Reading Oracle SQL Execution Plans

Dave Abercrombie
Principal Database Architect, Convio
http://aberdave.blogspot.com/

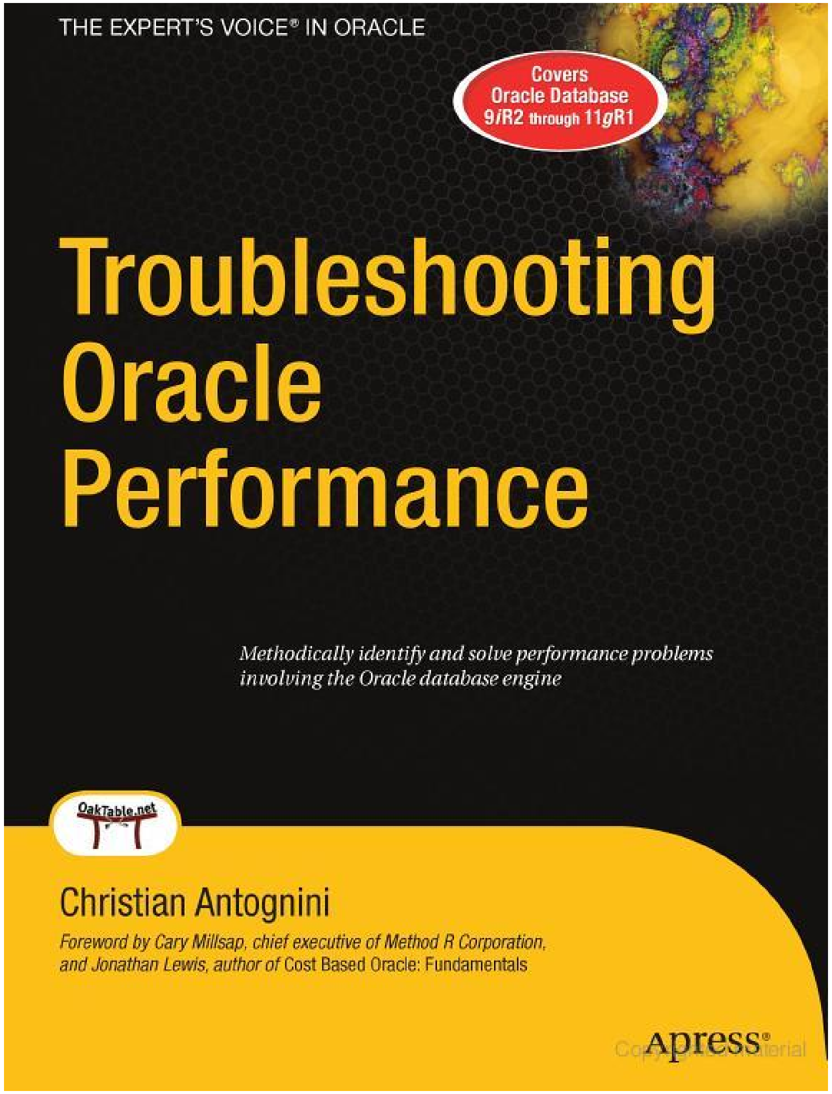
NoCOUG Spring Conference, May 20 2010

Agenda

- DBMS_XPLAN and Cardinality Feedback
- Parent – Child relationships
- Three types of operations
- Blocking vs. Non-blocking
- Examples of each type

- NOT: operation details
- NOT: tuning

Troubleshooting Oracle Performance



THE EXPERT'S VOICE® IN ORACLE

Covers Oracle Database 9iR2 through 11gR1

Troubleshooting Oracle Performance

Methodically identify and solve performance problems involving the Oracle database engine

OakTable.net

Christian Antognini

Foreword by Cary Millsap, chief executive of Method R Corporation, and Jonathan Lewis, author of Cost Based Oracle: Fundamentals

Apress®

By Christian Antognini

ISBN13: 978-1-59059-917-4
ISBN10: 1-59059-917-9
616 pp.
Published Jun 2008
http://apress.com/

