Executions Plans
after the first steps

Jonathan Lewis

jonathanlewis.wordpress.com

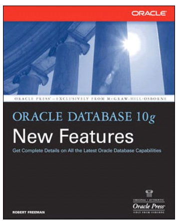
www.jlcomp.demon.co.uk

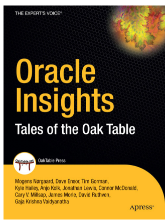
My History

Independent Consultant

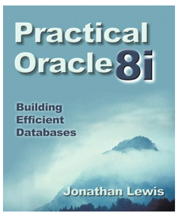
36 years in IT
31 using Oracle (5.1a on MSDOS 3.3)

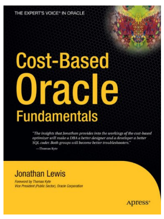
Strategy, Design, Review,
Briefings, Educational,
Trouble-shooting

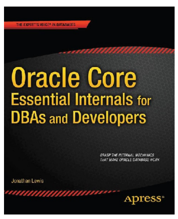




Founder Member of Oak Table Network
Best Presentation HROUG 2016
UKOUG Lifetime Award (IPA) 2013
ODTUG 2012 Best Presenter (d/b)
UKOUG Inspiring Presenter 2012
UKOUG Inspiring Presenter 2011
Select Editor's choice 2007
Oracle author of the year 2006
Oracle *ACE Director*
O1 visa for USA







Many slides have a foot-note. This is just two lines summarizing the highlights of the slide so that you have a reference when reading the handouts at a later date.

Acquisition (a)

```
explain plan for …
select * from table(dbms_xplan.display)
```

Problem:
This can produce plans that won't appear at run time.
- any bind variables are assumed to be character type
- there are no bound values to peek

SQL*Plus special

```
set autotrace traceonly explain
execute query
```

Problem:
As above, since this is just running "explain plan" under the covers.

Special case
For *"traceonly explain"* - Oracle will ***not*** run select statements, but it ***does*** run inserts, updates or deletes.

Note - the ***autotrace*** option in SQL*Developer will ask for values for any bind, run the query and pull the plan (and all the session statistics) from memory.

Acquisition (b)

SQL*Plus special case

```
set linesize …
set pagesize …
set trimspool on

set serveroutput off
execute a query
select * from table(dbms_xplan.display_cursor);
```

General call

```
select  *
from    table(dbms_xplan.display_cursor(
                {sql_id}, {child_number}, {format options}
        )
;
```


In SQL*Plus a call to ***display_cursor*** with no parameters will display the execution plan of the most recent statement - which might be ***dbms_output.get_lines()***.

Acquisition (c)

```
alter session set statistics_level = all;
alter session set "_rowsource_execution_statistics" = true
add /*+ gather_plan_statistics */ hint to the query
-- execute the query

select *
from table(
        dbms_xplan.display_cursor(null,null,'allstats [last]')
);
```

Reports

- the number of times each line of the plan was run.
- the number of rows supplied by each line to it parent
- the number of buffer gets (accumulating up the plan) due to each line
- the number of disk reads (accumulating up the plan) due to each line

Note: the hint strategy samples for some of the statistics.

Gathering rowsource execution statistics can add a significant CPU overhead to the query, depending on operating system and the operations used by the query.

Acquisition (d)

SQL Monitor

(requires performance and diagnostic licences)

- `add /*+ monitor */ hint to the query`
- `any query that runs more than 5 seconds`
- `any query that executes as a parallel query`

```
set long 1000000 longchunksize 32000

select
        dbms_sqltune.report_sql_monitor(
--              sql_id                  => '&m_sql_id',
--              start_time_filter       => sysdate - 30/(24 * 60),
                type                    =>'TEXT'  /*  'ACTIVE'  */
) text_line
from    dual
;
```


There is a graphic user interface with a very nice presentation available through both SQL*Developer and the Enterprise Manager, but a text output is very helpful.

Projection (a)

```
merge
into    ord
using   x
on      (ord.global_ext_id = x.ext_id)
when matched then
        update set ord.ord_id = x.ord_id
;

| Id  | Operation             | Name | Rows  | Bytes | Cost (%CPU)| Time     |
|   0 | MERGE STATEMENT       |      |  1100 | 28600 |   488   (1)| 00:00:01 |
|   1 |  MERGE                | ORD  |       |       |            |          |
|   2 |   VIEW                |      |       |       |            |          |
|*  3 |    HASH JOIN          |      |  1100 |  256K|   488   (1)| 00:00:01 |
|   4 |     TABLE ACCESS FULL| ORD  |  1000 |  114K|     7   (0)| 00:00:01 |
|   5 |     TABLE ACCESS FULL| X    |   100K|   11M|   480   (1)| 00:00:01 |

Predicate Information (identified by operation id):
   3 - access("ORD"."GLOBAL_EXT_ID"="X"."EXT_ID")
```


I happen to have perfect indexes on tables ***x*** and ***ord*** that include all the columns in the query - why am I getting full tablescans ?

Projection (b)

```
select * from table(dbms_xplan.display(null,null,'projection'));

Column Projection Information (identified by operation id):
-----------------------------------------------------------
   1 - SYSDEF[4], SYSDEF[32720], SYSDEF[1], SYSDEF[112], SYSDEF[32720]
   2 - "X"."ORD_ID"[NUMBER,22]

   3 - (#keys=1) "ORD"."GLOBAL_EXT_ID"[NUMBER,22],
       "X"."EXT_ID"[NUMBER,22], "ORD".ROWID[ROWID,10],
       "ORD"."ORD_ID"[NUMBER,22], "ORD"."PADDING"[VARCHAR2,100],
       "ORD"."V1"[VARCHAR2,10], "X"."ORD_ID"[NUMBER,22],
       "X"."PADDING"[VARCHAR2,100], "X"."V1"[VARCHAR2,10]

   4 - "ORD".ROWID[ROWID,10], "ORD"."ORD_ID"[NUMBER,22],
       "ORD"."GLOBAL_EXT_ID"[NUMBER,22], "ORD"."V1"[VARCHAR2,10],
       "ORD"."PADDING"[VARCHAR2,100]
   5 - "X"."ORD_ID"[NUMBER,22], "X"."EXT_ID"[NUMBER,22],
       "X"."V1"[VARCHAR2,10], "X"."PADDING"[VARCHAR2,100]
```


If we request the projection information (which columns are supplied by each rowsource) we see lots of redundant columns. Let's eliminate them.

