

QUERY TRANSFORMATION – Part 1

Introduction

Query transformation is a set of techniques used by the optimizer to rewrite a query and optimizer it better. Few optimization paths open up to the optimizer after query transformation. Some query transformations must be costed to be chosen and some do not need to be costed. For example, if a table can be eliminated completely from the join, then that transformation is applied and need to cost that transformation is minimal.

Test case

We will use the following test case to illustrate the concepts behind Query transformation. Some of the optimizations that we see here works from version 11gR1 onwards and so, these test cases might not work in the versions 10g and below.

```
create table backup.t1
(n1 number not null primary key,
 n2 number not null,
 n3 varchar2(256) );
insert into backup.t1
  select n1, n1, lpad ( n1, 250,'x') from
  (select level n1 from dual connect by level <=100);
create table backup.t2
(n1 number not null primary key,
 n2 number not null,
 n3 varchar2(256) );
alter table backup.t1 add constraint t1_fk
  foreign key (n2) references backup.t2(n1);
insert into backup.t2 select n1, n1, lpad ( n1, 250,'x')
  from (select level n1 from dual connect by level <=100);
insert into backup.t1 select n1, n1, lpad ( n1, 250,'x')
  from (select level n1 from dual connect by level <=100);
```

Join elimination (JE)

JE is a technique in which one or more tables can be eliminated from the execution plan without altering functional behavior. In the listing 1-1, query selects columns from the table t1 only, but there exists a join predicate between t1 and t2 in that query. Further, no columns are selected from table t2 in this query and join to t2 simply serves as to verify the existence of foreign key values. Enabled Foreign key constraint between these two tables establishes the existence check already and so, there is no need for explicit existence check in the query also. Join to table t2 can be eliminated by the optimizer safely.

```
select /*+ gather_plan_statistics */ t1.* from t1, t2 where t1.n2 =
t2.n1;

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

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers
1	TABLE ACCESS FULL	T1	1	82	100	00:00:00.01	13

Listing 1-1 :JE example case 1

As you see from the listing 1-1, Table T2 is removed from the execution plan. Since there is a valid foreign key constraint, optimizer eliminated the join condition to that table t2.

Let's also discuss another Join Elimination test case. In the Listing 1-2, predicate is "t1.n2 not in (select t2.n1 from t2)". As the enabled foreign key constraint dictates that this predicate will always be false and no rows will be returned. Optimizer promptly identified this condition and added a filter predicate in the step 1 with "NULL is NOT NULL" as a predicate. Step 1 is executed before step 2; Step 2 is never executed as the value of Starts column is zero in the execution plan.

```
SQL_ID d09kmzum9wgta, child number 0
```

```
-----
select /*+ gather_plan_statistics */ t1.* from t1 where t1.n2 not in
(select t2.n1 from t2 )
```

```
Plan hash value: 3332582666
```

```
-----
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
0	SELECT STATEMENT		1		0	00:00:00.01
* 1	FILTER		1		0	00:00:00.01
2	TABLE ACCESS FULL	T1	0	100	0	00:00:00.01

```
-----
```

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

```
-----
1 - filter(NULL IS NOT NULL)
```

Listing 1-2 :JE example case 2

Listing 1-3 provides another variation of JE.

```
select /*+ gather_plan_statistics */ t1.* from t1 where t1.n2 in (select t2.n1 from t2 )
```

```
Plan hash value: 3617692013
```

```
-----
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers
0	SELECT STATEMENT		1		100	00:00:00.01	14
1	TABLE ACCESS FULL	T1	1	100	100	00:00:00.01	14

```
-----
```

Listing 1-3 :JE example case 2

Following output shows the trace lines from 10053 trace file.

