',null,'basic'));
```

PLAN TABLE NAME

STATEMENT ID

FORMAT

How to generate an execution plan

Generate & display plan for last SQL statements executed in session

```
SQL> SELECT  p.prod_name, avg(s.amount_sold)
           FROM    sales s, products p
           WHERE   p.prod_id = s.prod_id
           GROUP BY p.prod_name;
```

```
SQL> SELECT * FROM
     table(dbms_xplan.display_cursor(null, null, 'basic'));
```

SQL_ID

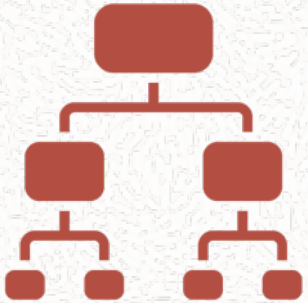
CHILD NUMBER

FORMAT

- Format* is highly customizable - Basic ,Typical, All
  - Additional low-level parameters show more detail

*More information on formatting on Optimizer blog

Program Agenda



1. What is an execution plan
2. How to generate a plan
3. Understanding execution plans
   - Cardinality
   - Access paths
   - Join methods
   - Join order

Cardinality

What is it?

Estimate of number rows that will be returned by each operation

How does the Optimizer Determine it?

Cardinality for a single column equality predicate = **total num of rows / num of distinct values**

For example: A table has **100** rows, a column has **5** distinct values
=> cardinality=**20** rows

More complicated predicates have more complicated cardinality calculation

Why should you care?

It influences everything! Access method, Join type, Join Order etc.

Identifying cardinality in an execution plan

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT				12 (100)	
1	NESTED LOOPS					
2	NESTED LOOPS		1	211	12 (9)	00:00:01
3	NESTED LOOPS		1	185	11 (10)	00:00:01
* 4	HASH JOIN		1	155	10 (10)	00:00:01
5	MERGE JOIN CARTESIAN		107	8774	6 (0)	00:00:01
* 6	TABLE ACCESS FULL	DEPARTMENTS	1	30	3 (0)	00:00:01
7	BUFFER SORT		107	5564	3 (0)	00:00:01
8	TABLE ACCESS FULL	EMPLOYEES	107	5564	3 (0)	00:00:01
9	TABLE ACCESS FULL	EMPLOYEES	107	7811	3 (0)	00:00:01
* 10	TABLE ACCESS BY INDEX ROWID	DEPARTMENTS	1	30	1 (0)	00:00:01
* 11	INDEX UNIQUE SCAN	DEPT_ID_PK	1		0 (0)	
* 12	INDEX UNIQUE SCAN	JOB_ID_PK	1		0 (0)	
13	TABLE ACCESS BY INDEX ROWID	JOBS	1	26	1 (0)	00:00:01

Predicate Information (identified by operation id):

```
   4 - access("E","MANAGER_ID"="E","EMPLOYEE_ID" AN
              "E","DEPARTMENT_ID"="D","DEPARTMENT_I
       filter("E","SALARY"+("E","SALARY"+"E","COMMI
              ARY"+"E","COMMISSION_PCT"))
   6 - filter("D","DEPARTMENT_NAME"='Sales')
  10 - filter("D","DEPARTMENT_NAME"='Sales')
  11 - access("E","DEPARTMENT_ID"="D","DEPARTMENT_ID")
  12 - access("E","JOB_ID"="J","JOB_ID")
```

Cardinality - estimated # of rows returned

Determine correct cardinality using a SELECT COUNT(*) from each table applying any WHERE Clause predicates belonging to that table

Checking cardinality estimates

```
SELECT /*+ gather_plan_statistics */
       p.prod_name, SUM(s.quantity_sold)
FROM    sales s, products p
WHERE   s.prod_id =p.prod_id
GROUP BY p.prod_name ;

SELECT * FROM table (
          DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'ALLSTATS LAST'));
```

Checking cardinality estimates

```
SELECT * FROM table (
        DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'ALLSTATS LAST'));
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	OMem	1Mem	Used-Mem
0	SELECT STATEMENT		1		71	00:00:00.57	1638			
1	HASH GROUP BY		1	71	71	00:00:00.57	1638	799K	799K	3079K (0)
* 2	HASH JOIN		1	918K	918K	00:00:00.85	1638	933K	933K	1279K (0)
3	TABLE ACCESS STORAGE FULL	PRODUCTS	1	72	72	00:00:00.01	3			
4	PARTITION RANGE ALL		1	918K	918K	00:00:00.37	1635			
5	TABLE ACCESS STORAGE FULL	SALES	28	918K	918K	00:00:00.20	1635			