Projection (c)

```
merge
into     (select ord.ord_id, ord.global_ext_id from ord) ord
using    (select ext_id, ord_id from x) x
on       (ord.global_ext_id = x.ext_id)
when matched then
         update set ord.ord_id = x.ord_id
;
```

```
| Id  | Operation                 | Name    | Rows   | Bytes  | Cost (%CPU)| Time     |
|   0 | MERGE STATEMENT           |         |  1100  | 28600  |   108   (2)| 00:00:01 |
|   1 |  MERGE                    | ORD     |        |        |            |          |
|   2 |   VIEW                    |         |        |        |            |          |
|*  3 |    HASH JOIN              |         |  1100  | 44000  |   108   (2)| 00:00:01 |
|   4 |     TABLE ACCESS FULL     | ORD     |  1000  | 30000  |     7   (0)| 00:00:01 |
|   5 |     INDEX FAST FULL SCAN| X_IDX2  |   100K|    976K|   100   (1)| 00:00:01 |

Predicate Information (identified by operation id):
   3 - access("ORD"."GLOBAL_EXT_ID"="X"."EXT_ID")
```


List only the relevant columns as inline subqueries and not only is the volume of data reduced, we've now managed to use a more efficient path to get some of it.

Projection (d)

```
select * from table(dbms_xplan.display(null,null,'projection'));
```

```
Column Projection Information (identified by operation id):
-----------------------------------------------------------
   1 - SYSDEF[4], SYSDEF[32720], SYSDEF[1], SYSDEF[112], SYSDEF[32720]
   2 - "X"."ORD_ID"[NUMBER,22]
   3 - (#keys=1) "ORD"."GLOBAL_EXT_ID"[NUMBER,22], "EXT_ID"[NUMBER,22],
       "ORD".ROWID[ROWID,10], "ORD"."ORD_ID"[NUMBER,22], "ORD_ID"[NUMBER,22]
   4 - "ORD".ROWID[ROWID,10], "ORD"."ORD_ID"[NUMBER,22],
       "ORD"."GLOBAL_EXT_ID"[NUMBER,22]
   5 - "ORD_ID"[NUMBER,22], "EXT_ID"[NUMBER,22]
```


When we check the projection information we can see that we're no longer carrying the redundant columns. This can make a huge difference to the hash join.

First child First (a)

There are two basic patterns in a plan:

– Single child parent
– Multi-child parent

There are two basic rules for ***simple*** plans:

– A child operation generates a rowsource for its parent*[1]*
– A parent operation calls its children in turn*[2][3]*

[1] And ***only*** for its parent

[2] In the order they appear in the execution plan

[3] But calls can be repeated

First child First (b)

Single-child parent

```
-----------------------------------------------------
| Id  | Operation                         | Name  |
-----------------------------------------------------
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| T1    |
|*  2 |   INDEX RANGE SCAN                | T1_PK |
-----------------------------------------------------
```

It is not possible to visit the table by rowid until the parent calls the child to supply a rowid (or list of rowids).

Recent versions of Oracle can pass a set of rowids to the parent from a range scan - in older versions the parent has to call the child repeatedly for *"the next"* rowid until it gets *"no more rowids"*.

First child First (c)

Multi-child parent (hash join)

```
--------------------------------------
| Id  | Operation                    |
--------------------------------------
|   1 |  HASH JOIN                   |
|   2 |   First rowsource generator  |
|   3 |   Second rowsource generator |
--------------------------------------
```

The hash join operation calls its first child (once) to generate a rowsource and creates a hash table from it, hoping that it can be built completely in memory

Then it calls the second child to derive and start supplying a second non-correlated row source and uses it to probe the hash table.

This is the simplest "first child first" - the first child is always the one that supplies the date that is used for the hash table.

First child First (d)

Multi-child parent (nested loop)

```
----------------------------
| Id  | Operation          |
----------------------------
|   1 |  NESTED LOOP       |
|   2 |   First rowsource  |
|   3 |   Second rowsource |
----------------------------
```

The nested loop operation calls its first child operation to derive and then start supplying rows one at a time.

Then, for each row from the first rowsource, it calls the second child operation to generate a correlated rowsource.

Since we may have to call the second operation many times we hope that it has an efficient method of generating data - which is where we typically see the first signs of recursion.

First child First (d2)

Multi-child parent (nested loop)

```
-------------------------------------------------------
| Id  | Operation                                     |
-------------------------------------------------------
|   1 |  NESTED LOOP                                  |
|   2 |   First rowsource                             |
|   3 |   TABLE ACCESS BY INDEX ROWID BATCHED| T1     |
|*  4 |    INDEX RANGE SCAN                  | T1_PK |
-------------------------------------------------------
```

We typically expect to see the second rowsource of a nested loop is a table access by rowid that has to call its child operation (and index access) before it can visit the table.

First child First (e)

Multi-child parent (merge join)

```
------------------------------------
| Id  | Operation                  |
------------------------------------
|   1 |  MERGE JOIN                |
|   2 |   First ordered rowsource  |
|   3 |   Second ordered rowsource |
------------------------------------
```

The merge join operation calls its first child to supply an ordered rowsource - which it may acquire in its entirety and attempt to keep in memory.

Then it calls the second child once for each row in the first rowsource to search for matching rows. The first time the second child operation is called it will derive a suitable ordered non-correlated rowsource and store it in memory (possibly spilling to disc) to make it possible for the searches to operate efficiently.

The merge join is an interesting hybrid. It starts with a non-correlated second child like the hash join, then does the equivalent of a nested loop into the resulting data set

First child First (f)

Multi-child parent (filter)

```
------------------------------
| Id    | Operation          |
------------------------------
|   1   |  FILTER            |
|   2   |   First rowsource  |
|   3   |   Second rowsource |
|   ... |   ...              |
|   N+1 |   Nth rowsource    |
------------------------------
```

There are three main filter operation. The more common multi-child filter calls its first child to supply a rowsource, and then for each row in turn it calls each following child operation to supply a correlated rowsource, until one of the child operations satisfies a condition that allows the parent to move on to the next row from the first rowsource.

We'll see the other interpretations of the filter operation later.