Part 3, Chapter 6

Apress Errata

Antognini Errata

Cardinality Feedback – two components

- 1) Put session into special mode

  Gathers execution details for each step

- 2) Use DBMS_XPLAN to get these details

  Compare optimizer estimates to actual performance

```
alter session set STATISTICS_LEVEL = ALL;

@your-query-here.sql

select * from table
     (dbms_xplan.DISPLAY_CURSOR(null, null, 'ALLSTATS'));

alter session set STATISTICS_LEVEL = TYPICAL;
```

Cardinality Feedback

- Requires that query actually be run
  - On **representative data and stats**, right?!
- Eliminates (most guesswork)
  - Shows where to focus investigation
- DBMS_XPLAN is very useful even without cardinality feedback (real plan, details, AWR)

DBMS_XPLAN methods

- Pipelined function (aka table function):

```
select * from table(dbms_xplan....);
```

Method	Use	Data source
DISPLAY	Explain plan	Plan table
DISPLAY_CURSOR	Real plan	**Cursor in SGA**
DISPLAY_AWR	History	AWR Repository
DISPLAY_SQLSET	SQL Tuning sets	SQLSET views

DBMS_XPLAN.DISPLAY_CURSOR

- Three arguments
  - **`sql_id`**
  - **`child_number`**
  - **`format`**
- Useful even without Cardinality Feedback
  - Gets the real plan

SQL_ID, argument #1

- Like a hash of SQL text
- NULL argument defaults to pervious SQL
  - But only with **`set serveroutput off`**
- Or, find your SQL_ID

```
select sql_id, executions,
buffer_gets, sql_text
from v$sql
where sql_text like '%&unique_string%'
```

- Use V$SQLSTATS in production
- Distinctive string in SQL
  - In comment, or as column name
- Change as needed – to force a reparse

CHILD_NUMBER, argument #2

- Parent cursor: SQL text
  `v$sqlarea`

- Child Cursor: Execution plan and environment
  `v$sql`

- NULL usually fine

FORMAT, argument #3

- ALLSTATS
  - **Required by the Cardinality Feedback method**
- LAST
  - Limits to **most recent** execution
- PEEKED_BINDS
  - Bind variables used at parse
- Single string,
  concatenated with space and plus sign
  - Example: `'typical +peeked_binds'`
- See Oracle docs for more options

Gathering all stats

- **Required by the Cardinality Feedback method**

- Session level:

  ```
  alter session set STATISTICS_LEVEL = ALL;
  ```

- SQL level (hint):

  ```
  select /*+ gather_plan_statistics */ ...
  ```

- **Adds overhead**, so set back to normal

  About 2,000 gets

  ```
  alter session set STATISTICS_LEVEL = TYPICAL;
  ```

Cardinality Feedback Recipe

```
spool using-your-favorite-convention.txt
alter session set STATISTICS_LEVEL = ALL;
set serveroutput off
@your-query-here.sql
select * from table
     (dbms_xplan.DISPLAY_CURSOR(null, null, 'ALLSTATS'));
alter session set STATISTICS_LEVEL = TYPICAL;
spool off
```

- Change SQL text to force re-parse between tests
- Add comments to SQL text or spool file
- Look for actual/estimated rows > ~100

Example 1

```
-------------------------------------------------------------------------------------------------
|Id | Operation                     | Name              | Starts | E-Rows | A-Rows |   A-Time    |
-------------------------------------------------------------------------------------------------
| 1 |  SORT ORDER BY                |                   |      1 |   1256 |  10387 |00:01:40.89 |
| 2 |   HASH JOIN SEMI              |                   |      1 |   1256 |  10387 |00:01:40.88 |
| 3 |    TABLE ACCESS BY INDEX ROWID| CONSTITUENT       |      1 |   1256 |    117K|00:01:40.47 |
| 4 |     INDEX RANGE SCAN          | ITOPS_BZ41319_CUS |      1 |    102 |    117K|00:00:00.73 |
| 5 |    INLIST ITERATOR            |                   |      1 |        |  24269 |00:00:00.05 |
| 6 |     INDEX RANGE SCAN          | GROUP_USER_INDEX  |      2 |  40875 |  24269 |00:00:00.02 |
-------------------------------------------------------------------------------------------------
```

