Oracle SQL Plan Execution: How it really works

Tanel Põder

http://www.tanelpoder.com

ORACLE | CERTIFIED MASTER

What is an execution plan?

For Oracle server:

- *Parsed*, optimized and compiled SQL code kept inside library cache

For DBAs and developers:

- Text or graphical representation of SQL execution flow

Often known as *explain plan*

- To be correct in terms, explain plan is just a tool, command in Oracle
- Explain plan outputs textual representation of execution plan into plan table
- DBAs/developers report human readable output from plan table

One slide for getting execution plan

Starting from 9.2 the usual way is:

- explain plan for <statement>
- select * from table(dbms_xplan.display)

In 10g

- the *autotrace* also uses dbms_xplan
- set autotrace on
- or select * from table(dbms_xplan.display_cursor())

In 11g

- DBMS_SQLTUNE.REPORT_SQL_MONITOR

Other methods

- sql_trace / 10046 trace + tkprof utility
- v$sql_plan
- setting event 10132 at level 1
- 3rd party tools (which use explain plan anyway)

Explain plan for has problems:

1) It treats all bind variables as VARCHAR2
2) It might not show you the real exec plan used!

Use V$SQL_PLAN_STATISTICS / dbms_xplan.display_cursor instead!

Avoid "explain plan for" approach if possible!!!

Parse stages

Syntactic check
- Syntax, keywords, sanity

Semantic check
- Whether objects referenced exist, are accessible (by permissions) and are usable

soft parse

View merging
- Queries are written to reference base tables
- Can merge both stored views and inline views

Query transformation
- Transitivity, etc (example: if a=1 and a=b then b=1)

