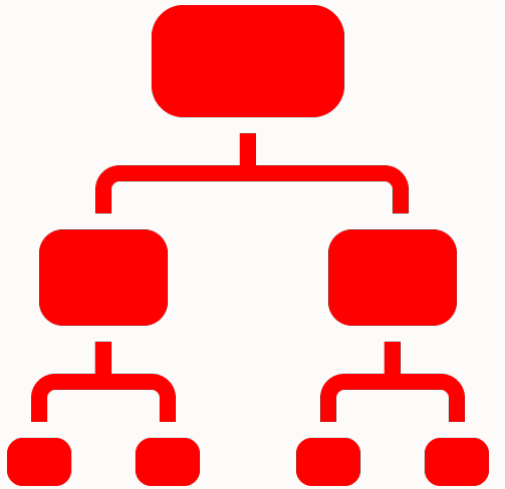
ORACLE

Explain the Explain Plan

PART 3 INTERPRETING EXECUTION PLANS FOR SQL STATEMENTS



Maria Colgan
Master Product Manager
Oracle Database
February 2020
@SQLMaria

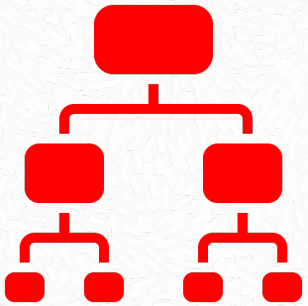
SQLMaria

Safe harbor statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions.

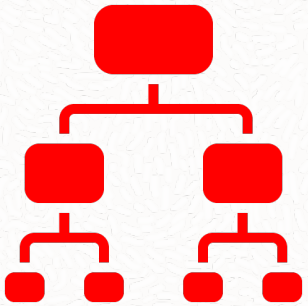
The development, release, timing, and pricing of any features or functionality described for Oracle's products may change and remains at the sole discretion of Oracle Corporation.

Program Agenda



1. What is an execution plan
2. How to generate a plan
3. Understanding execution plans
4. Execution Plan Example
5. Partitioning
6. Parallel Execution

Program Agenda



1. What is an execution plan
2. How to generate a plan
3. Understanding execution plans
4. Execution Plan Example

What is an execution plan?

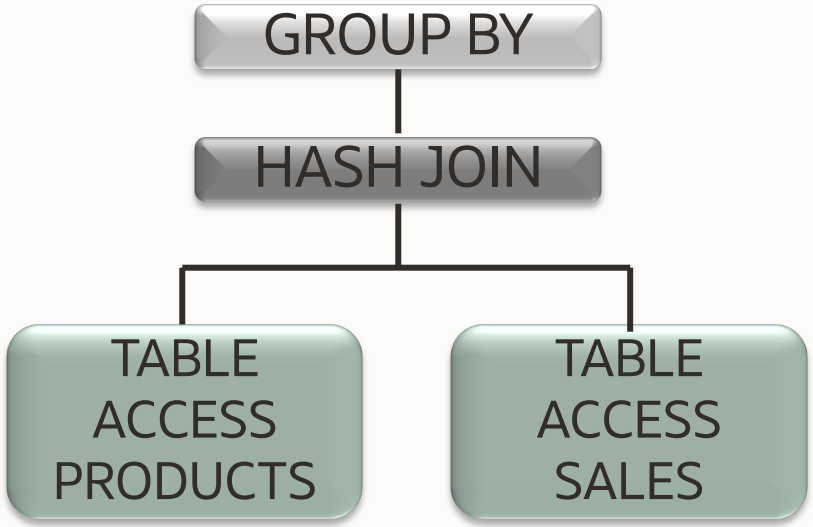
Query:

```
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

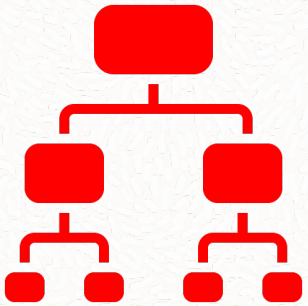
Additional information under the execution plan

```
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: 187119048

-------------------------------------------------------------------------------
| 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_11v9s0fh9t3z1aa1ba510 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)

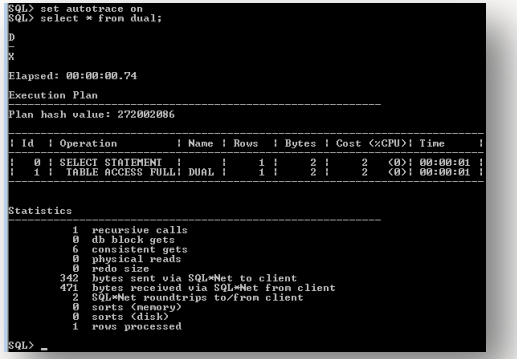
Program Agenda



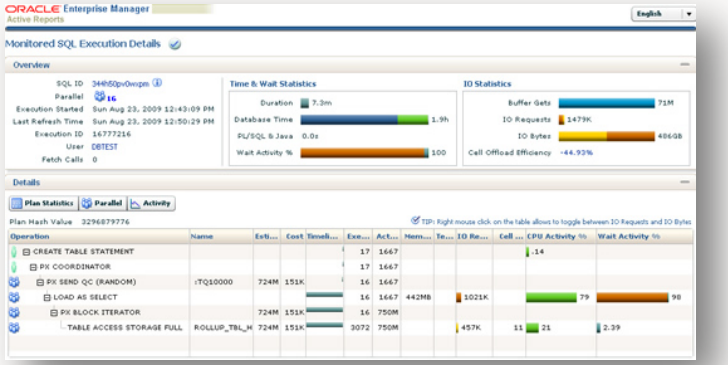
1. What is an execution plan
2. How to generate a plan
3. Understanding execution plans
4. Execution Plan Example

Many ways to view an execution plan

Autotrace

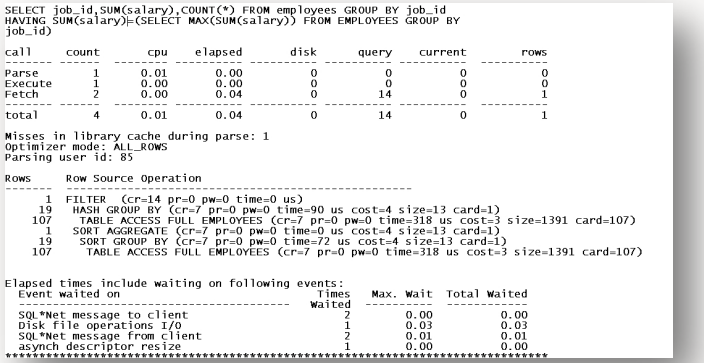


SQL Developer



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

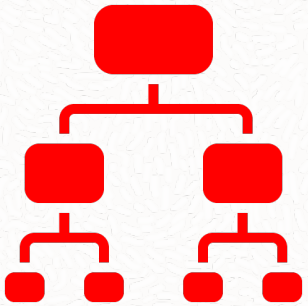
CHILD NUMBER

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
4. Execution Plan Example

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	00:00:				
1	PX COORDINATOR		1		71	00:00:				
2	PX SEND QC (RANDOM)	:TQ10002	0	71	0	00:00:				
3	HASH GROUP BY		0	71	0	00:00:				
4	PX RECEIVE		0	71	0	00:00:				
5	PX SEND HASH	:TQ10001	0	71	0	00:00:				
6	HASH GROUP BY		0	71	0	00:00:				
*7	HASH JOIN		0	918K	0	00:00:				
8	PX RECEIVE		0	72	0	00:00:				
9	PX SEND BROADCAST	:TQ10000	0	72	0	00:00:				
10	PX BLOCK ITERATOR		0	72	0	00:00:				
*11	TABLE ACCESS STORAGE FULL	PRODUCTS	0	72	0	00:00:				
12	PX BLOCK ITERATOR		0	918K	0	00:00:				
*13	TABLE ACCESS STORAGE FULL	SALES	0	918K	0	00:00:				

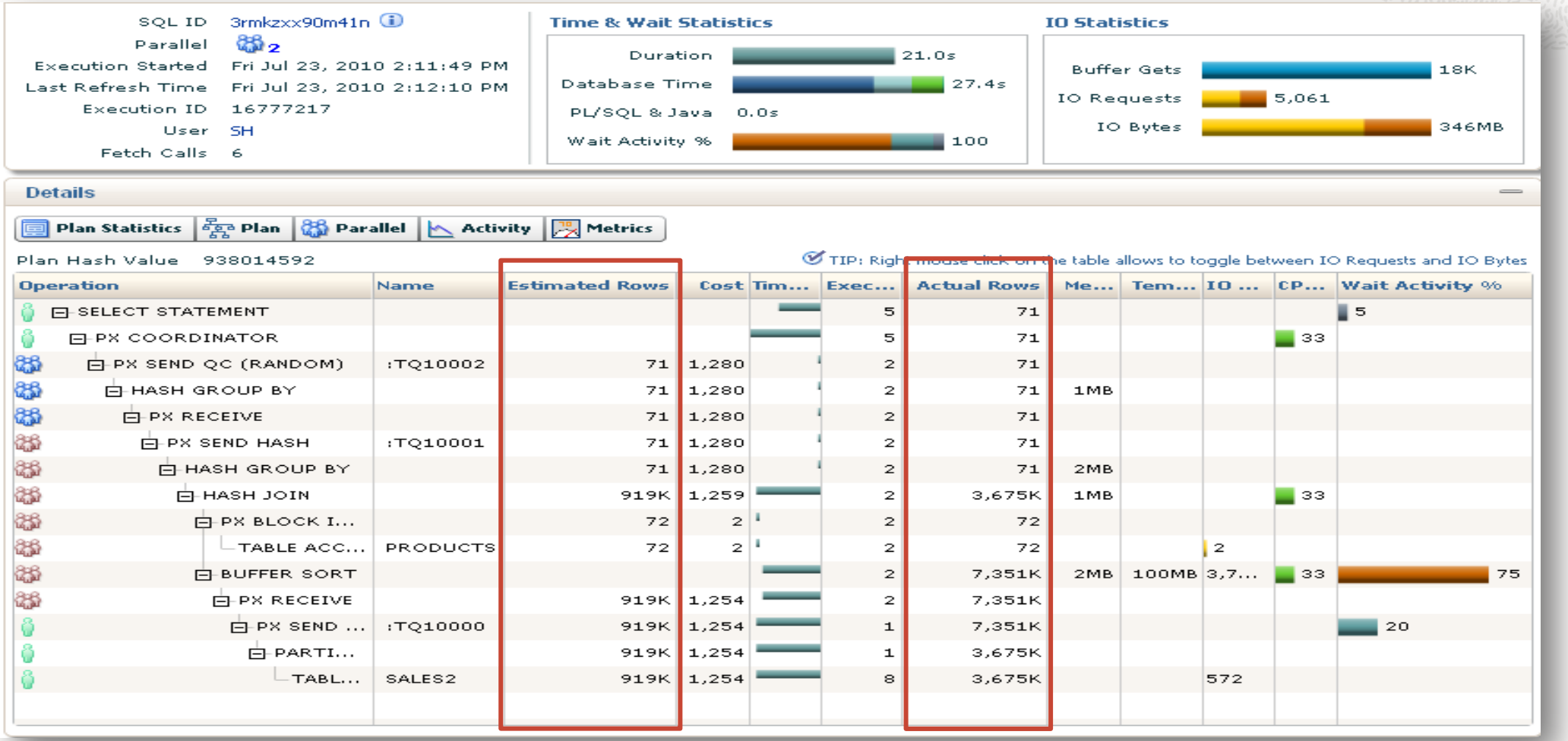
Note: a lot of the data is zero in the A-rows column because we only show last execution of the cursor which is done by the QC. Need to use ALLSTATS ALL to see info on all parallel server processes execution of cursors

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



SQL ID	3rmkzxx90m41n
Parallel	2
Execution Started	Fri Jul 23, 2010 2:11:49 PM
Last Refresh Time	Fri Jul 23, 2010 2:12:10 PM
Execution ID	16777217
User	SH
Fetch Calls	6

Time & Wait Statistics

Duration	21.0s
Database Time	27.4s
PL/SQL & Java	0.0s
Wait Activity %	100

IO Statistics

Buffer Gets	18K
IO Requests	5,061
IO Bytes	346MB

Details

Plan Statistics | Plan | Parallel | Activity | Metrics

Plan Hash Value 938014592

TIP: Right mouse click on the table allows to toggle between IO Requests and IO Bytes

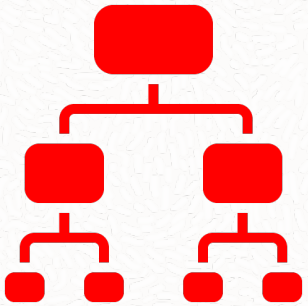
Operation	Name	Estimated Rows	Cost	Tim...	Exec...	Actual Rows	Me...	Tem...	IO ...	CP...	Wait Activity %
SELECT STATEMENT					5	71					5
PX COORDINATOR					5	71				33	
PX SEND QC (RANDOM)	:TQ10002	71	1,280		2	71					
HASH GROUP BY		71	1,280		2	71	1MB				
PX RECEIVE		71	1,280		2	71					
PX SEND HASH	:TQ10001	71	1,280		2	71					
HASH GROUP BY		71	1,280		2	71	2MB				
HASH JOIN		919K	1,259		2	3,675K	1MB			33	
PX BLOCK I...		72	2		2	72					
TABLE ACC...	PRODUCTS	72	2		2	72			2		
BUFFER SORT					2	7,351K	2MB	100MB	3,7...	33	75
PX RECEIVE		919K	1,254		2	7,351K					
PX SEND ...	:TQ10000	919K	1,254		1	7,351K					20
PARTI...		919K	1,254		1	3,675K					
TABL...	SALES2	919K	1,254		8	3,675K			572		

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
4. Execution Plan Example

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 |   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" 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