Compare estimated number of rows (E-Rows) with actual rows returned (A-Rows)

Checking cardinality estimates

Extra information you get with ALLSTATS

```
SELECT * FROM table (
          DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'ALLSTATS LAST'));
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	OMem	1Mem	Used-Mem
0	SELECT STATEMENT		1		71	00:00:00.57	1638			
1	HASH GROUP BY		1	71	71	00:00:00.57	1638	799K	799K	3079K (0)
* 2	HASH JOIN		1	918K	918K	00:00:00.85	1638	933K	933K	1279K (0)
3	TABLE ACCESS STORAGE FULL	PRODUCTS	1	72	72	00:00:00.01	3			
4	PARTITION RANGE ALL		1	918K	918K	00:00:00.37	1635			
5	TABLE ACCESS STORAGE FULL	SALES	28	918K	918K	00:00:00.20	1635			

Starts indicates the number of times that step, or operation was done

In this case the SALES table is partitioned and has 28 partitions

Checking cardinality estimates

Extra information you get with ALLSTATS

```
SELECT * FROM table (
          DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'ALLSTATS LAST'));
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	OMem	1Mem	Used-Mem
0	SELECT STATEMENT		1		71	00:00:00.57	1638			
1	HASH GROUP BY		1	71	71	00:00:00.57	1638	799K	799K	3079K (0)
* 2	HASH JOIN		1	918K	918K	00:00:00.85	1638	933K	933K	1279K (0)
3	TABLE ACCESS STORAGE FULL	PRODUCTS	1	72	72	00:00:00.01	3			
4	PARTITION RANGE ALL		1	918K	918K	00:00:00.37	1635			
5	TABLE ACCESS STORAGE FULL	SALES	28	918K	918K	00:00:00.20	1635			

Buffers indicates the number of buffers that need to be read for each step

Checking cardinality estimates

Extra information you get with ALLSTATS

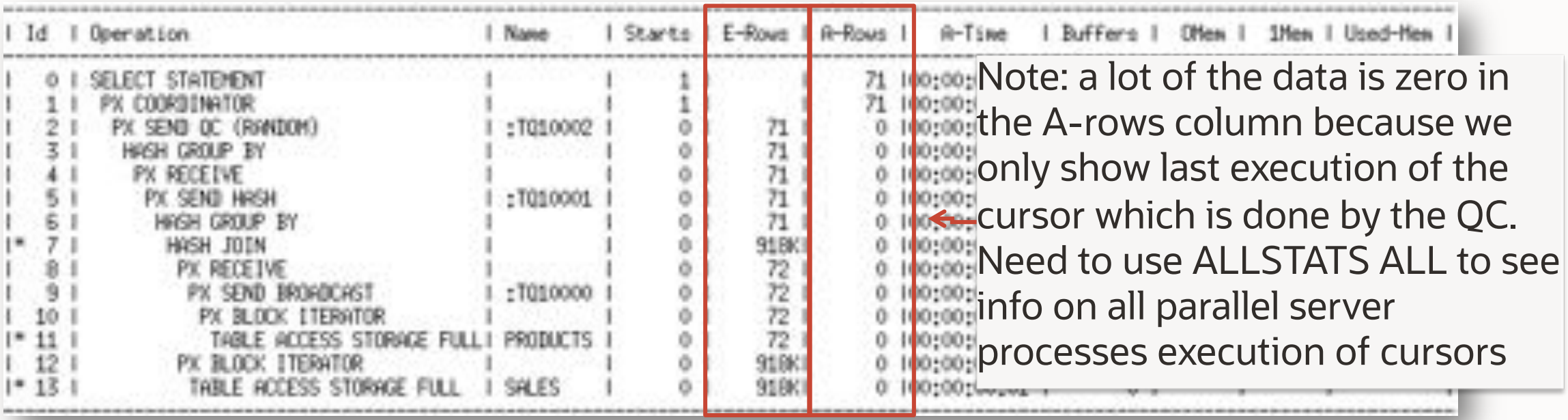
```
SELECT * FROM table (
          DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'ALLSTATS LAST'));
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	OMem	1Mem	Used-Mem
0	SELECT STATEMENT		1		71	00:00:00.57	1638			
1	HASH GROUP BY		1	71	71	00:00:00.57	1638	799K	799K	3079K (0)
* 2	HASH JOIN		1	918K	918K	00:00:00.85	1638	933K	933K	1279K (0)
3	TABLE ACCESS STORAGE FULL	PRODUCTS	1	72	72	00:00:00.01	3			
4	PARTITION RANGE ALL		1	918K	918K	00:00:00.37	1635			
5	TABLE ACCESS STORAGE FULL	SALES	28	918K	918K	00:00:00.20	1635			

OMem - estimated amount of memory needed
1Mem - amount of memory needed to perform the operation in 1 pass
Used-Mem - actual amount of memory used and number of passes required

Checking cardinality estimates for Parallel Execution

