Under The Hood Of Query Transformations?

Jože Senegačnik, Oracle ACE Director, Member of OakTable
DbProf d.o.o.
Ljubljana, Slovenia

Keywords
Query Transformations, Cost Based Optimization, Statistical Optimizer, Selectivity, Cardinality

Introduction

Whenever one sends a SQL statement to Oracle database the statement will first be parsed in order to check whether it is syntactically and semantically correct and further or prepare the optimal execution plan for its execution.

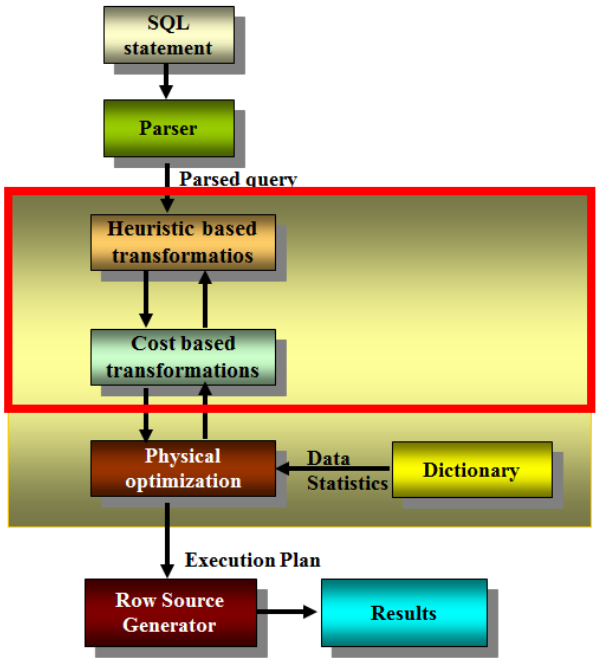

Figure 1: Parse flow

The preparation of execution plan is done by so called "Cost Based Optimizer" (CBO) which is making decisions on the base of the cost of certain operation. So the best execution plan is obviously the one which has the lower possible cost. In order to determine the lowest possible cost the CBO has to undergo certain phases which are usually called optimization as their aim is finding the optimal execution path. The "cost" is a representation of the time which is expected to be required for the execution. It is an internal Oracle measure which combines not only the cost of a physical access measured in number of I/O operations but also the cost of used CPU time.

There are two different optimization phases: **logical** and **physical**. The former is known also as the "query transformation" and is the subject of our discussion. The latter – so called "physical optimization" is actually dealing with:

- Possible access method to every table (full scan, index lookup,...)
- Possible join method for every join (HJ, SM, NL)
- Join order for the query tables (join(join(A,B), C)

We will not discuss further the physical optimization but will rather focus on the logical optimization.

The parse flow is depicted on Figure 1, while the Figure 2. shows the process of logical and physical optimization.