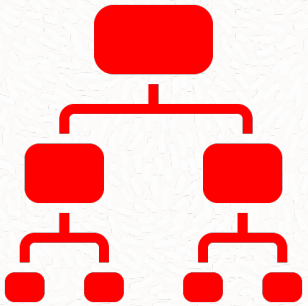
```
SELECT country_id, name
FROM   customers
WHERE country_name = 'USA';
```

```
-------------------------------------------------------------------------------
| 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

Program Agenda

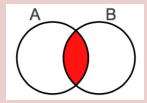
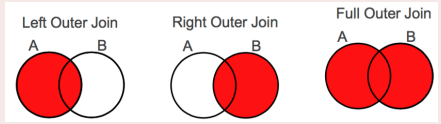
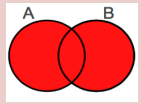
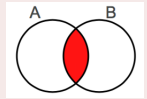
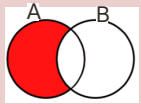


1. What is an execution plan
2. How to generate a plan
3. Understanding execution plans
   - Cardinality
   - Access paths
   - Join methods
   - Join order
4. Execution Plan Example

Join methods

Join Methods	Explanation
Nested Loops joins	For every row in the outer table, Oracle accesses all the rows in the inner table Useful when joining small subsets of data and there is an efficient way to access the second table (index look up)
Hash Joins	The smaller of two tables is scan and resulting rows are used to build a hash table on the join key in memory. The larger table is then scan, join column of the resulting rows are hashed and the values used to probe the hash table to find the matching rows. Useful for larger tables & if equality predicate
Sort Merge joins	Consists of two steps: 1. Sort join operation: Both the inputs are sorted on the join key. 2. Merge join operation: The sorted lists are merged together. Useful when the join condition between two tables is an inequality condition

Join types

Join Type	Explanation
Inner Joins 	Returns all rows that satisfy the join condition
Outer Joins 	Returns all rows that satisfy the join condition and also returns all of the rows from the table without the (+) for which no rows from the other table satisfy the join condition
Cartesian Joins 	Joins every row from one data source with every row from the other data source, creating the Cartesian Product of the two sets. Only good if tables are very small. Only choice if there is no join condition specified in query
Semi-Join 	Returns a row from the outer table when a matching row exists in the subquery data set. Typically used when there is an EXISTS or an IN predicate, where we aren't interested in returning rows from the subquery but merely checking a match exists
Anti-Join 	Returns a row from the outer table when a matching row does not exist in the subquery data set. Typically used when there is a NOT EXISTS or NOT IN predicate, where we aren't interested in returning rows from the subquery but merely checking a match doesn't exists

Identifying join methods 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 |
|  13 |   TABLE ACCESS BY INDEX ROWID  | JOBS        |     1 |
-----------------------------------------------------------------------------------------------

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
```

Look in the Operation section to check the right join method is used

If wrong join type is used check stmt is written correctly & cardinality estimates

Join method example 1

What join method would you expect for this query?

```
SELECT e.last_name, e.salary, d.department_name
FROM   hr.employees e, hr.departments d
WHERE  d.departments_name IN ('Marketing','Sales')
AND    e.department_id = d.department_id;
```

Employees has 107 rows

Departments has 27 rows

Foreign key relationship between Employees and Departments on dept_id

Join method example 1

What join method would you expect for this query?

```
SELECT e.last_name, e.salary, d.department_name
FROM   hr.employees e, hr.departments d
WHERE  d.departments_name IN ('Marketing','Sales')
AND    e.department_id = d.department_id;
```

```
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                     |    19 |   722 |     3   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                |                     |       |       |            |          |
|   2 |   NESTED LOOPS               |                     |    19 |   722 |     3   (0)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL         | DEPARTMENTS         |     2 |    32 |     2   (0)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN          | EMP_DEPARTMENT_IX   |    10 |       |     0   (0)| 00:00:01 |
|   5 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES           |    10 |   220 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   3 - filter("D"."DEPARTMENT_NAME"='Marketing' OR "D"."DEPARTMENT_NAME"='Sales')
   4 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
```

Join method example 2

What join method would you expect for this query?

```
SELECT o.customer_id, l.unit_price * l.quantity
FROM   oe.orders o, oe.order_items l
WHERE  l.order_id = o.order_id;
```

Orders has 105 rows

Order Items has 665 rows

Join method example 2

What join method would you expect for this query?

```
SELECT o.customer_id, l.unit_price * l.quantity
FROM   oe.orders o, oe.order_items l
WHERE  l.order_id = o.order_id;
```

```
-----------------------------------------------------------------------------
| Id  | Operation            |  Name         | Rows  | Bytes | Cost (%CPU)|
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |               |   665 | 13300 |     8  (25)|
|*  1 |  HASH JOIN           |               |   665 | 13300 |     8  (25)|
|   2 |   TABLE ACCESS FULL  | ORDERS        |   105 |   840 |     4  (25)|
|   3 |   TABLE ACCESS FULL  | ORDER_ITEMS   |   665 |  7980 |     4  (25)|
-----------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("L"."ORDER_ID"="O"."ORDER_ID")
```

Join method example 3

What join method would you expect for this query?

```
SELECT o.order_id, o.order_date ,e.name
 FROM  oe.orders o , hr.employees e;
```

Orders has 105 rows

Employees has 107 rows

Join method example 3

What join method would you expect for this query?

```
SELECT o.order_id, o.order_date ,e.name
 FROM  oe.orders o , hr.employees e;
```

```
Plan hash value: 3229651169

-------------------------------------------------------------------------------
| Id  | Operation                | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |             | 11235 |   120K|     33   (7)| 00:00:01 |
|   1 |  MERGE JOIN CARTESIAN    |             | 11235 |   120K|     33   (7)| 00:00:01 |
|   2 |   INDEX FULL SCAN        | ORDER_PK    |   105 |   420 |      1   (0)| 00:00:01 |
|   3 |   BUFFER SORT            |             |   107 |   749 |     32   (7)| 00:00:01 |
|   4 |    INDEX FAST FULL SCAN| EMP_NAME_IX |   107 |   749 |      0   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

Join method example 4

What join method would you expect for this query?

```
SELECT   s.quantity_sold
FROM     sales s, customers c
WHERE    s.cust_id =c.cust_id;
```

Sales table has 960 Rows

Customer table has 55,500 rows

Customer has a primary key created on cust_id

Sales has a foreign key created on cust_id

Join method example 4

What join method would you expect for this query?

```
SELECT   s.quantity_sold
FROM     sales s, customers c
WHERE    s.cust_id =c.cust_id;
```

No join is needed 

Table elimination transformation Optimizer realizes that the join to customers tables is redundant as no columns are selected Presence of primary –foreign key relationship means we can remove table

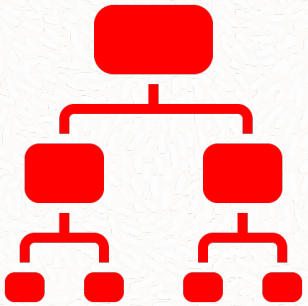
```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------
Plan hash value: 2489314924

------------------------------------------------------------------------------------------
| Id  | Operation           | Name  | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |       |   960 |  2880 |     5   (0)| 00:00:01 |       |       |
|   1 |  PARTITION RANGE ALL|       |   960 |  2880 |     5   (0)| 00:00:01 |     1 |    16 |
|   2 |   TABLE ACCESS FULL | SALES |   960 |  2880 |     5   (0)| 00:00:01 |     1 |    16 |
------------------------------------------------------------------------------------------
```

What causes wrong join method to be selected

Issue	Cause
Nested loop selected instead of hash join	
Hash join selected instead of nested loop	come in a clustered or ordered fashion so the probe into 2nd table is more efficient
Cartesian Joins	Cardinality underestimation

Adaptive Plans in 12c can address these problems on the fly by changing the join method after oracle sees what data is coming out of the left-hand side of the join

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
4. Execution Plan Example

Join order

- The order in which the tables are join in a multi table statement
- Ideally start with the table that will eliminate the most rows
- Strongly affected by the access paths available
- Some basic rules
- Joins guaranteed to produce at most one row always go first
  - Joins between two row sources that have only one row each
- When outer joins are used the table with the outer join operator must come after the other table in the predicate
- If view merging is not possible all tables in the view will be joined before joining to the tables outside the view

Identifying join methods 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 |       1TABLE ACCESS FULL       | DEPARTMENTS  |     1 |    30 |     3   (0)| 00:00:01 |
|   7 |       BUFFER SORT              |              |   107 |  5564 |     3   (0)| 00:00:01 |
|   8 |       2 TABLE ACCESS FULL      | EMPLOYEES    |   107 |  5564 |     3   (0)| 00:00:01 |
|   9 |     3TABLE ACCESS FULL         | EMPLOYEES    |   107 |  7811 |     3   (0)| 00:00:01 |
|* 10 |    4 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 | 5 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
```

Want to start with the table that reduce the result set the most

If the join order is not correct, check the statistics, cardinality & access methods

Finding the join order for complex SQL

It can be hard to determine Join Order for Complex SQL statements but it is easily visible in the outline data of plan

```
SELECT * FROM table(dbms_xplan.display_cursor(format=>'TYPICAL +OUTLINE'));
```

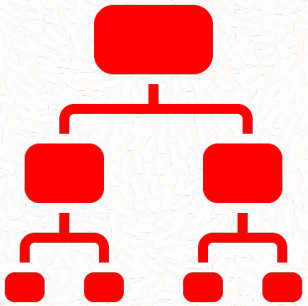
```
Outline Data
-------------

  /*+
      BEGIN_OUTLINE_DATA
      IGNORE_OPTIM_EMBEDDED_HINTS
      OPTIMIZER_FEATURES_ENABLE('11.2.0.2')
      DB_VERSION('11.2.0.2')
      ALL_ROWS
      OUTLINE_LEAF(@"SEL$5428C7F1")
      MERGE(@"SEL$2")
      MERGE(@"SEL$3")
      OUTLINE(@"SEL$1")
      OUTLINE(@"SEL$2")
      OUTLINE(@"SEL$3")
      FULL(@"SEL$5428C7F1" "D"@"SEL$3")
      INDEX_RS_ASC(@"SEL$5428C7F1" "E"@"SEL$3" ("EMPLOYEES"."DEPARTMENT_ID"))
      INDEX_RS_ASC(@"SEL$5428C7F1" "E"@"SEL$2" ("EMPLOYEES"."MANAGER_ID"))
      INDEX_RS_ASC(@"SEL$5428C7F1" "J"@"SEL$2" ("JOBS"."JOB_ID"))
      INDEX(@"SEL$5428C7F1" "D"@"SEL$2" ("DEPARTMENTS"."DEPARTMENT_ID"))
      LEADING(@"SEL$5428C7F1" "D"@"SEL$3" "E"@"SEL$3" "E"@"SEL$2" "J"@"SEL$2" "D"@"SEL$2")
      USE_NL(@"SEL$5428C7F1" "E"@"SEL$3")
```

The leading hint tells you the join order

What causes the wrong join order

Causes
Incorrect single table cardinality estimates
Incorrect join cardinality estimates

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
4. Execution Plan Example

Example SQL Statement

Find all the employees who make as much or more than their manager

```
SELECT  e1.last_name, e1.job_title, e1.total_comp
FROM    (SELECT  e.manager_id, e.last_name, j.job_title,
                 e.salary+(e.salary+e.commission_pct) total_comp
         FROM    employees e, jobs j, departments d
         WHERE   d.department_name = 'Sales'
         AND     e.department_id   = d.department_id
         AND     e.job_id          = j.job_id ) e1,
        (SELECT e.employee_id, e.salary+(e.salary+e.commission_pct) tc
         FROM    employees e, departments d
         WHERE   d.department_name = ‘Sales'
         AND     e.department_id   = d.department_id ) e2
WHERE   e1.manager_id = e2.employee_id
AND     e1.total_comp >= e2.tc;
```

Is it a good execution plan?