- Operation ID #5 expected 102 rows, but got 117,000 – investigate this optimizer confusion

Parent – Child relationships

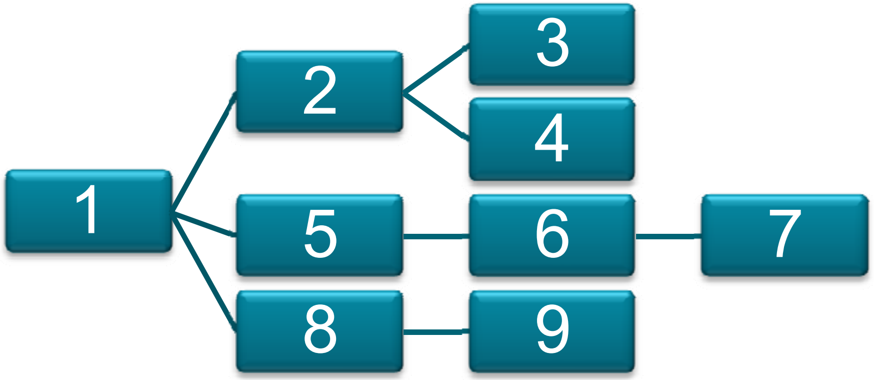
- A parent has one or more children
- A child has a single parent
- Only one root without a parent
- Children indented relative to parent
- Parent right before children (lower ID)
- `v$sql_plan_statistics_all.parent_id`

Parent – Child Example

```
----------------------------------------
|  ID | Operation                       |
----------------------------------------
|   1 |  UPDATE                         |
|   2 |   NESTED LOOPS                  |
| * 3 |    TABLE ACCES FULL             |
| * 4 |    INDEX UNIQUE SCAN            |
|   5 |   SORT AGGREGATE                |
|   6 |    TABLE ACCESS BY INDEX ROWID  |
| * 7 |     INDEX RANGE SCAN            |
|   8 |   TABLE ACCESS BY INDEX ROWID   |
| * 9 |    INDEX UNIQUE SCAN            |
----------------------------------------
```

Parent – Child (tree hierarchy)

```
-----------------------------------------
|  ID | Operation                       |
-----------------------------------------
|   1 |  UPDATE                         |
|   2 |   NESTED LOOPS                  |
| * 3 |    TABLE ACCES FULL             |
| * 4 |    INDEX UNIQUE SCAN            |
|   5 |   SORT AGGREGATE                |
|   6 |    TABLE ACCESS BY INDEX ROWID  |
| * 7 |     INDEX RANGE SCAN            |
|   8 |   TABLE ACCESS BY INDEX ROWID   |
| * 9 |    INDEX UNIQUE SCAN            |
-----------------------------------------
```



Three Types of Operations

- About 200 exist, of these three types:

  Stand-alone

  Unrelated-Combine

  Related-Combine

Note: these terms invented by Christian Antognini, and are generally not used elsewhere

Blocking vs. non-blocking

- Blocking operations
  - Process data in sets
  - Example: **SORT** – the first row might be anywhere in set

- Non-blocking operations
  - Process data one row at a time
  - Example: **FILTER** – each row evaluated independently

Note: these names are a little counterintuitive. Think of “blocking” as “sets” or “blocks” of data, rather than as “interfering” or “obstructing”.