```
SELECT * FROM table (
          DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'ALLSTATS LAST'));
```



Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	OMem	1Mem	Used-Mem
0	SELECT STATEMENT		1		71					
1	PX COORDINATOR		1		71					
2	PX SEND QC (RANDOM)	:TQ10002	0	71	0					
3	HASH GROUP BY		0	71	0					
4	PX RECEIVE		0	71	0					
5	PX SEND HASH	:TQ10001	0	71	0					
6	HASH GROUP BY		0	71	0					
7	HASH JOIN		0	918K	0					
8	PX RECEIVE		0	72	0					
9	PX SEND BROADCAST	:TQ10000	0	72	0					
10	PX BLOCK ITERATOR		0	72	0					
11	TABLE ACCESS STORAGE FULL	PRODUCTS	0	72	0					
12	PX BLOCK ITERATOR		0	918K	0					
13	TABLE ACCESS STORAGE FULL	SALES	0	918K	0					

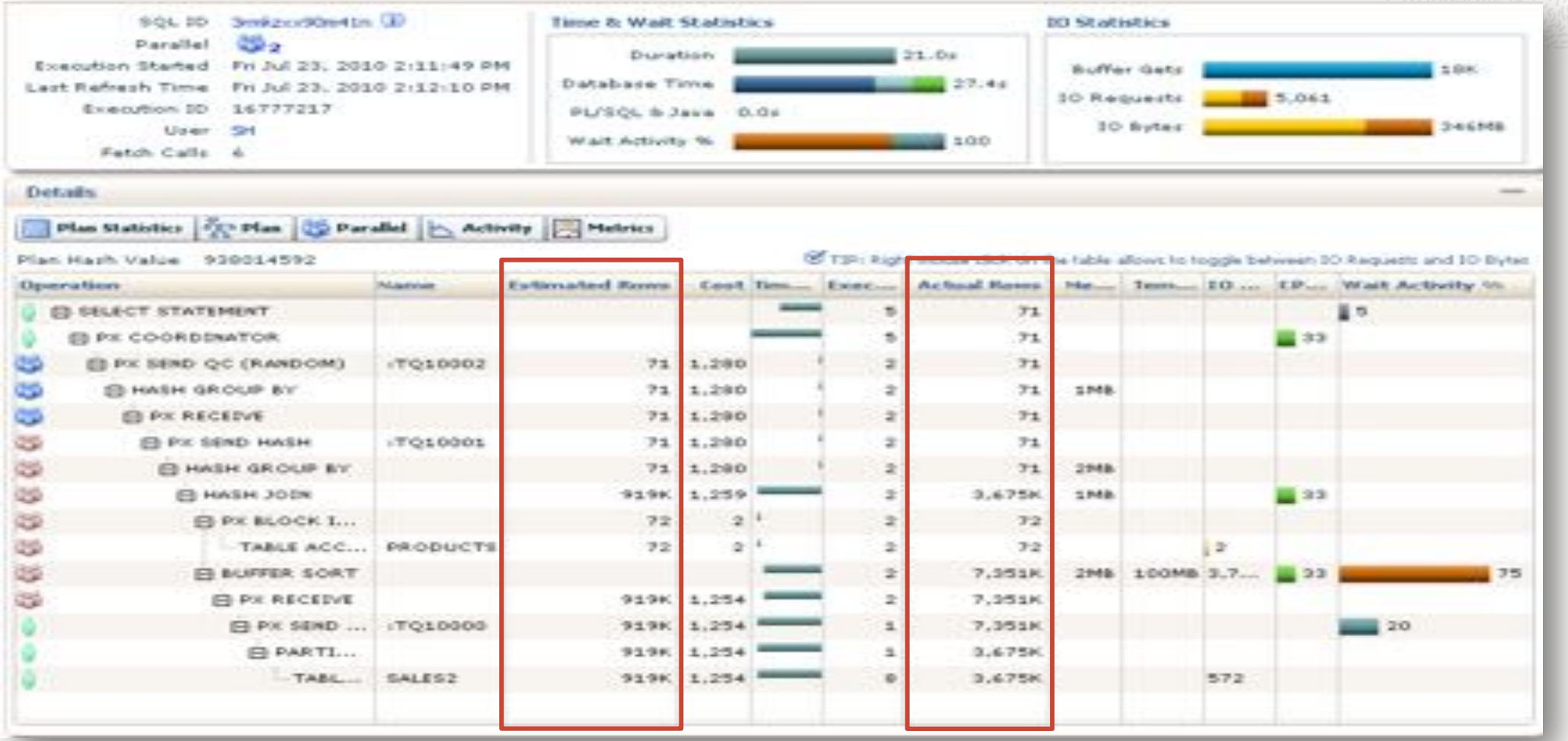
Note: a lot of the data is zero in the A-rows column because we only show last execution of the cursor which is done by the QC. Need to use ALLSTATS ALL to see info on all parallel server processes execution of cursors

Checking cardinality estimates for Parallel Execution

