

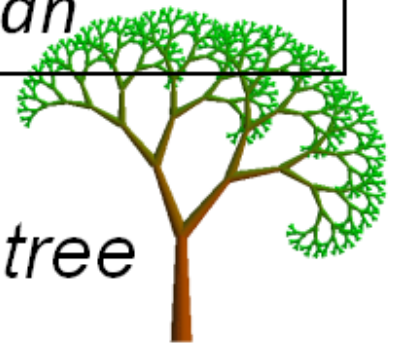


Query Tuning Foundations

Execution Plan

C - Interpret the Plan

Walk through the tree



Muriel Fry - Version V1.4 - Sept 2009

- ***Execution Plan***
 - A - Row Source Operations
 - B - Find the Plan
 - ***C - Interpret the Plan***



Execution Plan

- The execution plan of a SQL statement is composed of small building blocks called “row sources.”
- The combination of row sources for a statement is called the execution plan.
- By using parent-child relationships, the explain plan can be displayed in a tree (Text or Graphical)

ORACLE®

Interpret the Plan

An execution plan is the output of the optimizer and is presented to the execution engine for implementation. It instructs the execution engine as to the operations it needs to perform to retrieve the data required by a query most efficiently. The implementation of these operations is down to the execution engine to implement.

When a statement is executed, the optimizer performs many steps. Each of these steps either retrieves rows of data physically from the database or prepares them in some way for the user issuing the statement. The combination of steps that are used to run a statement is called an *execution plan*.

An execution plan includes an access method for each table that the statement accesses and an ordering of the tables (the join order). The optimizer also uses different methods to combine the rows from multiple tables (join method). The steps of the execution plan are not performed in the order in which they are numbered. there is parent – child relationship between steps.



Objectives

- After completing this lesson, you should be able to do the following:
 - Interpret Execution plan
 - Simplify complex Execution Plan to be able to interpret it

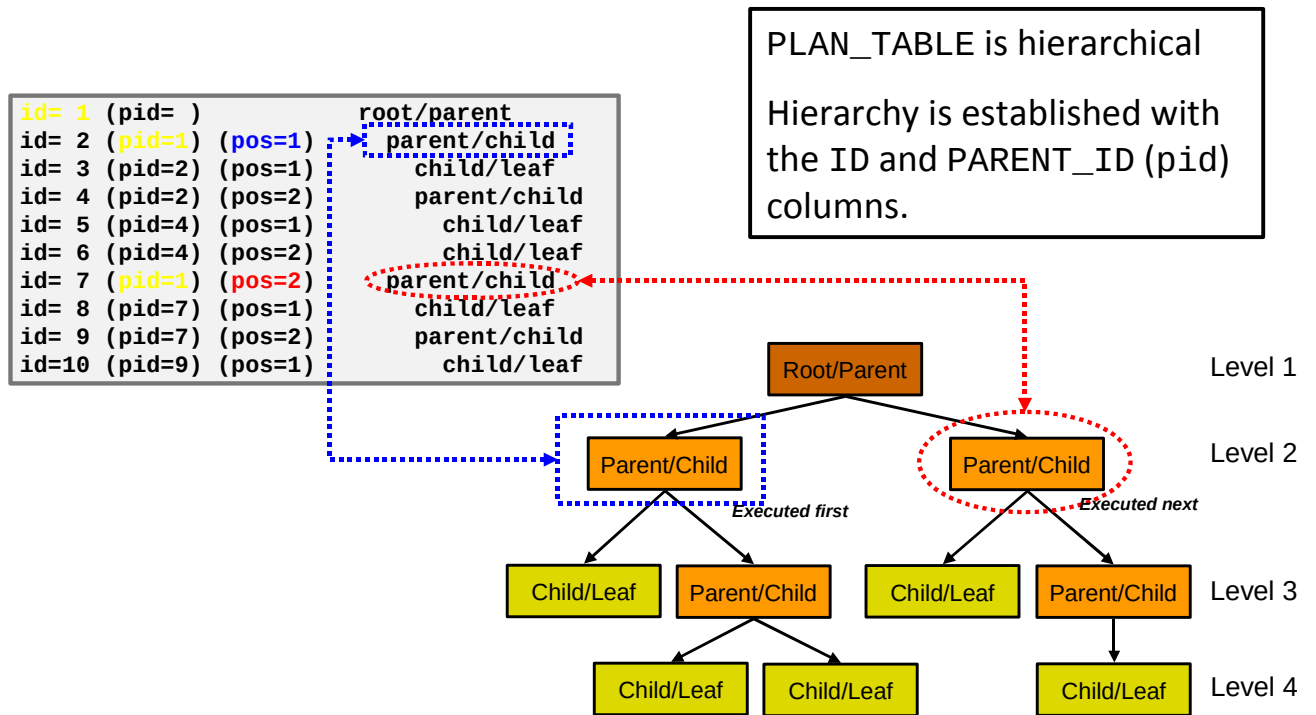
ORACLE

Objectives

Objective from this part is to be more familiar with plan itself.

- How to gather it (there are different source available)
- How to display it (there are different detail level available)
- How to walk through the tree and interpret plan

Transform it into a tree



ORACLE

Transform it into a Tree

Whatever the method to gather and display the explain plan, the basic format and goal is the same.

Scripts provided in \$ORACLE_HOME/rdbms/admin directory to display plan

These scripts have been improved with DBMS_XPLAN availability

```
@?/rdbms/admin/utlxpls
@?/rdbms/admin/utlxplp
```

Example : DBMS_XPLAN.DISPLAY usage to display output from PLAN_TABLE

```
select plan_table_output from table(dbms_xplan.display());
```

Explain plan output is a representation of a tree of row sources. Each step (line in the execution plan or node in the tree) represents a row source.

The explain plan utility indents nodes to indicate that they are the children of the parent above it. The order of the nodes under the parent indicates the order of execution of the nodes within that level. If two steps are indented at the same level, the first one is executed first. In the tree format, the left-most leaf on each level of the tree is where the execution starts. The steps of the execution plan are not performed in the order in which they are numbered. there is parent – child relationship between steps.

in PLAN_TABLE and v\$SQL_PLAN

in trc file - STAT # lines

- | | |
|-------------------|--------|
| • ID (id) | - id= |
| • PARENT_ID (pid) | - pid= |
| • POSITION (pos) | - pos= |

One way to read an execution plan is by turning it into a graph of sorts—an evaluation tree.

- Start from the top, with id= 1, which is tree root node.
- Next, find the things that feed this root node.
- That will be accomplished by operation which have parent_id = pid= 1.

To draw plan as a tree, the basic formula is:

- Take the ID with the lowest number and place it at the top.
- Look for rows which have a PID (parent) equal to this value
- Place these in the tree below the Parent according to their POS values from lowest to highest ordered from left to right
- Once all IDs for a parent have been found, move down to the next ID and repeat the process, finding new rows with the same PID

The first thing to determine in an explain plan is which node is executed first. The method in the slide above explains this, but sometimes with complicated plans it is difficult to do this and also difficult to follow the steps through to the end. Often people can become worried by a large plan. Large plans are exactly the same as smaller ones just with more entries The same basic rules apply. You can always collapse plan to hide a branch of the tree which is not involved in performance problem.

Standard Explain plan interpretation

- Start at the top.
- Work down the row sources until you get to one which produces data but does NOT consume any.
- This is the start row source
- Look at the siblings of this row source. These row sources are executed next.
- Once the children are executed then the parent is executed next
- Now that this parent and its children are completed, work back up the tree and look at the siblings of the parent row source and it's parents. Execute as before.
- Move back up the plan until all row sources are exhausted.

Standard Tree Interpretation:

- Start at the top
- Move down the Tree to the left until you reach the left most node.
- This is executed first.
- Look at the siblings of this row source. This is row source is executed next.
- Once the children are executed then the parent is executed next
- Now that this parent and it's children are completed, work back up the tree and look at the siblings of the parent row source and it's parents. Execute as before.
- Move back up the tree until all row sources are exhausted.

If you remember the few basic rules of explain plans then with some experience you should be able to read most plans with little difficulty.

1 - Interpret Plan - Example 1

```
select /*+ RULE */ ename,job,sal,dname from emp,dept
where dept.deptno = emp.deptno
and not exists (select *
                from salgrade
                where emp.sal between losal and hisal);
```

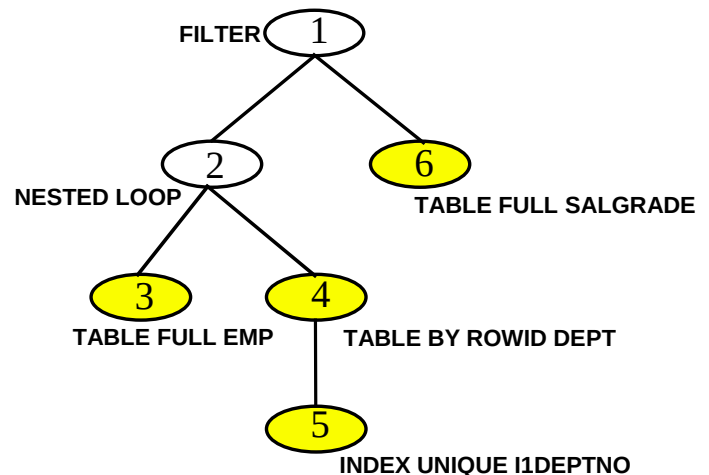
Id	Operation	Name
0	SELECT STATEMENT	
* 1	FILTER	
2	NESTED LOOPS	
3	TABLE ACCESS FULL	EMP
4	TABLE ACCESS BY INDEX ROWID	DEPT
* 5	INDEX UNIQUE SCAN	PK_DEPT
* 6	TABLE ACCESS FULL	SALGRADE

Predicate Information (identified by operation id):

1 - filter(IS NULL)
 5 - access("DEPT"."DEPTNO"="EMP"."DEPTNO")
 6 - filter(("HISAL">=:B1 AND "LOSAL"<=:B2))

Note

- rule based optimizer used (consider using cbo)



ORACLE

Interpret Plan – Example 1

Lets start with an example query to illustrate how to interpret an explain plan. The slide shows a query with its associated explain plan and the same plan in tree format.

The query is:

```
SELECT /*+ RULE */ ename, job, sal, dname
FROM emp,dept
WHERE dept.deptno = emp.deptno
AND NOT EXISTS (SELECT *
                FROM salgrade
                WHERE emp.sal BETWEEN losal AND hisal)
```

This query is trying to find employees who have a salary outside of the range of salaries in the salary grade table.

The query is a select from 2 tables with a subquery based on another table to check the salary grades. The

explain plan generated by parsing this query is shown in the main slide and is also presented in a tree format.