```
JE: Considering Join Elimination on query block SEL$5DA710D3 (#1)
```

```
*****
```

```
Join Elimination (JE)
```

```
*****
```

```
JE: cfro: T1 objn:74684 col#:2 dfro:T2 dcol#:2
```

```
JE: cfro: T1 objn:74684 col#:2 dfro:T2 dcol#:2
```

```
Query block (0E0D43D0) before join elimination:
```

```
SQL:***** UNPARSED QUERY IS *****
```

```
SELECT "T1"."N1" "N1","T1"."N2" "N2","T1"."N3" "N3" FROM "CBO3"."T2"
```

```
"T2","CBO3"."T1" "T1" WHERE "T1"."N2"="T2"."N1"
```

```
JE: eliminate table: T2
```

```
Registered qb: SEL$14EF7918 0xe0d43d0 (JOIN REMOVED FROM QUERY BLOCK
```

```
SEL$5DA710D3; SEL$5DA710D3; "T2"@SEL$2")
```

Filter Predicate(s) Generation from constraints

Various filter predicates are generated and added to the execution plan using enabled and validated constraints (check, not null constraints).

In the Listing 1-4, columns n1 and n2 has enabled valid NOT NULL constraints that precludes null values in the columns n1 and n2. Query in the listing 1-4 has predicate "n1 is null or n2 is null" which can never be true. This fact is used by the optimizer to improve the execution plan. Filter predicate (NULL IS NOT NULL) is added in step 1 which will be FALSE. So, Step 2 is never executed as the value of Starts column is 0 in the execution plan. This means that step (2) in the execution plan was never executed and table T1 was never accessed.

```
select /*+ gather_plan_statistics */ * from t1 where n1 is null or n2 is null;
select * from table(dbms_xplan.display_cursor('','','allstats last'));
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
* 1	FILTER		1		0	00:00:00.01
2	TABLE ACCESS FULL	T1	0	100	0	00:00:00.01

Predicate Information (identified by operation id):

1 - filter(NULL IS NOT NULL)

Listing 1-4: Filter predicate generation from NOT NULL constraint.

Let's add a check constraint to this column to explain this further. In the listing 1-5 a check constraint is added which specifies that "n1 <200". But, the query in this listing also has a predicate "n1>200". A predicate is added to the SQL using the check constraint. Query specified predicate n1> 200 and generated predicate n1<200 will nullify each other leading to an always FALSE condition. Optimizer identified this condition and added a filter predicate in step 1: NULL IS NOT NULL.

```
alter table t1 add constraint t1_n1_lt_150 check (n1 <200);
select /*+ gather_plan_statistics */ t1.* from t1 where n1>200;
select * from table(dbms_xplan.display_cursor('','','allstats last'));
```

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time
* 1	FILTER		1		0	00:00:00.01
* 2	TABLE ACCESS FULL	T1	0	20	0	00:00:00.01

Predicate Information (identified by operation id):

1 - filter(NULL IS NOT NULL)
2 - filter("N1">200)

Listing 1-5: Filter predicate generation from a check constraint

Following lines from the trace file generated from event 10053 shows that a predicate n1<200 is added; This auto-generated predicate and existing predicate canceled each other leading to an eternally FALSE condition.

```
kkogcp: try to generate transitive predicate from check constraints for SEL$5DA710D3 (#0)
constraint: "T1"."N1"<200
predicates with check constraints: "T1"."N2"="T2"."N1" AND "T1"."N1"<200
after transitive predicate generation: "T1"."N2"="T2"."N1" AND "T1"."N1"<200
```

finally: "T1"."N2"="T2"."N1"

apadriv-start: call(in-use=1056, alloc=16344), compile(in-use=44792, alloc=46272)

kkoqbc-start

SJC: Set to Join Conversion

In some cases, the optimizer can convert a set operator to a join operator. Interestingly, this feature is not enabled by default (up to 11gR1). In the listing 1-6, we enable this parameter. A MINUS set operation has been converted to a join operation.

```
alter session set "_convert_set_to_join"=true;
```

```
select /*+ gather_plan_statistics */ n2 from t1 minus select n1 from t2
```

Plan hash value: 3050591313

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	...
0	SELECT STATEMENT		1		0	00:00:00.01	...
1	HASH UNIQUE		1	99	0	00:00:00.01	...
2	NESTED LOOPS ANTI		1	99	0	00:00:00.01	...
3	TABLE ACCESS FULL	T1	1	100	100	00:00:00.01	...
* 4	INDEX UNIQUE SCAN	SYS_C0010995	100	1	100	00:00:00.01	...