Optimization

Query execution plan (QEP) generation

Loading SQL and execution plan in library cache

hard parse

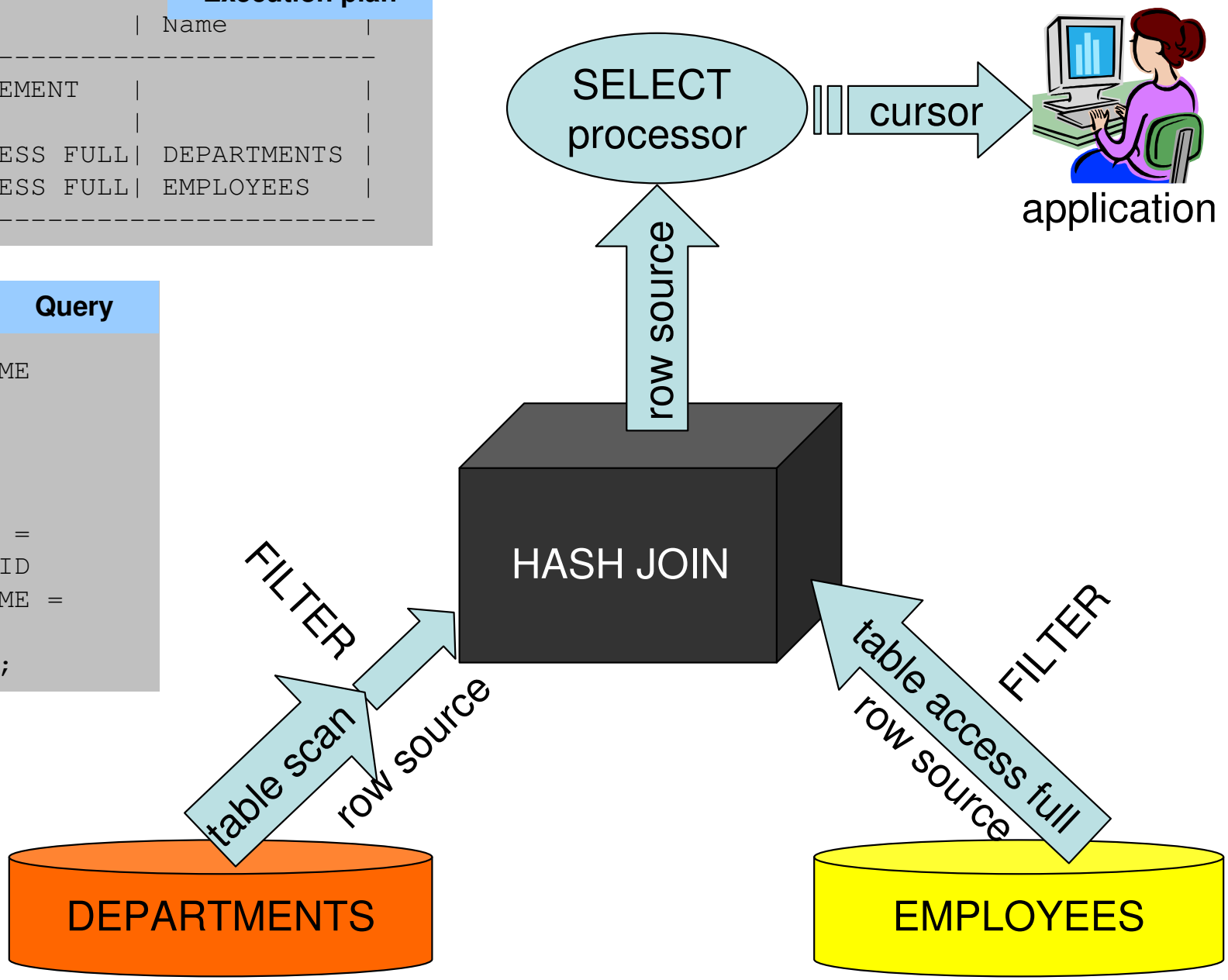
SQL execution basics

Execution plan

```
-----------------------------------------
| Id  | Operation          | Name       |
-----------------------------------------
|   0 | SELECT STATEMENT   |            |
|*  1 |  HASH JOIN         |            |
|*  2 |   TABLE ACCESS FULL| DEPARTMENTS |
|*  3 |   TABLE ACCESS FULL| EMPLOYEES  |
-----------------------------------------
```

Query

```
SELECT
    E.LAST_NAME,
    D.DEPARTMENT_NAME
FROM
    EMPLOYEES E,
    DEPARTMENTS D
WHERE
    E.DEPARTMENT_ID =
      D.DEPARTMENT_ID
AND D.DEPARTMENT_NAME =
      'Sales'
AND E.SALARY > 2000;
```

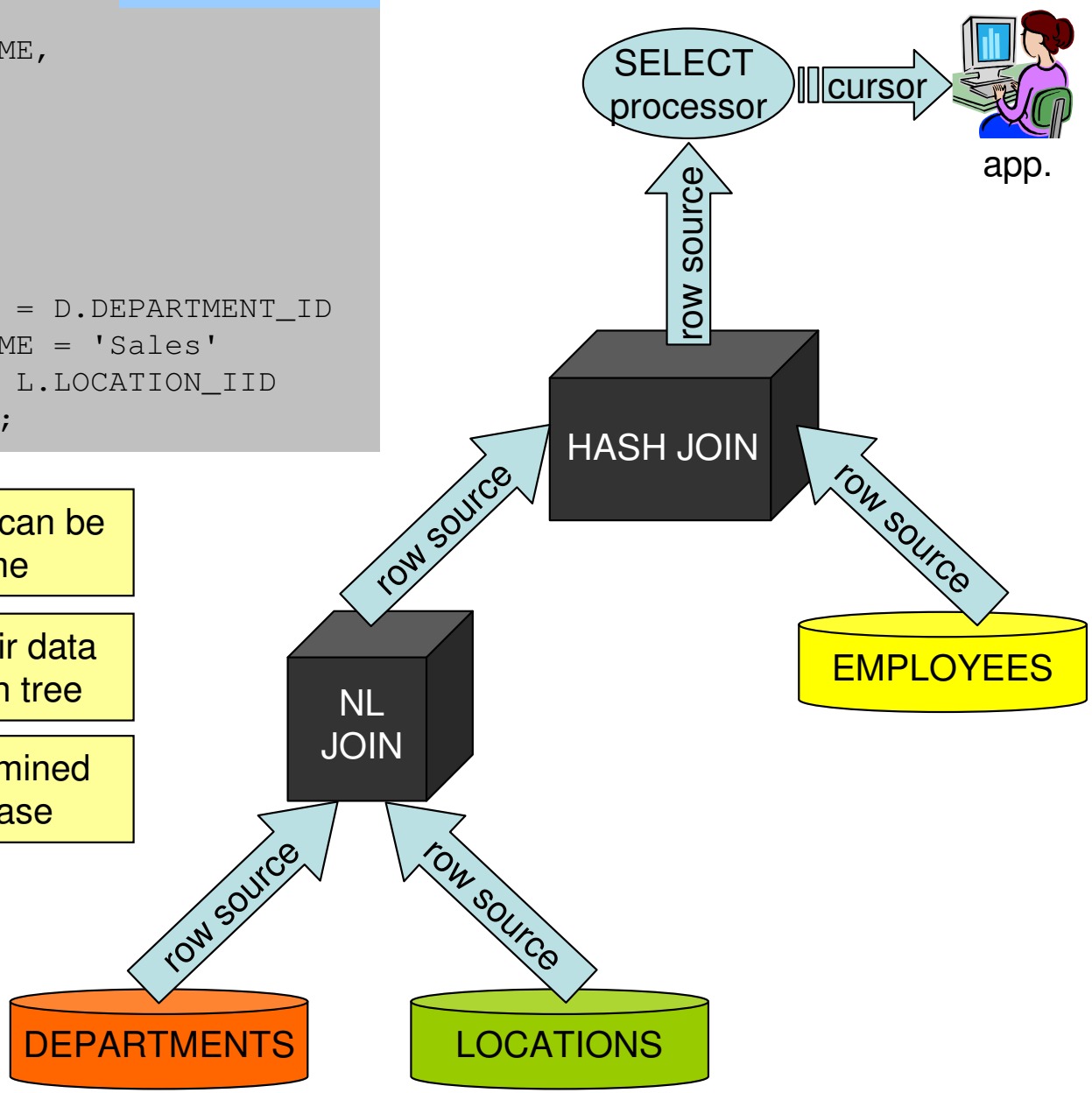


SQL execution basics - multitable joins

Multiple joins

```
SELECT
    E.LAST_NAME,
    D.DEPARTMENT_NAME,
    L.CITY
FROM
    EMPLOYEES E,
    DEPARTMENTS D,
    LOCATIONS L
WHERE
    E.DEPARTMENT_ID = D.DEPARTMENT_ID
AND D.DEPARTMENT_NAME = 'Sales'
AND D.LOCATION_ID = L.LOCATION_IID
AND E.SALARY > 2000;
```