Here is its 10046 trace file

----- Begin 10046 extract trace -----

```
*** 2006-04-20 17:37:09.346
PARSING IN CURSOR #7 len=199 dep=0 uid=55 oct=3 lid=55 tim=813452704809 hv=2760060471
ad='1bd61fb0'
SeLeCt /*+ RULE */ ename,job,sal,dname from emp,dept
where dept.deptno = emp.deptno
      and not exists (select *
                      from salgrade
                      where emp.sal between losal and hisal)
END OF STMT
PARSE #7:c=0,e=108,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=3,tim=813452704801
EXEC #7:c=0,e=83,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=3,tim=813452752367
FETCH #7:c=0,e=937,p=0,cr=59,cu=0,mis=0,r=0,dep=0,og=3,tim=813452755008
STAT #7 id=1 cnt=0 pid=0 pos=1 obj=0 op='FILTER (cr=59 time=925 us)'
STAT #7 id=2 cnt=14 pid=1 pos=1 obj=0 op='NESTED LOOPS (cr=23 time=645 us)'
STAT #7 id=3 cnt=14 pid=2 pos=1 obj=1759 op='TABLE ACCESS FULL EMP (cr=7 time=221 us)'
STAT #7 id=4 cnt=14 pid=2 pos=2 obj=1757 op='TABLE ACCESS BY INDEX ROWID DEPT (cr=16)'
STAT #7 id=5 cnt=14 pid=4 pos=1 obj=1758 op='INDEX UNIQUE SCAN PK_DEPT (cr=2 time=135 us)'
STAT #7 id=6 cnt=12 pid=1 pos=2 obj=1762 op='TABLE ACCESS FULL SALGRADE (cr=36 time=)'
XCTEND rlbk=0, rd_only=1
```

Rows	Row Source Operation
0	FILTER (cr=59 time=925 us)
14	NESTED LOOPS (cr=23 time=645 us)
14	TABLE ACCESS FULL EMP (cr=7 time=221 us)
14	TABLE ACCESS BY INDEX ROWID DEPT (cr=16 time=333 s)
14	INDEX UNIQUE SCAN PK_DEPT (cr=2 time=135 us)
12	TABLE ACCESS FULL SALGRADE (cr=36 time=195 us)

----- End 10046 extract trace -----

Execution Order - Example 1

```
select /*+ RULE */ ename,job,sal,dname from emp,dept
where dept.deptno = emp.deptno
and not exists (select *
                from salgrade
                where emp.sal between losal and hisal);
```

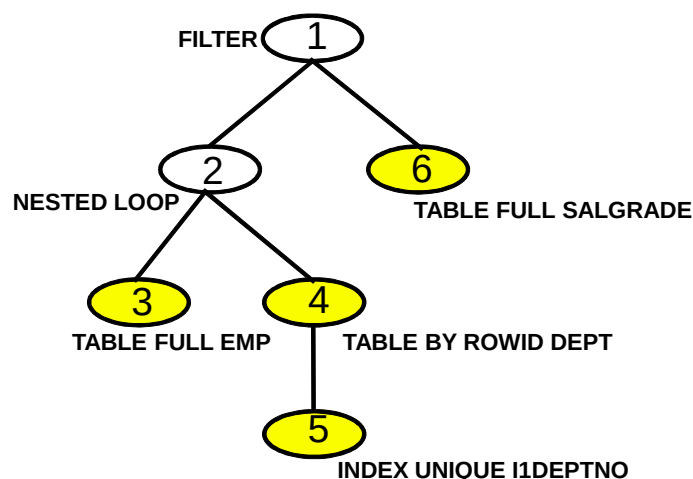
Id	Operation	Name
0	SELECT STATEMENT	
* 1	FILTER	
2	NESTED LOOPS	
3	TABLE ACCESS FULL	EMP
4	TABLE ACCESS BY INDEX ROWID	DEPT
* 5	INDEX UNIQUE SCAN	PK_DEPT
* 6	TABLE ACCESS FULL	SALGRADE

Predicate Information (identified by operation id):

1 - filter(IS NULL)
5 - access("DEPT"."DEPTNO"="EMP"."DEPTNO")
6 - filter(("HISAL">=:B1 AND "LOSAL"<=:B2))

Note

- rule based optimizer used (consider using cbo)



ORACLE

Execution Order

To start off with we are going to reveal what the execution order of this query is and then discuss how that was arrived at from the explain plan.

In this example case the execution order for the explain plan is: **3 – 5 – 4 – 2 – 6 – 1**

Or in words:

3: The plan starts with a Full table scan of EMP (ID=3)

5: The rows are passed back to the controlling nested loops join step (ID=2) which uses them to execute the lookup of rows in the PK_DEPT index in ID=5 next which is the child of ID=4.

4: The rowids from the index are used to lookup the other information from the DEPT table in ID=4

2: ID=2 nested loops join step is executed until completion.

6: Once ID=2 has exhausted its row sources, a the FULL table scan of SALGRADE in ID=6 (at the same level in the tree as ID=2 therefore its sibling) is executed.

1: This is used to filter the rows from ID2 in ID1.

Note that children are executed before parents, so although structures for joins must be setup prior to the child execution, the children are notated as executed first. Probably the easiest way to think of this is think of it as the order in which execution completes, so for the NESTED LOOPS join at ID=2, the two children (ID=3 and ID=4 (together with its child)) must have completed their execution before ID = 2 can be completed).

What follows is a detailed explanation of how to read the plan to get this information for both the explain plan and the tree examples.

Build the tree - Example 1

```
select /*+ RULE */ ename,job,sal,dname from emp,dept
where dept.deptno = emp.deptno
and not exists (select *
                from salgrade
                where emp.sal between losal and hisal);
```

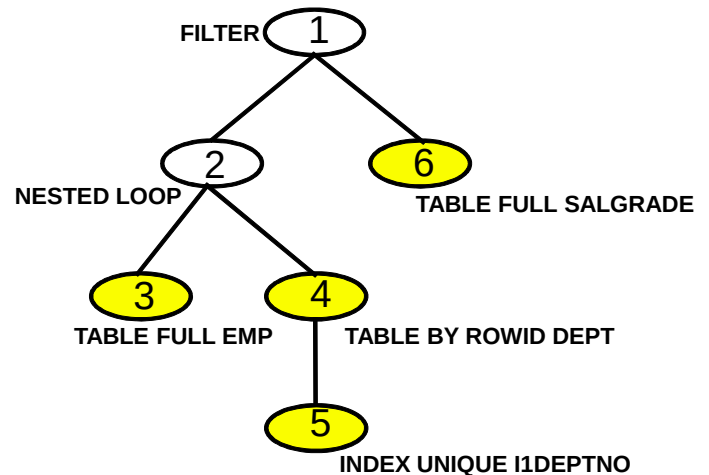
Id	Operation	Name
0	SELECT STATEMENT	
* 1	FILTER	
2	NESTED LOOPS	
3	TABLE ACCESS FULL	EMP
4	TABLE ACCESS BY INDEX ROWID	DEPT
* 5	INDEX UNIQUE SCAN	PK_DEPT
* 6	TABLE ACCESS FULL	SALGRADE

Predicate Information (identified by operation id):

1 - filter(IS NULL)
 5 - access("DEPT"."DEPTNO"="EMP"."DEPTNO")
 6 - filter(("HISAL">=:B1 AND "LOSAL"<=:B2))

Note

- rule based optimizer used (consider using cbo)



ORACLE

Build the Tree

1) To build the tree, start from the top, with step 1, which will be the root node in the tree.

```
STAT #1 id=1 cnt=0 pid=0 pos=1 obj=0 op='FILTER'
```

Id	Operation	Name
0	SELECT STATEMENT	
* 1	FILTER	
...		

At this stage, the tree looks like :

1

2) Next, find the things that feed this root node.

That will be accomplished in steps 2 and 6,

which are at the same level of indentation, because they feed into step 1.

Children of Node 1)

STAT #1 id=1

STAT #1 id=2 cnt=11 pid=1 pos=1 obj=0 op='NESTED LOOPS

STAT #1 id=6 cnt=10 pid=1 pos=2 obj=130942 op='TABLE ACCESS FULL

SALGRADE

Id	Operation	Name	

* 1	FILTER		
2	NESTED LOOPS		-> pos=1
...			
* 6	TABLE ACCESS FULL	SALGRADE	-> pos=2

tree is



3) Next Find operations that feed these nodes 2 and 6

Children of Node 2)

STAT #1 id=2 cnt=11 pid=1

STAT #1 id=3 cnt=12 pid=2 pos=1 obj=130939

STAT #1 id=4 cnt=11 pid=2 pos=2 obj=129943

...

2	NESTED LOOPS		
3	TABLE ACCESS FULL	EMP	-> pos=1
4	TABLE ACCESS BY INDEX ROWID	DEPT	-> pos=2

...



Children of Node 3) access Table EMP

Children of Node 4)

STAT #1 id=4

STAT #1 id=5 cnt=11 pid=4 pos=1 obj=129944

...

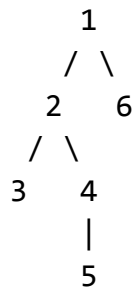
4	TABLE ACCESS BY INDEX ROWID	DEPT	
* 5	INDEX UNIQUE SCAN	PK_DEPT	-> pos=1

...



Children of Node 6) Access Table SALGRADE

Bring The whole tree together, we can draw the following evaluation tree:



When plan is dumped from v\$sql_plan with statistics_level set to ALL. There is column **Starts** which can be used to see HOW Many times operation "ran"

```
SQL> alter session set statistics_level=ALL;
Session altered.
SQL> alter system flush shared_pool;
System altered.
```

```
SQL> SeLeCt /*+ RULE */ ename,job,sal,dname from emp,dept
  2  where dept.deptno = emp.deptno
  3    and not exists (select *
  4                      from salgrade
  5                      where emp.sal between losal and hisal)
  6  /
no rows selected
```

```
SQL> select *
2 from table(dbms_xplan.display_cursor(null,null,-
3 'TYPICAL IOSTATS LAST'));
```

SQL_ID 3b26s5yk86bjr, child number 0

Plan hash value: 1175760222

Id	Operation	Name	Starts	A-Rows	A-Time	Buffers

* 1	FILTER		1	0	00:00:00.01	59
2	NESTED LOOPS		1	14	00:00:00.01	23
3	TABLE ACCESS FULL	EMP	1	14	00:00:00.01	7
4	TABLE ACCESS BY INDEX ROWID	DEPT	14	14	00:00:00.01	16
* 5	INDEX UNIQUE SCAN	PK_DEPT	14	14	00:00:00.01	2
* 6	TABLE ACCESS FULL	SALGRADE	12	12	00:00:00.01	36

Predicate Information (identified by operation id):

```
-----  
1 - filter( IS NULL)  
    filter(NOT EXISTS(SELECT 0 FROM "SALGRADE" "SALGRADE"  
                      WHERE "HISAL">=:B1 AND "LOSAL"<=:B2))  
5 - access("DEPT"."DEPTNO"="EMP"."DEPTNO")  
6 - filter(("HISAL">=:B1 AND "LOSAL"<=:B2))
```

Note

```
-----  
- rule based optimizer used (consider using cbo)
```

A-Rows is corresponding to tkprof "Rows"

Buffers is corresponding to tkprof "cr"

Starts helps to read plan, as we can see how many times operation processed

Due to RULE hint, we have a "Note" section saying RBO is used

- For each row from EMP table, we get employee name (ename), employee salary (sal), employee job (job).
- We access DEPT table using deptno from previous resultset by its unique index (PK_DEPT) to get department name (dname).
- At this step (ID=2), we get all employee name (ename), employee salary (sal), employee job (job), employee department (dname). We need to filter out employee who have a salary outside of the range of salaries in the salary grade table
- To do that, for each row from ID= 2, we access SALGRADE table in Full Table scan to check if employee salary is outside of the range of salaries. (Actually, we do this operation only 12 times (instead of 14 times which is the nb of employees in emp table) because there is an optimisation at runtime and we do the check for each distinct salary – There are 12 distinct salary in emp table).