1st child 1st - recursive descent (a)

A could have been generated by a parent operation combining the rowsources from its child operation so, for example, the first child of a hash join might be another hash join

```
| Id | Operation                    |
|  1 |  HASH JOIN                   |
|  2 |   First rowsource generator  |
|  3 |   Second rowsource generator |
```

↓

```
| Id  | Operation                    |
|  1  |  HASH JOIN                   |  We start here
|  2  |   Hash Join                  |  Which means we start here
|  2a |    Rowsource operation A     |  Which means we start here
|  2b |    Rowsource operation B     |
|  3  |   Second rowsource generator |
```

1^{st} child 1^{st} - recursive descent (b)

But rowsource operation A could have been the result of a hash join.

```
| Id  | Operation                   |
|  1  |  HASH JOIN                  |
|  2  |   Hash Join                 |
|  2a |    Rowsource operation A    |
|  2b |    Rowsource operation B    |
|  3  |   Second rowsource generator |
```

↓

```
| Id   | Operation                    |
|  1   |  HASH JOIN                   |  We start here
|  2   |   Hash Join                  |  Which means we start here
|  2a  |    Hash Join                 |  Which means we start here
|  2ax |     Rowsource operation AX   |  Which means we start here
|  2ay |     Rowsource operation AY   |
|  2b  |    Rowsource operation B     |
|  3   |   Second rowsource generator |
```


1^{st} child 1^{st} - recursive descent (c)

Let's assume all our base rowsource operations are tablescans

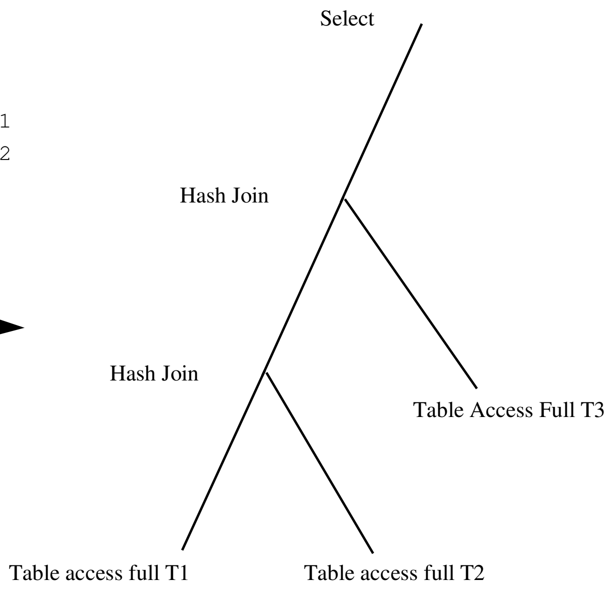
```
AX  ->   tablescan of t4
AY  ->   tablescan of t3
B   ->   tablescan of t2
2nd ->   tablescan of t1
```

↓

```
| Id   | Operation                   |
|  1   |  HASH JOIN                  |  We start here
|  2   |   Hash Join                 |  Which means we start here
|  2a  |    Hash Join                |  Which means we start here
|  2ax |     table access full t4    |  Which means we start here
|  2ay |     table access full t3    |    then scan and probe
|  2b  |    table access full t2     |      then scan and probe
|  3   |   table access full t1      |        then scan and probe
```

Text to Tree (a)

```
SELECT STATEMENT
    HASH JOIN
        HASH JOIN
            TABLE ACCESS FULL T1
            TABLE ACCESS FULL T2
        TABLE ACCESS FULL T3
```

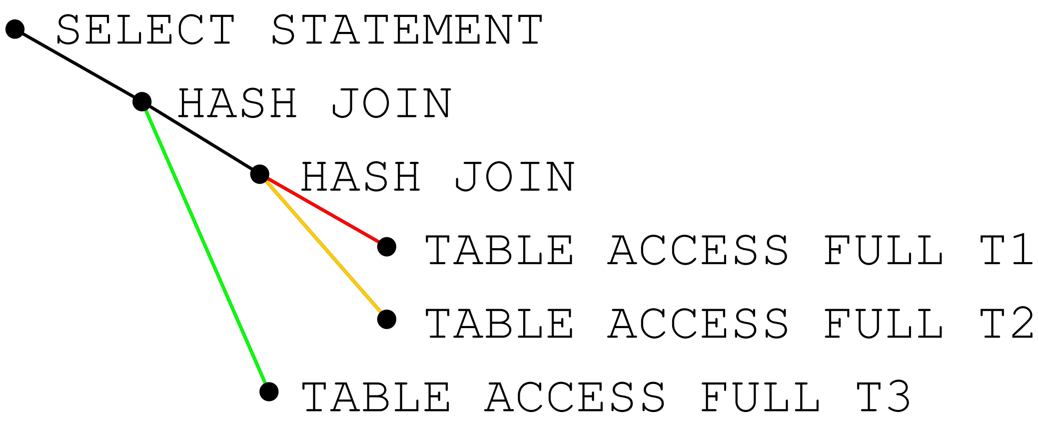


Jonathan Lewis

How can we turn a textual execution plan into a tree-diagram. The tree on the right is the picture for the plan on the left.

Topic
page 21

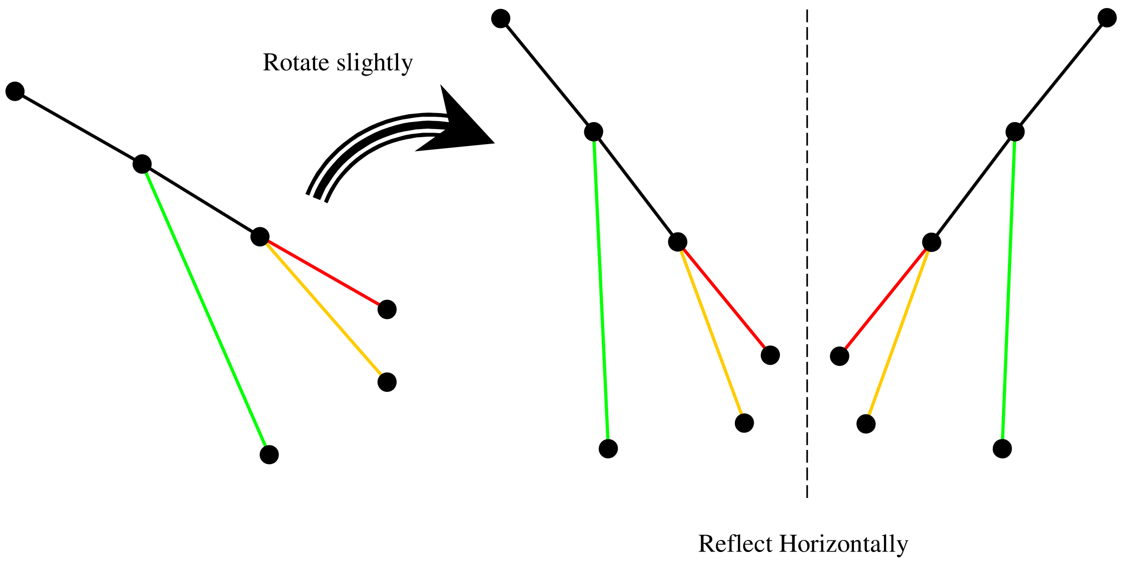
Text to Tree (a)



