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)

```sql
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
```

dbms_xplan demo
SQL statement lifecycle

- Parse
  - Statement checked, optimized, cached
- Bind
- Execute
  - Statement executed
  - DML, DDL, DCL are executed immediately
  - Selects aren't necessarily - 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
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

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>14</td>
<td>518</td>
<td>3 (0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>1</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>14</td>
<td>518</td>
<td>3 (0)</td>
<td>00:00:01</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>37</td>
<td>3 (0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>* 1</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>1</td>
<td>37</td>
<td>3 (0)</td>
<td>00:00:01</td>
</tr>
</tbody>
</table>

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;

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>37</td>
<td>1 (0)</td>
</tr>
<tr>
<td>1</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>EMP</td>
<td>1</td>
<td>37</td>
<td>1 (0)</td>
</tr>
<tr>
<td>* 2</td>
<td>INDEX UNIQUE SCAN</td>
<td>PK_EMP</td>
<td>1</td>
<td></td>
<td>0 (0)</td>
</tr>
</tbody>
</table>

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
SQL> select * from emp
2   where empno > 7000
3   and ename like 'KING%';
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>27</td>
<td>3 (0)</td>
</tr>
<tr>
<td>* 1</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>EMP</td>
<td>1</td>
<td>27</td>
<td>3 (0)</td>
</tr>
<tr>
<td>* 2</td>
<td>INDEX RANGE SCAN</td>
<td>PK_EMP</td>
<td>9</td>
<td></td>
<td>2 (0)</td>
</tr>
</tbody>
</table>

Predicate Information (identified by operation id):

```plaintext
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;
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>5</td>
<td>25 (0)</td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>INDEX FAST FULL SCAN</td>
<td>PK_EMP</td>
<td>54121</td>
<td>264K</td>
<td>25 (0)</td>
</tr>
</tbody>
</table>
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%';

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>37</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>NESTED LOOPS</td>
<td></td>
<td>1</td>
<td>37</td>
<td>4</td>
</tr>
<tr>
<td>* 2</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>1</td>
<td>17</td>
<td>3</td>
</tr>
<tr>
<td>* 3</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>DEPT</td>
<td>1</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>* 4</td>
<td>INDEX UNIQUE SCAN</td>
<td>PK_DEPT</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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%';
```

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>37</td>
<td>9 (12)</td>
</tr>
<tr>
<td>* 1</td>
<td>HASH JOIN</td>
<td></td>
<td>1</td>
<td>37</td>
<td>9 (12)</td>
</tr>
<tr>
<td>* 2</td>
<td>TABLE ACCESS FULL</td>
<td>DEPT</td>
<td>1</td>
<td>20</td>
<td>2 (0)</td>
</tr>
<tr>
<td>* 3</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>4</td>
<td>68</td>
<td>6 (0)</td>
</tr>
</tbody>
</table>

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;

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1245</td>
<td>46065</td>
<td>64 (10)</td>
</tr>
<tr>
<td>1</td>
<td>MERGE JOIN</td>
<td></td>
<td>1245</td>
<td>46065</td>
<td>64 (10)</td>
</tr>
<tr>
<td>* 2</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>DEPT</td>
<td>1</td>
<td>20</td>
<td>2 (0)</td>
</tr>
<tr>
<td>* 3</td>
<td>INDEX FULL SCAN</td>
<td>PK_DEPT</td>
<td>4</td>
<td></td>
<td>1 (0)</td>
</tr>
<tr>
<td>* 4</td>
<td>SORT JOIN</td>
<td></td>
<td>3735</td>
<td>63495</td>
<td>62 (10)</td>
</tr>
<tr>
<td>* 5</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>3735</td>
<td>63495</td>
<td>61 (9)</td>
</tr>
</tbody>
</table>
## View merging

- **Optimizer** merges subqueries, inline and stored views and runs queries directly on base tables.
- Not always possible though.

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

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SELECT STATEMENT</td>
<td></td>
<td>7</td>
<td>210</td>
<td>5 (20)</td>
</tr>
<tr>
<td>*</td>
<td>HASH JOIN</td>
<td></td>
<td>7</td>
<td>210</td>
<td>5 (20)</td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS FULL</td>
<td>DEPT</td>
<td>4</td>
<td>52</td>
<td>2 (0)</td>
</tr>
<tr>
<td>*</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>EMP</td>
<td>7</td>
<td>119</td>
<td>2 (0)</td>
</tr>
<tr>
<td>*</td>
<td>INDEX RANGE SCAN</td>
<td>EMP_ENAME</td>
<td>8</td>
<td></td>
<td>1 (0)</td>
</tr>
</tbody>
</table>
### Subquery unnesting

- Subqueries can be unnested, converted to anti- and semijoins

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

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>37</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>NESTED LOOPS</td>
<td></td>
<td>1</td>
<td>37</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>NESTED LOOPS</td>
<td></td>
<td>1</td>
<td>24</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>SORT UNIQUE</td>
<td></td>
<td>1</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>TABLE ACCESS FULL</td>
<td>BONUS</td>
<td>1</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>* 5</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>EMP</td>
<td>1</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>* 6</td>
<td>INDEX RANGE SCAN</td>
<td>EMP_ENAME</td>
<td>37</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>DEPT</td>
<td>1</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>* 8</td>
<td>INDEX UNIQUE SCAN</td>
<td>PK_DEPT</td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

**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;

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>1</td>
<td>50</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>NESTED LOOPS</td>
<td></td>
<td>1</td>
<td>50</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>NESTED LOOPS</td>
<td></td>
<td>1</td>
<td>37</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>TABLE ACCESS FULL</td>
<td>BONUS</td>
<td>1</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>* 4</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>EMP</td>
<td>1</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>* 5</td>
<td>INDEX RANGE SCAN</td>
<td>EMP.ENAME</td>
<td>37</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>* 6</td>
<td>TABLE ACCESS BY INDEX ROWID</td>
<td>DEPT</td>
<td>1</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>* 7</td>
<td>INDEX UNIQUE SCAN</td>
<td>PK_DEPT</td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

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 V$SQL, V$SESSION
    - 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