Interpretation - Example 1

```
select /*+ RULE */ ename,job,sal,dname from emp,dept
where dept.deptno = emp.deptno
and not exists (select *
                from salgrade
                where emp.sal between losal and hisal);
```

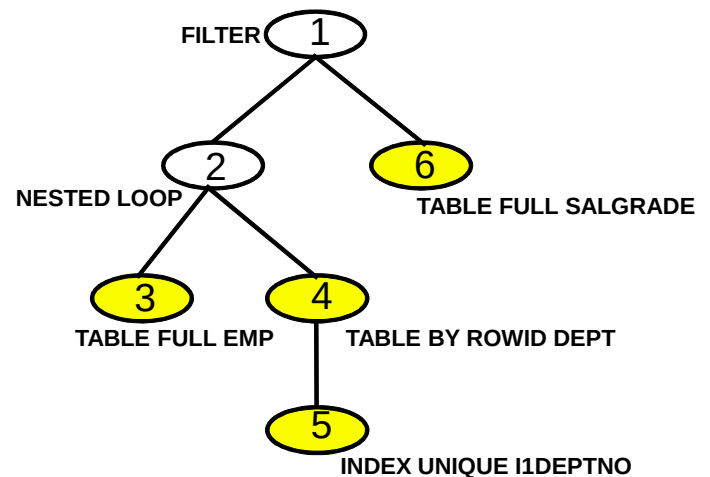
Id	Operation	Name
0	SELECT STATEMENT	
* 1	FILTER	
2	NESTED LOOPS	
3	TABLE ACCESS FULL	EMP
4	TABLE ACCESS BY INDEX ROWID	DEPT
* 5	INDEX UNIQUE SCAN	PK_DEPT
* 6	TABLE ACCESS FULL	SALGRADE

Predicate Information (identified by operation id):

1 - filter(IS NULL)
 5 - access("DEPT"."DEPTNO"="EMP"."DEPTNO")
 6 - filter(("HISAL">=:B1 AND "LOSAL"<=:B2))

Note

- rule based optimizer used (consider using cbo)



ORACLE

Interpretation

Standard Explain plan interpretation

- Start at the top.
- Work down the row sources until you get to one which produces data but does NOT consume any.
- This is the start row source
- Look at the siblings of this row source. These row sources are executed next.
- Once the children are executed then the parent is executed next
- Now that this parent and its children are completed, work back up the tree and look at the siblings of the parent row source and it's parents. Execute as before.
- Move back up the plan until all row sources are exhausted.

Worked Explain Plan Interpretation example:

Start at the top.

In this case this is ID = 0 and is usually the SELECT statement. (Can also be INSERT, UPDATE etc)

Work down the row sources until you get to one which produces data but does NOT consume any.

Children are executed before parents so to find the first row source we need to find the first node that has no children. Working this through:

- From the indentation the first child is ID=1, FILTER – but this has a child (ID=2 NESTED LOOPS) so this cannot be the first node.
- Moving to ID=2, NESTED LOOPS, again this has 2 children at the same level of indentation beneath it, so this cannot be first.
- The next node is ID=3, TABLE ACCESS FULL. This node is the first to have no children. It produces rows and consumes none.

This is the start row source

- ID=3, TABLE ACCESS FULL

Look at the siblings of this row source. These row sources are executed next.

Remember that if a row source has child row sources the child is executed prior to the parent.

- In this case ID=3 has one sibling at the same level of indentation (ID=4 - TABLE ACCESS BY INDEX ROWID).
- These both have the same parent ID=2 NESTED LOOPS (which is a join expecting 2 row sources).
- ID=3 TABLE ACCESS FULL produces rows and returns them to its parent ID=2 NESTED LOOPS this then joins it to ID=4.
- ID=4 also has a child ID=5 INDEX UNIQUE SCAN. Children are executed before parents so ID=5 will be executed next and will return the rows to its parent ID=4 which in turn will be returned to its parent ID=3.

Once the children are executed then the parent is executed next

- ID=2 Is the parent and is executed next

Now that this parent and its children are completed, work back up the tree and look at the siblings of the parent row source and its parents. Execute as before.

- ID=2 has one sibling at the same level of indentation - ID=6 TABLE ACCESS FULL. These both have the same parent ID=1 FILTER.
- ID=2 produces rows which are then used by ID=1 to filter the rows using ID=6.

Move back up the plan until all row sources are exhausted.

Standard Tree Interpretation:

- Start at the top
- Move down the Tree to the left until you reach the left most node.
- This is executed first.
- Look at the siblings of this row source. This is row source is executed next.
- Once the children are executed then the parent is executed next
- Now that this parent and it's children are completed, work back up the tree and look at the siblings of the parent row source and it's parents. Execute as before.
- Move back up the tree until all row sources are exhausted.

Worked Tree Interpretation example:

Start at the top - ID=1 in this case

Move down the Tree to the left until you reach the left most node.

- In this case we would look at node 2 but this has a child to the left of it (Node 3). Node 3 has no child and is the left-most in the tree.

This is executed first.

Look at the siblings of this row source. This is row source is executed next.

- The sibling (node 4) is found at the same level as the current node (node 3) but the sibling has a child. Remember that if a row source has child row sources the child is executed prior to the parent. Node 4 has a child (Node 5). Node 5 is the left most (and only) child so is executed next.

Once the children are executed then the parent is executed next.

- Node 4 is then executed. There are no more siblings so the parent is executed next (Node 2)

Now that this parent and it's children are completed, work back up the tree and look at the siblings of the parent row source and it's parents. Execute as before.

- Node 2 and its children are now executed so now need to look at the same level as Node 2 for siblings and we find Node 6. This is executed next.

Move back up the tree until all row sources are exhausted.

Lastly Node 1 is executed.

2 - Interpret Plan example 2

```
SQL> select m.last_name as dept_manager
2   ,      d.department_name
3   ,      l.street_address, l.postal_code, l.city
4 from    employees m
5         join departments d on (d.manager_id = m.employee_id)
6         join locations l on (l.location_id = d.location_id)
7 where   l.city = 'Seattle';
```

Execution Plan

```
-----
0   SELECT STATEMENT Optimizer=CHOOSE
1 0   NESTED LOOPS
2 1     NESTED LOOPS
3 2       TABLE ACCESS (BY INDEX ROWID) OF 'LOCATIONS'
4 3         INDEX (RANGE SCAN) OF 'LOC_CITY_IX'
5 2       TABLE ACCESS (BY INDEX ROWID) OF 'DEPARTMENTS'
6 5         INDEX (RANGE SCAN) OF 'DEPT_LOCATION_IX'
7 1       TABLE ACCESS (BY INDEX ROWID) OF 'EMPLOYEES'
8 7         INDEX (UNIQUE SCAN) OF 'EMP_EMP_ID_PK'
```

ORACLE

Interpreting EXPLAIN Output: Example 2

This query retrieves employee, department names and addresses for employees whose departments are located in 'Seattle' and who have managers..

Notice in this case the explain plan is slightly differently formatted. We have ID in the first column and PID in the second column, it is reflected by indentation as well. First POS is above subsequent POS

The execution plan shows two nested loops join operations.

```
SQL> select m.last_name as dept_manager
2   ,      d.department_name
3   ,      l.street_address
4   ,      l.postal_code
5   ,      l.city
6 from    employees m
7         join departments d on (d.manager_id = m.employee_id)
8         join locations l on (l.location_id = d.location_id)
```

```
9 where l.city = 'Seattle';
```

DEPT_MANAG	DEPARTMENT_NAME	STREET_ADDRESS	POSTAL_CODE	CITY
Whalen	Administration	2004 Charade Rd	98199	Seattle
Raphaely	Purchasing	2004 Charade Rd	98199	Seattle
King	Executive	2004 Charade Rd	98199	Seattle
Greenberg	Finance	2004 Charade Rd	98199	Seattle
Higgins	Accounting	2004 Charade Rd	98199	Seattle

SQL_ID ddkdcvzx26jwu, child number 0

select ...

Plan hash value: 316119014

Id	Operation	Name	Starts	E-Rows	Cost (%CPU)	A-Rows	Buffers
1	NESTED LOOPS		1	2	5 (0)	5	14
2	NESTED LOOPS		1	2	3 (0)	5	7
3	TABLE ACCESS BY INDEX ROWID	LOCATIONS	1	1	2 (0)	1	3
* 4	INDEX RANGE SCAN	LOC_CITY_IX	1	1	1 (0)	1	2
* 5	TABLE ACCESS BY INDEX ROWID	DEPARTMENTS	1	2	1 (0)	5	4
* 6	INDEX RANGE SCAN	DEPT_LOCATION_IX	1	4	0 (0)	21	2
7	TABLE ACCESS BY INDEX ROWID	EMPLOYEES	5	1	1 (0)	5	7
* 8	INDEX UNIQUE SCAN	EMP_EMP_ID_PK	5	1	0 (0)	5	2

Predicate Information (identified by operation id):

- 4 - access(L.CITY='Seattle')
- 5 - filter(D.MANAGER_ID IS NOT NULL)
- 6 - access(L.LOCATION_ID=D.LOCATION_ID)
- 8 - access(D.MANAGER_ID=M.EMPLOYEE_ID)

Follow our steps from the previous example:

Start at the top. ID=0

Work down the row sources until you get to one which produces data but does NOT consume any. In this case ID 0, 1, 2 and 3 all consume data. ID=4 is the first row source that does not consume any.

This is the start row source. ID4 is executed first. The index range scan produces rowids which are used to

lookup in the LOCATIONS table in ID=3 .

Look at the siblings of this row source. These row sources are executed next. The sibling at the same level as ID=3 is ID=5. Node ID=5 has a child ID=6 which is executed prior to it. This is another index range scan producing rowids which are used to lookup in the DEPARTMENTS table in ID=5

Once the children are executed then the parent is executed next. The NESTED LOOPS join at ID=2 is executed next bringing together the underlying data.

Now that this parent and its children are completed, work back up the tree and look at the siblings of the parent rowsource and it's parents. Execute as before. The sibling of ID=2 at the same level in the plan is ID=7. This has a child ID=8 which is executed first. The index unique scan produces rowids which are used to lookup in the EMPLOYEES table in ID=7.