- Only two row sources can be joined together at a time
- Row sources pass their data "up" the execution plan tree
- The join order is determined during optimization phase



SQL execution terminology

ACCESS PATH

- A means to access physical data in database storage
- From tables, indexes, external tables, database links

ROW SOURCE

- A virtual stream of rows
- Can come through access paths from tables, indexes
- Or from other child row sources

FILTER *PREDICATE*

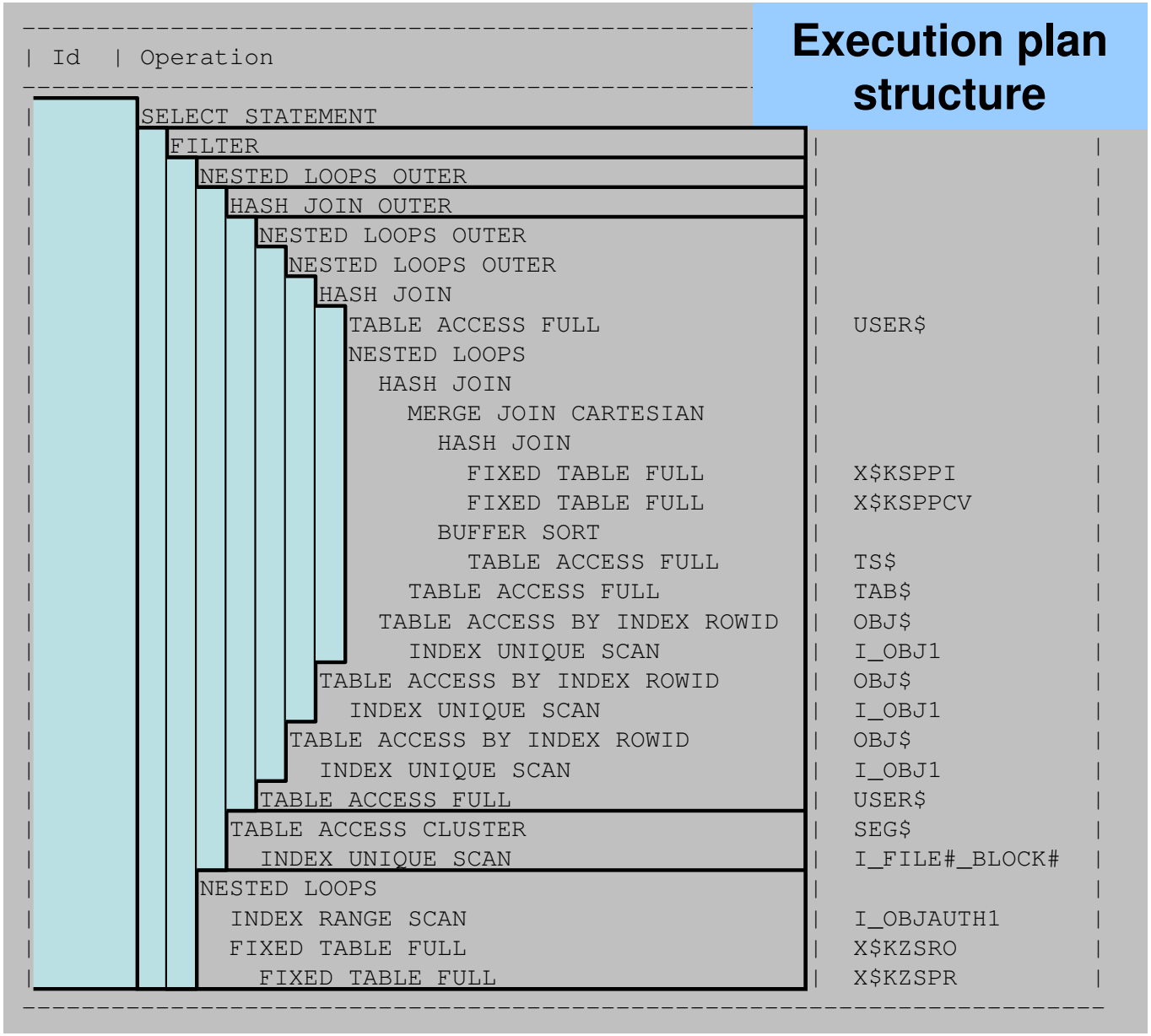
- A property of row source - can discard rows based on defined conditions - *filter predicates*