1. Is the estimated number of rows being returned accurate?

```
-------------------------------------------------------------------------------------------------
| 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" 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_
  12 - access("E"."JOB_ID"="J"."JOB_ID")

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

3.Are the access method correct?

2. Are the cardinality estimates accurate?

Means no stats gathered strong indicator this won't be best possible plan

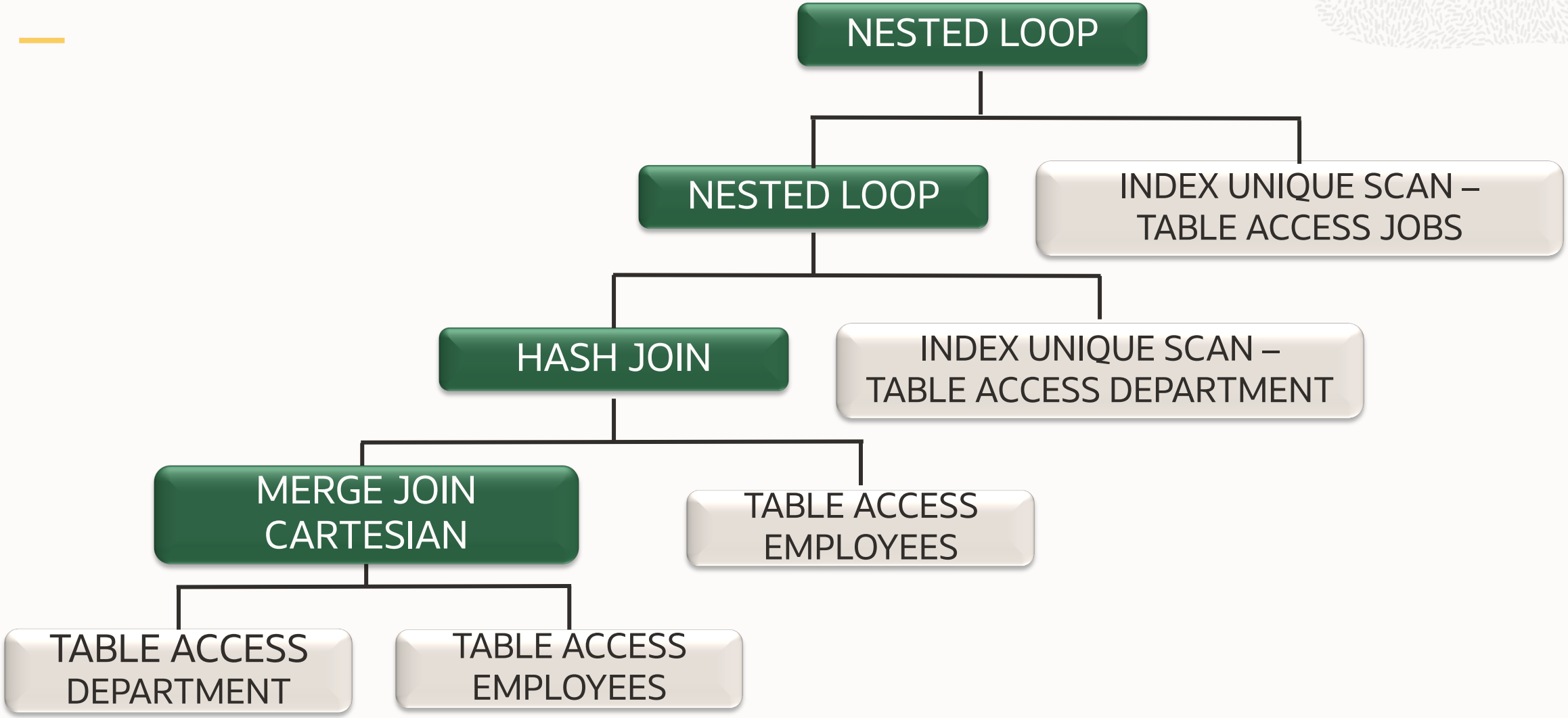
Example cont'd execution plan

```
-------------------------------------------------------------------------------------------
| 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 |       1 TABLE ACCESS FULL      | DEPARTMENTS |     1 |    30 |     3   (0)| 00:00:01 |
|   7 |        BUFFER SORT             |             |   107 |  5564 |     3   (0)| 00:00:01 |
|   8 |       2 TABLE ACCESS FULL      | EMPLOYEES   |   107 |  5564 |     3   (0)| 00:00:01 |
|   9 |     3 TABLE ACCESS FULL        | EMPLOYEES   |   107 |  7811 |     3   (0)| 00:00:01 |
|* 10 |    4 TABLE ACCESS BY INDEX ROWID| DEPARTMENTS |     1 |    30 |     1   (0)| 00:00:01 |
|* 11 |     4 INDEX UNIQUE SCAN        | DEPT_ID_PK  |     1 |       |     0   (0)|          |
|* 12 |    INDEX UNIQUE SCAN           | JOB_ID_PK   |     1 |       |     0   (0)|          |
|  13 |  5 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"."COMMISSI
              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")

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

4. Are the right join methods being used?

5. Is the join order correct? Is the table that eliminates the most rows accessed first?

What does the plan tree look like?



Solution

```
| Id  | Operation                     | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
|   0 | SELECT STATEMENT              |                     |       |       |     8 (100)|          |
|   1 |  NESTED LOOPS                 |                     |       |       |            |          |
|   2 |   NESTED LOOPS                |                     |     1 |   102 |     8   (0)| 00:00:01 |
|   3 |    NESTED LOOPS               |                     |     1 |    86 |     7   (0)| 00:00:01 |
|   4 |     NESTED LOOPS              |                     |     1 |    59 |     6   (0)| 00:00:01 |
|   5 |      NESTED LOOPS             |                     |    10 |   290 |     4   (0)| 00:00:01 |
|*  6 |       TABLE ACCESS FULL       | DEPARTMENTS         |     1 |    16 |     3   (0)| 00:00:01 |
|   7 |       TABLE ACCESS BY INDEX ROWID| EMPLOYEES        |    10 |   130 |     1   (0)| 00:00:01 |
|   8 |        INDEX RANGE SCAN       | EMP_DEPARTMENT_IX   |    10 |       |     0   (0)|          |
|   9 |      TABLE ACCESS BY INDEX ROWID | EMPLOYEES        |     1 |    30 |     1   (0)| 00:00:01 |
|  10 |       INDEX RANGE SCAN        | EMP_MANAGER_IX      |     6 |       |     0   (0)|          |
|  11 |     TABLE ACCESS BY INDEX ROWID | JOBS              |     1 |    27 |     1   (0)| 00:00:01 |
|* 12 |      INDEX UNIQUE SCAN        | JOB_ID_PK           |     1 |       |     0   (0)|          |
|* 13 |    INDEX UNIQUE SCAN          | DEPT_ID_PK          |     1 |       |     0   (0)|          |
|* 14 |   TABLE ACCESS BY INDEX ROWID | DEPARTMENTS         |     1 |    16 |     1   (0)| 00:00:01 |

Predicate Information (identified by operati

   6 - filter("D"."DEPARTMENT_NAME"='Sales')
   8 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
   9 - filter("E"."SALARY"+("E"."SALARY"+"E"."COMMISSION_PCT")>="E"."S
              "."COMMISSION_PCT"))
  10 - access("E"."MANAGER_ID"="E"."EMPLOYEE_ID")
  12 - access("E"."JOB_ID"="J"."JOB_ID")
  13 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
  14 - filter("D"."DEPARTMENT_NAME"='Sales')
```

1. Only 1 row is actually returned, and the cost is 4 lower now

2. Cardinalities are correct and with each join number of rows reduced

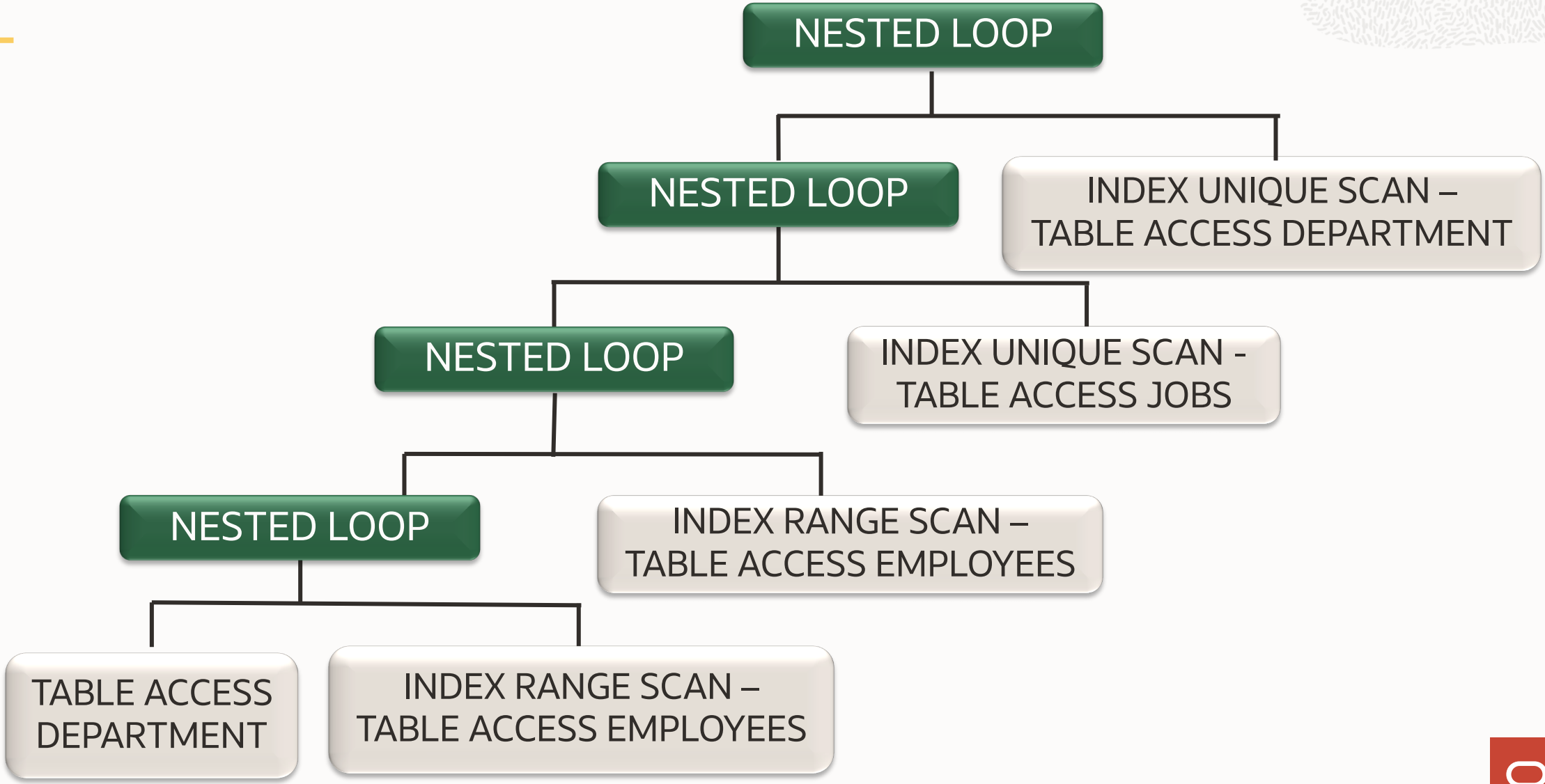
3. Access methods have changed for some tables

4. Join methods have changed to be all NL

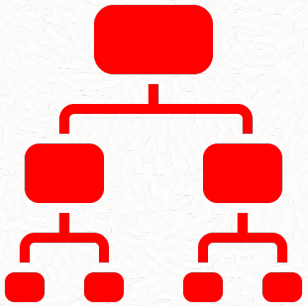
5. The join order has changed

1
2
3
4
5

What does the plan tree look like?



Program Agenda

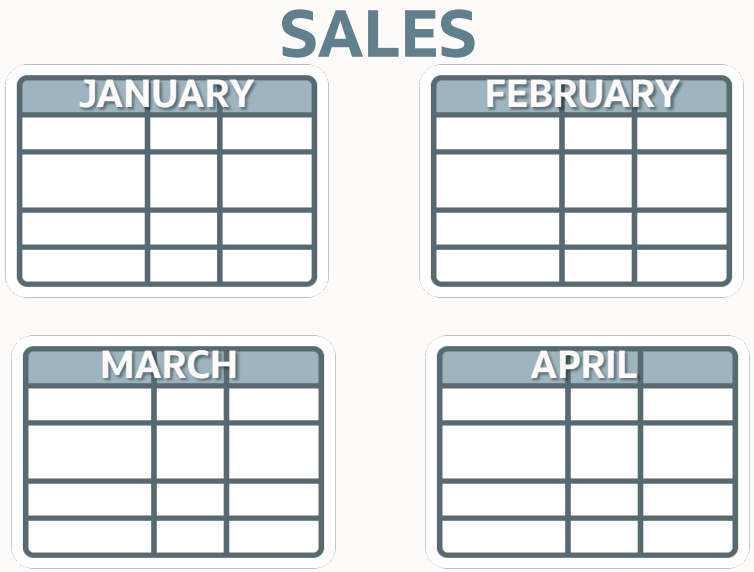


1. What is an execution plan
2. How to generate a plan
3. Understanding execution plans
4. Execution Plan Example
5. Partitioning
6. Parallel Execution

Partitioning Provides Flexibility & Efficiency at Scale

Large Table

Difficult to Manage



Partitions

Divide and Conquer

Easier to Manage

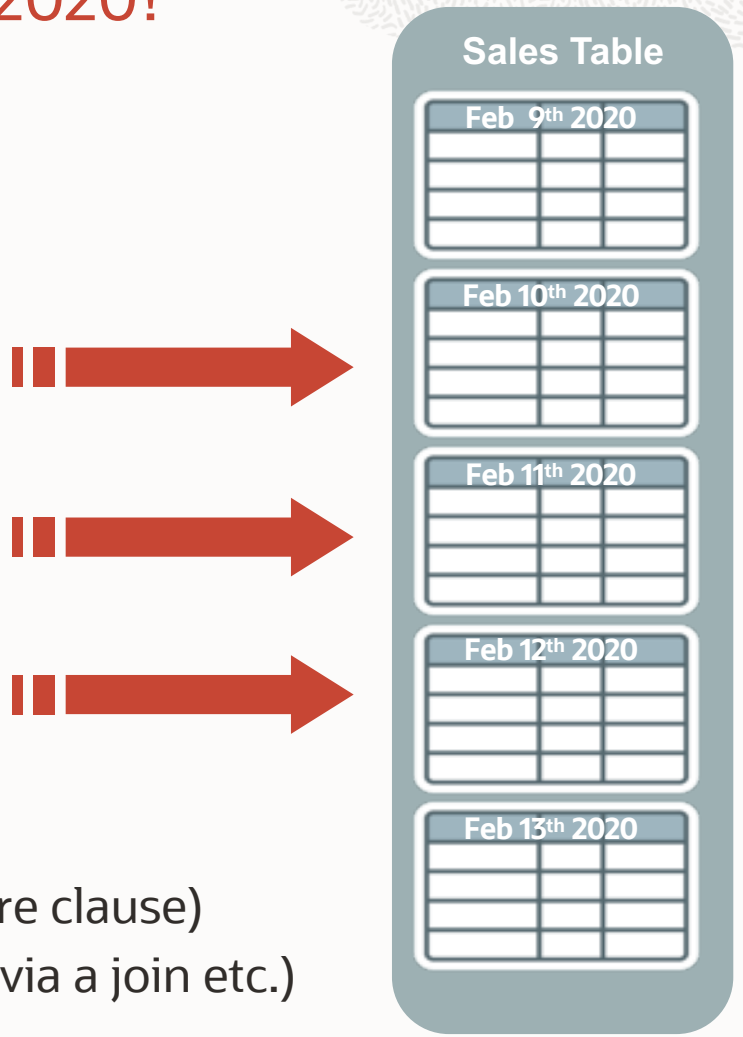
Improve Performance

Transparent to applications

Partition pruning

What was the total sales for the weekend of Feb 10 - 12 2020?

```
SELECT SUM(s.sales_amount)
FROM sales s
WHERE s.sales_date
      BETWEEN to_date('02/10/2020','MM/DD/YYYY')
      AND     to_date('02/10/2020','MM/DD/YYYY');
```



Only the 3 relevant partitions are accessed

- Partition Pruning takes two forms
  - Static pruning occurs when partitions are known in advance (via where clause)
  - Dynamic pruning occurs when partitions are only known at runtime (via a join etc.)