Type 1: Stand-alone Operations

- Definition: all operations having at most one child

- Vast majority are this type (~180 out of ~200)

- Rules:

  Child executed before parent
  (with two important exceptions)

  Child executed at most once

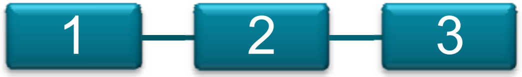
  Child “feeds” rows to its parent

Stand-alone example (start)

```
Select deptno, count(*) from emp where job = 'CLERK’' and sal < 1200  group by deptno;

---------------------------------------------------------------------
|  ID | Operation                       | Name        | Starts | A-rows |
---------------------------------------------------------------------
|   1 |  HASH GROUP BY                  |             | 1       | 2       |
| * 2 |   TABLE ACCESS BY INDEX ROWID   | EMP         | 1       | 3       |
| * 3 |    INDEX RANGE SCAN             | EMP_JOB_I   | 1       | 4       |
---------------------------------------------------------------------

2 – filter("SAL"<1200)
3 – access("JOB"='CLERK')
```



- All are stand-alone
- 1 and 2 have children, so they cannot execute first
- Execution must therefore start with 3

Stand-alone example (details)

```
Select deptno, count(*) from emp where job = 'CLERK’' and sal < 1200  group by deptno;

---------------------------------------------------------------------
|  ID | Operation                       | Name       | Starts | A-rows |
---------------------------------------------------------------------
|   1 |  HASH GROUP BY                  |            | 1      | 2      |
| * 2 |   TABLE ACCESS BY INDEX ROWID   | EMP        | 1      | 3      |
| * 3 |    INDEX RANGE SCAN             | EMP_JOB_I  | 1      | 4      |
---------------------------------------------------------------------

2 – filter("SAL"<1200)
3 – access("JOB"='CLERK')
```

- Operation #3 scans index for JOB, feeding four rowids to parent #2
- Operation #2 goes to table blocks using rowids, finding three rows (sal<1200) that it feeds to #1
- Operation #1 does "group by" returning 2 rows

Stand-alone rule exceptions

- Basic rule: "Child executed before parent"

- But, in two important exceptions, a parent may decide that:
  - It makes no sense to finish child execution, or
  - It makes no sense to even start child execution

- In other words, parents can sometimes control child execution.

Stand-alone exception: COUNT STOPKEY

```
select * from emp where rownum <= 10;

------------------------------------------------------
|  ID | Operation           | Name | Starts | A-rows |
------------------------------------------------------
|   1 |  COUNT STOPKEY      |      |      1 |     10 |
| * 2 |   TABLE ACCESS FULL | EMP  |      1 |     10 |
------------------------------------------------------

1 - filter(ROWNUM<=10)
```

- Parent operation #1 stops child operation #2 after 10 rows.

- BUT: "blocking" operations cannot be stopped, because they need to be fully processed before returning first row to their parent (example follows)

Blocking operations cannot be stopped

```
select * from (select * from emp order by sal desc) where rownum < 10;

-----------------------------------------------------------
|  ID | Operation                | Name | Starts | A-rows |
-----------------------------------------------------------
| * 1 |  COUNT STOPKEY           |      | 1      | 10     |
|   2 |   VIEW                   |      | 1      | 10     |
| * 3 |    SORT ORDER BY STOPKEY |      | 1      | 10     |
|   4 |     TABLE ACCESS FULL    | EMP  | 1      | 14     |
-----------------------------------------------------------
```

- "Blocking" operations cannot be stopped, because they need to be fully processed before returning first row to their parent

- Child operation #4 (emp full scan) cannot be stopped because of the "order by".

Stand-alone exception: FILTER

```
select * from emp where job = 'CLERK' and 1 = 2;

-------------------------------------------------------------------
|  ID | Operation                    | Name       | Starts | A-rows |
-------------------------------------------------------------------
| * 1 |  FILTER                      |            | 1      | 0      |
|   2 |   TABLE ACCESS BY INDEX ROWID | EMP        | 0      | 0      |
| * 3 |    INDEX RANGE SCAN          | EMP_JOB_I  | 0      | 0      |
-------------------------------------------------------------------

1 - filter(NULL IS NOT NULL)
3 - access("JOB"='CLERK')
```

- Standard rules suggest that execution starts with operation #3,
- BUT: the FILTER operation controls its children to prevent any execution, since no rows can pass it anyway