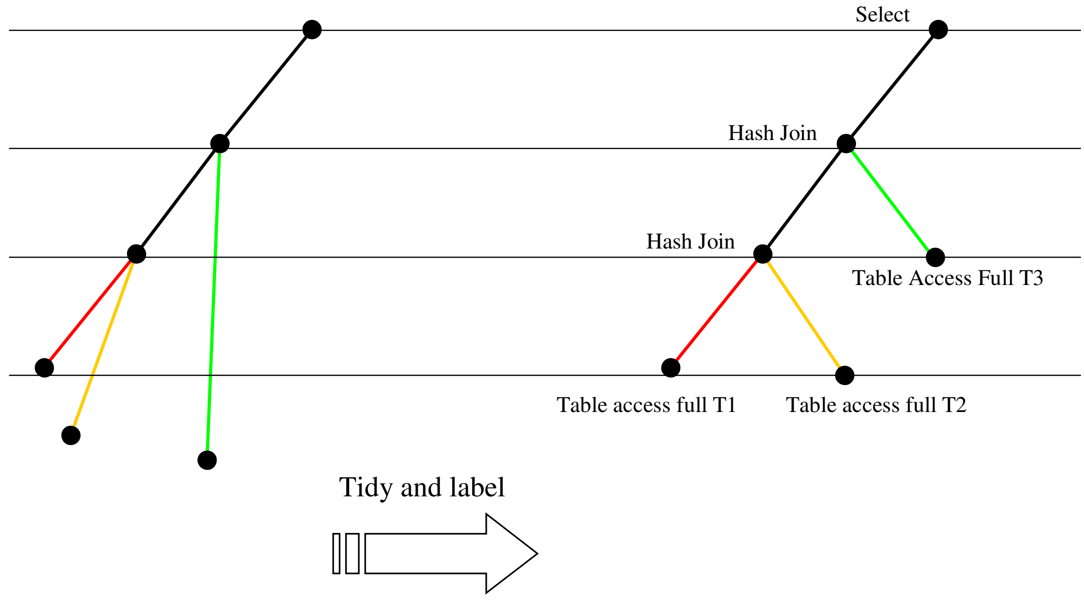
Jonathan Lewis

Topic
page 22

Text to Tree (a)



Text to Tree (a)



1^{st} child 1^{st} - recursive descent (d)

But what if it's the second rowsource that is the more complex one ?

```
| Id | Operation                 |
|  1 |  HASH JOIN                |
|  2 |   First rowsource generator  |
|  3 |   Second rowsource generator |
```

⬇

```
| Id  | Operation                     |
|  1  |  HASH JOIN                    |  We start here
|  2  |   First rowsource generator   |  which means we build this
|  3  |   Hash join                   |  but can't probe until
|  3p |    Rowsource operation P      |  we build this
|  3q |    Rowsource operation Q      |  and probe with 3q
```


1^{st} child 1^{st} - recursive descent (e)

But maybe the rowsource at 3q is also a little complicated

```
| Id  | Operation                     |
|  1  |  HASH JOIN                    |
|  2  |   First rowsource generator   |
|  3  |   Hash join                   |
|  3p |    Rowsource operation P      |
|  3q |    Rowsource operation Q      |
```

⬇

```
| Id   | Operation                      |
|  1   |  HASH JOIN                     |  We start here
|  2   |   First rowsource generator    |  Which means we build this
|  3   |   Hash join                    |  but can't probe until
|  3p  |    Rowsource operation P       |  we build this
|  3q  |     Hash join                  |  but can't probe until
|  3qm |      Rowsource operation QM    |  we build this
|  3qn |      Rowsource operation QN    |  and probe with 3qn
```

1^{st} child 1^{st} - recursive descent (f)

Assume, again, that all our base rowsource operations are tablescans

```
QN  ->   tablescan of t4
QM  ->   tablescan of t3
P   ->   tablescan of t2
1st ->   tablescan of t1
```

⬇

```
| Id   | Operation                       |
|  1   |  HASH JOIN                      |  We start here
|  2   |   table access full t1          |  Which means we build this
|  3   |   Hash join                     |  but can't probe until
|  3p  |    table access full t2         |  we build this
|  3q  |     Hash join                   |  but can't probe until
|  3qm |      table access full t3       |  we build this
|  3qn |      table access full t4       |  and probe with this
```


1^{st} child 1^{st} - recursive descent (g)

```
| Id   | Operation                 |   | Id | Operation                 |
|  1   |  HASH JOIN                |   |  1 |  HASH JOIN                |
|  2   |   Hash Join               |   |  2 |   table access full t1    |
|  3   |    Hash Join              |   |  3 |   Hash join               |
|  4   |     table access full t4  |   |  4 |    table access full t2   |
|  5   |     table access full t3  |   |  5 |    Hash join              |
|  6   |    table access full t2   |   |  6 |      table access full t3 |
|  7   |   table access full t1    |   |  7 |      table access full t4 |
```

```
/*+ leading(t4 t3 t2 t1) */
```

```
/*+
    leading(t4,t3,t2,t1)
    swap_join_inputs(t3)
    swap_join_inputs(t2)
    swap_join_inputs(t1)
*/
```

Both plans join tables t4 and t3. then join t2, then join t1

The ***join order*** is identical, the ***order of access*** is reversed

The difference between join order, order of appearance in the plan, and order of initial access gets more complicated in parallel execution.

Optional plans

```
| Id  | Operation               |
|   1 |  HASH JOIN              |
|   2 |   Table access full t1 |      -- build a hash table
|   3 |   Table access full t2 |      -- probe the hash table
```

What to you think happens if there's no relevant data in t1 ?

```
select * from t1 where 1 = 2;
```

```
| Id  | Operation            | Name | Rows   | Bytes | Cost (%CPU)|
|   0 | SELECT STATEMENT     |      |        |       |     1 (100)|
|*  1 |  FILTER              |      |        |       |            |
|   2 |   TABLE ACCESS FULL| T1   |  1000 |  8803K|   255   (2)|

Predicate Information (identified by operation id):
   1 - filter(NULL IS NOT NULL)
```


One variation of the FILTER operation is the one that says: *"in what circumstances should I execute my child operation"*. The plan is still following the standard rule.

Filter operation (a)

```
| Id  | Operation             | Name | Rows   | Bytes | Cost  |
|   0 | SELECT STATEMENT      |      |     10 |     40 |    33 |
|*  1 |  FILTER               |      |        |        |       |
|   2 |   HASH GROUP BY       |      |     10 |     40 |    33 |
|   3 |    TABLE ACCESS FULL| T1   |  3000 | 12000 |    15 |

Predicate Information (identified by operation id):
   1 - filter(COUNT(*)>10)
```

```
select  n1, count(*)
from    t1
group by
        n1
having
        count(*) > 10
/
```


But a plan of exactly the same (sort of) shape can have a different meaning. In this case we always execute the child first - then do some "late" elimination.