Identifying partition pruning in a plan

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop
0	SELECT STATEMENT		1407	54873	166 (83)	00:00:02		
1	PX COORDINATOR							
2	PX SEND QC (ORDER)	:TQ10003	1407	54873	166 (83)	00:00:02		
3	VIEW		1407	54873	166 (83)	00:00:02		
4	SORT GROUP BY		1407	148K	166 (83)	00:00:02		
5	PX RECEIVE		1407	148K	166 (83)	00:00:02		
6	PX SEND RANGE	:TQ10002	1407	148K	166 (83)	00:00:02		
7	HASH GROUP BY		1407	148K	166 (83)	00:00:02		
* 8	HASH JOIN		1407	148K	165 (83)	00:00:02		
9	PART JOIN FILTER CREATE	:BF0000	1408	78848	7 (29)	00:00:01		
10	PX RECEIVE		1408	78848	7 (29)	00:00:01		
11	PX SEND PARTITION (KEY)	:TQ10001	1408	78848	7 (29)	00:00:01		
* 12	HASH JOIN		1408	78848	7 (29)	00:00:01		
13	PX RECEIVE		139	3614	2 (0)	00:00:01		
14	PX SEND BROADCAST	:TQ10000	139	3614	2 (0)	00:00:01		
15	PX BLOCK ITERATOR		139	3614	2 (0)	00:00:01		
16	TABLE ACCESS FULL	HOGAN_PCODE_HD_REF	139	3614	2 (0)	00:00:01		
17	PX BLOCK ITERATOR		1408	42240	4 (25)	00:00:01	1	4
18	TABLE ACCESS FULL	T_ACCT_MASTER_HD	1408	42240	4 (25)	00:00:01	1	64
19	PX PARTITION RANGE SINGLE		18919	960K	157 (85)	00:00:02	14	14
20	PX PARTITION HASH JOIN-FILTER		18919	960K	157 (85)	00:00:02	:BF0000	:BF0000
* 21	TABLE ACCESS FULL	T_TRAN_DETAIL_HD	18919	960K	157 (85)	00:00:02	53	56

Pstart and Pstop list the partition touched by the query

If you see the word ‘KEY’ listed it indicate dynamic pruning

Partition pruning

Numbering of partitions

```
SELECT COUNT(*)
FROM RHP_TAB
WHERE CUST_ID = 9255 AND TIME_ID = '2020-02-10';
```

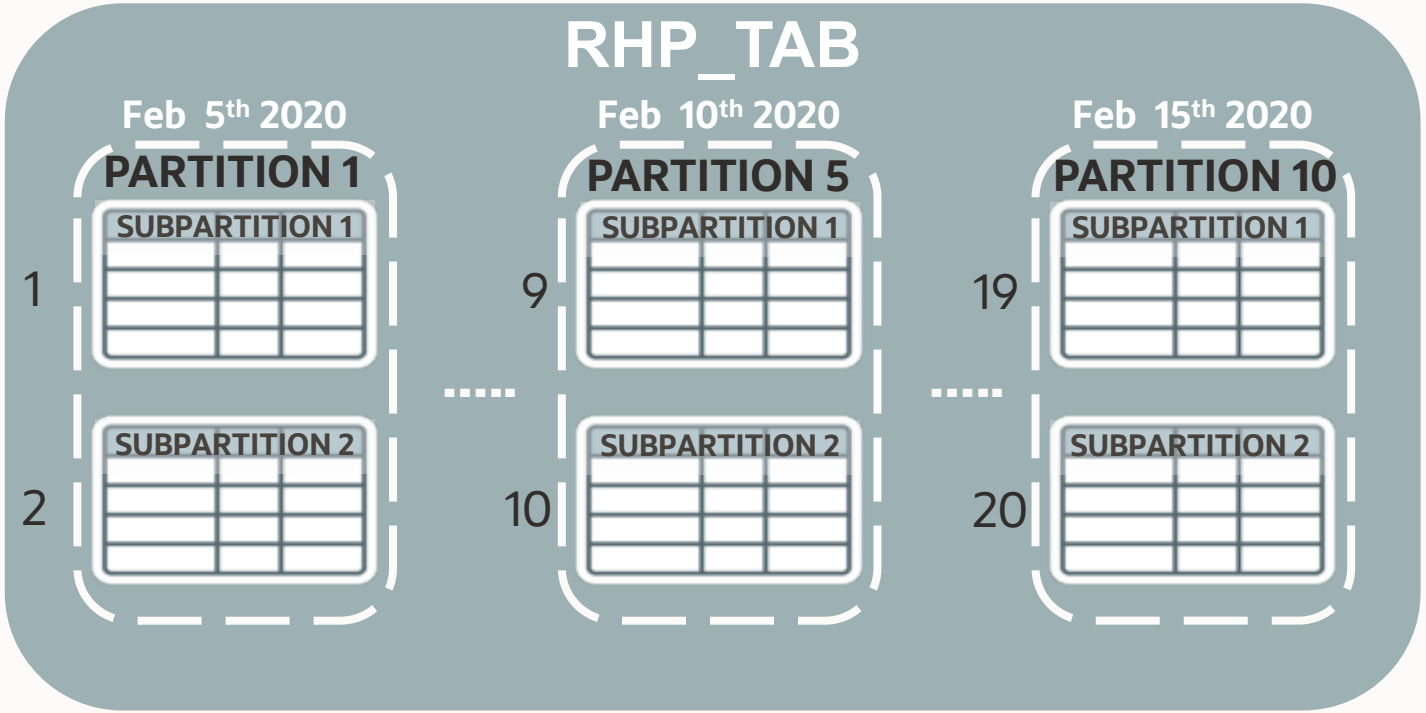
Why so many numbers in the Pstart / Pstop columns?

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop
0	SELECT STATEMENT				5 (100)			
1	SORT AGGREGATE		1	13				
2	PARTITION RANGE SINGLE		1	13	5 (0)	00:00:01	5	5
3	PARTITION HASH SINGLE		1	13	5 (0)	00:00:01	2	2
* 4	TABLE ACCESS FULL	RHP_TAB	1	13	5 (0)	00:00:01	10	10

Partition pruning

Numbering of partitions

- An execution plan show partition numbers for static pruning
- Each partition is numbered 1 to N
- Within each partition subpartitions are numbered 1 to M
- Each physical object in the table is given an overall partition number from 1 to N*M



Partition pruning

Numbering of partitions

```
SELECT COUNT(*)
FROM RHP_TAB
WHERE CUST_ID = 9255 AND TIME_ID = '2020-02-10';
```

Why so many numbers in the Pstart / Pstop columns?

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop
0	SELECT STATEMENT				5 (100)			
1	SORT AGGREGATE		1	13				
2	PARTITION RANGE SINGLE		1	13	5 (0)	00:00:01	5	5
3	PARTITION HASH SINGLE		1	13	5 (0)	00:00:01	2	2
* 4	TABLE ACCESS FULL	RHP_TAB	1	13	5 (0)	00:00:01	10	10

Range partition #

Sub-partition #

Overall partition #

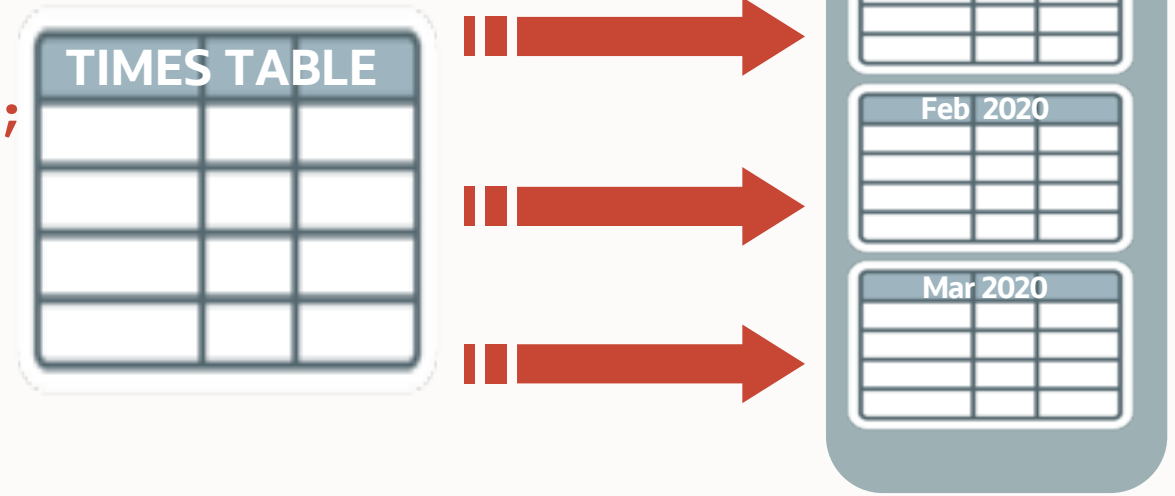
Partition pruning

Dynamic partition pruning

- Advanced Pruning mechanism for complex queries
- Recursive statement evaluates the relevant partitions at runtime
  - Look for the word ‘KEY’ in PSTART/PSTOP columns

```
SELECT sum(amount_sold)
FROM   sales s, times t
WHERE  t.time_id = s.time_id
AND    t.calendar_month_desc IN
      (‘JAN-20’,‘FEB-20’,‘MAR-20’);
```

- Dynamic partition pruning via a join



Partition pruning

Dynamic partition pruning

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop
0	SELECT STATEMENT				13 (100)			
1	SORT AGGREGATE		1	28				
2	NESTED LOOPS		2	56	13 (0)	00:00:01		
*3	TABLE ACCESS FULL	TIMES	2	32	13 (8)	00:00:01		
4	PARTITION RANGE ITERATOR		2	24	0 (0)		KEY	KEY
*5	TABLE ACCESS FULL	SALES	2	24	0 (0)		KEY	KEY

Identifying partition pruning in a plan

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop	TQ	IN-OUT	PQ Distrib
0	SELECT STATEMENT		1407	54873	166 (83)	00:00:02					
1	PX COORDINATOR										
2	PX SEND QC (ORDER)	:TQ10003	1407	54873	166 (83)	00:00:02			Q1,03	P->S	QC (ORDER)
3	VIEW		1407	54873	166 (83)	00:00:02			Q1,03	PCWP	
4	SORT GROUP BY		1407	148K	166 (83)	00:00:02			Q1,03	PCWP	
5	PX RECEIVE		1407	148K	166 (83)	00:00:02			Q1,03	PCWP	
6	PX SEND RANGE	:TQ10002	1407	148K	166 (83)	00:00:02			Q1,02	P->P	RANGE
7	HASH GROUP BY		1407	148K	166 (83)	00:00:02			Q1,02	PCWP	
* 8	HASH JOIN		1407	148K	165 (83)	00:00:02			Q1,02	PCWP	
9	PART JOIN FILTER CREATE	:BF0000	1408	78848	7 (29)	00:00:01			Q1,02	PCWP	
10	PX RECEIVE		1408	78848	7 (29)	00:00:01			Q1,02	PCWP	
11	PX SEND PARTITION (KEY)	:TQ10001	1408	78848	7 (29)	00:00:01			Q1,01	P->P	PART (KEY)
* 12	HASH JOIN		1408	78848	7 (29)	00:00:01			Q1,01	PCWP	
13	PX RECEIVE		139	3614	2 (0)	00:00:01			Q1,01	PCWP	
14	PX SEND BROADCAST	:TQ10000	139	3614	2 (0)	00:00:01			Q1,00	P->P	BROADCAST
15	PX BLOCK ITERATOR		139	3614	2 (0)	00:00:01			Q1,00	PCWC	
16	TABLE ACCESS FULL	HOGAN_PCODE_HD_REF	139	3614	2 (0)	00:00:01			Q1,00	PCWP	
17	PX BLOCK ITERATOR		1408	42240	4 (25)	00:00:01	1	4	Q1,01	PCWC	
18	TABLE ACCESS FULL	T_ACCT_MASTER_HD	1408	42240	4 (25)	00:00:01	1	64	Q1,01	PCWP	
19	PX PARTITION RANGE SINGLE		18919	960K	157 (85)	00:00:02	14	14	Q1,02	PCWC	
20	PX PARTITION HASH JOIN-FILTER		18919	960K	157 (85)	00:00:02	:BF0000	:BF0000	Q1,02	PCWC	
* 21	TABLE ACCESS FULL	T_TRAN_DETAIL_HD	18919	960K	157 (85)	00:00:02	53	56	Q1,02	PCWP	

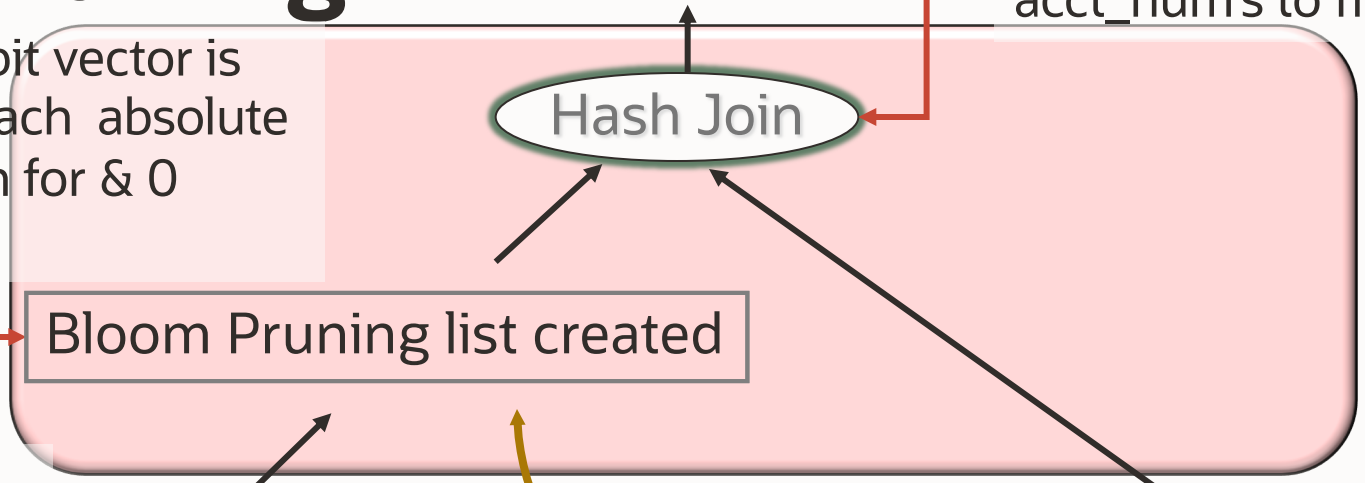
- What does :BF0000 mean?