Type 2: Unrelated-Combine Operations

- Definition: Multiple children, **independently executed**

```
AND-EQUAL, BITMAP AND, BITMAP OR,
BITMAP  MINUS, CONCATENATION,
CONNECT BY WITHOUT FILTERING,
HASH JOIN, INTERSECTION,
MERGE JOIN,  MINUS,
MULTI-TABLE INSERT, SOL MODEL,
TEMP TABLE TRANSFORMATION,
and UNION-ALL
```

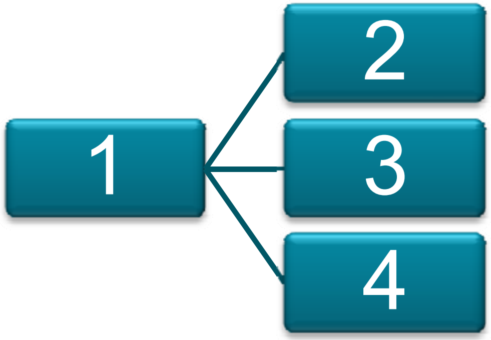
Unrelated-Combine Operation Rules

- Children executed before parent
- Children executed sequentially, in ID order
- Each child must complete before moving on to the next child
- Every child "feeds" rows to the parent

Unrelated-Combine Example (tree)

```
select ename from emp
union all
select dname from dept
union all
select '%' from dual;

-----------------------------------------------------
| ID | Operation            | Name | Starts | A-rows |
-----------------------------------------------------
|  1 |  UNION-ALL           |      | 1      | 19     |
|  2 |   TABLE ACCESS FULL  | EMP  | 1      | 14     |
|  3 |   TABLE ACCESS FULL  | DEPT | 1      | 4      |
|  4 |   FAST DUAL          |      | 1      | 1      |
-----------------------------------------------------
```



Unrelated-Combine Example (details)

```
-----------------------------------------------------
| ID | Operation          | Name | Starts | A-rows |
-----------------------------------------------------
|  1 |  UNION-ALL         |      | 1      | 19     |
|  2 |   TABLE ACCESS FULL | EMP  | 1      | 14     |
|  3 |   TABLE ACCESS FULL | DEPT | 1      | 4      |
|  4 |   FAST DUAL         |      | 1      | 1      |
-----------------------------------------------------
```

- Operation #1 has three children, with #2 having the lowest ID, so execution starts with #2.
- After #2 sends its 14 rows to the parent #1, operation #3 starts executing.
- After #3 sends its 4 rows to the parent #1, operation #4 starts executing.
- After #4 sends its 1 row to parent #1, the parent builds a single results set and returns it to caller.

Type 3: Related-Combine Operations

- Definition: Multiple children, and one child controls the execution of all other children

```
NESTED LOOPS,
UPDATE*,
FILTER*,
CONNECT BY WITH FILTERING,
and BITMAP KEY ITERATION
```

* note: **UPDATE** and **FILTER** can also be "stand-alone", depending on number of children

Related-Combine Operation Rules

- Children executed before parent
- Child with lowest ID controls execution of the others
- Children execute in ID order, but interleaved (not sequentially)
- The controlling child is executed (at most) once, the others may be executed many or zero times.
- Not every child "feeds" the parent.

Nested Loops (a "related-combine")

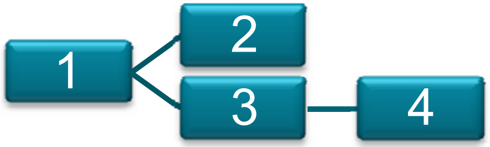
- A join, so always has exactly two children
- Child with smaller ID is the "driving rowsource" aka "outer loop"
- Other child is the "inner loop"
- Inner loop is executed once for every row returned by outer loop.