```
SELECT * FROM table (
          DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'ALLSTATS ALL'));
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	OMem	1Mem	O/1/M
0	SELECT STATEMENT		1		71	00:00:00.65	51			
1	PX COORDINATOR		1		71	00:00:00.65	51			
2	PX SEND QC (RANDOM)	:TQ10002	0	71	0	00:00:00.01	0			
3	HASH GROUP BY		16	71	10	00:00:01.00	0	858K	858K	16/0/0
4	PX RECEIVE		16	71	498	00:00:00.76	0			
5	PX SEND HASH	:TQ10001	0	71	0	00:00:00.01	0			
6	HASH GROUP BY		16	71	520	00:00:02.93	446	813K	813K	16/0/0
*7	HASH JOIN		16	918K	127K	00:00:03.65	446	1089K	1089K	16/0/0
8	PX RECEIVE		16	72	1152	00:00:01.09	0			
9	PX SEND BROADCAST	:TQ10000	0	72	0	00:00:00.01	0			
10	PX BLOCK ITERATOR		16	72	40	00:00:00.01	2			
*11	TABLE ACCESS STORAGE FULL	PRODUCTS	1	72	40	00:00:00.01	2			
12	PX BLOCK ITERATOR		16	918K	127K	00:00:02.00	446			
*13	TABLE ACCESS STORAGE FULL	SALES	223	918K	127K	00:00:00.09	446			

Check cardinality using SQL Monitor

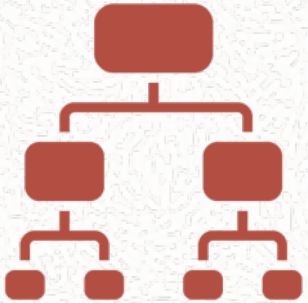


Easiest way to compare the estimated number of rows returned with actual rows returned

Solutions to incorrect cardinality estimates

Cause	Solution
Stale or missing statistics	DBMS_STATS
Data Skew	Create a histogram
Multiple single column predicates on a table	Create a column group using DBMS_STATS.CREATE_EXTENDED_STATS
Function wrapped column	Create statistics on the funct wrapped column using DBMS_STATS.CREATE_EXTENDED_STATS
Multiple columns used in a join	Create a column group on join columns using DBMS_STATS.CREATE_EXTENDED_STAT
Complicated expression containing columns from multiple tables	Use dynamic sampling level 4 or higher