Pstart and Pstop list the partition touched by the query

Bloom Pruning

5. Hash Join: Join completed by probing into the hash table from the trans acct_num's to find actual matching rows

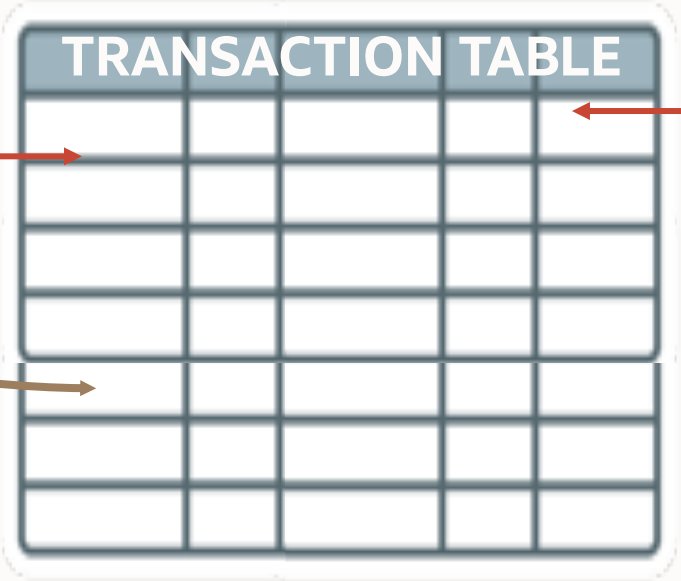
2. Bloom Pruning list: A bit vector is created that has a 1 for each absolute partition there is a match for & 0 where there is no match



4. Reduced row sent: Only rows that match the additional predicates from partitions that match the bit vector get sent hash join

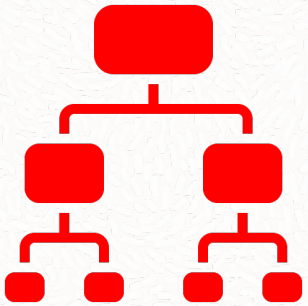
1. Table scan: ACCTS table is scanned and matching rows to hash join

Bit vector of partitions



3. Table Scan: Only partitions on the TRANS table that have a 1 in the bit vector will be scanned

Program Agenda



1. What is an execution plan
2. How to generate a plan
3. Understanding execution plans
4. Execution Plan Example
5. Partitioning
6. Parallel Execution

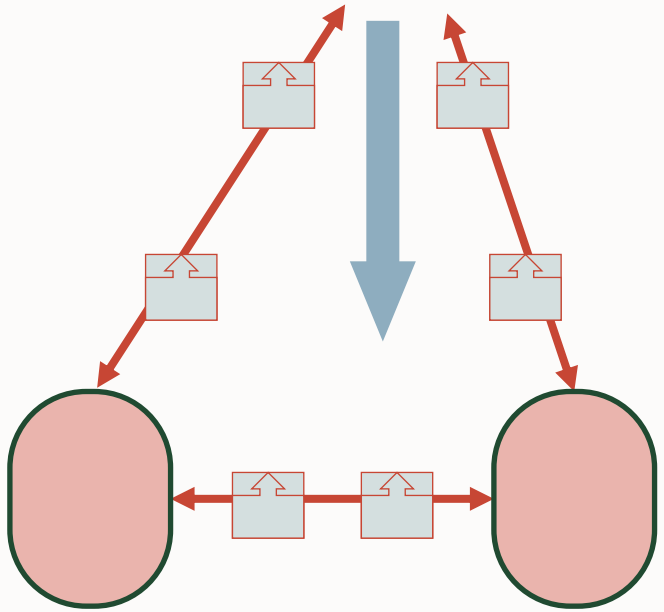
Parallel Execution

How it works

When user issues a parallel SQL statement the background process becomes the Query Coordinator

QC gets parallel servers from global pool and distributes the work to them

Parallel servers communicate among themselves & the QC using messages that are passed via memory buffers in the shared pool



Parallel servers - individual sessions that perform work in parallel Allocated from a pool of globally available parallel server processes & assigned to a given operation

Identifying parallel execution in the plan

Steps completed by PX Coordinator versus Parallel Server Processes

```
SELECT c.cust_last_name, s.time_id, s.amount_sold
FROM sales s, customers c
WHERE s.cust_id = c.cust_id;
```

Query Coordinator

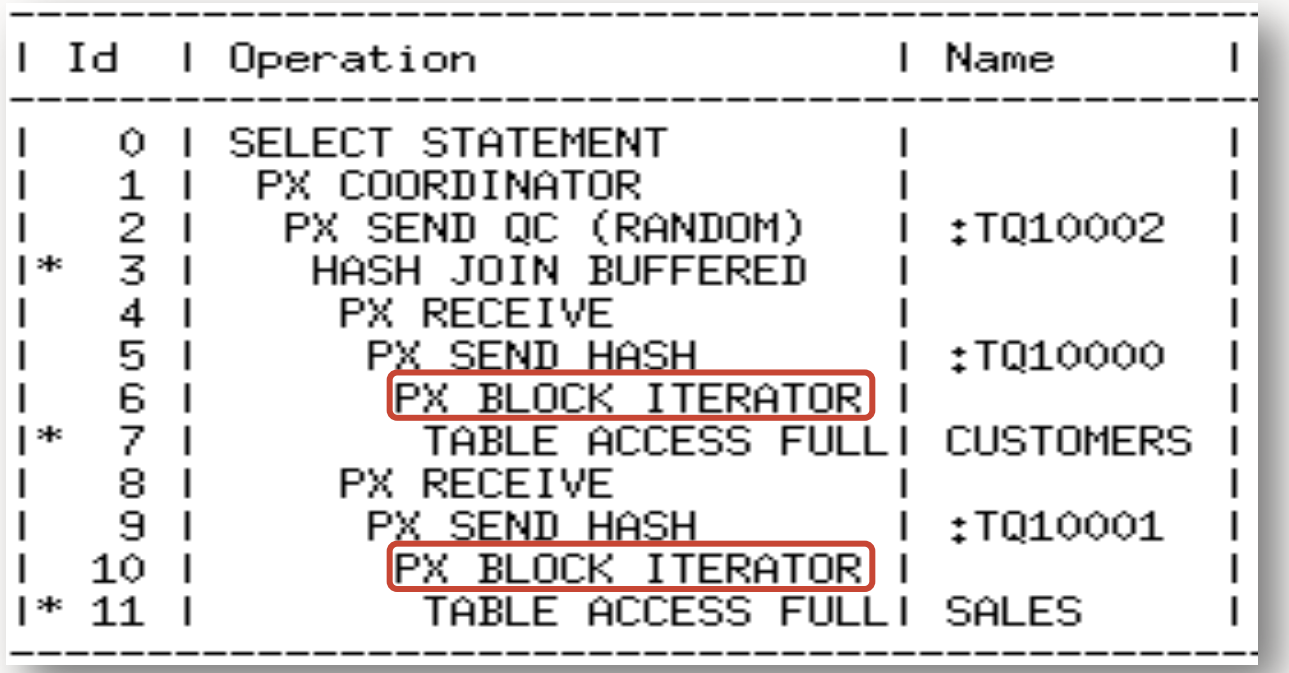
Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT				311 (100)	
1	PX COORDINATOR					
2	PX SEND QC (RANDOM)	:TQ10002	1049K	31M	311 (2)	00:00:04
* 3	HASH JOIN BUFFERED		1049K	31M	311 (2)	00:00:04
4	PX RECEIVE		55500	704K	112 (0)	00:00:02
5	PX SEND HASH	:TQ10000	55500	704K	112 (0)	00:00:02
6	PX BLOCK ITERATOR		55500	704K	112 (0)	00:00:02
* 7	TABLE ACCESS FULL	CUSTOMERS	55500	704K	112 (0)	00:00:02
8	PX RECEIVE		1049K	18M	196 (2)	00:00:03
9	PX SEND HASH	:TQ10001	1049K	18M	196 (2)	00:00:03
10	PX BLOCK ITERATOR					
* 11	TABLE ACCESS FULL	SALES				

Parallel Servers do majority of the work

Identifying granules of parallelism in the plan

How work is divided up among the parallel server processes

- Data is divided into granules
  - Block ranges
  - Partition
- Each parallel server is allocated one or more granules
- The granule method is specified on the line above the scan operation in the plan

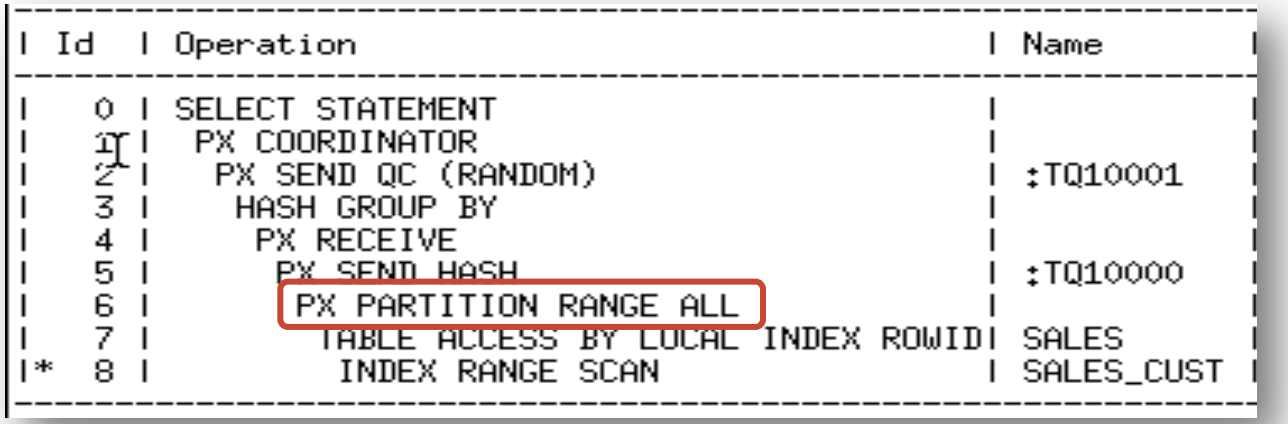


```
-------------------------------------------------------------------
| Id  | Operation                          | Name        |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |             |
|   1 |  PX COORDINATOR                    |             |
|   2 |   PX SEND QC (RANDOM)              | :TQ10002    |
|*  3 |    HASH JOIN BUFFERED              |             |
|   4 |     PX RECEIVE                     |             |
|   5 |      PX SEND HASH                  | :TQ10000    |
|   6 |       PX BLOCK ITERATOR            |             |
|*  7 |        TABLE ACCESS FULL           | CUSTOMERS   |
|   8 |     PX RECEIVE                     |             |
|   9 |      PX SEND HASH                  | :TQ10001    |
|  10 |       PX BLOCK ITERATOR            |             |
|* 11 |        TABLE ACCESS FULL           | SALES       |
-------------------------------------------------------------------
```

Identifying granules of parallelism in the plan

How work is divided up among the parallel server processes

- Data is divided into granules
  - Block ranges
  - Partition
- Each parallel server is allocated one or more granules
- The granule method is specified on the line above the scan operation in the plan



Id	Operation	Name
0	SELECT STATEMENT	
1	PX COORDINATOR	
2	PX SEND QC (RANDOM)	:TQ10001
3	HASH GROUP BY	
4	PX RECEIVE	
5	PX SEND HASH	:TQ10000
6	PX PARTITION RANGE ALL	
7	TABLE ACCESS BY LOCAL INDEX ROWID	SALES
* 8	INDEX RANGE SCAN	SALES_CUST