JOIN

- Filters and merges rows based on matching rows from child rowsources. Matching is defined by *join predicates*
- Any join operator can join only two inputs

First rule for reading an execution plan

Parent operations get input only from their children



Second rule for reading an execution plan

Data access starts from the first line without children

Execution plan structure

```
-----------------------------------------------------------------------------
| Id  | Operation                                | Name                     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT                         |                          |
|*  1 |   FILTER                                 |                          |
|   2 |     NESTED LOOPS OUTER                   |                          |
|*  3 |       HASH JOIN OUTER                    |                          |
|   4 |         NESTED LOOPS OUTER               |                          |
|   5 |           NESTED LOOPS OUTER             |                          |
|*  6 |             HASH JOIN                    |                          |
|   7 |               TABLE ACCESS FULL          | USER$                    |
|   8 |               NESTED LOOPS               |                          |
|*  9 |                 HASH JOIN                |                          |
|  10 |                   MERGE JOIN CARTESIAN   |                          |
|* 11 |                     HASH JOIN            |                          |
|* 12 |                       FIXED TABLE FULL   | X$KSPPI                  |
|  13 |                       FIXED TABLE FULL   | X$KSPPCV                 |
|  14 |                     BUFFER SORT          |                          |
|  15 |                       TABLE ACCESS FULL  | TS$                      |
|* 16 |                   TABLE ACCESS FULL      | TAB$                     |
|* 17 |                 TABLE ACCESS BY INDEX ROWID | OBJ$                     |
|* 18 |                   INDEX UNIQUE SCAN      | I_OBJ1                   |
|  19 |               TABLE ACCESS BY INDEX ROWID | OBJ$                     |
|* 20 |                 INDEX UNIQUE SCAN        | I_OBJ1                   |
|  21 |             TABLE ACCESS BY INDEX ROWID  | OBJ$                     |
|* 22 |               INDEX UNIQUE SCAN          | I_OBJ1                   |
|  23 |           TABLE ACCESS FULL              | USER$                    |
|  24 |         TABLE ACCESS CLUSTER             | SEG$                     |
|* 25 |           INDEX UNIQUE SCAN              | I_FILE#_BLOCK#           |
|  26 |       NESTED LOOPS                       |                          |
|* 27 |         INDEX RANGE SCAN                 | I_OBJAUTH1               |
|* 28 |         FIXED TABLE FULL                 | X$KZSRO                  |
|* 29 |           FIXED TABLE FULL               | X$KZSPR                  |
-----------------------------------------------------------------------------
```

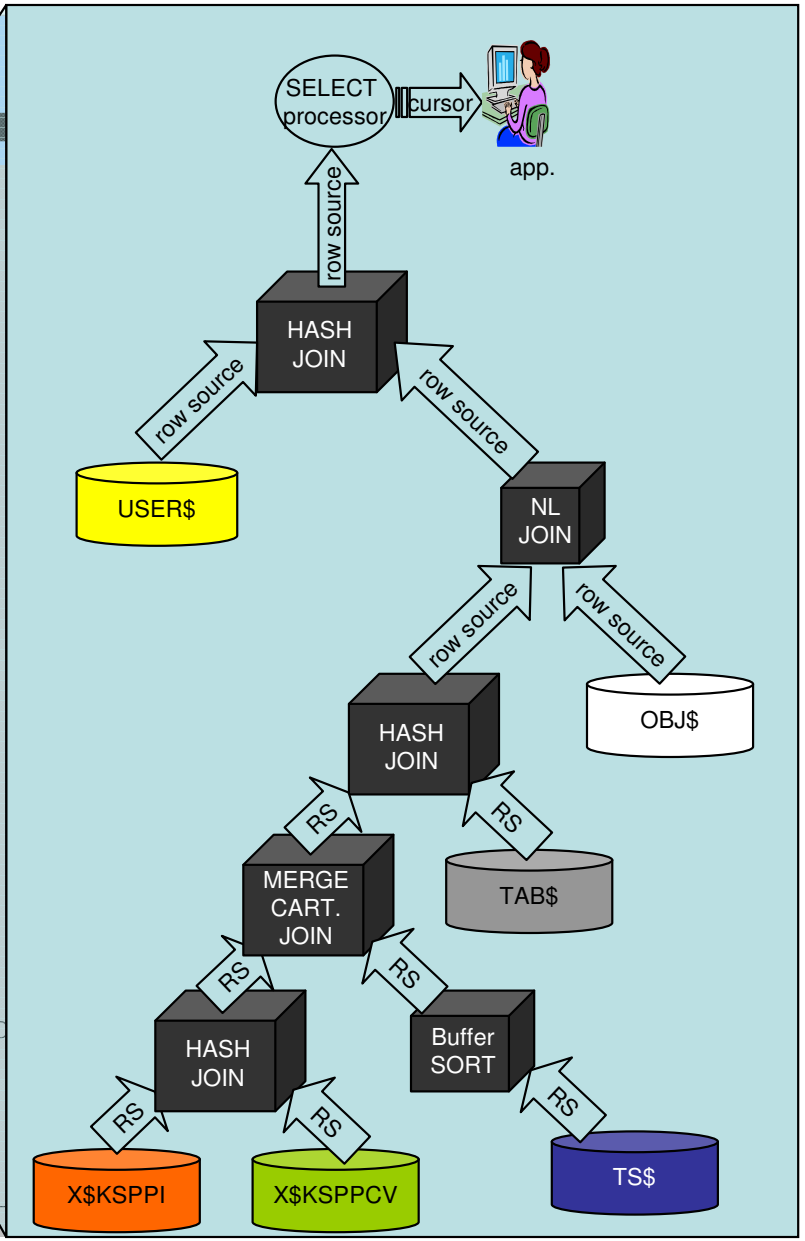
First operation with no children (leaf operation) **accesses** data