Filter operation (b)

```
| Id  | Operation           | Name   | Rows  | Bytes | Cost  (%CPU)|
|   0 | SELECT STATEMENT    |        |  3000 |   547K|    15   (0)|
|*  1 |  FILTER             |        |       |       |             |
|   2 |   TABLE ACCESS FULL| T1     |  3000 |   547K|    13   (0)|
|*  3 |   INDEX RANGE SCAN | T2_I1  |     1 |     4 |     2   (0)|

Predicate Information (identified by operation id):
   1 - filter( EXISTS (SELECT 0 FROM "T2" "T2" WHERE "T2"."N1">1000))
   3 - access("T2"."N1">1000)

select  *
from    t1
where   exists (
                select  null
                from    t2
                where   t2.n1 > 1000
        )
;
```

Jonathan Lewis
© 2002 - 2018

This looks like a "standard" filter operation - but it's an example that breaks "1st child 1st". The "constant subquery" is run first to test whether or not to run the scan of t1.

Topic
page 31

Filter operation (c)

```
set serveroutput off
alter session set statisics_level = all;

-- run query

select * from table(dbms_xplan.display_cursor(null,null,'allstats last'))


| Id |Operation           | Name   | Starts | E-Rows | A-Rows |    A-Time    | Buffers |
|  0 |SELECT STATEMENT    |        |      1 |        |      0 |00:00:00.01 |       2 |
|* 1 | FILTER             |        |      1 |        |      0 |00:00:00.01 |       2 |
|  2 |  TABLE ACCESS FULL| T1     |      0 |   3000 |      0 |00:00:00.01 |       0 |
|* 3 |  INDEX RANGE SCAN | T2_I1  |      1 |      1 |      0 |00:00:00.01 |       2 |

Predicate Information (identified by operation id):
   1 - filter( IS NOT NULL)
   3 - access("T2"."N1">1000)
```

Jonathan Lewis
© 2002 - 2018

We could use extended tracing (perhaps flushing the buffer cache first) to show that the tablescan doesn't happen - but enabling execution stats is quicker and easier.

Topic
page 32

Filter operation (d)

Variations on a simple correlated subquery.

```
select *
from    t1
where   n1 = (select /*+ no_push_subq */ max(n1) from t1)
;

| Id  | Operation                    | Name  | Rows | Bytes | Cost (%CPU)|
|   0 | SELECT STATEMENT             |       | 3000 |   547K|    15   (0)|
|*  1 |  FILTER                      |       |      |       |            |
|   2 |   TABLE ACCESS FULL          | T1    | 3000 |   547K|    13   (0)|
|   3 |   SORT AGGREGATE             |       |    1 |     4 |            |
|   4 |    INDEX FULL SCAN (MIN/MAX)| T1_I1 |    1 |     4 |     2   (0)|

Predicate Information (identified by operation id):
1 - filter("N1"= (SELECT /*+ NO_PUSH_SUBQ */ MAX("N1") FROM "T1" "T1"))
```


If I block subquery pushing the subquery nominally runs "for each row" - but in fact, thanks to "scalar subquery caching" it runs only once. Classic FILTER operation.

Filter operation (d)

```
select  *
from    t1
where   n1 = (select /*+ push_subquery */ max(n1) from t1)
;

| Id  | Operation             | Name | Rows  | Bytes | Cost (%CPU)|
|   0 | SELECT STATEMENT      |      |    15 |  2805 |    26   (0)|
|*  1 |  TABLE ACCESS FULL    | T1   |    15 |  2805 |    13   (0)|
|   2 |   SORT AGGREGATE      |      |     1 |     4 |            |
|   3 |    TABLE ACCESS FULL| T1   |  3000 | 12000 |    13   (0)|

Predicate Information (identified by operation id):
   1 - filter("N1"= (SELECT MAX("N1") FROM "T1" "T1"))
```


In any vaguely recent version of Oracle this subquery will be pushed by default. It is still a simple filter test (nominally) for each row.

Filter operation (e)

```
select  *
from    t1
where   n1 = (select max(n1) from t1)
;
```

```
| Id  | Operation                            | Name  | Rows  | Bytes | Cost (%CPU)|
|   0 | SELECT STATEMENT                     |       |    15 |  2805 |     2   (0)|
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| T1    |    15 |  2805 |     2   (0)|
|*  2 |   INDEX RANGE SCAN                   | T1_I1 |    15 |       |     1   (0)|
|   3 |    SORT AGGREGATE                    |       |     1 |     4 |            |
|   4 |     INDEX FULL SCAN (MIN/MAX)        | T1_I1 |     1 |     4 |     2   (0)|
```

```
Predicate Information (identified by operation id):
   2 - access("N1"= (SELECT MAX("N1") FROM "T1" "T1"))
```


Add an index on t1(n1) and the ***shape*** of the plan doesn't change much, but it's no longer a filter - the subquery is a **driving** subquery.

Query Blocks (a)

First child first with recursive descent is a good guideline for a **single query block**.
Many queries (like the filter with subquery) start with multiple query blocks

```
Select
        /*+
                no_query_transformation
                qb_name(main)
        */
        *
from    t1
where
        (id1) in (
                select  /*+ qb_name(subq_in) */ x1 from t21
        )
and     (id1, id2, n1) not in (
                select  /*+ qb_name(subq_not)*/ x1, x2, x3 from t23
        )
;
```

Query Blocks (b)

```
select * from table(dbms_xplan.display(null,null,'alias -predicate'));

| Id  | Operation           | Name | Rows  | Bytes | Cost  |
|   0 | SELECT STATEMENT    |      |     1 |   123 |   367K|
| * 1 |  FILTER             |      |       |       |       |
|   2 |   TABLE ACCESS FULL| T1   |  100K|    11M|   278 |
| * 3 |   TABLE ACCESS FULL| T21  |     1 |    13 |     2 |
| * 4 |   TABLE ACCESS FULL| T23  |     1 |    39 |     2 |


Query Block Name / Object Alias (identified by operation id):
   1 - MAIN
   2 - MAIN     / T1@MAIN
   3 - SUBQ_IN  / T21@SUBQ_IN
   4 - SUBQ_NOT / T23@SUBQ_NOT
```


We can almost see the literal translation of our query into a plan. The query block names are all visible, and each table (RHS) is in the query block (LHS) it started in

Query Blocks (b)