Move back up the plan until all row sources are exhausted. Finally this is brought together with the NESTED LOOPS at ID=1 which passes the results back to ID=0

Execution order: 4 – 3 – 6 – 5 – 2 – 8 – 7 – 1 - 0

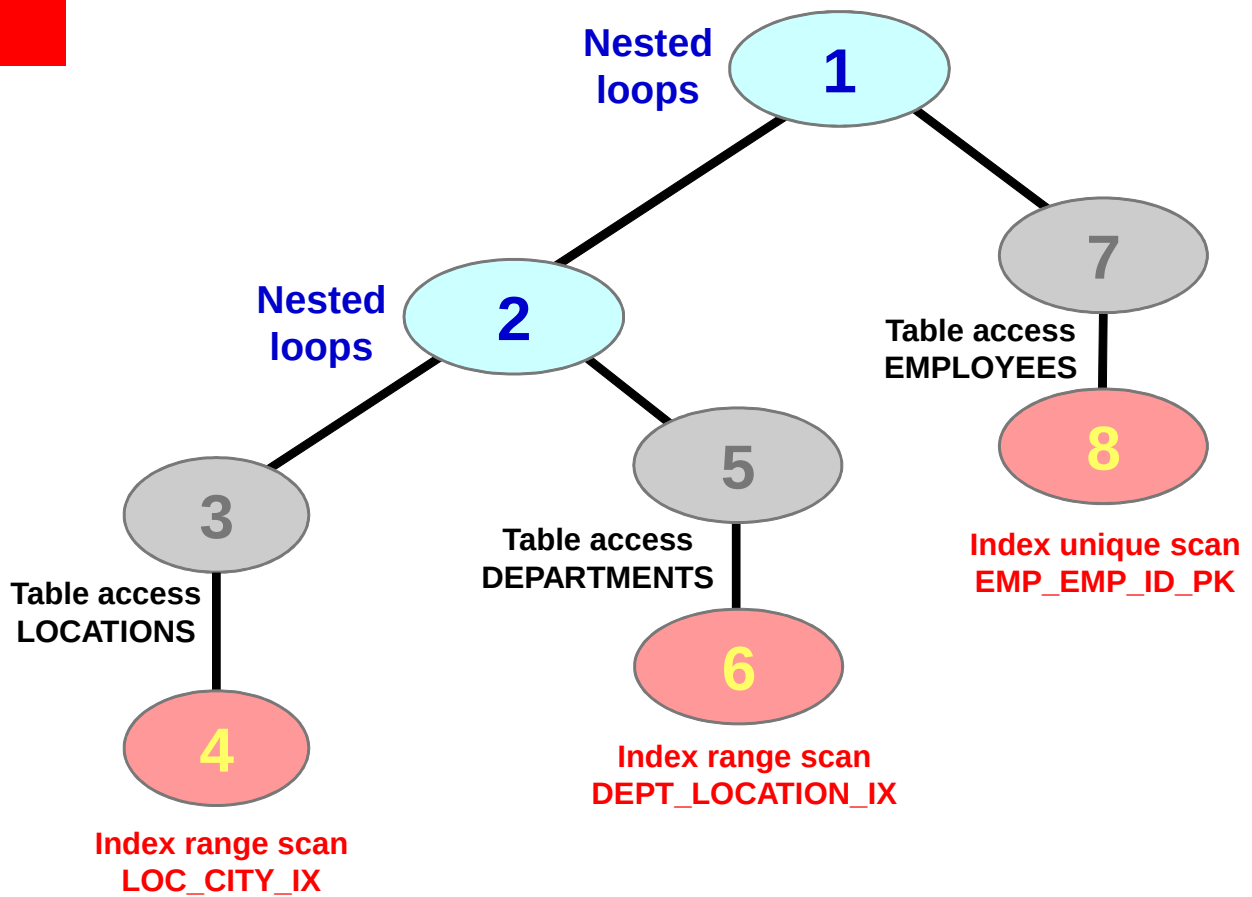
Text Description:

The inner nested loops is executed first LOCATIONS as the driving table, using an index access on the CITY column. This is because we are searching for departments in Seattle only.

The result is joined with the DEPARTMENTS table, using the index on the LOCATION_ID join column; the result of this first join operation is the driving row source for the second nested loops join.

The second join probes the index on the EMPLOYEE_ID column of the EMPLOYEES table. We can do that because we know (from the first join) the employee ID of all managers of departments in Seattle. Note that this is a unique scan because it is based on the primary key.

Finally, the EMPLOYEES table is accessed to retrieve the last name.



ORACLE

Example

This diagram shows the operations described in the previous example. In this representation it is clearer how row sources generate rows that in turn are processed by other row sources before the final result set of rows is returned (from the nested loops join operation). It is also clearer that the rows originate from leafs in the tree and then are sent upwards in the tree.

Execution order: 4 – 3 – 6 – 5 – 2 – 8 – 7 – 1 – 0

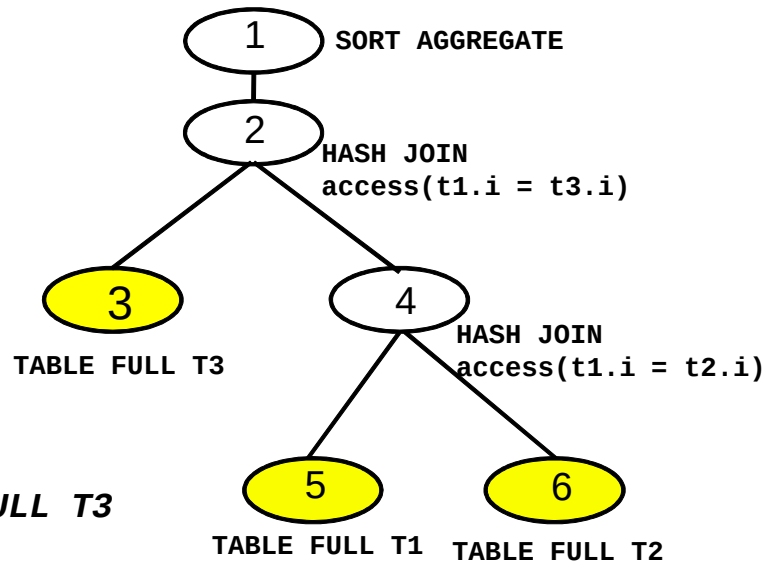
3.3 – Interpret Plan Example 3

Operations order & Joins order

```
select max(t1.i)
  from t1,t2,t3
 where t1.i = t2.i
    and t1.i = t3.i
```

QUERY_PLAN

```
-----
      SELECT STATEMENT
1      SORT AGGREGATE
2 1      HASH JOIN
3 2      TABLE ACCESS  FULL T3
4 2      HASH JOIN
5 4      TABLE ACCESS  FULL T1
6 4      TABLE ACCESS  FULL T2
```



ORACLE

Interpret Plan : Operations order & join order

Look at plan and try to find Order in which plan is executed and deduce what is join order (Order in which we are joining tables)

We have ID in the first column and PID in the second column. POS is reflected by the indentation.

That's important to recognize what is join order in a plan to be able to find your plan into 10053 trace file.

Check below if you are right

- Oracle will hash table T3 (Operation ID = 2) into memory.

- Then hash table T1 (operation ID = 4) into memory.
- Then the scan of table T2 will begin (Operation ID=5).
- Oracle will pick a row from T2 and probe T1 (T1.i = T2.i).
- If the row survives, Oracle will probe T3 (T1.i = T3.i).
- If row survives, Oracle will send row to next operation
- Oracle output max from previous resultset

Execution order: 3 – 5 – 6 – 4 – 2 – 1 - 0

Join order : T1 – T2 – T3

Oracle swapped ((T1-T2) resultset and T3 to do hash join) as T3 was smaller. To do Hash Join, Oracle builds hash table with smaller input.

Non-Side Swapped Plan is:

```

1  HASH JOIN
2 1  HASH JOIN
3 2  TABLE ACCESS FULL EMP
4 2  TABLE ACCESS FULL DEPT
5 1  TABLE ACCESS FULL SALGRADE

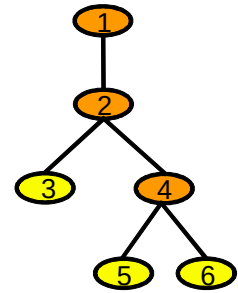
```

See the following for more information:

<Note:171940.1> How to switch the driving table in a hash join

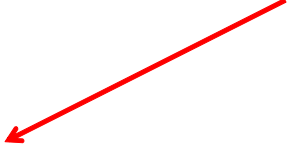


Example 3 (continued)



This is same example as previous slide

- Order in which tables are accessed :Order column
- Join order is t1 - t2 - t3



Operation	Line ID	Object	Object Type	Order	Rows	Size (KB)	Cost	Time (sec)	CPU Cost	I/O Cost
SELECT STATEMENT	0			7	1	0.009	121	2	18133014	117
SORT AGGREGATE	1			6	1	0.009				
HASH JOIN	2			5	10000	87.891	121	2	18133014	117
TABLE ACCESS FULL	3	T3	TABLE	1	10000	29.297	40	1	2703523	39
HASH JOIN	4			4	10000	58.594	80	1	10418269	78
TABLE ACCESS FULL	5	T1	TABLE	2	10000	29.297	40	1	2703523	39
TABLE ACCESS FULL	6	T2	TABLE	3	10000	29.297	40	1	2703523	39

ORACLE

Interpret Plan : Operations order & join order

Enterprise Manager may help. It is displaying column "Order" and column "Id" So that's easy to find execution order.

Execution order: 3 – 5 – 6 – 4 – 2 – 1 - 0

4 - Interpret Plan example 4 Star transformation plan

```
SELECT
    t.calendar_quarter_desc,
    SUM(s.amount_sold) sales_amount
FROM sales s, times t, channels ch
WHERE s.time_id = t.time_id AND
      s.channel_id = ch.channel_id AND
      ch.channel_desc IN ('Internet', 'Catalog') AND
      t.calendar_quarter_desc IN ('1999-Q1', '1999-Q2')
GROUP BY t.calendar_quarter_desc;
```

ORACLE

Interpret Plan

What were the sales and profits for “internet” and “catalog” channels of stores over the last two quarters?«

Typical join methods tend to **start** the query from a dimension table (CHANNEL by exemple) join into the fact table (SALES) using 1 index (and so returns a lot of data). and then, perform further “filtering” by joining to other dimensions (TIMES by example).

Star transformation, which is an optimization, apply ALL dimension table filters (here CHANNELS and TIMES) BEFORE to access FACT table (Here SALES).