Nested Loops example (related-combine)

```
select *
from emp, dept
where emp.deptno = dept.deptno
and emp.comm is null
and dept.dname != 'SALES';

-------------------------------------------------------------------
|  ID | Operation                        | Name    | Starts | A-rows |
-------------------------------------------------------------------
|   1 |  NESTED LOOPS                    |         |      1 |      8 |
| * 2 |   TABLE ACCESS FULL              | EMP     |      1 |     10 |
| * 3 |   TABLE ACCESS BY INDEX ROWID    | DEPT    |     10 |      8 |
| * 4 |    INDEX UNIQUE SCAN             | DEPT_PK |     10 |     10 |
-------------------------------------------------------------------

2 - filter("EMP"."COMM" IS NULL)
3 - filter("DEPT"."DNAME"<>'SALES')
4 - access("EMP"."DEPTNO"="DEPT"."DEPTNO")
```



Nested Loops Example (details)

```
-------------------------------------------------------------------
|  ID | Operation                     | Name    | Starts | A-rows |
-------------------------------------------------------------------
|   1 |  NESTED LOOPS                 |         |      1 |      8 |
| * 2 |   TABLE ACCESS FULL           | EMP     |      1 |     10 |
| * 3 |   TABLE ACCESS BY INDEX ROWID | DEPT    |     10 |      8 |
| * 4 |    INDEX UNIQUE SCAN          | DEPT_PK |     10 |     10 |
-------------------------------------------------------------------

2 - filter("EMP"."COMM" IS NULL)
3 - filter("DEPT"."DNAME"<>'SALES')
4 - access("EMP"."DEPTNO"="DEPT"."DEPTNO")
```

- Operation #1 has two children, and #2 has the lowest ID, so execution starts with controlling #2.
- After #2 full scans EMP, it tells #3 to do 10 loops.
- Using "stand-alone" rules, operation #4 executes first, sending its 10 rowids to #3, one at a time.
- Operation #3 looks at DEPT table blocks one at a time, filtering out two rows, sending 8 rows to #1

FILTER (a "related-combine")

- Can be considered "stand-alone" if it has a single child.
- If it has two or more children, it works similar to NESTED LOOPS.

FILTER example (related-combine)

```
select *
from emp
where not exists ( select 0
                     from dept
                    where dept.dename = 'SALES'
                      and dept.deptno = emp.deptno)
  and not exists ( select 0
                     from bonus
                    where bonus.ename = emp.ename);
```

- Note row counts:
  - Three distinct values of DNAME
  - Six EMP rows for SALES

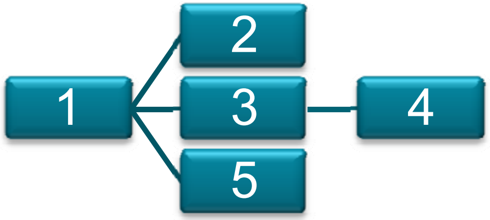
```
select dname, count(*)
from emp, dept
where emp.deptno = dept.deptno
group by dname;

DNAME         COUNT(*)
------------ --------
ACCOUNTING          3
RESEARCH            5
SALES               6
```

FILTER example (related-combine)

```
select *
from emp
where not exists ( select 0
                    from dept
                   where dept.dename = 'SALES'
                     and dept.deptno = emp.deptno)
  and not exists ( select 0
                    from bonus
                   where bonus.ename = emp.ename);
```

```
-------------------------------------------------------------------
|  ID | Operation                     | Name    | Starts | A-rows |
-------------------------------------------------------------------
| * 1 |  FILTER                       |         |      1 |      8 |
|   2 |   TABLE ACCESS FULL           | EMP     |      1 |     14 |
| * 3 |   TABLE ACCESS BY INDEX ROWID | DEPT    |      3 |      1 |
| * 4 |    INDEX UNIQUE SCAN          | DEPT_PK |      3 |      3 |
| * 5 |   TABLE ACCESS FULL           | BONUS   |      8 |      0 |
-------------------------------------------------------------------
```



```
1 - filter(( ... )) note: Oracle v$ views can be buggy
3 - filter("DEPT"."DNAME"='SALES')
4 - access("DEPT"."DEPTNO"=:B1)
5 - filter("BONUS"."ENAME"=:B1)
```