Remove the /*+ no_query_transformation */ hint:

```
| Id  | Operation                    | Name   | Rows  | Bytes | Cost  |
|   0 | SELECT STATEMENT             |        |     1 |   136 |    22 |
|   1 |  NESTED LOOPS                |        |     1 |   136 |    20 |
|   2 |   NESTED LOOPS               |        |     1 |   136 |    20 |
|   3 |    SORT UNIQUE               |        |     1 |    13 |     2 |
|   4 |     TABLE ACCESS FULL        | T21    |     1 |    13 |     2 |
|*  5 |    INDEX RANGE SCAN          | T1_I2  |     1 |       |     1 |
|*  6 |     TABLE ACCESS FULL        | T23    |     1 |    39 |     2 |
|   7 |   TABLE ACCESS BY INDEX ROWID| T1     |     1 |   123 |     2 |

Query Block Name / Object Alias (identified by operation id):
   1 - SEL$94CC97E7
   4 - SEL$94CC97E7 / T21@SUBQ_IN
   5 - SEL$94CC97E7 / T1@MAIN
   6 - SUBQ_NOT     / T23@SUBQ_NOT
   7 - SEL$94CC97E7 / T1@MAIN
```


Query blocks main and subq_in have disappeared - transformed into a query block called ***sel$94cc97e7***. Notice how operation 6 is a "pushed" filter subquery.

Query Blocks (c)

Add (just) the /*+ unnest */ hint to subquery "subq_not":

```
| Id | Operation                       | Name   | Rows  | Bytes | Cost  |
|  0 | SELECT STATEMENT                |        |     1 |   175 |    53 |
|  1 |  MERGE JOIN ANTI NA             |        |     1 |   175 |    53 |
|  2 |   SORT JOIN                     |        |     1 |   136 |    35 |
|  3 |    NESTED LOOPS                 |        |     1 |   136 |    20 |
|  4 |     NESTED LOOPS                |        |     1 |   136 |    20 |
|  5 |      SORT UNIQUE                |        |     1 |    13 |     2 |
|  6 |       TABLE ACCESS FULL         | T21    |     1 |    13 |     2 |
|* 7 |      INDEX RANGE SCAN           | T1_I2  |     1 |       |     1 |
|  8 |     TABLE ACCESS BY INDEX ROWID| T1     |     1 |   123 |     2 |
|* 9 |   SORT UNIQUE                   |        |     1 |    39 |    18 |
| 10 |    TABLE ACCESS FULL            | T23    |     1 |    39 |     2 |

Query Block Name / Object Alias (identified by operation id):
   1 - SEL$17E058DA
   6 - SEL$17E058DA / T21@SUBQ_IN
   7 - SEL$17E058DA / T1@MAIN
   8 - SEL$17E058DA / T1@MAIN
  10 - SEL$17E058DA / T23@SUBQ_NOT
```


Forcing the optimizer to unnest the ***subq_not*** subquery we end up with a single query block - with a name derived from the three original names.

Query Blocks (d)

Add /*+ no_unnest */ to both subqueries

```
| Id  | Operation                       | Name   | Rows  | Bytes | Cost  |
|   0 | SELECT STATEMENT                |        |     1 |   136 |    22 |
|   1 |  NESTED LOOPS                   |        |     1 |   136 |    20 |
|   2 |   NESTED LOOPS                  |        |     1 |   136 |    20 |
|   3 |    SORT UNIQUE                  |        |     1 |    13 |     2 |
|   4 |     TABLE ACCESS FULL           | T21    |     1 |    13 |     2 |
|*  5 |    INDEX RANGE SCAN             | T1_I2  |     1 |       |     1 |
|*  6 |     TABLE ACCESS FULL           | T23    |     1 |    39 |     2 |
|   7 |   TABLE ACCESS BY INDEX ROWID| T1     |     1 |   123 |     2 |

Query Block Name / Object Alias (identified by operation id):
   1 - SEL$94CC97E7
   4 - SEL$94CC97E7 / T21@SUBQ_IN
   5 - SEL$94CC97E7 / T1@MAIN
   6 - SUBQ_NOT     / T23@SUBQ_NOT
   7 - SEL$94CC97E7 / T1@MAIN
```


Query blocks main and subq_in have disappeared - transformed into a query block called sel$94cc97e7. Notice how operation 6 is a "pushed" filter subquery.

Query Blocks (e)

ANSI isn't friendly!

```
select
        t1.object_name, t2.object_type, t3.owner
from
        t1
join
        t2
on      t2.object_id = t1.object_id
join
        t3
on      t3.data_object_id = t2.data_object_id
/
```


This looks like a simple three table join.
How many query blocks do you think are involved ?

Query Blocks (f)

The optimizer has a series of generic transformations to transform "ANSI" SQL into legacy Oracle SQL before optimising. The result is that a simple join of N tables turns into a starting N-1 query blocks:

```
| Id  | Operation            | Name | Rows  | Bytes | Cost  |
|   0 | SELECT STATEMENT     |      |  6178 |   331K|   683 |
|*  1 |  HASH JOIN           |      |  6178 |   331K|   683 |
|*  2 |   TABLE ACCESS FULL  | T3   |  6141 | 49128 |   223 |
|*  3 |   HASH JOIN          |      |  6141 |   281K|   457 |
|*  4 |    TABLE ACCESS FULL | T2   |  6141 | 98256 |   223 |
|   5 |    TABLE ACCESS FULL | T1   | 84495 |  2557K|   223 |

Query Block Name / Object Alias (identified by operation id):
   1 - SEL$9E43CB6E
   2 - SEL$9E43CB6E / T3@SEL$2
   4 - SEL$9E43CB6E / T2@SEL$1
   5 - SEL$9E43CB6E / T1@SEL$1
```


How do you hint a query block when you don't even know what query block it was originally in ? (Unless you've looked through the alias information (and outline)).

Multiple Query Blocks (a)

- Subqueries in the from clause
- Scalar subqueries in updates
- "with" subqueries (common table expressions - CTEs)
- Scalar subqueries in the select list

MQB - updates (a)

```
update  t1 target
set     data_object_id = (
                select  max(t2.data_object_id)
                from    t2
                where   t2.object_name = target.object_name
        ),
        owner = (
                select  max(t3.owner)
                from    t3
                where   t3.object_type = target.object_type
        )
where
        object_id = (
                select  max(source.object_id)
                from    t1  source
                where   source.owner = target.owner
        )
;
```


We have an update that has to identify some rows, and then uses scalar subqueries to find values to update two separate columns in the table.

