NYOUG Spring Meeting 2005

Interpreting Execution Plans

Tanel Põder
http://integrid.info

Introduction

- Name: Tanel Põder
- Occupation: Independent consultant
- Company: integrid.info
- Oracle experience: 8 years as DBA
- Oracle Certified Master
- OakTable Network Member
- EMEA Oracle User Group director

- This presentation is about:
  - Understanding execution plans
- This presentation is not about:
  - Generating execution plans
  - Tuning queries

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 process 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 recommended way is
  - explain plan into plan table
  - select * from table(dbms_xplan.display)
- Other methods
  - sql_trace/10046 trace + tkprof
  - v$sql_plan
  - event 10132 (at level 1)
  - 3rd party tools (which use explain plan anyway)

```
set termout off
store set tmp/env_&_connect_identifier..sql replace
save tmp/explain_&_connect_identifier..sql replace
0 explain plan for
run
set termout on
select * from table(dbms_xplan.display);
@@tmp/env_&_CONNECT_IDENTIFIER..sql
get tmp/explain_&_CONNECT_IDENTIFIER..sql
set termout on
```

SQL statement lifecycle

- Parse
  - Statement checked, optimized, cached
- Bind
- Execute
  - Statement executed
  - DML,DDL,DCL are executed immediately
  - Selects aren't neccessarily - exception is sorting
- Fetch
  - Will start actually retrieving result rows
  - How many at a time depends on arraysize
- Rebind, execute
- Rebind, execute
- ...

Parse stages

- Syntactic check
  - Syntax, keywords, sanity
- Semantic check
  - Whether objects referenced exist, are accessible (by permissions) and are usable
- View merging
  - Queries are written to reference base tables
  - Can merge both stored views and inline views
- Query transformation
  - Transitivity, etc
- Optimization
- Query execution plan (QEP) generation
- Storing SQL and execution plan in cache

soft parse

hard parse

Reading execution plan

- SQL rowsource execution starts from the top rightmost operation
- Row sources are generated from tables
- Next stages in plan execution use rowsources from previous operations
- Only two row sources can be joined together at a time!
  - However, some operations are cascading, thus the whole join doesn't have to be done to pass results on for further operations
  - Nested loop is cascading
  - Hash join is semi-cascading
  - Sort-merge join is not cascading

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

```
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

- *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* operation fetches only matching rows

```
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)
```

Some terminology

- Access path - a means to read data from tables, indexes (also external tables)
- Filter - an operation for throwing out non-matching rows (or computing aggregates)

```
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
  - As whole index segment is just scanned as Oracle finds 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 required columns are indexed
  - 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

```
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 and non-equijoins
  - 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 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

```
SQL> create or replace view employees
  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 employees
  2  where ename = 'KING';

-------------------------------------------------------------------------------
| 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  );
-------------------------------------------------------------------------
| 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")
```

Multilevel joins

- Top rightmost operation is executed first
  - Then the one(s) below it at same indentation level
  - Then jump one level back in indentation and continue
  - Cascading operators allow space efficient joining

```
SQL> select e.empno, e.ename, e.dname, b.comm
  2  from employees e, bonus b
  3  where e.ename = b.ename;
-------------------------------------------------------------------------
| Id  | Operation                     | Name       | Rows  | Bytes | Cost (
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |            |     1 |    50 |      4
|   1 |  NESTED LOOPS                 |            |     1 |    50 |      4
|   2 |   NESTED LOOPS                |            |     1 |    37 |      3
|   3 |    TABLE ACCESS FULL          | BONUS      |     1 |    20 |      2
|*  4 |    TABLE ACCESS BY INDEX ROWID| EMP        |     1 |    17 |      1
|*  5 |     INDEX RANGE SCAN          | EMP_ENAME  |    37 |       |      1
|   6 |   TABLE ACCESS BY INDEX ROWID | DEPT       |     1 |    13 |      1
|*  7 |    INDEX UNIQUE SCAN          | PK_DEPT    |     1 |       |      0
-------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   4 - filter("E"."DEPTNO" IS NOT NULL)
   5 - access("E"."ENAME"="B"."ENAME")
   7 - access("E"."DEPTNO"="D"."DEPTNO")
```

Conclusion

- To gain understanding, experimenting is needed
  - Gather statistics
  - Use explain plan/dbms_xplan
  - Run the statement and gather execution statistics
    - Logical IOs, CPU usage, sorts etc from VSQL, VSESSION
    - V$SQL_PLAN_STATISTICS[_ALL], CBO/10053 trace
- Usual problems
  - Not accurate enough statistics
  - Bad SQL
  - optimizer_index_caching, optimizer_index_cost_adj
  - _complex_view_merging, _unnest_subquery
  - sort_area_size, hash_area_size
  - Optimizer bugs
- Metalink note: “Interpreting explain plans”

Questions?

Thank you!

Tanel Põder
tanel@integrid.info
http://integrid.info