FILTER Example (details, 1 of 3)

```
-------------------------------------------------------------------
|  ID | Operation                     | Name    | Starts | A-rows |
-------------------------------------------------------------------
| * 1 |  FILTER                       |         |      1 |      8 |
|   2 |   TABLE ACCESS FULL           | EMP     |      1 |     14 |
| * 3 |   TABLE ACCESS BY INDEX ROWID | DEPT    |      3 |      1 |
| * 4 |    INDEX UNIQUE SCAN          | DEPT_PK |      3 |      3 |
| * 5 |   TABLE ACCESS FULL           | BONUS   |      8 |      0 |
-------------------------------------------------------------------

1 - filter(( ... ))
3 - filter("DEPT"."DNAME"='SALES')
4 - access("DEPT"."DEPTNO"=:B1)
5 - filter("BONUS"."ENAME"=:B1)
```

- Operation #1 has three children (#2, #3, #5), and #2 has the lowest ID, so execution starts at #2.
- After #2 full scans EMP, it returns 14 rows to #1
- To a first approximation, Operation #1 would control its other children (#3 an #5) to execute 14 times, once per row from #2. However, Oracle does some caching, once per distinct value.

FILTER Example (cont. details, 2 of 3)

```
-------------------------------------------------------------------
|  ID | Operation                    | Name    | Starts | A-rows |
-------------------------------------------------------------------
| * 1 |  FILTER                      |         |      1 |      8 |
|   2 |   TABLE ACCESS FULL          | EMP     |      1 |     14 |
| * 3 |   TABLE ACCESS BY INDEX ROWID | DEPT    |      3 |      1 |
| * 4 |    INDEX UNIQUE SCAN          | DEPT_PK |      3 |      3 |
| * 5 |   TABLE ACCESS FULL          | BONUS   |      8 |      0 |
-------------------------------------------------------------------

1 - filter(( ... ))
3 - filter("DEPT"."DNAME"='SALES')
4 - access("DEPT"."DEPTNO"=:B1)
5 - filter("BONUS"."ENAME"=:B1)
```

- Using "stand-alone" rules, Operation #4 executes three times, passing its rowids to its parent #3.
- Operation #3 looks at the table blocks for the rows specified by #4, looking at DNAME for 'SALES'. It finds one matching row, but since this is a NOT EXISTS, it causes the six 'SALES' rows to be excluded in #1 (no rows passed from #3)

FILTER Example (cont. details, 3 of 3)

```
-------------------------------------------------------------------
|  ID | Operation                     | Name    | Starts | A-rows |
-------------------------------------------------------------------
| * 1 |  FILTER                       |         |      1 |      8 |
|   2 |   TABLE ACCESS FULL           | EMP     |      1 |     14 |
| * 3 |   TABLE ACCESS BY INDEX ROWID | DEPT    |      3 |      1 |
| * 4 |    INDEX UNIQUE SCAN          | DEPT_PK |      3 |      3 |
| * 5 |   TABLE ACCESS FULL           | BONUS   |      8 |      0 |
-------------------------------------------------------------------

1 - filter(( ... ))
3 - filter("DEPT"."DNAME"='SALES')
4 - access("DEPT"."DEPTNO"=:B1)
5 - filter("BONUS"."ENAME"=:B1)
```

- Operation #5 full scans BONUS using :B1 passed from #1. Since (like #3) this operation is used only to implement restrictions, no rows are passed to parent #1. Anyway, no matches were found, so no more rows get restricted.
- Operation #1 passes eight rows to the caller (fourteen from #2 minus six from #3).

See book for further examples

- UPDATE
- CONNECT BY WITH FILTERING

Summary

- Strict parent/child, rooted tree hierarchy
- About 200 operations exist, of these three types:
  - Stand-alone
  - Unrelated-Combine (14)
  - Related-Combine (5)
- Blocking or Non-blocking
  - Blocking is set based (e.g., sort)
  - Non-Blocking is row-based (e.g., simple filter)
- Rules for each type, apply recursively
- Confirmed with
  - "all stats" plans: A-rows and A-time,
  - Extended SQL Event 10046, tracing, and
  - Tanel Poder's PlanViz, Iggy Fernadez's tool (PDF)