MQB - updates (b)

```
| Id  | Operation               | Name    | Rows   | Bytes | Cost  (%CPU)|
|   0 | UPDATE STATEMENT        |         |     25 |  3175 | 20832    (1)|
|   1 |  UPDATE                 | T1      |        |       |             |

|*  2 |   HASH JOIN             |         |     25 |  3175 |   807    (2)|
|   3 |    VIEW                 | VW_SQ_1 |     25 |  1975 |   407    (3)|
|   4 |     SORT GROUP BY       |         |     25 |   275 |   407    (3)|
|   5 |      TABLE ACCESS FULL| T1      | 84495 |   907K|   400    (1)|
|   6 |    TABLE ACCESS FULL   | T1      | 84495 |  3960K|   400    (1)|

|   7 |   SORT AGGREGATE        |         |      1 |    28 |             |
|*  8 |    TABLE ACCESS FULL   | T2      |      2 |    56 |   400    (1)|

|   9 |   SORT AGGREGATE        |         |      1 |    15 |             |
|* 10 |    TABLE ACCESS FULL   | T3      |   2914 | 43710 |   400    (1)|
```

NB: the cost of an update without the set subqueries is derived as the cost of the statement `select target.rowid from …`

The update operation has three direct children. The first child identifies the rows to update, the 2nd and subequent children show the plans for the "set" subqueries.

MQB - "with" subquery (a)

```
with objects as (
        select object_type, object_name, owner, object_id
        from   all_objects            -- a local table copy of the view
),
object_types as (
        select distinct owner, object_type
        from   objects
),
owners as (
        select distinct owner
        from   object_types
)
select  ot.owner, count(*)
from    object_types ot
where   ot.owner = (select max(ow.owner) from owners ow)
group by ot.owner
;
```


I have a cascade of CTEs here - and Oracle can decide which ones are worth turning into "temporary tables".

MQB - "with" subquery (b)

```
| Id  | Operation                   | Name                       | Rows  | Bytes |
|   0 | SELECT STATEMENT            |                            |    25 |   150 |
|   1 |  TEMP TABLE TRANSFORMATION  |                            |       |       |
|   2 |   LOAD AS SELECT            | SYS_TEMP_0FD9D6646_D6D4524 |       |       |
|   3 |    HASH UNIQUE              |                            |   513 |  7695 |
|   4 |     TABLE ACCESS FULL       | ALL_OBJECTSs                | 84498 |  1237K|
|   5 |   HASH GROUP BY             |                            |    25 |   150 |
|*  6 |    VIEW                     |                            |   513 |  3078 |
|   7 |     TABLE ACCESS FULL       | SYS_TEMP_0FD9D6646_D6D4524 |   513 |  7695 |
|   8 |     SORT AGGREGATE          |                            |     1 |    66 |
|   9 |      VIEW                   |                            |   513 | 33858 |
|  10 |       TABLE ACCESS FULL     | SYS_TEMP_0FD9D6646_D6D4524 |   513 |  7695 |

Predicate Information (identified by operation id):
   6 - filter("OT"."OWNER"= (SELECT MAX("OWNER") FROM  (SELECT
              /*+ CACHE_TEMP_TABLE ("T1") */ "C0" "OWNER","C1" "OBJECT_TYPE"
              FROM "SYS"."SYS_TEMP_0FD9D6646_D6D4524" "T1") "OBJECT_TYPES"))
```

Loading the temporary table and running the query are both child rows to "temp table transformation"
Note the "pushed" subquery effect at operation 8.

The optimizer has decided it can inline and merge the ***objects*** declaration then create and use a temporary table ***object_types***, deriving ***owners*** as an inline view.

MQB - "with" subquery (c)

```
with objects as (
        select /*+ materialize */
               object_type, object_name, owner, object_id
        from   all_objects            -- a local table copy of the view
),
object_types as (
        select /*+ materialize */ distinct owner, object_type
        from   objects
),
owners as (
        select /*+ materialize */ distinct owner
        from   object_types
)
select  ot.owner, count(*)
from    object_types ot
where   ot.owner = (select max(ow.owner) from owners ow)
group by ot.owner
;
```


There are two hints to control "with" subqueries. "Materialize" tells Oracle to create a temporary table, "Inline" tells Oracle not to.

MQB - "with" subquery (d)

```
| Id  | Operation                    | Name                         | Rows  | Bytes |
|   0 | SELECT STATEMENT             |                              |    25 |   150 |
|   1 |  TEMP TABLE TRANSFORMATION   |                              |       |       |
|   2 |   LOAD AS SELECT             | SYS_TEMP_0FD9D6649_D6D4524   |       |       |
|   3 |    TABLE ACCESS FULL         | ALL_OBJECTS                  | 84498 |  3795K|
|   4 |   LOAD AS SELECT             | SYS_TEMP_0FD9D664A_D6D4524   |       |       |
|   5 |    HASH UNIQUE               |                              |   513 |  7695 |
|   6 |     VIEW                     |                              | 84498 |  1237K|
|   7 |      TABLE ACCESS FULL       | SYS_TEMP_0FD9D6649_D6D4524   | 84498 |  3795K|
|   8 |   LOAD AS SELECT             | SYS_TEMP_0FD9D664B_D6D4524   |       |       |
|   9 |    HASH UNIQUE               |                              |    25 |   150 |
|  10 |     VIEW                     |                              |   513 |  3078 |
|  11 |      TABLE ACCESS FULL       | SYS_TEMP_0FD9D664A_D6D4524   |   513 |  7695 |
|  12 |   HASH GROUP BY              |                              |    25 |   150 |
|* 13 |    VIEW                      |                              |   513 |  3078 |
|  14 |     TABLE ACCESS FULL        | SYS_TEMP_0FD9D664A_D6D4524   |   513 |  7695 |
|  15 |     SORT AGGREGATE           |                              |     1 |    66 |
|  16 |      VIEW                    |                              |    25 |  1650 |
|  17 |       TABLE ACCESS FULL      | SYS_TEMP_0FD9D664B_D6D4524   |    25 |   150 |

    13 - filter("OT"."OWNER"= (SELECT MAX("OW"."OWNER") FROM  (
         SELECT /*+ CACHE_TEMP_TABLE ("T1") */ "C0" "OWNER"
         FROM "SYS"."SYS_TEMP_0FD9D664B_D6D4524" "T1") "OW"))
```


We've loaded three temporary tables - deriving each one from the previous generated temporary table. Then used the last two to execute the query.

MQB - select list (a)

```
select
        ss1.location,
        ss1.sales,
        (       select
                        /*+ index (ss2 ss2_fk_area) */
                        sum(sales)
                from
                        ss_test_2 ss2
                where
                        ss2.area = ss1.area
                and     ss2.location_type = ss1.location_type
        )       area_sales
from
        ss_test ss1
where
        ss1.location_type not in ('Type_001','Type_002')
and     ss1.location_type like 'Type_00%'
;
```


One way of getting correlated summaries in a query is simply to execute a correlated subquery for each row.