A "star transformation" is not named as such in displayed PLAN. Instead, it can be identified by the following set of operations:

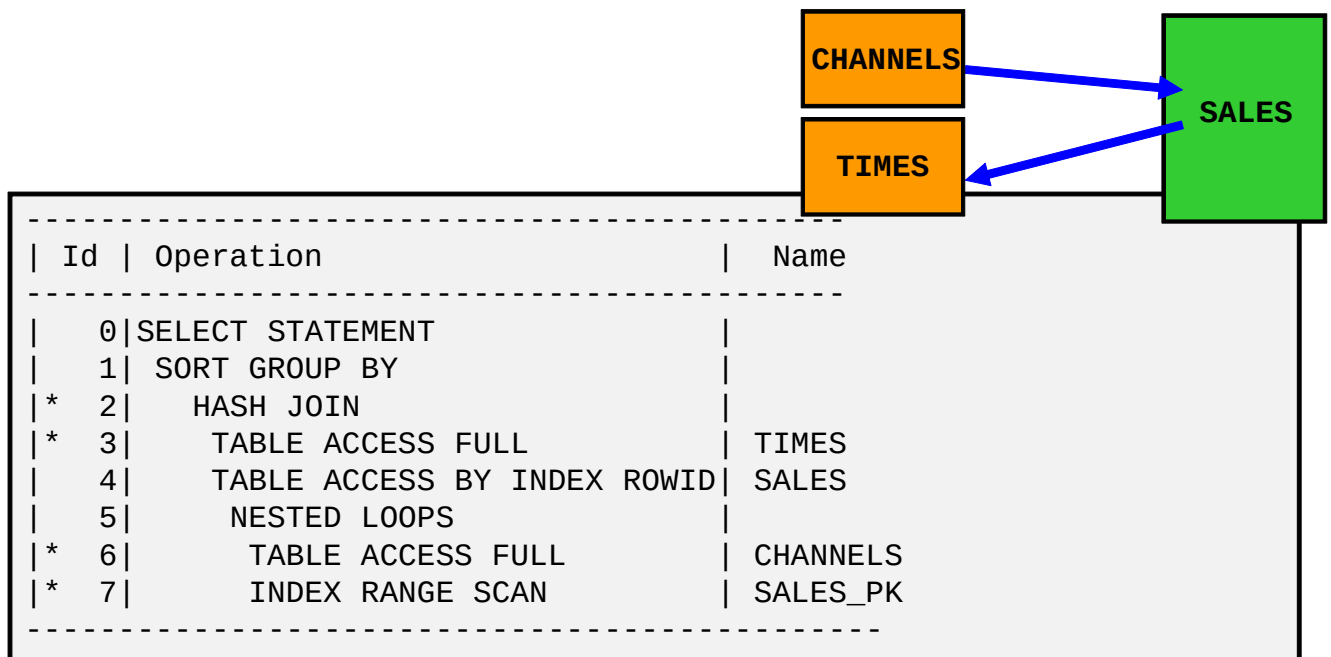
```
TABLE ACCESS BY ROWID OF (<fact-table-name>)
BITMAP AND
BITMAP MERGE
```

```
      (data retrieved from a dimension)
    BITMAP MERGE
      (data retrieved from a dimension)
    BITMAP MERGE
      (data retrieved from a dimension)
```

Next 2 slides are showing example plan from this query

- Without star transformation
- With star transformation

Example 4 – No star transformation



ORACLE

Interpret Plan – No Star Transformation

In that case, there is NO star transformation.

We access 1st Dimension to retrieve CHANNEL_ID for which CHANNEL_DESC IN ('Internet','Catalog').

Then, using this CHANNEL_ID value, CHANNEL table (DIMENSION table) is joined to SALES table (FACT table).

Then, resultset is joined to TIMES table (another dimension table) by a HASH join which was estimated the most efficient one by the optimizer.

Without STAR transformation, we access SALES table after having filtered rows from 'CHANNELS WHERE clause' but not 'TIMES WHERE clause'.

We could have something more efficient if we were able to apply ALL dimension table filters (here CHANNELS and TIMES) BEFORE to access FACT table (Here SALES) (see next slide)

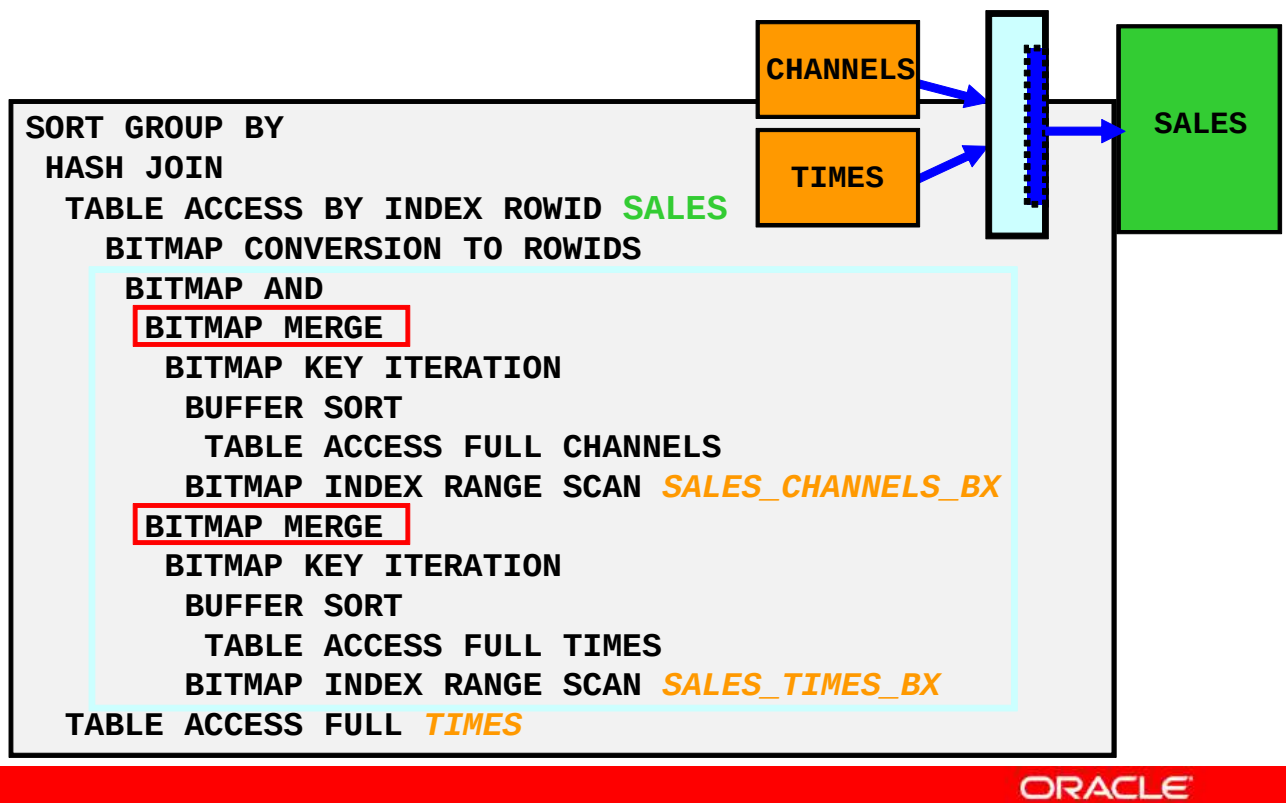
PLAN without star transformation could be like :

Id	Operation	Name

0	SELECT STATEMENT	
1	SORT GROUP BY	
* 2	HASH JOIN	
* 3	TABLE ACCESS FULL	TIMES
4	TABLE ACCESS BY INDEX ROWID	SALES
5	NESTED LOOPS	
* 6	TABLE ACCESS FULL	CHANNELS
* 7	INDEX RANGE SCAN	SALES_PK SALES(CHANNEL_ID, TIME_ID)

- 4 - access(TIME_ID=TIME_ID)
- 5 - filter(calendar_quarter_desc = '1999-Q1' OR = '1999-Q2')
- 6 - filter(CHANNEL_DESC ='Internet' OR CHANNEL_ID = 'Catalog')
- 7 - access(CHANNEL_ID=CHANNEL_ID)

Example 4 – Star Transformation



Interpret Plan – Star Transformation

This is the same query as the previous slide. The plan is different, because the query has been transformed by star transformation optimisation.

Star transformation improves the performance of star queries by applying a transformation that adds new subqueries to the original SQL. These new subqueries will allow the FACT table to be accessed much more efficiently using bitmap indexes (**applying ALL dimension table filters**).

```
SELECT
t.calendar_quarter_desc
SUM(s.amount_sold) sales_amount
FROM
sales s,
times t,
channels ch
WHERE
s.time_id = t.time_id AND
```

```
s.channel_id = ch.channel_id AND
ch.channel_desc IN ('Internet','Catalog') AND
t.calendar_quarter_desc IN ('1999-Q1','1999-Q2')
GROUP BY t.calendar_quarter_desc;
```

Star transformation is done in 2 phases :

First Phase :

Access FACT table by filtering ALL dimension Filters (Using BITMAP)

```
SELECT
s.amount_sold
FROM
sales s
WHERE
time_id IN
(SELECT time_id
FROM times
WHERE calendar_quarter_desc IN('1999-Q1','1999-Q2'))
AND channel_id IN
(SELECT channel_id
FROM channels
WHERE channel_desc IN ('Internet','Catalog'))
```

Oracle uses the bitmap indexes on the foreign key columns of the **SALES** table to identify and retrieve only the necessary rows from the **SALES** table. This is the transformation step of the algorithm, because the original star query has been transformed into this above subquery representation.

Each bitmap corresponds to a separate dimension table, and each bitmap represents the set of rows of the fact table that satisfy that individual dimension's constraints.

In our example, we have 2 dimensions. These 2 bitmaps are combined into a single bitmap using the bitmap AND operation. This final bitmap represents the set of rows in the fact table that satisfy all of the constraints on the dimension tables.

Once the result set is identified, the bitmap is used to access the actual data from the **SALES** table. Only those rows that are required for the end user's query are retrieved from the fact table.

Examine the **CHANNELS** table for channel called « Internet," and find the primary key of each such channel. For each corresponding Channel primary key, it probes the corresponding fact bitmap index **SALES_CHANNELS_BX** and fetches the corresponding bitmap. **MERGE** the resulting sections into a single large **bitmap** for the possible target rows of the **SALES** table.

Repeat the process for '1999-Q1' and the **SALES_TIMES_BX** index to produce a second large **bitmap**.

AND the two **bitmaps** together to produce a single **bitmap** identifying all **sales** rows that match both conditions.

Convert the resulting **bitmap** into a list of rowids, and fetch the rows from the table.

Because we have started with a star transformation to identify the critical **SALES** rows, we should now join this 'intermediate result set' back to the **dimension** tables with a minimum of extra work.

- Generates 'IN' sub-query using the join predicate of the fact for only those dimensions which has single table predicate (semi-join)
- For each distinct value of dim1.key1 get corresponding bitmap from fact and 'OR' them together to get a single bitmap stream
- For each branch of semi-join (each generated sub-query) do 'AND' of all bitmap streams to generate final 'interesting' bitmap

Second Phase

In the second phase of the query (the 'join-back' phase), the dimension tables are joined back to the data set from the first phase. Since, in this query, dimension-table column which appears in the select-list is only **t.calendar_quarter_desc**, this tables needs to be joined.