Access paths and how they are parallelized

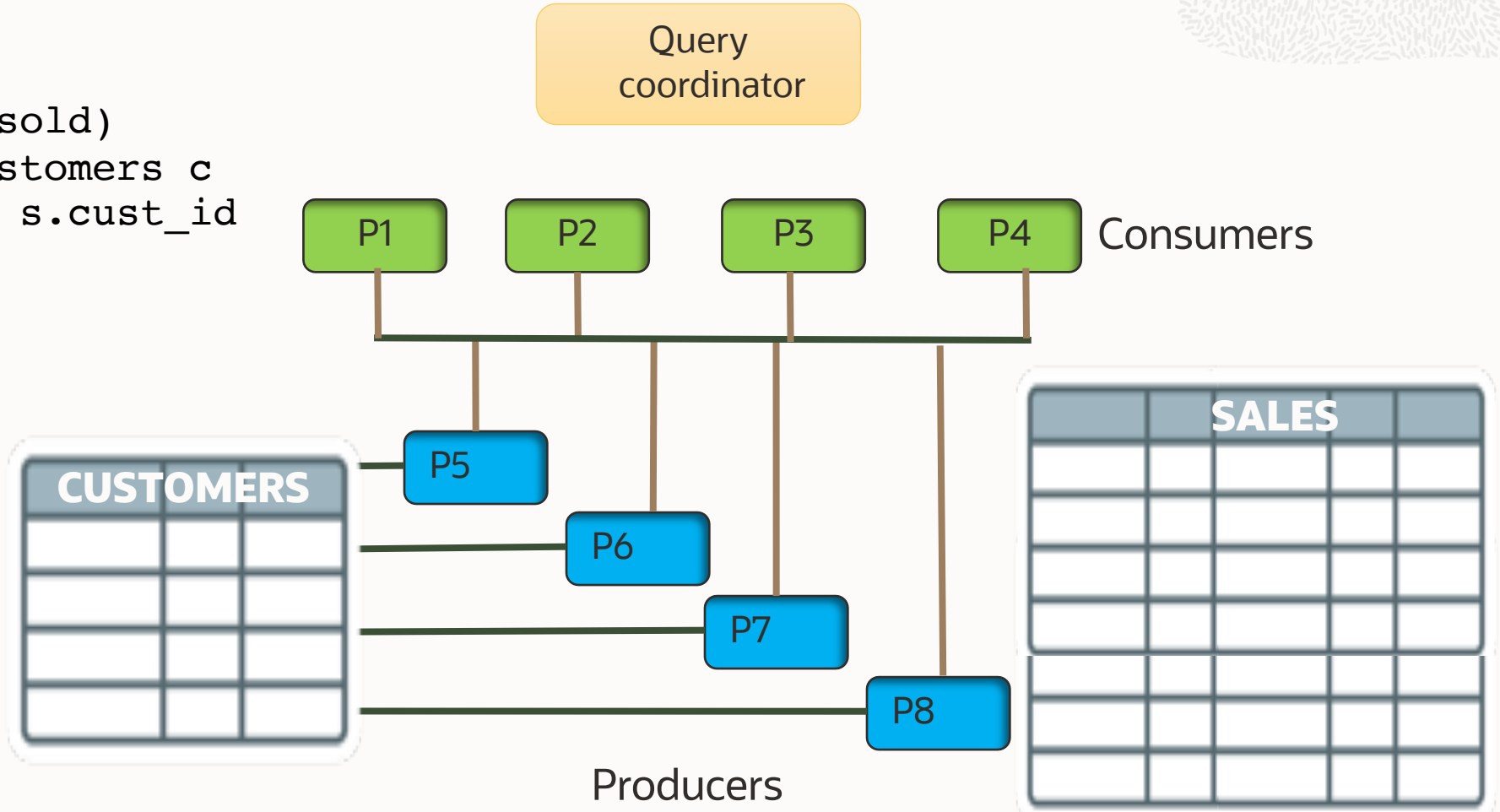
Access Paths	Parallelization method
Full table scan	Block Iterator
Table accessed by Rowid	Partition
Index unique scan	Partition
Index range scan (descending)	Partition
Index skip scan	Partition
Full index scan	Partition
Fast full index scan	Block Iterator
Bitmap indexes (in Star Transformation)	Block Iterator

How parallel execution works

Parallel Hash Join

```
SELECT sum(amount_sold)
FROM   sales s, customers c
WHERE  c.cust_id = s.cust_id
```

1. Hash join always begins with a scan of the smaller table. In this case that's is the customer table. The 4 producers scan the customer table and send the resulting rows to the consumers



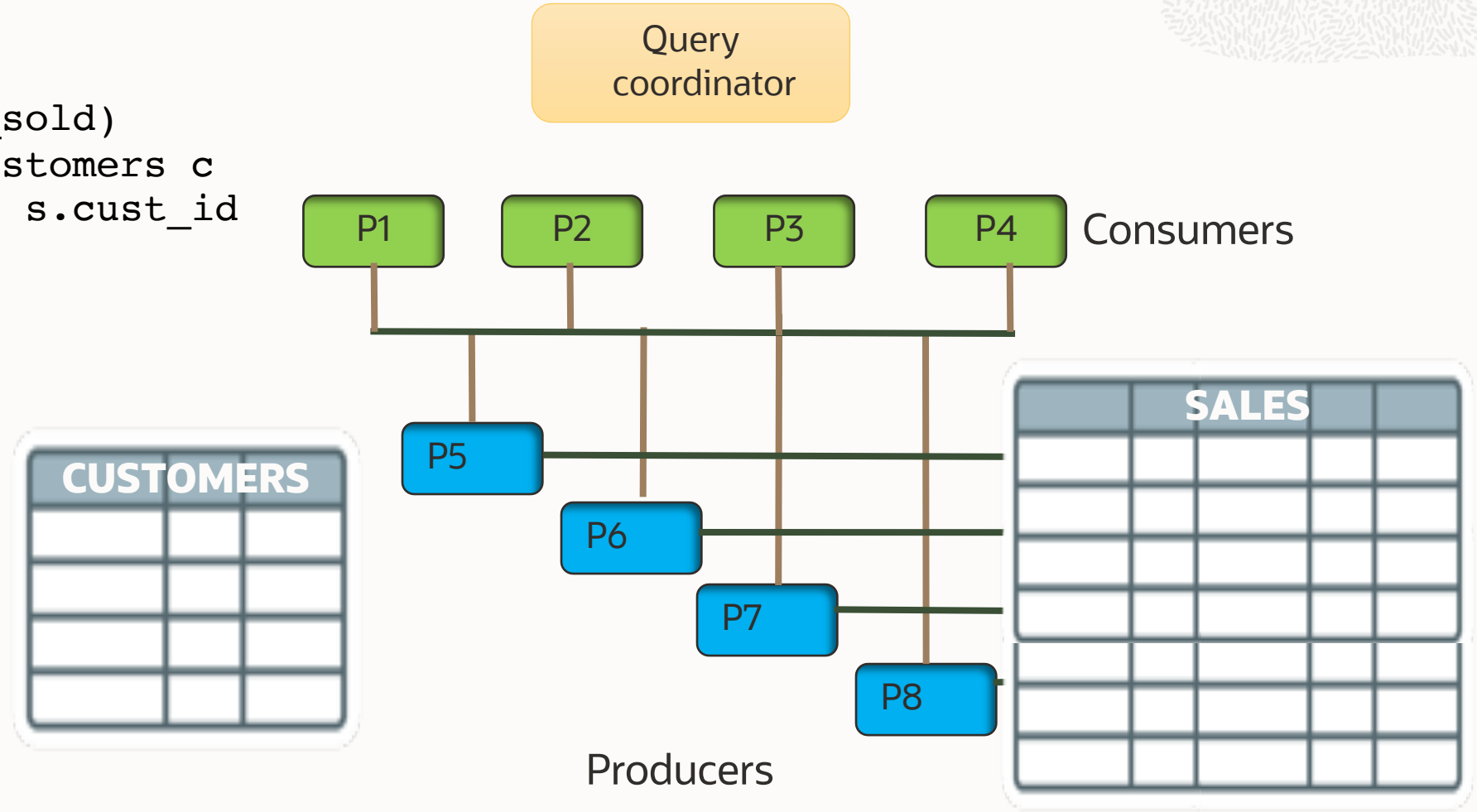
Hash Join with a Parallel Degree of 4

How parallel execution works

Parallel Hash Join

```
SELECT sum(amount_sold)
FROM   sales s, customers c
WHERE  c.cust_id = s.cust_id
```

2. Once the 4 producers finish scanning the customer table, they start to scan the Sales table and send the resulting rows to the consumers

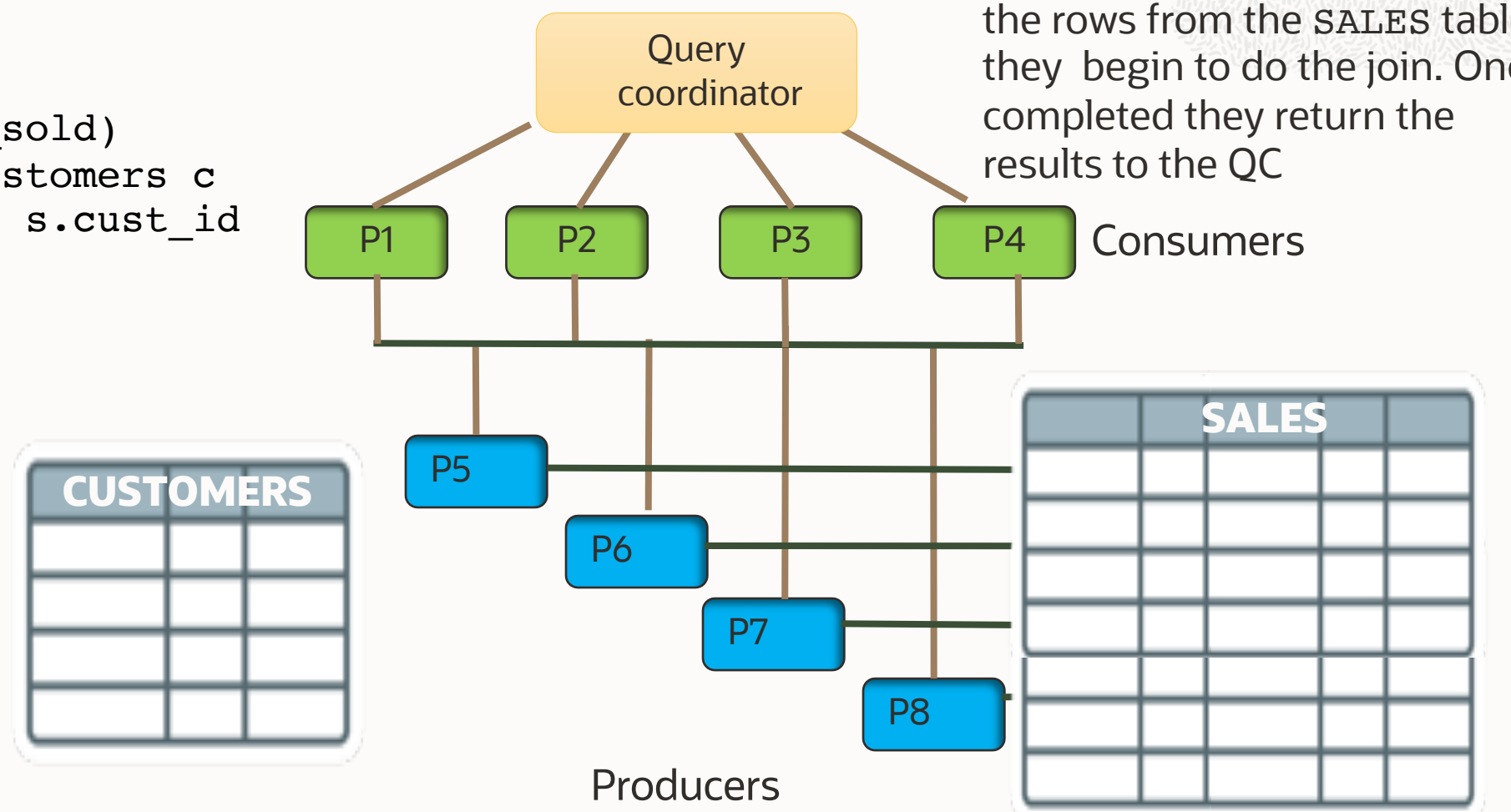


Hash Join with a Parallel Degree of 4

How parallel execution works

Parallel Hash Join

```
SELECT sum(amount_sold)
FROM   sales s, customers c
WHERE  c.cust_id = s.cust_id
```



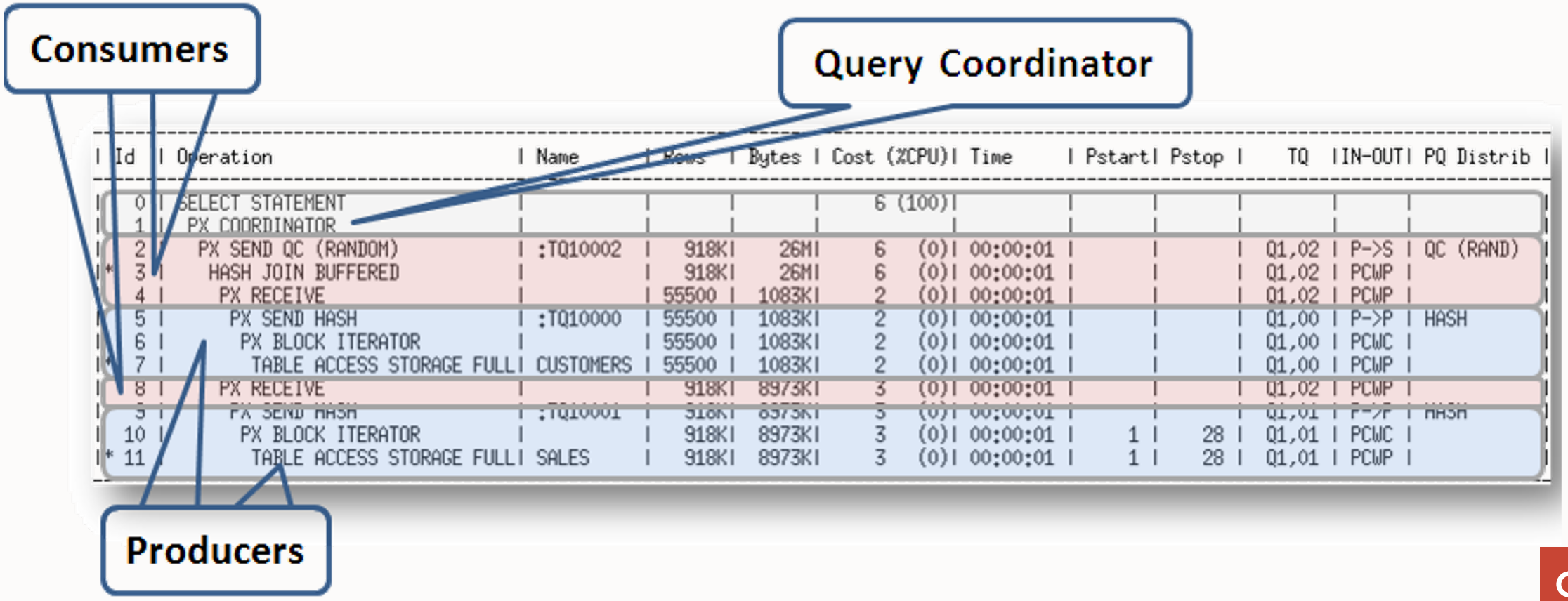
3. Once the consumers receive the rows from the SALES table they begin to do the join. Once completed they return the results to the QC