Cascading rowsources

Data access starts from the first line without children

```
-----------------------------------------------------
| Id  | Operation                                   |
-----------------------------------------------------
|   0 | SELECT STATEMENT                            |
|*  1 |   FILTER                                    |
|   2 |     NESTED LOOPS OUTER                      |
|*  3 |       HASH JOIN OUTER                       |
|   4 |         NESTED LOOPS OUTER                  |
|   5 |           NESTED LOOPS OUTER                |
|*  6 |             HASH JOIN                       |
|   7 |               TABLE ACCESS FULL             |
|   8 |               NESTED LOOPS                  |
|*  9 |                 HASH JOIN                   |
|  10 |                   MERGE JOIN CARTESIAN      |
|* 11 |                     HASH JOIN               |
|* 12 |                       FIXED TABLE FULL      |
|  13 |                       FIXED TABLE FULL      |
|  14 |                     BUFFER SORT             |
|  15 |                       TABLE ACCESS FULL     |
|* 16 |                     TABLE ACCESS FULL       |
|* 17 |                   TABLE ACCESS BY INDEX ROWID |
|* 18 |                     INDEX UNIQUE SCAN       |
|  19 |             TABLE ACCESS BY INDEX ROWID     |
|* 20 |               INDEX UNIQUE SCAN             |
|  21 |           TABLE ACCESS BY INDEX ROWID       |
|* 22 |             INDEX UNIQUE SCAN               |
|  23 |         TABLE ACCESS FULL                   |
|  24 |       TABLE ACCESS CLUSTER                  |
|* 25 |         INDEX UNIQUE SCAN                   |
|  26 |     NESTED LOOPS                            |
|* 27 |       INDEX RANGE SCAN                      |
|* 28 |       FIXED TABLE FULL                      |
|* 29 |         FIXED TABLE FULL                    |
-----------------------------------------------------
```

Execution structu



SQL execution plan recap

Execution plan lines are just Oracle kernel functions!

- In other words, each row source is a function

Data can only be accessed using *access path functions*

- Only access paths can access physical data
- Access paths process physical data, return *row sources*

Data processing starts from first line without children

- In other words the first leaf access path in execution plan

Row sources feed data to their parents

- Can be non-cascading, semi-cascading or cascading

A join operation can input only two row sources

- However, it is possible to combine result of more than 2 row sources for some operations (not for joins though)
- Index combine, bitmap merging, filter, union all, for example

Troubleshooting: Reading DBMS_XPLAN execution plan profile

```
SQL> select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------
SQL_ID  56bs32ukywdsq, child number 0
-------------------------------------
select count(*) from dba_tables
Plan hash value: 736297560
-------------------------------------------------------------------------------------
| Id  | Operation                     | Name     | Starts | E-Rows | A-Rows |   A-Time   |
-------------------------------------------------------------------------------------
|   1 |  SORT AGGREGATE               |          |      1 |      1 |      1 |00:00:00.38 |
|*  2 |   HASH JOIN RIGHT OUTER       |          |      1 |   1690 |   1688 |00:00:00.37 |
|   3 |    TABLE ACCESS FULL          | USER$    |      1 |     68 |     68 |00:00:00.01 |
|*  4 |    HASH JOIN OUTER            |          |      1 |   1690 |   1688 |00:00:00.37 |
|*  5 |     HASH JOIN
|   6 |      TABLE ACCESS FU
|*  7 |      HASH JOIN
|   8 |       NESTED LOOPS O
|*  9 |        HASH JOIN RIG
|  10 |         TABLE ACCESS
|* 11 |         HASH JOIN
|  12 |          MERGE JOIN
|* 13 |           HASH JOIN
|* 14 |            FIXED TAB
|  15 |            FIXED TAB
|  16 |           BUFFER SOR
|  17 |            TABLE ACC
|* 18 |          TABLE ACCES
|* 19 |        INDEX UNIQUE
|* 20 |       TABLE ACCESS F
|  21 |     TABLE ACCESS FUL
-------------------------------------------------------------------------------------
```