The existence of the subqueries containing CHANNEL in the first phase of the queries obviated the need to join this table in second phase, and the query optimizer intelligently eliminates this join.

Oracle will use the most efficient method for accessing and joining the dimension tables. Many dimension are very small, and table scans are typically the most efficient access method for these dimension tables. The optimizer automatically determines which access method is most appropriate for a given dimension table, based upon the optimizer's knowledge about the sizes and data distributions of each dimension table.

A hash join is often the most efficient algorithm for joining the dimension tables.

The final answer is returned to the user once all of the dimension tables have been joined "back".

How to read Below Plan

[illegible]

Rows	Row Source Operation
1	VIEW (cr=987610 pr=92643 tm=617760587 us)
1	UNION-ALL (cr=987610 pr=92643 tm=617760575 us)
0	NESTED LOOPS (cr=2363 tm=12999 us)
0	NESTED LOOPS (cr=2363 tm=12993 us)
1	TABLE ACCESS BY INDEX ROWID FILE\$ (cr=2 tm=54 us)
1	INDEX UNIQUE SCAN I_FILE1 (cr=1 tm=31 us)
0	TABLE ACCESS FULL UET\$ (cr=2361 tm=12935 us)
0	VIEW (cr=0 tm=0 us)
0	UNION ALL PUSHED PREDICATE (cr=0 tm=0 us)
0	NESTED LOOPS (cr=0 tm=0 us)
0	NESTED LOOPS (cr=0 tm=0 us)
0	TABLE ACCESS CLUSTER USER\$ (cr=0 tm=0 us)
0	MERGE JOIN CARTESIAN (cr=0 tm=0 us)
0	NESTED LOOPS (cr=0 tm=0 us)
0	BUFFER SORT (cr=0 tm=0 us)
0	NESTED LOOPS (cr=0 tm=0 us)
0	MERGE JOIN CARTESIAN (cr=0 tm=0 us)
0	TABLE ACCESS CLUSTER USER\$ (cr=0 tm=0 us)
1	NESTED LOOPS (cr=985247 pr=92643 tm=617747538 us)
13	NESTED LOOPS (cr=985231 pr=92643 tm=1870047923 us)
94668	VIEW (cr=593549 pr=310 tm=10641190 us)
94668	UNION-ALL (cr=593549 pr=310 tm=10546516 us)
94421	NESTED LOOPS (cr=589417 pr=310 tm=10149552 us)
247	NESTED LOOPS (cr=1741 tm=18151 us)
0	HASH JOIN (cr=2391 tm=104482 us)
13	FIXED TAB FIXED INDEX X\$KTFFBUE(ind:1)(cr=391682 pr=92333 tm=1879347583)
1	TABLE ACCESS BY INDEX ROWID FILE\$ (cr=16 tm=303 us)
13	INDEX UNIQUE SCAN I_FILE1 (cr=3 tm=129 us)

Collapse using Indentation

ORACLE®

This plan comes from a Dictionary Query and it is so long that it is very difficult to apply previous method to interpret it and find where is first operation ...

Keep in mind, you can always COLLAPSE a plan to be back to a readable one.

Then, You just need to EXPAND branch you want to investigate (where time is being spent). Time reported in tkprof is cumulative :

parent operation time = Operation time itself + sum(children operation time).

Example to collapse :

Start at Parent operation (First **ORANGE** (-->) line from the top downwards) below which you'll collapse
(This operation has a parent operation GREEN line just before)

Rows	Row Source	Operation
------	------------	-----------

```

-----
1 VIEW (cr=987610 pr=92643 tm=617760587 us)
1 UNION-ALL (cr=987610 pr=92643 tm=617760575 us)
0 NESTED LOOPS (cr=2363 tm=12999 us)
0 NESTED LOOPS (cr=2363 tm=12993 us)
1 | TABLE ACCESS BY INDEX ROWID FILE$ (cr=2 tm=54 us)
1 | | INDEX UNIQUE SCAN I_FILE1 (cr=1 tm=31 us)
0 | TABLE ACCESS FULL UET$ (cr=2361 tm=12935 us)
0 VIEW (cr=0 tm=0 us)
0 | UNION ALL PUSHED PREDICATE (cr=0 tm=0 us)
0 | | NESTED LOOPS (cr=0 tm=0 us)
--> 0 | | | NESTED LOOPS (cr=0 tm=0 us)
0 | | | HASH JOIN (cr=0 tm=0 us)
0 | | | MERGE JOIN CARTESIAN (cr=0 tm=0 us)
0 | | | NESTED LOOPS (cr=0 tm=0 us)
0 | | | INDEX UNIQUE SCAN I_FILE2 (cr=0 tm=0 us)
0 | | | TABLE ACCESS CLUSTER TS$ (cr=0 tm=0 us)
0 | | | INDEX UNIQUE SCAN I_TS# (cr=0 tm=0 us)
0 | | | BUFFER SORT (cr=0 tm=0 us)
0 | | | TABLE ACCESS CLUSTER SEG$ (cr=0 tm=0 us)
0 | | | INDEX UNIQUE SCAN I_FILE#_BLOCK# (cr=0 tm=0 us)
0 | | | VIEW (cr=0 tm=0 us)
0 | | | UNION-ALL (cr=0 tm=0 us)
0 | | | TABLE ACCESS FULL TAB$ (cr=0 tm=0 us)
0 | | | TABLE ACCESS FULL TABPART$ (cr=0 tm=0 us)
0 | | | TABLE ACCESS FULL CLU$ (cr=0 tm=0 us)
0 | | | TABLE ACCESS FULL IND$ (cr=0 tm=0 us)
0 | | | TABLE ACCESS FULL INDPART$ (cr=0 tm=0 us)
0 | | | TABLE ACCESS FULL LOB$ (cr=0 tm=0 us)
0 | | | TABLE ACCESS FULL TABSUBPART$ (cr=0 tm=0 us)
0 | | | TABLE ACCESS FULL INDSUBPART$ (cr=0 tm=0 us)
0 | | | TABLE ACCESS FULL LOBFRAG$ (cr=0 tm=0 us)
0 | | | TABLE ACCESS BY INDEX ROWID OBJ$ (cr=0 tm=0 us)
0 | | | INDEX UNIQUE SCAN I_OBJ1 (cr=0 tm=0 us)
0 | | | TABLE ACCESS CLUSTER USER$ (cr=0 tm=0 us)
0 | | | INDEX UNIQUE SCAN I_USER# (cr=0 tm=0 us)
0 | | | MERGE JOIN CARTESIAN (cr=0 tm=0 us)

```

- Move right one level/indentation (in above example indentation is one character)
- Move down (follow **ORANGE** line) until you find a sibling of **ORANGE** operation or one of its parent.
- Collapse (Delete BLACK lines between these 2 **ORANGE** lines)
- Move down (follow **ORANGE** line) until you find a sibling of **ORANGE** operation or one of its parent.
- Collapse (Delete BLACK lines between 2nd **ORANGE** line and 2nd **GREEN** line)

Rows	Row	Source	Operation
	1	VIEW	(cr=987610 pr=92643 tm=617760587 us)
	1	UNION-ALL	(cr=987610 pr=92643 tm=617760575 us)
	0	NESTED LOOPS	(cr=2363 tm=12999 us)
	0	NESTED LOOPS	(cr=2363 tm=12993 us)
	1		TABLE ACCESS BY INDEX ROWID FILE\$ (cr=2 tm=54 us)
	1		INDEX UNIQUE SCAN I_FILE1 (cr=1 tm=31 us)
	0		TABLE ACCESS FULL UET\$ (cr=2361 tm=12935 us)
	0	VIEW	(cr=0 tm=0 us)
	0		UNION ALL PUSHED PREDICATE (cr=0 tm=0 us)
	0		NESTED LOOPS (cr=0 tm=0 us)
-->	0		NESTED LOOPS (cr=0 tm=0 us)
	0		TABLE ACCESS CLUSTER USER\$ (cr=0 tm=0 us)
	0		MERGE JOIN CARTESIAN (cr=0 tm=0 us)



Let's Read a more complex Plan.

Here is a common query against DBA_EXTENT :

```
SELECT owner , segment_name , segment_type
FROM dba_extents
WHERE file_id = &AFN
      AND &BLOCKNO BETWEEN block_id
                        AND block_id + blocks -1;
```

ORACLE

Let's Read a more complex Plan.

This below Dictionary Query became too slow (30 min) after upgrade, it used to take 6 min.

That's usual query to find segment owner, name and type when we have File_id and Block Number. This SQL comes from Note 181306.1.

There are between 94000 and 110000 segments in DB

```
SELECT owner , segment_name , segment_type
FROM dba_extents
WHERE file_id = 15
      AND 1904005 BETWEEN block_id AND block_id + blocks -1
```

call	count	cpu elapsed		disk	query	current	rows
-----	-----	-----	-----	-----	-----	-----	-----
Parse	1	17.31	17.00	1	221088	0	0
Execute	1	0.00	0.00	0	0	0	0

Fetch	2	275.50	1864.48	92643	703546	633	1

total	4	292.81	1881.48	92644	924634	633	1

Misses in library cache during parse: 1

Optimizer goal: ALL_ROWS

Parsing user id: SYS

Elapsed times include waiting on following events:

Event waited on	Times Waited	Max. Wait	Total Waited
-----	-----	-----	-----
db file sequential read	92359	0.29	1586.70

Is it a slow IO problem ?

Plan *seems* complex ...