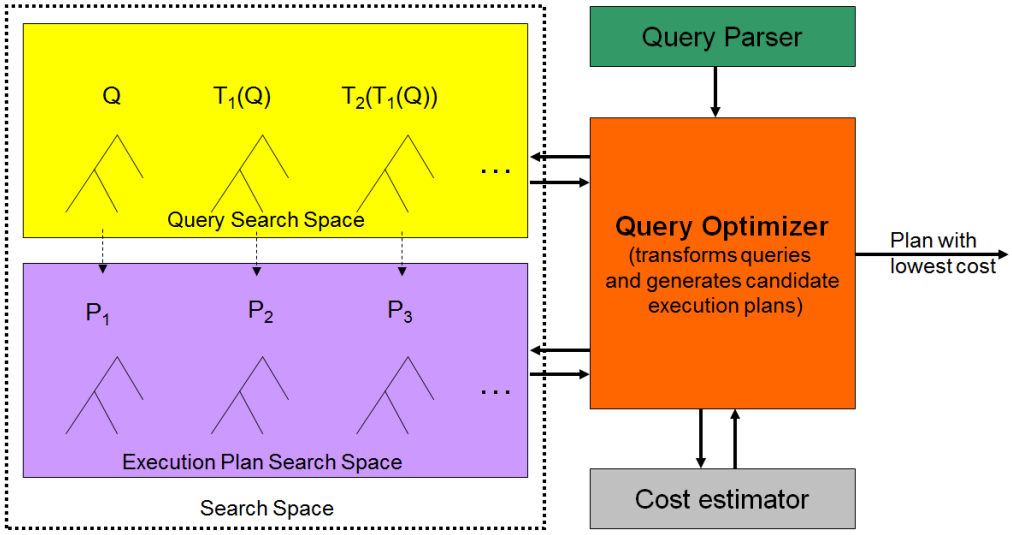

Figure 2: Logical and Physical Optimization

The query optimizer a.k.a CBO tries to perform different transformations for each statement. If we look at Figure 2. In the yellow quadrant we can see the original statement Q and its first transformation T1(Q) and its second transformation T2(T1(Q)) which was possible because of first transformation T1(Q). Important to remember here is that transformations can be used as a base for further transformations which were not possible for the original statement Q.

The CBO tries to produce the first execution plan in the shortest possible time by using some internal shortcuts and then spend more time by examining different possible physical optimizations. The cost of every produced plan is compared with the cost of the plan which had the lowest cost until then and if the cost is lower that plan becomes the winning execution plan.

One of the possible shortcuts in the optimization is so called “single row table” table. A “Single Row Table” will always produce at most only one row so any later join operation with such data source will always produce the cardinality of at most the second data source or in other words joining 10 rows from one table with 1 row from second table will produce only 10 rows. The knowledge about “Single Row Tables” is based on the constraints definition, in this case primary or unique constraint.

So what is the goal of logical optimization a.k.a query transformation? The answer is short and straightforward: to enhance the query performance. Therefore transformation should generate semantically equivalent form of statement, which produces the same results, but significantly differs in performance. We can think about the transformation as somebody would rewrite our query in the form which would do the same job but would run much faster. Everybody who is involved in performance tuning of SQL statement knows that one of the very common approaches to query tuning is so called

"rewriting technique". While preforming query transformation the CBO tries to do exactly the same – rewrite the SQL statement in order to achieve better performance. The result of transformation is usually possibility to do different physical optimization of the same query thus opening new access paths (like using index access, different join method,…) or reducing the number of times that one data source should be scanned (like using very efficient analytical functions for optimization of sums, …). Important to know is also that since Oracle 10g the majority of transformations is cost based what means that the CBO calculates the cost of execution of transformed query. Only some of the transformations are done heuristically.

Another very important fact to notice is that the transformation can span several blocks within the SQL statement. So the purpose is not just to rewrite single query block like the main query or the subquery but the transformation can be done across the whole statement. As transformations open possibility to another transformations this fact is very important.

Another very important fact to notice is that the transformations are done "silently" – actually the only way to spot that a transformation was actually performed is the execution plan of the transformed query. Besides the actual execution plan which shows some kind of operation which is not a valid expression in SQL language syntax like ANTI JOIN or SEMI JOIN operation one can spot the transformation by spotting "unknown" names in the name column of the execution plan like in the following case:

```
-----------------------------------------------------+-----------------------------------+
| Id  | Operation                     | Name         | Rows  | Bytes | Cost  | Time      |
-----------------------------------------------------+-----------------------------------+
| 0   | SELECT STATEMENT              |              |       |       |  8237 |           |
| 1   |  HASH UNIQUE                  |              |     1 |   410 |  8237 |  00:02:39 |
| 2   |   NESTED LOOPS                |              |       |       |       |           |
| 3   |    NESTED LOOPS               |              |    72 |   29K |  8236 |  00:02:39 |
| 4   |     VIEW                      | VW_DTP_B2E7E6CA|    72 |   14K |  8091 |  00:02:38 |
| 5   |      HASH UNIQUE              |              |    72 |   14K |  8091 |  00:02:38 |
| 6   |       TABLE ACCESS FULL       | T4           |  977K |  195M |  8046 |  00:02:37 |
| 7   |     INDEX RANGE SCAN          | T3_I         |     1 |       |     2 |  00:00:01 |
| 8   |    TABLE ACCESS BY INDEX ROWID| T3           |     1 |   206 |     3 |  00:00:01 |
-----------------------------------------------------+-----------------------------------+
Predicate Information:
----------------------
7 - access("T3"."ID"="ITEM_1")
```