Starts	number of times the rowsource was initialized
E-rows	CBO number estimated rows coming from rowsource
A-rows	actual *measured* number of rows during last execution
A-time	actual *measured (and extrapolated)* time spent inside a rowsource function or under its children (cumulative)
Buffer	number of buffer gets done within rowsource during last execution

Troubleshooting: Reading XMS/XMSH execution plan profile

```
SQL> @xms

SQL hash value:          2783852310    Cursor address:                00000003DCA9EF28    |   Statement firs

 Ch Pr    Op                                          Object         ms spent   Estimated Real #rows   Op. ite-
 ld ed    ID Operation                                Name             in op. output rows   returned    rations
--- -- ---- ------------------------------------------ ---------- ---------- ----------- ---------- ----------
  0        0 SELECT STATEMENT
           1  SORT AGGRE
    A      2   HASH JOIN
           3    TABLE AC
    A      4    HASH JOI
    A      5     HASH JO
           6      TABLE
    A      7      HASH J
           8       NESTE
    A      9        HASH
          10         TAB
    A     11         HAS
          12          ME
    A     13           H
     F    14
          15
          16           B
          17
     F    18          TA
    A     19         INDE
     F    20        TABLE
          21       TABLE A

 Ch Op
 ld ID     Predicate In
--- ------ -----------
  0      2  - access("CX
         4  - access("T"
         5  - access("O"
         7  - access("O"
         9  - access("T"
```

ms spent in op.	milliseconds spent in rowsource function (cumulative)
Estimated rows	CBO rowcount estimate
Real # rows	Real *measured* rowcount from rowsource
Op. iterations	Number of times the rowsource fetch was initialized
Logical reads	Consistent buffer gets
Logical writes	Current mode buffer gets (Note that some CUR gets may not always be due writing...)
Physical reads	Physial reads done by the rowsource function
Physical writes	Physical writes done by the rowsource function
Optimizer cost	Least significant thing for measuring the *real execution efficiency* of a statement

Advanced Troubleshooting - Reading process stack

```
$ pstack 5855
#0  0x00c29402 in __kernel_vsyscall ()
#1  0x005509e4 in semtimedop () from /lib/libc.so.6
#2  0x0e5769b7 in sskgpwwait ()
#3  0x0e575946 in skgpwwait ()
#4  0x0e2c3adc in ksliwat ()
#5  0x0e2c3449 in kslwaitctx. ()
#6  0x0b007261 in kjusuc ()
#7  0x0c8a7961 in ksipgetctx ()
#8  0x0e2d4dec in ksqcmi ()
#9  0x0e2ce9b8 in ksqgtlctx ()
#10 0x0e2cd214 in ksqgelctx. ()
#11 0x08754afa in ktcwit1 ()
#12 0x0e39b2a8 in kddggb ()
#13 0x08930c80 in kdddel ()
#14 0x0892af0f in kaudel ()
#15 0x08c3d21a in delrow ()
#16 0x08e6ce16 in qerdlFetch ()
#17 0x08c403c5 in delexe ()
#18 0x0e3c3fa9 in opiexe ()
#19 0x08b54500 in kpoal8 ()
#20 0x0e3be673 in opiodr ()
#21 0x0e53628a in ttcpip ()
#22 0x089a87ab in opitsk ()
#23 0x089aaa00 in opiino ()
#24 0x0e3be673 in opiodr ()
#25 0x089a4e76 in opidrv ()
#26 0x08c1626f in sou2o ()
#27 0x08539aeb in opimai_real ()
#28 0x08c19a42 in ssthrdmain ()
#29 0x08539a68 in main ()
```

Where to look up the meaning of Oracle kernel function names?

1) Metalink:

175982.1 ORA-600 Lookup Error Categories

453521.1 ORA-04031 “KSFQ Buffers” ksmlgpalloc

Search: <function> "executable entry point”

2) Oracle views

v$latch_misses (lm.sql)

v$latchholder (latchprofx.sql)

v$fixed_view_definition (d.sql, f.sql)

3) Internet search

Advanced Troubleshooting - Getting stack traces

OS stack dumper

- pstack - Solaris, Linux, HP-UX
- procstack - AIX
- gdb bt, mdb $c
- Procwatcher (Metalink note: 459694.1)

Windows

- windbg, procexp - but no symbolic function names in oracle.exe :(

Oracle internal

- oradebug short_stack
- oradebug dump errorstack
- alter session set events '942 trace name errorstack'

Advanced - Interpreting rowsource functions with os_explain

```
select /*+ ordered use_nl(b) use_nl(c) use_nl(d)
           full(a) full(b) full(c) full(d) */
count(*)
from    sys.obj$ a, sys.obj$ b, sys.obj$ c, sys.obj$ d
where
    a.owner# = b.owner# and b.owner# = c.owner#
and c.owner# = d.owner# and rownum <= 10000000000
```

```
-------------------------------------------
| Id  | Operation                  | Name |
-------------------------------------------
|   1 |  SORT AGGREGATE            |      |
|*  2 |   COUNT STOPKEY            |      |
|   3 |    NESTED LOOPS            |      |
|   4 |     NESTED LOOPS           |      |
|   5 |      NESTED LOOPS          |      |
|   6 |       TABLE ACCESS FULL    | OBJ$ |
|*  7 |       TABLE ACCESS FULL    | OBJ$ |
|*  8 |      TABLE ACCESS FULL     | OBJ$ |
|*  9 |     TABLE ACCESS FULL      | OBJ$ |
-------------------------------------------
```

```
$ pstack 1595 | ./os_explain
   kpoal8
    SELECT FETCH:
     GROUP BY SORT: Fetch
      COUNT: Fetch
       NESTED LOOP JOIN: Fetch
        TABLE ACCESS: Fetch
         kdsttgr
          kdstf0100101km
           expeal
            expepr
```