Rows	Row Source Operation
-----	-----
1	VIEW (cr=987610 pr=92643 tm=617760587 us)
1	UNION-ALL (cr=987610 pr=92643 tm=617760575 us)
0	NESTED LOOPS (cr=2363 tm=12999 us)
0	NESTED LOOPS (cr=2363 tm=12993 us)
1	TABLE ACCESS BY INDEX ROWID FILE\$ (cr=2 tm=54 us)
1	INDEX UNIQUE SCAN I_FILE1 (cr=1 tm=31 us)
0	TABLE ACCESS FULL UET\$ (cr=2361 tm=12935 us)
0	VIEW (cr=0 tm=0 us)
0	UNION ALL PUSHED PREDICATE (cr=0 tm=0 us)
0	NESTED LOOPS (cr=0 tm=0 us)
0	NESTED LOOPS (cr=0 tm=0 us)
0	HASH JOIN (cr=0 tm=0 us)
0	MERGE JOIN CARTESIAN (cr=0 tm=0 us)
0	NESTED LOOPS (cr=0 tm=0 us)
0	INDEX UNIQUE SCAN I_FILE2 (cr=0 tm=0 us)
0	TABLE ACCESS CLUSTER TS\$ (cr=0 tm=0 us)
0	INDEX UNIQUE SCAN I_TS# (cr=0 tm=0 us)
0	BUFFER SORT (cr=0 tm=0 us)
0	TABLE ACCESS CLUSTER SEG\$ (cr=0 tm=0 us)
0	INDEX UNIQUE SCAN I_FILE#_BLOCK# (cr=0 tm=0 us)
0	VIEW (cr=0 tm=0 us)

```

0          UNION-ALL (cr=0 tm=0 us)
0          TABLE ACCESS FULL TAB$ (cr=0 tm=0 us)
0          TABLE ACCESS FULL TABPART$ (cr=0 tm=0 us)
0          TABLE ACCESS FULL CLU$ (cr=0 tm=0 us)
0          TABLE ACCESS FULL IND$ (cr=0 tm=0 us)
0          TABLE ACCESS FULL INDPART$ (cr=0 tm=0 us)
0          TABLE ACCESS FULL LOB$ (cr=0 tm=0 us)
0          TABLE ACCESS FULL TABSUBPART$ (cr=0 tm=0 us)
0          TABLE ACCESS FULL INDSUBPART$ (cr=0 tm=0 us)
0          TABLE ACCESS FULL LOBFRAG$ (cr=0 tm=0 us)
0          TABLE ACCESS BY INDEX ROWID OBJ$ (cr=0 tm=0 us)
0          INDEX UNIQUE SCAN I_OBJ1 (cr=0 tm=0 us)
0          TABLE ACCESS CLUSTER USER$ (cr=0 tm=0 us)
0          INDEX UNIQUE SCAN I_USER# (cr=0 tm=0 us)
0          MERGE JOIN CARTESIAN (cr=0 tm=0 us)
0          NESTED LOOPS (cr=0 tm=0 us)
0          MERGE JOIN CARTESIAN (cr=0 tm=0 us)
0          NESTED LOOPS (cr=0 tm=0 us)
0          INDEX UNIQUE SCAN I_FILE2 (cr=0 tm=0 us)
0          TABLE ACCESS CLUSTER TS$ (cr=0 tm=0 us)
0          INDEX UNIQUE SCAN I_TS# (cr=0 tm=0 us)
0          BUFFER SORT (cr=0 tm=0 us)
0          TABLE ACCESS CLUSTER SEG$ (cr=0 tm=0 us)
0          INDEX UNIQUE SCAN I_FILE#_BLOCK# (cr=0 tm=0 us)
0          TABLE ACCESS CLUSTER USER$ (cr=0 tm=0 us)
0          INDEX UNIQUE SCAN I_USER# (cr=0 tm=0 us)
0          BUFFER SORT (cr=0 tm=0 us)
0          TABLE ACCESS FULL UNDO$ (cr=0 tm=0 us)
0          NESTED LOOPS (cr=0 tm=0 us)
0          MERGE JOIN CARTESIAN (cr=0 tm=0 us)
0          NESTED LOOPS (cr=0 tm=0 us)
0          TABLE ACCESS BY INDEX ROWID FILE$ (cr=0 tm=0 us)
0          INDEX UNIQUE SCAN I_FILE2 (cr=0 tm=0 us)
0          TABLE ACCESS CLUSTER TS$ (cr=0 tm=0 us)
0          INDEX UNIQUE SCAN I_TS# (cr=0 tm=0 us)
0          BUFFER SORT (cr=0 tm=0 us)
0          TABLE ACCESS CLUSTER SEG$ (cr=0 tm=0 us)
0          INDEX UNIQUE SCAN I_FILE#_BLOCK# (cr=0 tm=0 us)
0          TABLE ACCESS CLUSTER USER$ (cr=0 tm=0 us)
0          INDEX UNIQUE SCAN I_USER# (cr=0 tm=0 us)
1          NESTED LOOPS (cr=985247 pr=92643 tm=617747538 us)
13         NESTED LOOPS (cr=985231 pr=92643 tm=1870047923 us)
94668       VIEW (cr=593549 pr=310 tm=10641190 us)
94668       UNION-ALL (cr=593549 pr=310 tm=10546516 us)
94421       NESTED LOOPS (cr=589417 pr=310 tm=10149552 us)

```

```

94421      NESTED LOOPS (cr=400570 pr=310 tm=7977855 us)
94421      NESTED LOOPS (cr=211725 pr=289 tm=5900578 us)
94421      NESTED LOOPS (cr=211722 pr=289 tm=5334041 us)
94421      HASH JOIN (cr=22877 pr=289 tm=3634432 us)
94668      TABLE ACCESS FULL SEG$ (cr=2361 tm=284204 us)
136059     VIEW (cr=20486 pr=289 tm=2197629 us)
136059     UNION-ALL (cr=20486 pr=289 tm=1989261 us)
19074     TABLE ACCESS FULL TAB$ (cr=4482 pr=289 tm=863191)
5849     TABLE ACCESS FULL TABPART$ (cr=97 tm=22446 us)
10      TABLE ACCESS FULL CLU$ (cr=4481 tm=20827 us)
24825     TABLE ACCESS FULL IND$ (cr=4481 tm=81636 us)
44331     TABLE ACCESS FULL INDPART$ (cr=482 tm=166713 us)
216     TABLE ACCESS FULL LOB$ (cr=4481 tm=23875 us)
10521     TABLE ACCESS FULL TABSUBPART$ (cr=566 tm=38911 us)
31199     TABLE ACCESS FULL INDSUBPART$ (cr=1413 tm=113490)
34      TABLE ACCESS FULL LOBFRAG$ (cr=3 tm=265 us)
94421     TABLE ACCESS CLUSTER TS$ (cr=188845 tm=1574482 us)
94421     INDEX UNIQUE SCAN I_TS# (cr=3 tm=449097 us)
94421     INDEX UNIQUE SCAN I_FILE2 (cr=3 tm=381868 us)
94421     TABLE ACCESS BY INDEX ROWID OBJ$(cr=188845 pr=21 tm=1945896 us)
94421     INDEX UNIQUE SCAN I_OBJ1 (cr=94424 pr=20 tm=1130632 )
94421     TABLE ACCESS CLUSTER USER$ (cr=188847 tm=1923754 us)
94421     INDEX UNIQUE SCAN I_USER# (cr=3 tm=423277 us)
247     NESTED LOOPS (cr=1741 tm=18151 us)
247     NESTED LOOPS (cr=1245 tm=13694 us)
247     NESTED LOOPS (cr=749 tm=8984 us)
247     NESTED LOOPS (cr=747 tm=7739 us)
247     TABLE ACCESS FULL UNDO$ (cr=4 tm=1009 us)
247     TABLE ACCESS CLUSTER SEG$ (cr=743 tm=6314 us)
247     INDEX UNIQUE SCAN I_FILE#_BLOCK# (cr=249 tm=1765 us)
247     INDEX UNIQUE SCAN I_FILE2 (cr=2 tm=947 us)
247     TABLE ACCESS CLUSTER USER$ (cr=496 tm=4334 us)
247     INDEX UNIQUE SCAN I_USER# (cr=2 tm=967 us)
247     TABLE ACCESS CLUSTER TS$ (cr=496 tm=3936 us)
247     INDEX UNIQUE SCAN I_TS# (cr=2 tm=876 us)
0      HASH JOIN (cr=2391 tm=104482 us)
0      NESTED LOOPS (cr=2361 tm=102448 us)
0      NESTED LOOPS (cr=2361 tm=102443 us)
0      TABLE ACCESS FULL SEG$ (cr=2361 tm=102439 us)
0      TABLE ACCESS CLUSTER USER$ (cr=0 tm=0 us)
0      INDEX UNIQUE SCAN I_USER# (cr=0 tm=0 us)
0      TABLE ACCESS CLUSTER TS$ (cr=0 tm=0 us)
0      INDEX UNIQUE SCAN I_TS# (cr=0 tm=0 us)
0      TABLE ACCESS FULL FILE$ (cr=0 tm=0 us)
13     FIXED TAB FIXED INDEX X$KTFBUE(ind:1)(cr=391682 pr=92333 tm=1879347583 us)

```

```

1      TABLE ACCESS BY INDEX ROWID FILE$ (cr=16 tm=303 us)
13     INDEX UNIQUE SCAN I_FILE1 (cr=3 tm=129 us)

```

Look at Back DBA_EXTENTS definition to understand what we are looking for

DBA_EXTENTS view is UNION ALL from 2 SQLs

- BLUE -> DICTIONARY Managed Tablespaces
- GREEN -> LOCALLY Managed Tablespaces

```

create or replace view DBA_EXTENTS
  (OWNER, SEGMENT_NAME, PARTITION_NAME, SEGMENT_TYPE, TABLESPACE_NAME,
   EXTENT_ID, FILE_ID, BLOCK_ID,
   BYTES, BLOCKS, RELATIVE_FNO)
as
select ds.owner, ds.segment_name, ds.partition_name, ds.segment_type,
       ds.tablespace_name,
       e.ext#, f.file#, e.block#, e.length * ds.blocksize, e.length,
       e.file#
from sys.uet$ e, sys.sys_dba_segs ds, sys.file$ f
where e.segfile# = ds.relative_fno
   and e.segblock# = ds.header_block
   and e.ts# = ds.tablespace_id
   and e.ts# = f.ts#
   and e.file# = f.relfile#
   and bitand(NVL(ds.segment_flags,0), 1) = 0
   and bitand(NVL(ds.segment_flags,0), 65536) = 0
union all
select /*+ ordered use_nl(e) use_nl(f) */
       ds.owner, ds.segment_name, ds.partition_name, ds.segment_type,
       ds.tablespace_name,
       e.ktfbueextno, f.file#, e.ktfbuebno,
       e.ktfbueblks * ds.blocksize, e.ktfbueblks, e.ktfbuefno
from sys.sys_dba_segs ds, sys.x$ktfbue e, sys.file$ f
where e.ktfbuesegfno = ds.relative_fno
   and e.ktfbuesegbno = ds.header_block
   and e.ktfbuesegtsn = ds.tablespace_id
   and e.ktfbuesegtsn = f.ts#
   and e.ktfbuefno = f.relfile#
   and bitand(NVL(ds.segment_flags, 0), 1) = 1
   and bitand(NVL(ds.segment_flags,0), 65536) = 0
/

```

Collapse PLAN to simplify (THIS IS VERY IMPORTANT to be back to simple things)

UNION ALL with 2 branches

- Branch 1 – DICTIONARY Managed Tablespaces
- Branch 2 – LOCALLY Managed Tablespaces

Rows	Row Source Operation
1	VIEW (cr=987610 pr=92643 tm=617760587 us)
1	UNION-ALL (cr=987610 pr=92643 tm=617760575 us)
0	NESTED LOOPS (cr=2363 tm=12999 us)
0	NESTED LOOPS (cr=2363 tm=12993 us)
0	VIEW SYS_DBA_SEGS (cr=0 tm=0 us)
1	NESTED LOOPS (cr=985247 pr=92643 tm=617747538 us)
13	NESTED LOOPS (cr=985231 pr=92643 tm=1870047923 us)
94668	VIEW SYS_DBA_SEGS (cr=593549 pr=310 tm=10641190 us)
13	FIXED TAB FIXED INDEX X\$KTFBUE(ind:1)(cr=391682 pr=92333 tm=1879347583 us)
1	TABLE ACCESS BY INDEX ROWID FILE\$ (cr=16 tm=303 us)
13	INDEX UNIQUE SCAN I_FILE1 (cr=3 tm=129 us)

- Branch 1 – DICTIONARY Managed Tablespaces is NOT a problem
There is NO segment on Dictionary Managed Tablespace
- Branch 2 – LOCALLY Managed Tablespaces is where time is being spent
ALL segments (94668 at query time are in Locally Managed Tablespace)

Where do I start to look at plan ... ?