Hash Join with a Parallel Degree of 4

How parallel execution works

Parallel Hash Join

```
SELECT sum(amount_sold)
FROM   sales s, customers c
WHERE  c.cust_id = s.cust_id
```

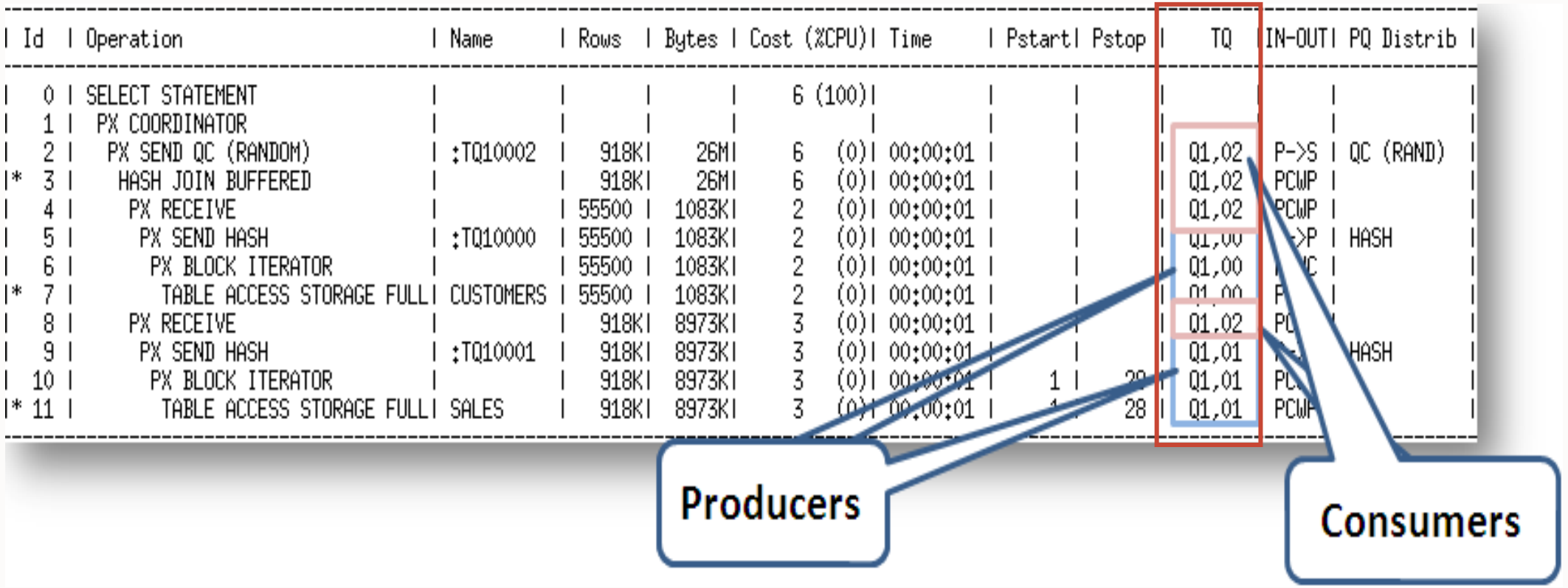


How parallel execution works

Parallel Hash Join

```
SELECT sum(amount_sold)
FROM   sales s, customers c
WHERE  c.cust_id = s.cust_id
```

TQ column shows parallel server sets



Parallel distribution

- Necessary when producers & consumers sets are used
- Producers must pass or distribute their data into consumers
- Operator into which the rows flow decides the distribution
- Distribution can be local or across other nodes in RAC
- Five common types of redistribution

Parallel distribution

- HASH
  - Hash function applied to value of the join column
  - Distribute to the consumer working on the corresponding hash partition
- Round Robin
  - Randomly but evenly distributes the data among the consumers
- Broadcast
  - The size of one of the result sets is small
  - Sends a copy of the data to all consumers

Parallel distribution

- Range
  - Typically used for parallel sort operations
  - Individual parallel servers work on data ranges
  - QC doesn't sort just present the parallel server results in the correct order
- Partitioning Key Distribution – PART (KEY)
  - Assumes that the target table is partitioned
  - Partitions of the target tables are mapped to the parallel servers
  - Producers will map each scanned row to a consumer based on partitioning column
- LOCAL suffix on the redistribution methods in a RAC database
  - An optimization in RAC the rows are distributed to only the consumers on the same RAC node

Indentifying parallel distribution in the plan

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop	TQ	IN-OUT	PQ Distrib
0	SELECT STATEMENT		1407	54873	166 (83)	00:00:02					
1	PX COORDINATOR										
2	PX SEND QC (ORDER)	:TQ10003	1407	54873	166 (83)	00:00:02			Q1,03	P->S	QC (ORDER)
3	VIEW		1407	54873	166 (83)	00:00:02			Q1,03	PCWP	
4	SORT GROUP BY		1407	148K	166 (83)	00:00:02			Q1,03	PCWP	
5	PX RECEIVE		1407	148K	166 (83)	00:00:02			Q1,03	PCWP	
6	PX SEND RANGE	:TQ10002	1407	148K	166 (83)	00:00:02			Q1,02	P->P	RANGE
7	HASH GROUP BY		1407	148K	166 (83)	00:00:02			Q1,02	PCWP	
* 8	HASH JOIN		1407	148K	165 (83)	00:00:02			Q1,02	PCWP	
9	PART JOIN FILTER CREATE	:BF0000	1408	78848	7 (29)	00:00:01			Q1,02	PCWP	
10	PX RECEIVE		1408	78848	7 (29)	00:00:01			Q1,02	PCWP	
11	PX SEND PARTITION (KEY)	:TQ10001	1408	78848	7 (29)	00:00:01			Q1,01	P->P	PART (KEY)
* 12	HASH JOIN		1408	78848	7 (29)	00:00:01			Q1,01	PCWP	
13	PX RECEIVE		139	3614	2 (0)	00:00:01			Q1,01	PCWP	
14	PX SEND BROADCAST	:TQ10000	139	3614	2 (0)	00:00:01			Q1,00	P->P	BROADCAST
15	PX BLOCK ITERATOR		139	3614	2 (0)	00:00:01			Q1,00	PCWC	
16	TABLE ACCESS FULL	HOGAN_PCODE_HD_REF	139	3614	2 (0)	00:00:01			Q1,00	PCWP	
17	PX BLOCK ITERATOR		1408	42240	4 (25)	00:00:01	1	4	Q1,01	PCWC	
18	TABLE ACCESS FULL	T_ACCT_MASTER_HD	1408	42240	4 (25)	00:00:01	1	64	Q1,01	PCWP	
19	PX PARTITION RANGE SINGLE		18919	960K	157 (85)	00:00:02	14	14	Q1,02	PCWC	
20	PX PARTITION HASH JOIN-FILTER		18919	960K	157 (85)	00:00:02	:BF0000	:BF0000	Q1,02	PCWC	
* 21	TABLE ACCESS FULL	T_TRAN_DETAIL_HD	18919	960K	157 (85)	00:00:02	53	56	Q1,02	PCWP	

Shows how the PQ servers distribute rows between each other

Adaptive Distribution Method

NEW IN 12.1

Hybrid-HASH Distribution method

Cardinality based distribution skew common scenario

- Crucial for parallel join of very small data sets with very large data sets

Distribution method decision based on expected number of rows

New adaptive distribution method HYBRID-HASH

- Statistic collectors inserted in front of PX process on the left hand side of the join
- If actual number of rows less than threshold, switch from HASH to Broadcast
  - Threshold number of total rows < 2x DOP

Enabled by default

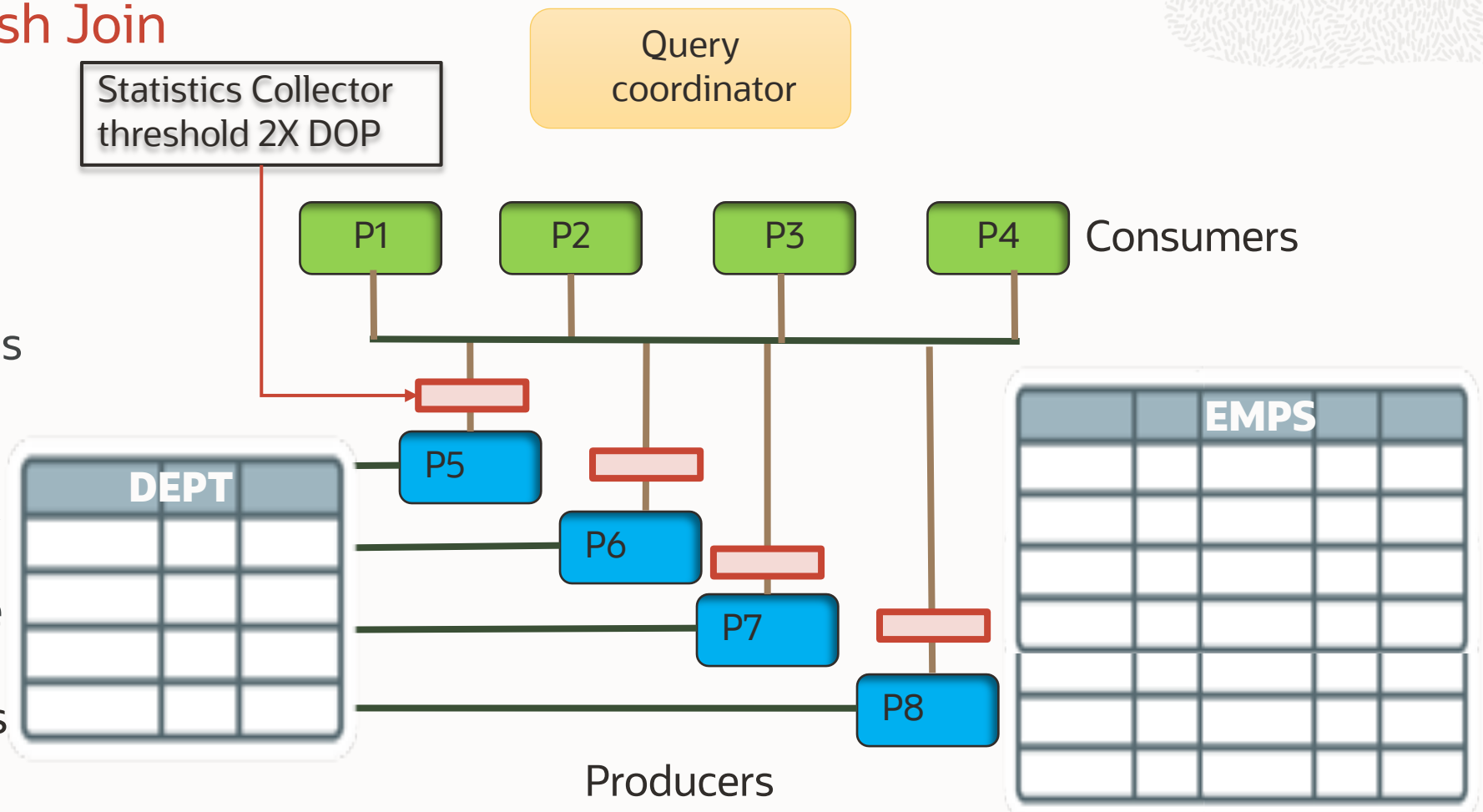
Adaptive Distribution Methods

Hybrid Parallel Hash Join

Hybrid hash join between `DEPTS` and `EMPS`
Distribution method based on runtime stats

Statistics collector inserted in front of PX processes scanning the `CUSTOMERS` table