Program Agenda

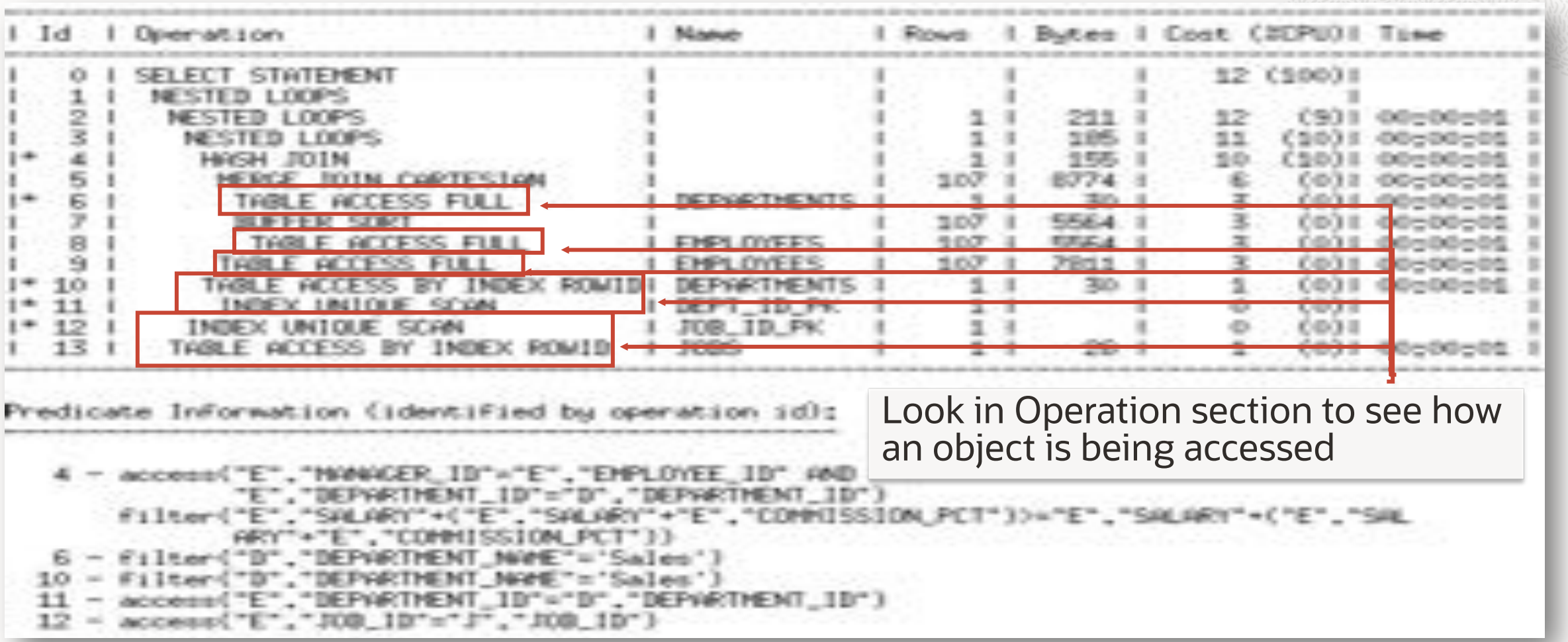


1. What is an execution plan
2. How to generate a plan
3. Understanding execution plans
   - Cardinality
   - Access paths
   - Join methods
   - Join order

Access Paths – Getting the data

Access Path	Explanation
Full table scan	Reads all rows from table & filters out those that do not meet the where clause predicates. Used when no index, DOP set etc.
Table access by Rowid	Rowid specifies the datafile & data block containing the row and the location of the row in that block. Used if rowid supplied by index or directly in a where clause predicate
Index unique scan	Only one row will be returned. Used when table contains a UNIQUE or a PRIMARY KEY constraint that guarantees that only a single row is accessed e.g. equality predicate on PRIMARY KEY column
Index range scan	Accesses adjacent index entries returns ROWID values Used with equality on non-unique indexes or range predicate on unique indexes (<.>, between etc.)
Index skip scan	Skips the leading edge (column) of the index & uses the rest Advantageous if there are few distinct values in the leading column and many distinct values in the non-leading column or columns of the index
Full index scan	Processes all leaf blocks of an index, but only enough branch blocks to find 1st leaf block. Used when all necessary columns are in index & order by clause matches index structure or if a sort merge join is done
Fast full index scan	Scans all blocks in index used to replace a Full Table Scan when all necessary columns are in the index. Using multi-block IO & can going parallel
Index joins	Hash join of several indexes that together contain all the table columns that are referenced in the query. Won't eliminate a sort operation
Bitmap indexes	Uses a bitmap for key values and a mapping function that converts each bit position to a rowid. Can efficiently merge indexes that correspond to several conditions in a WHERE clause

Identifying access paths in an execution plan



Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT				12 (100)	
1	NESTED LOOPS					
2	NESTED LOOPS		1	231	12 (9)	00:00:01
3	NESTED LOOPS		1	185	11 (10)	00:00:01
* 4	HASH JOIN		1	155	10 (10)	00:00:01
5	MERGE JOIN CARTESIAN		107	8774	6 (0)	00:00:01
* 6	TABLE ACCESS FULL	DEPARTMENTS	1	30	3 (0)	00:00:01
7	BUFFER SORT		107	5564	3 (0)	00:00:01
8	TABLE ACCESS FULL	EMPLOYEES	107	5564	3 (0)	00:00:01
9	TABLE ACCESS FULL	EMPLOYEES	107	7811	3 (0)	00:00:01
* 10	TABLE ACCESS BY INDEX ROWID	DEPARTMENTS	1	30	1 (0)	00:00:01
* 11	INDEX UNIQUE SCAN	DEPT_ID_PK	1		0 (0)	
* 12	INDEX UNIQUE SCAN	JOB_ID_PK	1		0 (0)	
13	TABLE ACCESS BY INDEX ROWID	JOBS	1	26	1 (0)	00:00:01