Rows	Row Source Operation
1	VIEW (cr=987610 pr=92643 tm=617760587 us)
1	UNION-ALL (cr=987610 pr=92643 tm=617760575 us)
0	NESTED LOOPS (cr=2363 tm=12999 us)
0	NESTED LOOPS (cr=2363 tm=12993 us)
1	TABLE ACCESS BY INDEX ROWID FILE\$ (cr=2 tm=54 us)
1	INDEX UNIQUE SCAN I_FILE1 (cr=1 tm=31 us)
0	TABLE ACCESS FULL UET\$ (cr=2361 tm=12935 us)
0	VIEW (cr=0 tm=0 us)
0	UNION ALL PUSHED PREDICATE (cr=0 tm=0 us)
0	NESTED LOOPS (cr=0 tm=0 us)
0	NESTED LOOPS (cr=0 tm=0 us)
0	TABLE ACCESS CLUSTER USER\$ (cr=0 tm=0 us)
0	MERGE JOIN CARTESIAN (cr=0 tm=0 us)
0	NESTED LOOPS (cr=0 tm=0 us)
0	BUFFER SORT (cr=0 tm=0 us)
0	NESTED LOOPS (cr=0 tm=0 us)
0	MERGE JOIN CARTESIAN (cr=0 tm=0 us)
0	TABLE ACCESS CLUSTER USER\$ (cr=0 tm=0 us)
1	NESTED LOOPS (cr=985247 pr=92643 tm=187747538 us)
13	NESTED LOOPS (cr=985231 pr=92643 tm=1870047923 us)
94668	VIEW (cr=593549 pr=310 tm=10641190 us)
94668	UNION-ALL (cr=593549 pr=310 tm=10546516 us)
94421	NESTED LOOPS (cr=589417 pr=310 tm=10149552 us)
247	NESTED LOOPS (cr=1741 tm=18151 us)
0	HASH JOIN (cr=2391 tm=104482 us)
13	FIXED TAB FIXED INDEX X\$KTFBUE(ind:1)(cr=391682 pr=92333 tm=1879347583)
1	TABLE ACCESS BY INDEX ROWID FILE\$ (cr=16 tm=303 us)
13	INDEX UNIQUE SCAN I_FILE1 (cr=3 tm=129 us)

Where time is being spent ?

Rows	Row Source Operation
1	VIEW (cr=987610 pr=92643 tm=617760587 us)
1	UNION-ALL (cr=987610 pr=92643 tm=617760575 us)
0	NESTED LOOPS (cr=2363 tm=12999 us)
0	NESTED LOOPS (cr=2363 tm=12993 us)
0	VIEW SYS_DBA_SEGS (cr=0 tm=0 us)
1	NESTED LOOPS (cr=985247 pr=92643 tm=187747538 us)
13	NESTED LOOPS (cr=985231 pr=92643 tm=1870047923 us)
94668	VIEW SYS_DBA_SEGS (cr=593549 pr=310 tm=10641190 us)
13	FIXED TAB FIXED INDEX X\$KTFBUE(ind:1)(cr=391682 pr=92333 tm=1879347583)
1	TABLE ACCESS BY INDEX ROWID FILE\$ (cr=16 tm=303 us)
13	INDEX UNIQUE SCAN I_FILE1 (cr=3 tm=129 us)

Simplify

ORACLE

Where time is being spent exactly ?

13 FIXED TAB FIXED INDEX X\$KTFBUE(ind:1)(cr=391682 pr=92333 time = 1879347583 us)

- X\$KTFBUE was accessed by its index 94668 times (NL inner table)
- There are 94668 segments in DB (at query time)
- With execution plan with rowsource execution statistics you'll get Start = 94668
- Almost all time is being spent in X\$KTFBUE

What is X\$KTFBUE ?

X\$KTFBUE [k]ernel [t]ablespace [f]ile [b]itmapped [u]sed [e]xtents

This is not really a table, this is a callback function. There are 2 access path available :

- by its index on KTFBUESEGTSN KTFBUESEGFNO KTFBUESEGBNO
index can only be used with equality clause on 3 columns : KTFBUESEGTSN , KTFBUESEGFNO and KTFBUESEGBNO
- KTFBUESEGTSN** NUMBER tablespace number of segment
KTFBUESEGFNO NUMBER segment relative file number

KTFBUESEGBNO NUMBER segment block number

- by Full Table Scan

As soon as there is NOT 3 equality predicates

From tests done, it takes around 1 hour to do Full Table Scan on X\$KTFBUE

Why do we spent so much time to access x\$KTFBUE ?

- All segments are in LMT tablespaces
- join order is **SYS_DBA_SEGS** - X\$KTFBUE – FILE\$

- which is good because we can access X\$KTFBUE by its index (by doing a FTS on x\$KTFBUE, it takes over 1 hour)
- We do not push any predicate to restrict SEGMENTS returned by **SYS_DBA_SEGS**, *so we access X\$KTFBUE by its index for ALL segments*

BUT We only want to get information on segments in **file_id = 15**

- If we know file (File_id = 15) ,Then Tablespace in which file belongs is known
- If we we change join predicate

from

```
and e.ktfbuesegtsn = ds.tablespace_id
and e.ktfbuesegtsn = f.ts#
```

to

```
and e.ktfbuesegtsn = ds.tablespace_id
and ds.tablespace_id = f.ts# -- change here
```

See below, difference is additionnal filter at ***Id=5 (DS.TABLESPACE_ID=F.TS#)***

and so only 18664 segments (instead of 110003 – Nb of segments at query time) are joined to X\$ktfbue

Collapsed PLAN BEFORE JOIN PREDICATE CHANGE

Keep LMT part and Force join order : FILE\$ - SYS DBA SEGS - X\$KTFBUE

Id	Operation	Name	Rows	Cost
----	-----------	------	------	------

	1		NESTED LOOPS				1		1098	
	2		NESTED LOOPS				5		1097	
	3		TABLE ACCESS BY INDEX ROWID		FILE\$		1		1	
	*	4		INDEX UNIQUE SCAN		I_FILE1		1		
	5		VIEW		SYS_DBA_SEGS		5		1096	
	*	50		FIXED TABLE FIXED INDEX		X\$KTFBUE (ind:1)		1		1

Predicate Information (identified by operation id):

```

4 - access(F.FILE#=15)
50 - filter(E.KTFBUEBNO<=1904005 AND
          E.KTFBUEBNO+E.KTFBUEBLKS-1>=1904005 AND
          E.KTFBUESEGFNO=DS.RELATIVE_FNO AND
          E.KTFBUESEGBNO=DS.HEADER_BLOCK AND
          E.KTFBUESEGTSN=DS.TABLESPACE_ID AND E.KTFBUESEGTSN=F.TS# AND
          E.KTFBUEFNO=F.RELFILE#)

```

Rows	Row Source Operation
1	NESTED LOOPS (cr=1150466 pr=110264 pw=0 time=1261573638 us)
110003	NESTED LOOPS (cr=691322 pr=328 pw=0 time=15545862 us)
1	TABLE ACCESS BY INDEX ROWID OBJ#(17) (cr=2 time=47 us)
1	INDEX UNIQUE SCAN OBJ#(41) (cr=1 pr=0 pw=0 time=23 us)
110003	VIEW (cr=691320 pr=328 pw=0 time=15325805 us)
1	FIXED TABLE FIXED INDEX X\$KTFBUE (ind:1) (cr=459144 pr=109936 time=1741363594 us)

Collapsed PLAN AFTER JOIN PREDICATE CHANGE

Keep LMT part and Force join order : FILE\$ - SYS_DBA_SEGS - X\$KTFBUE (+ change join predicate)

	Id		Operation		Name		Rows		Cost	
	1		NESTED LOOPS				1		1098	
	2		NESTED LOOPS				1		1097	
	3		TABLE ACCESS BY INDEX ROWID		FILE\$		1		1	
	*	4		INDEX UNIQUE SCAN		I_FILE1		1		
	*	5		VIEW		SYS_DBA_SEGS		1		1096
	*	50		FIXED TABLE FIXED INDEX		X\$KTFBUE (ind:1)		1		1

Predicate Information (identified by operation id):

```

4 - access(F.FILE#=15)
5 - filter(DS.TABLESPACE_ID=F.TS#)
50 - filter(E.KTFBUEBNO<=1904005 AND
          E.KTFBUEBNO+E.KTFBUEBLKS-1>=1904005 AND
          E.KTFBUESEGFNO=DS.RELATIVE_FNO AND
          E.KTFBUESEGBNO=DS.HEADER_BLOCK AND
          E.KTFBUESEGTSN=DS.TABLESPACE_ID AND
          E.KTFBUEFNO=F.RELFILE#)

```

```

Rows      Row Source Operation
-----
1  NESTED LOOPS  (cr=676209 pr=19124 pw=0 time=301272629 us)
18664  NESTED LOOPS  (cr=599961 pr=321 pw=0 time=10755477 us)
1  TABLE ACCESS BY INDEX ROWID OBJ#(17) (cr=2 pw=0 time=71 us)
1  INDEX UNIQUE SCAN OBJ#(41) (cr=1 time=33 us)(object id 41)
18664  VIEW  (cr=599959 pr=321 pw=0 time=10736736 us) ---
1  FIXED TABLE FIXED INDEX X$KTFBUE (ind:1) (cr=76248 pr=18803 time=290484962 us)

```

```

SELECT owner , segment_name , segment_type
FROM dba_extents
WHERE file_id = 15
AND 1904005 BETWEEN block_id AND block_id + blocks -1

```

EXECUTION TIME BEFORE THE JOIN CHANGE

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	17.31	17.00	1	221088	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	275.50	1864.48	92643	703546	633	1
total	4	292.81	1881.48	92644	924634	633	1

EXECUTION TIME AFTER THE JOIN CHANGE

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	1.22	1.17	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	62.21	299.57	19124	620097	274	1
total	4	63.43	300.74	19124	620097	274	1

That is purpose of Bug.5029334 SLOW QUERY ON DBA_EXTENTS

View DBA_EXTENTS changed in 10.2.0.4

Main words are

- Simplify to get a readable plan
- Understand what Oracle is doing
- Check if we can do less work to go faster