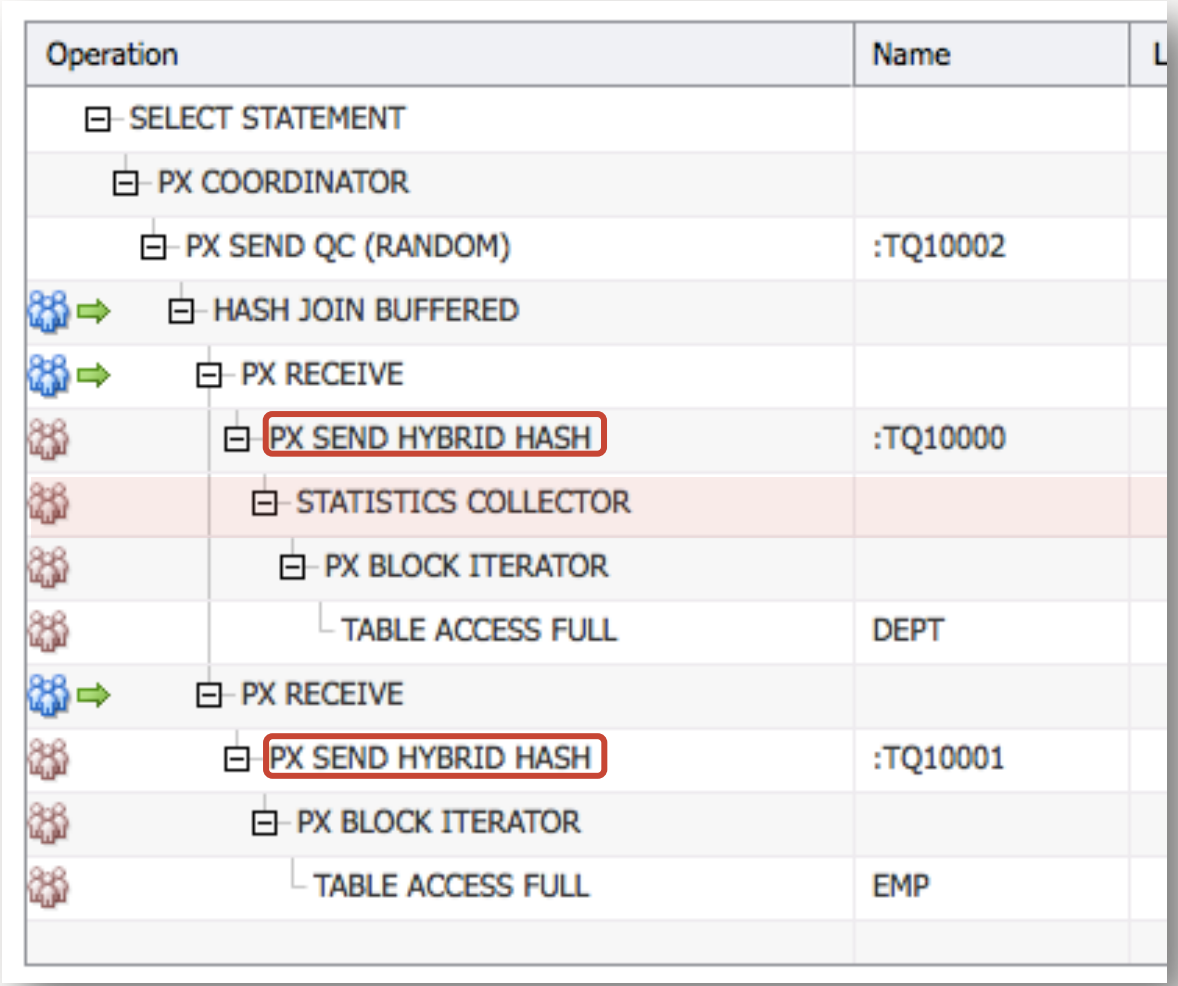
If # rows returned less than threshold, rows distributed via Broadcast



Hash Join with a Parallel Degree of 4

Adaptive Distribution Methods

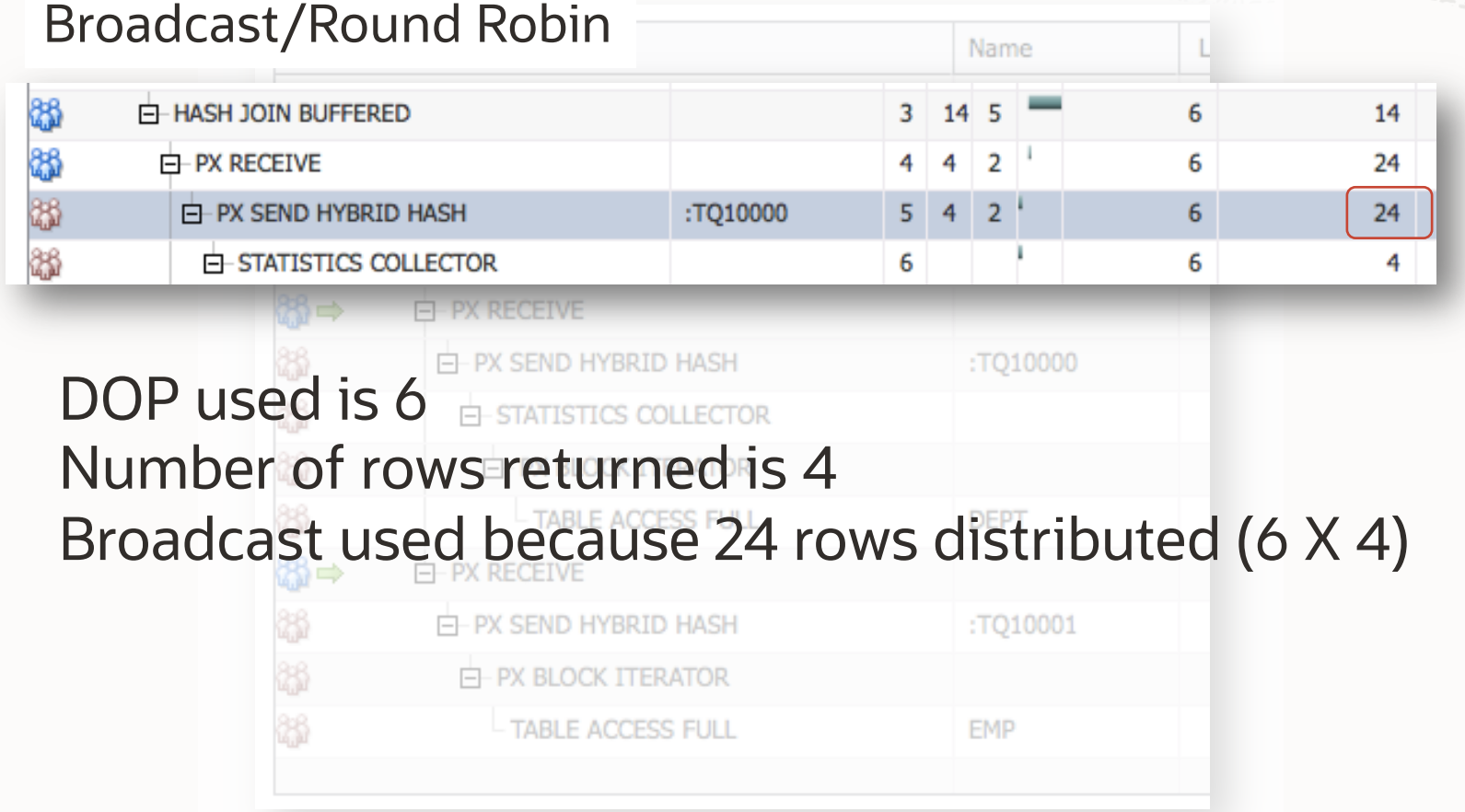
- Hybrid hash join between `EMP` and `DEPT`
- Distribution method based on runtime stats
- Statistics collector inserted in front of PX processes scanning `DEPT`



Adaptive Distribution Methods

- If DEPT uses BROADCAST
  - EMP uses ROUND-ROBIN

Broadcast/Round Robin

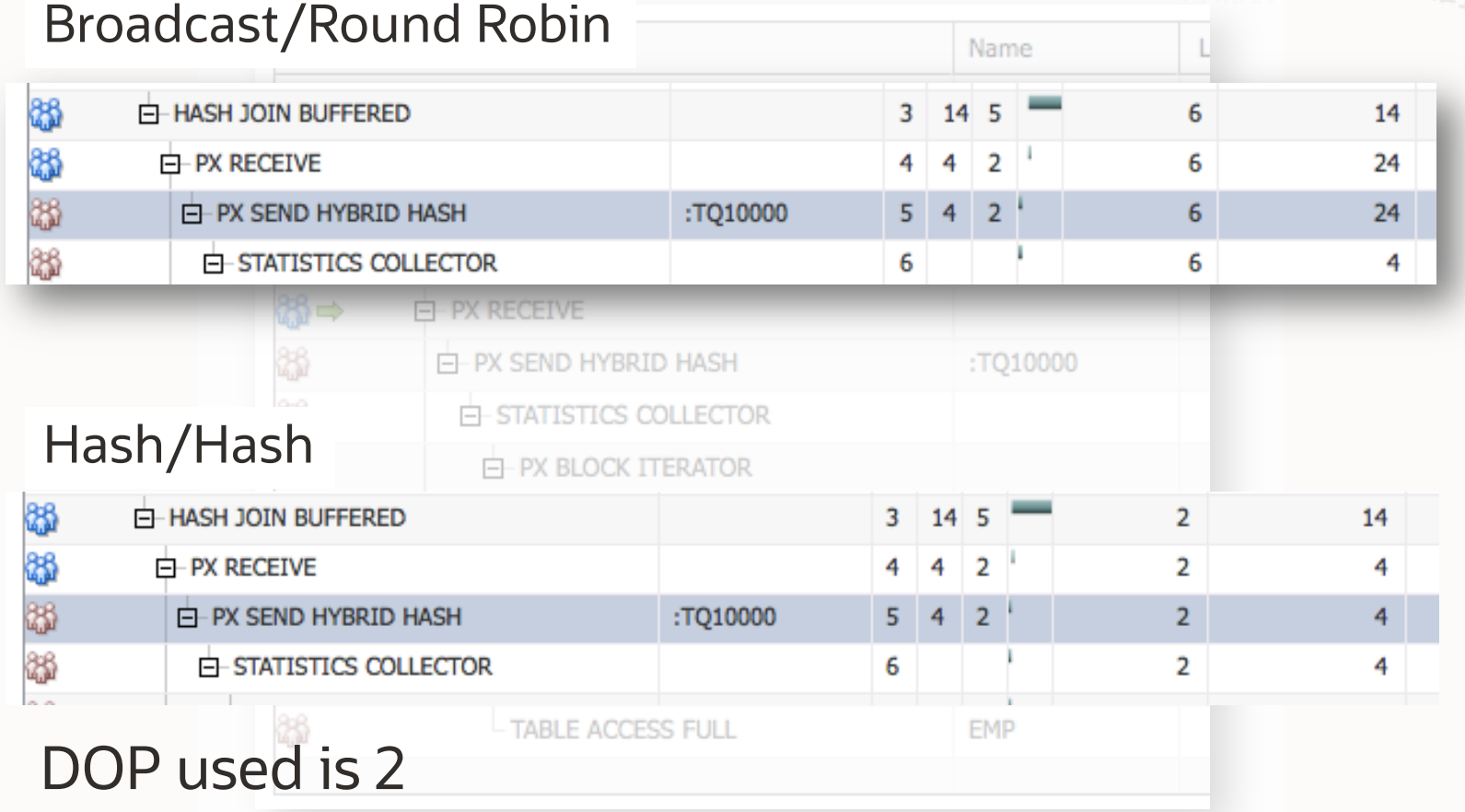


DOP used is 6
Number of rows returned is 4
Broadcast used because 24 rows distributed (6 X 4)

Adaptive Distribution Methods

- If DEPT uses BROADCAST
  - EMP uses ROUND-ROBIN
- If DEPT used HASH
  - EMP uses HASH

Broadcast/Round Robin



Hash/Hash

DOP used is 2
Hash used because only 4 rows distributed

Identifying parallel execution in a plan

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop	TQ	IN-OUT	PQ Distrib
0	SELECT STATEMENT		1407	54873	166 (83)	00:00:02					
1	PX COORDINATOR										
2	PX SEND QC (ORDER)	:TQ10003	1407	54873	166 (83)	00:00:02			Q1,03	P->S	QC (ORDER)
3	VIEW		1407	54873	166 (83)	00:00:02			Q1,03	PCWP	
4	SORT GROUP BY		1407	148K	166 (83)	00:00:02			Q1,03	PCWP	
5	PX RECEIVE		1407	148K	166 (83)	00:00:02			Q1,03	PCWP	
6	PX SEND RANGE	:TQ10002	1407	148K	166 (83)	00:00:02			Q1,02	P->P	RANGE
7	HASH GROUP BY		1407	148K	166 (83)	00:00:02			Q1,02	PCWP	
*8	HASH JOIN		1407	148K	165 (83)	00:00:02			Q1,02	PCWP	
9	PART JOIN FILTER CREATE	:BF0000	1408	78848	7 (29)	00:00:01			Q1,02	PCWP	
10	PX RECEIVE		1408	78848	7 (29)	00:00:01			Q1,02	PCWP	
11	PX SEND PARTITION (KEY)	:TQ10001	1408	78848	7 (29)	00:00:01			Q1,01	P->P	PART (KEY)
*12	HASH JOIN		1408	78848	7 (29)	00:00:01			Q1,01	PCWP	
13	PX RECEIVE		139	3614	2 (0)	00:00:01			Q1,01	PCWP	
14	PX SEND BROADCAST	:TQ10000	139	3614	2 (0)	00:00:01			Q1,00	P->P	BROADCAST
15	PX BLOCK ITERATOR		139	3614	2 (0)	00:00:01			Q1,00	PCWC	
16	TABLE ACCESS FULL	HOGAN_PCODE_HD_REF	139	3614	2 (0)	00:00:01			Q1,00	PCWP	
17	PX BLOCK ITERATOR		1408	42240	4 (25)	00:00:01	1	4	Q1,01	PCWC	
18	TABLE ACCESS FULL	T_ACCT_MASTER_HD	1408	42240	4 (25)	00:00:01	1	64	Q1,01	PCWP	
19	PX PARTITION RANGE SINGLE		18919	960K	157 (85)	00:00:02	14	14	Q1,02	PCWC	
20	PX PARTITION HASH JOIN-FILTER		18919	960K	157 (85)	00:00:02	:BF0000	:BF0000	Q1,02	PCWC	
*21	TABLE ACCESS FULL	T_TRAN_DETAIL_HD	18919	960K	157 (85)	00:00:02	53	56	Q1,02	PCWP	

IN-OUT column shows which step is run in parallel and if it is a single parallel server set or not

- PCWP - Parallel Combined With Parent - operation occurs when the database performs this step simultaneously with the parent step
- P->P - Parallel to Parallel – data is being sent from one parallel operation to another
- P->S - Parallel to Serial - data is being sent to serial operation always happen on the step below the QC

NOTE If the line begins with an S then that step is executed serial – check DOP & access method

SQLMaria®

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