Child row source function must map directly to a child line in exec plan

Simple full table scan

Full table scan scans all the rows in the table

- All table blocks are scanned up to the HWM
- Even if all rows have been deleted from table
- Oracle uses multiblock reads where it can
- Most efficient way when querying majority of rows
  - And majority of columns

```
SQL> select * from emp;

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 4080710170

------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |    14 |   518 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| EMP  |    14 |   518 |     3   (0)| 00:00:01 |
------------------------------------------------------------------------------
```

Full table scan with a filter predicate

Filter operation throws away non-matching rows

- By definition, not the most efficient operation
- Filter conditions can be seen in predicate section

```
SQL> select * from emp where ename = 'KING';
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 4080710170

------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    37 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMP  |     1 |    37 |     3   (0)| 00:00:01 |
------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("ENAME"='KING')
```

Simple B*tree index+table access

Index tree is walked from root to leaf

- Key values and ROWIDs are gotten from index
- Table rows are gotten using ROWIDs
- *Access* operator fetches only matching rows
  - As opposed to *filter* which filters through the whole child rowsource

```
SQL> select * from emp where empno = 10;

-------------------------------------------------------------------------
| Id  | Operation                    | Name    | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |     1 |    37 |     1   (0)|
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP     |     1 |    37 |     1   (0)|
|*  2 |   INDEX UNIQUE SCAN          | PK_EMP  |     1 |       |     0   (0)|
-------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("EMPNO"=10)
```

Predicate attributes

Predicate = access

- A means to avoid processing (some) unneeded data at all

Predicate = filter

- Everything from child row source is processed / filtered
- The non-matching rows are *thrown away*

```
SQL> select * from emp
  2  where empno > 7000
  3  and ename like 'KING%';
-------------------------------------------------------------------------
| Id  | Operation                    | Name    | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |     1 |    27 |     3   (0)|
|*  1 |  TABLE ACCESS BY INDEX ROWID| EMP     |     1 |    27 |     3   (0)|
|*  2 |   INDEX RANGE SCAN           | PK_EMP  |     9 |       |     2   (0)|
-------------------------------------------------------------------------
Predicate Information (identified by operation id):

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------
   1 - filter("ENAME" LIKE 'KING%')
   2 - access("EMPNO">7000)
```

Index fast full scan

Doesn't necessarily return keys in order

- The whole index segment is just scanned as Oracle finds its blocks on disk (in contrast to tree walking)
- Multiblock reads are used
- As indexes don't usually contain all columns that tables do, FFS is more efficient if all used columns are in index
- Used mainly for aggregate functions, min/avg/sum,etc
- Optimizer must know that all table rows are represented in index! (null values and count example)

```
SQL> select min(empno), max(empno) from emp;

-----------------------------------------------------------------------
| Id  | Operation             | Name    | Rows  | Bytes | Cost (%CPU)|
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT      |         |     1 |     5 |    25   (0)|
|   1 |  SORT AGGREGATE       |         |     1 |     5 |            |
|   2 |   INDEX FAST FULL SCAN| PK_EMP  | 54121 |  264K |    25   (0)|
-----------------------------------------------------------------------
```

Nested Loop Join

Nested loop join

- Read data from outer row source (upper one)
- *Probe* for a match in inner row source for each outer row

```
SQL> select d.dname, d.loc, e.empno, e.ename
  2  from emp e, dept d
  3  where e.deptno = d.deptno
  4  and d.dname = 'SALES'
  5  and e.ename like 'K%';

-----------------------------------------------------------------------
| Id  | Operation                    | Name    | Rows  | Bytes | Cost  |
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT             |         |     1 |    37 |     4 |
|   1 |  NESTED LOOPS                |         |     1 |    37 |     4 |
|*  2 |   TABLE ACCESS FULL          | EMP     |     1 |    17 |     3 |
|*  3 |   TABLE ACCESS BY INDEX ROWID| DEPT    |     1 |    20 |     1 |
|*  4 |    INDEX UNIQUE SCAN         | PK_DEPT |     1 |       |       |
-----------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("E"."DEPTNO" IS NOT NULL AND "E"."ENAME" LIKE 'K%')
   3 - filter("D"."DNAME"='SALES')
   4 - access("E"."DEPTNO"="D"."DEPTNO")
```

Hash Join

Only for equijoins/non-equijoins (outer joins in 10g)