Figure 3: Distinct Placement execution plan

In this case the VW_DTP_B2E7E6CA is the view which is performed because of **dist**inct **p**lacement transformation where the original SQL statement is rewritten like the following:

```
select distinct t3.c1, t4.c1
from t3, t4
where t3.id = t4.id;
```

```
SELECT DISTINCT "T3"."C1" "C1",
       "VW_DTP_B2E7E6CA"."ITEM_2" "C1"
FROM
       (SELECT DISTINCT "T4"."ID" "ITEM_1",
              "T4"."C1" "ITEM_2"
        FROM "JOC"."T4" "T4") "VW_DTP_B2E7E6CA",
             "JOC"."T3" "T3"
       WHERE "T3"."ID"="VW_DTP_B2E7E6CA"."ITEM_1"
```

Figure 4: Transformed query after DTP transformation performed

So in this case the CBO creates an inline view named according to the transformation (DTP).

Another possibility to spot a silent transformation was performed is under the “Predicate information” section of the DBMS_XPLAN output like in this case:

```
select /*+ opt_param('_convert_set_to_join','true') */ x.c4
from t1 x
minus
select y.c4
from t1 y;
```

```
-------------------------------------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |     3 |    18 |     8  (25)| 00:00:01 |
|   1 |  HASH UNIQUE        |      |     3 |    18 |     8  (25)| 00:00:01 |
|*  2 |   HASH JOIN ANTI    |      |    10 |    60 |     7  (15)| 00:00:01 |
|   3 |    TABLE ACCESS FULL| T1   |  1000 |  3000 |     3   (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL| T1   |  1000 |  3000 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access(SYS_OP_MAP_NONNULL("X"."C4")=SYS_OP_MAP_NONNULL("Y"."C4"))
```

Figure 5: Set-Join Conversion transformation

The untransformed original SQL statement has the following execution plan:

```
select c4  from t1 minus select c2 from t2 ;
```

```
-------------------------------------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |  1000 |  6000 |     8  (63)| 00:00:01 |
|   1 |  MINUS              |      |       |       |            |          |
|   2 |   SORT UNIQUE       |      |  1000 |  3000 |     4  (25)| 00:00:01 |
|   3 |    TABLE ACCESS FULL| T1   |  1000 |  3000 |     3   (0)| 00:00:01 |
|   4 |   SORT UNIQUE       |      |  1000 |  3000 |     4  (25)| 00:00:01 |
|   5 |    TABLE ACCESS FULL| T2   |  1000 |  3000 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

Figure 6:Execution plan without SJC transformation

From the above execution plans we can see that when the “Set-Join Conversion” transformation was performed an internal SQL engine function (SYS_OP_MAP_NONNULL) was used and instead of MINUS operation we have HASH JOIN ANTI operation in the transformed query.

The easiest place to analyze which transformations were attempted to be performed and also which were actually done is the cost based optimizer trace file produced with event 10053.

```
SQL> alter session set events ‘10053 trace name context forever’;
```

The trace file is written to standard trace location which depends on the version of the database.

As we read the trace file (which can be quite long, especially when multiple tables are involved in SQL statement) we can find at the beginning the legend for abbreviations used further on in the trace file for different transformations.
For the above transformation “Set-Join Conversion” one can find the following text in the CBO trace file (the actual text is highly dependent on the version of database used as the CBO trace file is changed practically at every release) :

```
SJC: Considering set-join conversion in query block SET$1 (#0)
*************************
Set-Join Conversion (SJC)
*************************
SJC:   Checking validity of SJC on query block SET$1 (#0)
SJC:   Passed validity checks.
SJC: SJC: Applying SJC on query block SET$1 (#0)
Registered qb: SEL$09AAA538 0x99f85c60 (SET QUERY BLOCK SET$1; SET$1)
---------------------
QUERY BLOCK SIGNATURE
---------------------
  signature (): qb_name=SEL$09AAA538 nbfros=2 flg=0
    fro(0): flg=0 objn=247624 hint_alias="X"@"SEL$1"
    fro(1): flg=0 objn=247624 hint_alias="Y"@"SEL$2"