Predicate Information (identified by operation id):

```
   4 - access("E","MANAGER_ID"="E","EMPLOYEE_ID" AND
              "E","DEPARTMENT_ID"="D","DEPARTMENT_ID")
       filter("E","SALARY"+("E","SALARY"+"E","COMMISSION_PCT")>="E","SALARY"+("E","SAL
              ARY"+"E","COMMISSION_PCT"))
   6 - filter("D","DEPARTMENT_NAME"='Sales')
  10 - filter("D","DEPARTMENT_NAME"='Sales')
  11 - access("E","DEPARTMENT_ID"="D","DEPARTMENT_ID")
  12 - access("E","JOB_ID"="J","JOB_ID")
```

Look in Operation section to see how an object is being accessed

If the wrong access method is being used check cardinality, join order…

Access path example 1

What plan would you expect for this query?

Table customers contains 10K rows & has a primary key on cust_id

```
SELECT country_id, name
FROM   customers
WHERE  cust_id IN (100,200,100000);
```

```
-------------------------------------------------------------------------------------
| Id  | Operation                    | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |           |     3 |    39 |     3   (0)| 00:00:01 |
|   1 |  INLIST ITERATOR             |           |       |       |            |          |
|   2 |   TABLE ACCESS BY INDEX ROWID| CUSTOMERS |     3 |    39 |     3   (0)| 00:00:01 |
|*  3 |    INDEX UNIQUE SCAN         | C_ID_IDX  |     3 |       |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------
   3 - access("CUST_ID"=100 OR "CUST_ID"=200 OR "CUST_ID"=100000)
```

Access path example 2

What plan would you expect for this query?

Table customers contains 10K rows & has a primary key on cust_id

```
SELECT country_id, name
FROM   customers
WHERE  cust_id BETWEEN 100 AND 150;
```

```
-------------------------------------------------------------------------------------------
| Id  | Operation                           | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |           |     1 |    13 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| CUSTOMERS |     1 |    13 |     3   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | C_ID_IDX  |     1 |       |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

Access path example 3

What plan would you expect for this query?

Table customers contains 10K rows & has a primary key on cust_id

```
SELECT country_id, name
FROM   customers
WHERE country_name = 'USA';
```

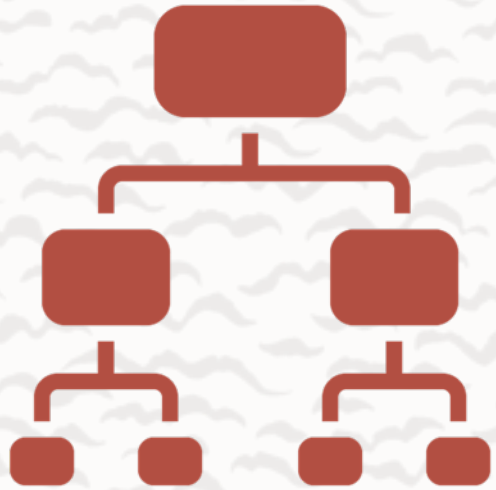
```
-------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |    30 |   480 |     5   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL | CUSTOMERS |    30 |   480 |     5   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

Common access path issues

Issue	Cause
Uses a table scan instead of index	DOP on table but not index, value of MBRC
Picks wrong index	Stale or missing statistics Cost of full index access is cheaper than index look up followed by table access Picks index that matches most # of column

Time for Q & A

Join us next time
August 18th 12pm EST
PART 2
Explain the Explain Plan



Join the Conversation

https://twitter.com/SQLMaria

https://blogs.oracle.com/optimizer/

https://sqlmaria.com

https://www.facebook.com/SQLMaria

Related White Papers

- Explain the Explain Plan
- Understanding Optimizer Statistics
- Best Practices for Gathering Optimizer Statistics
- What to expect from the Optimizer in 19c
- What to expect from the Optimizer in 12c
- What to expect from the Optimizer in 11g