- Builds an array with hashed key values from smaller row source
- Scans the bigger row source, builds and compares hashed key values on the fly, returns matching ones

```
SQL> select d.dname, d.loc, e.empno, e.ename
  2  from emp e, dept d
  3  where e.deptno = d.deptno
  4  and d.dname = 'SALES'
  5  and e.ename between 'A%' and 'M%';
-------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    37 |     9  (12)|
|*  1 |  HASH JOIN         |      |     1 |    37 |     9  (12)|
|*  2 |   TABLE ACCESS FULL| DEPT |     1 |    20 |     2   (0)|
|*  3 |   TABLE ACCESS FULL| EMP  |     4 |    68 |     6   (0)|
-------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("E"."DEPTNO"="D"."DEPTNO")
   2 - filter("D"."DNAME"='SALES')
   3 - filter("E"."DEPTNO" IS NOT NULL AND "E"."ENAME"<='M%'
              AND"E"."ENAME">='A%')
```

Sort-Merge Join

Requires both rowsources to be sorted

- Either by a sort operation
- Or sorted by access path (index range and full scan)

Cannot return any rows before both rowsources are sorted (non-cascading)

NL and Hash join should be normally preferred

```
SQL> select /*+ USE_MERGE(d,e) */ d.dname, d.loc, e.empno, e.ename
  2  from emp e, dept d
  3  where e.deptno = d.deptno
  4  and d.dname = 'SALES'
  5  and e.ename between 'A%' and 'X%'
  6  order by e.deptno;
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)
0	SELECT STATEMENT		1245	46065	64 (10)
1	MERGE JOIN		1245	46065	64 (10)
* 2	**TABLE ACCESS BY INDEX ROWID**	DEPT	1	20	2 (0)
3	**INDEX FULL SCAN**	PK_DEPT	4		1 (0)
* 4	SORT JOIN		3735	63495	62 (10)
* 5	TABLE ACCESS FULL	EMP	3735	63495	61 (9)

View merging

Optimizer merges subqueries, inline and stored views and runs queries directly on base tables

- Not always possible though due semantic reasons

```
SQL> create or replace view empview
  2  as
  3  select e.empno, e.ename, d.dname
  4  from emp e, dept d
  5  where e.deptno = d.deptno;

SQL> select * from empview
  2  where ename = 'KING';
```

Can be controlled using:
Parameter: _complex_view_merging
_simple_view_merging
Hints: MERGE, NO_MERGE

```
-------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |    7 |   210 |     5  (20)|
|*  1 |  HASH JOIN                   |             |    7 |   210 |     5  (20)|
|   2 |   TABLE ACCESS FULL          | DEPT        |    4 |    52 |     2   (0)|
|*  3 |   TABLE ACCESS BY INDEX ROWID| EMP         |    7 |   119 |     2   (0)|
|*  4 |    INDEX RANGE SCAN          | EMP_ENAME   |    8 |       |     1   (0)|
-------------------------------------------------------------------------------
```

Subquery unnesting

Subqueries can be unnested, converted to anti- and semijoins

```
SQL> select * from employees e
  2  where exists (
  3     select ename from bonus b
  4     where e.ename = b.ename
  5  );
```

Can be controlled using:
- Parameter: _unnest_subqueries
- Hints: UNNEST, NO_UNNEST

```
-------------------------------------------------------------------------
| Id  | Operation                     | Name       | Rows  | Bytes | Cost (
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |            |     1 |    37 |     5
|   1 |  NESTED LOOPS                 |            |     1 |    37 |     5
|   2 |   NESTED LOOPS                |            |     1 |    24 |     4
|   3 |    SORT UNIQUE                |            |     1 |     7 |     2
|   4 |     TABLE ACCESS FULL         | BONUS      |     1 |     7 |     2
|*  5 |    TABLE ACCESS BY INDEX ROWID| EMP        |     1 |    17 |     1
|*  6 |     INDEX RANGE SCAN          | EMP_ENAME  |    37 |       |     1
|   7 |   TABLE ACCESS BY INDEX ROWID | DEPT       |     1 |    13 |     1
|*  8 |    INDEX UNIQUE SCAN          | PK_DEPT    |     1 |       |     0
-------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   5 - filter("E"."DEPTNO" IS NOT NULL)
   6 - access("E"."ENAME"="B"."ENAME")
   8 - access("E"."DEPTNO"="D"."DEPTNO")
```

SQL execution plan recap (again)

Execution plan lines are just Oracle kernel functions!

- In other words, each row source is a function

Data can only be accessed using *access path functions*

- Only access paths can access physical data
- Access paths process physical data, return *row sources*

Data processing starts from first line without children

- In other words the first leaf access path in execution plan

Row sources feed data to their parents

- Can be non-cascading, semi-cascading or cascading

A join operation can input only two row sources

- However, it is possible to combine result of more than 2 row sources for some operations (not for joins though)
- Index combine, bitmap merging, filter, union all, for example

Questions?

Tanel Põder
http://www.tanelpoder.com

ORACLE | CERTIFIED MASTER