SJC: performed
```

Figure 7: SJC transformation details in CBO trace file

Sometimes you can see that some transformation was attempted to be performed, however it was actually not done due to some reason. The most crucial information obtained from the CBO trace file is the transformed query – the text of transformed SQL statement which is not always available as for some transformations there is no change in the actual text of the statement because the transformation can’t be expressed as an SQL operation. Most typical operation of such type is EXISTS operation which transforms to SEMI JOIN and NOT EXISTS operation which transforms in ANTI JOIN. The transformation performed is called “Subquery unnesting”

```
SQL> select cust_id,cust_first_name,cust_last_name
       from customers c
       where exists ( select 1 from sales s where s.cust_id = c.cust_id);

-----------------------------------------------------------
| Id  | Operation                      | Name           |
-----------------------------------------------------------
|   0 | SELECT STATEMENT               |                |
|*  1 |  HASH JOIN SEMI                |                |
|   2 |   TABLE ACCESS FULL            | CUSTOMERS      |
|   3 |   PARTITION RANGE ALL          |                |
|   4 |    BITMAP CONVERSION TO ROWIDS |                |
|   5 |     BITMAP INDEX FAST FULL SCAN| SALES_CUST_BIX |
-----------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("S"."CUST_ID"="C"."CUST_ID")
```

Figure 8: Subquery unnesting transformation

```
...
*****************************
Cost-Based Subquery Unnesting
*****************************
SU: Unnesting query blocks in query block SEL$1 (#1) that
    are valid to unnest.
Subquery Unnesting on query block SEL$1
(#1)SU: Performing unnesting that does not require costing.
SU: Considering subquery unnest on query block SEL$1 (#1).
SU: Checking validity of unnesting subquery SEL$2 (#2)
SU: Passed validity checks.
SU: Transforming EXISTS subquery to a join.
...
```

Figure 9: Subquery unnesting transformation in CBO trace file

From the above execution plan we can see that the “Predicate information” section contains a valid join condition "S"."CUST_ID"="C"."CUST_ID" which is actually a semi join operation. However in SQL syntax we can’t express semi-join operation and thus one should carefully look at the join condition and also at the same time the join operation method which is SEMI JOIN operation in our case.

In the presentation we will walk through the following transformations:

- JPPD - join predicate push-down
- FPD - filter push-down
- PM - predicate move-around
- CVM - complex view merging
- SPJ - select-project-join
- SJC - set join conversion
- SU - subquery unnesting
- OBYE - order by elimination
- CNT - count(col) to count(*) transformation
- JE - Join Elimination
- JF - join factorization
- SLP - select list pruning
- DP – distinct/group by placement

and two new transformations added in Oracle 12c (12.1.0.1)

- Scalar Subquery Unnesting transformation
- Null Accepting Semi-Joins

So the presentation is actually part of this paper where the above transformations are briefly introduced through very simple SQL statements.

Conclusion

The query transformations are logical optimization, which are performed in order to improve the performance of certain query by opening new possibilities for physical optimization. Most of the time the performed transformations are not even spotted by the developer or DBA as they are silently performed during the parse phase of the SQL statement execution. People usually spot them when they look at the execution plan and see that the execution plan is not like the one they were expecting to see. Practically for all transformations there are optimizer hints which could be used to force them or

prevent them (NO_* hints). The last resort to disable all possible transformations is to use NO_QUERY_TRANSFORMATION hint.
Another important fact to know is that with every single release Oracle is adding new transformations which were developed in order to boost the performance of SQL statements.

Kontaktadresse:
Jože Snegačnik
DbProf d.o.o.
Smrjene 153
SI-1291 Škofljica
Slovenia

Telefon: +386 41 72 44 61
E-Mail joze.senegacnik@dbprof.com
Internet: www.dbprof.com