MQB - select list (b)

```
| Id  | Operation                     | Name         | Rows  | Bytes | Cost  |
|   0 | SELECT STATEMENT              |              |  8574 |  293K|    11 |
|   1 |  SORT AGGREGATE               |              |     1 |   24 |       |
|*  2 |   TABLE ACCESS BY INDEX ROWID| SS_TEST_2    |    10 |  240 |    58 |
|*  3 |    INDEX RANGE SCAN           | SS2_FK_AREA |   200 |      |     2 |
|*  4 |  TABLE ACCESS FULL            | SS_TEST      |  8574 |  293K|    11 |

Predicate Information (identified by operation id):
   2 - filter("SS2"."LOCATION_TYPE"=:B1)
   3 - access("SS2"."AREA"=:B1)
   4 - filter("SS1"."LOCATION_TYPE"<>'Type_001' AND
              "SS1"."LOCATION_TYPE"<>'Type_002' AND
              "SS1"."LOCATION_TYPE" LIKE 'Type_00%')
```

The last child in the plan is the main query and is the first thing to operate. The previous child operations are then nominally executed once for each row in the last child.

Note how the final cost is clearly wrong - it doesn't reflect the cost of executing the scalar subquery at all (let alone 8,574 times). In 12c the cost changes to 55,111

MQB - select list (c)

```
| Id  | Operation               | Name        | Rows  | Bytes | Cost  |
|   0 | SELECT STATEMENT        |             |  8574 |   510K|    62 |
|*  1 |  HASH JOIN RIGHT OUTER|             |  8574 |   510K|    62 |
|   2 |   VIEW                  | VW_SSQ_1    |   672 | 17472 |    50 |
|   3 |    HASH GROUP BY        |             |   672 | 16128 |    50 |
|*  4 |     TABLE ACCESS FULL | SS_TEST_2 |  9500 |   222K|    11 |
|*  5 |   TABLE ACCESS FULL     | SS_TEST     |  8574 |   293K|    11 |

Predicate Information (identified by operation id):
   1 - access("ITEM_1"(+)="SS1"."AREA" AND
              "ITEM_2"(+)="SS1"."LOCATION_TYPE")
   4 - filter("SS2"."LOCATION_TYPE"<>'Type_001' AND
              "SS2"."LOCATION_TYPE"<>'Type_002' AND
              "SS2"."LOCATION_TYPE" LIKE 'Type_00%')
   5 - filter("SS1"."LOCATION_TYPE"<>'Type_001' AND
              "SS1"."LOCATION_TYPE"<>'Type_002' AND
              "SS1"."LOCATION_TYPE" LIKE 'Type_00%')
```


12c can unnest the scalar subquery and transform the query into an outer join (outer to allow for the scalar subquery returning no rows). The unnest hint will block this.

MQB - select list (d)

```
select  grp, id
        case
                when grp = 4 then
                        to_char((
                                select count(*)
                                from   pt_range pt2
                                where  id = to_number(pt1.small_vc)
                                ),'9999'
                        )
                when grp = 5 then
                        to_char((
                                select  count(*)
                                from    pt_range pt2
                                where   id = 10*to_number(pt1.small_vc)
                                ),'XXXX'
                        )
                else null
        end     as test,
from    pt_range        pt1
where   id between 200 and 300
```


There are still some cases - even in 18.3 where the indentation you get from dbms_xplan is wrong. The two scalar subqueries are clearly at the same "depth".

MQB - select list (e)

```
| Id  | Operation                    | Name     | Rows  | Bytes | Pstart| Pstop |
|   0 | SELECT STATEMENT             |          |   102 |  1020 |       |       |
|   1 |  SORT AGGREGATE              |          |     1 |     4 |       |       |
|   2 |   PARTITION RANGE SINGLE     |          |     1 |     4 |   KEY |   KEY |
|*  3 |    INDEX UNIQUE SCAN         | PT_PK    |     1 |     4 |   KEY |   KEY |
|   4 |     SORT AGGREGATE           |          |     1 |     4 |       |       |
|   5 |      PARTITION RANGE SINGLE|          |     1 |     4 |   KEY |   KEY |
|*  6 |       INDEX UNIQUE SCAN      | PT_PK    |     1 |     4 |   KEY |   KEY |
|   7 |  PARTITION RANGE SINGLE      |          |   102 |  1020 |     2 |     2 |
|*  8 |   TABLE ACCESS FULL          | PT_RANGE |   102 |  1020 |     2 |     2 |

Query Block Name / Object Alias (identified by operation id):
   1 - SEL$2
   3 - SEL$2 / PT2@SEL$2
   4 - SEL$3
   6 - SEL$3 / PT2@SEL$3
   7 - SEL$1
   8 - SEL$1 / PT1@SEL$1

Predicate Information (identified by operation id):
   3 - access("ID"=TO_NUMBER(:B1))
   6 - access("ID"=10*TO_NUMBER(:B1))
   8 - filter("ID"<=300)
```


According to the plan the second scalar subquery is somehow sub-ordinate to / called before the first one. Note the KEY/KEY partition information.

How often (a)

```
update t1
set n2  = (
                select
                        /*+ no_unnest */
                        t2.n1
                from    t2
                where   t2.id = t1.n1
        )
where   exists (
                select
                        /*+ no_unnest */
                        t2.n1
                from    t2
                where   t2.id = t1.n1
        )
;
```


We set ***statistics_level = all*** and ***set serveroutput off*** before running this query.
The two scalar subqueries are identical, and we're about to update 988 rows of 1,000.

How often (b)

```
select * from table(dbms_xplan.display_cursor(null,null,'allstats last'));

| Id | Operation            | Name | Starts | E-Rows | A-Rows |   A-Time   | Buffers |
|  0 | UPDATE STATEMENT     |      |      1 |        |      0 |00:00:00.06|    2585 |
|  1 |  UPDATE              | T1   |      1 |        |      0 |00:00:00.06|    2585 |
|* 2 |   FILTER             |      |      1 |        |    988 |00:00:00.01|     315 |
|  3 |    TABLE ACCESS FULL| T1   |      1 |   1000 |   1000 |00:00:00.01|      19 |
|* 4 |    TABLE ACCESS FULL| T2   |    148 |      1 |    147 |00:00:00.01|     296 |
|* 5 |   TABLE ACCESS FULL | T2   |    147 |      1 |    147 |00:00:00.01|     294 |

Predicate Information (identified by operation id):
   2 - filter( IS NOT NULL)               -- missing subquery predicates
   4 - filter("T2"."ID"=:B1)              -- bind variables for correlated values
   5 - filter("T2"."ID"=:B1)
```


We don't execute the "where" and "update" subqueries 1,000 and 988 times respectively because of scalar subquery caching - but should we ever run the 2nd ?

What else happened ?

- SQL Monitor
- Case Study - in very small print.