Predicate Information (identified by operation id):

```
4 - access("N2"="N1")
```

Listing 1-6: SJC

There is also a new hint set_to_join with this new feature.

```
/*+  
...  
  OPT_PARAM('_convert_set_to_join' 'true')  
...  
  SET_TO_JOIN(@"SET$1")  
...  
*/
```

SU: Subquery Unnesting

Subqueries can be unnested in to a join. Listing 1-7 shows that a subquery is unnested in to a view and then joined to other row sources. In this listing, a correlated subquery is moved in to a view VW_SQ_1 and unnested to a Nested Loops Join condition. There are many different variations of Subquery Unnesting possible, but crux of the matter is that subqueries can be unnested, joined and then costed.

```
select /*+ gather_plan_statistics */ n1 from t1 where n1 >  
  (select max(n2) from t2 where t2.n1 = t1.n1)
```

Plan hash value: 2311753844

Id	Operation	Name	Starts	E-Rows	A-Rows	...
0	SELECT STATEMENT		1		0	...
1	NESTED LOOPS		1	1	0	...
2	VIEW	VW_SQ_1	1	5	100	...
* 3	FILTER		1		100	...
4	HASH GROUP BY		1	5	100	...
5	TABLE ACCESS BY INDEX ROWID	T2	1	100	100	...
* 6	INDEX RANGE SCAN	SYS_C0010995	1	100	100	...
* 7	INDEX UNIQUE SCAN	SYS_C0010992	100	1	0	...

Predicate Information (identified by operation id):

```
-----
3 - filter(MAX("N2")<200)
6 - access("T2"."N1"<200)
7 - access("ITEM_1"="T1"."N1")
  filter("N1">"MAX(N2)")
```

SU is one reason why there are many performance issues after a database upgrade to 10g and above. Cost of unnested subquery will go up or down leading to an unfortunate choice of not-so-optimal execution plan.

Use of ORDERED hint can really play havoc with SU feature. For example, in the listing 1-8, join between t1 and t2 is preferred followed by other joins. You would expect to see the leading table in the join to be T1, but the leading row source is VW_SQ_1.

```
select /*+ gather_plan_statistics ORDERED */ t1.n1, t2.n1 from t1 , t2
where t1.n1 = t2.n1 and t1.n1 > (select max(n2) from t2 where t2.n1 =
t1.n1)
```

Plan hash value: 3904485247

Id	Operation	Name	Starts	E-Rows	A-Rows
0	SELECT STATEMENT		1		0
1	NESTED LOOPS		1	1	0
2	NESTED LOOPS		1	1	0
3	VIEW	VW_SQ_1	1	5	100
* 4	FILTER		1		100
5	HASH GROUP BY		1	5	100
6	TABLE ACCESS BY INDEX ROWID	T2	1	100	100
* 7	INDEX RANGE SCAN	SYS_C0010995	1	100	100
* 8	INDEX UNIQUE SCAN	SYS_C0010992	100	1	0
* 9	INDEX UNIQUE SCAN	SYS_C0010995	0	1	0

Predicate Information (identified by operation id):

```
-----
4 - filter(MAX("N2")<200)
7 - access("T2"."N1"<200)
8 - access("ITEM_1"="T1"."N1")
  filter("T1"."N1">"MAX(N2)")
9 - access("T1"."N1"="T2"."N1")
  filter("T2"."N1"<200)
```

Is CBO not honouring our hint? It is honouring our hint. Except that ORDERED hint was applied after the SU transformation and so, unnested view is in the leading row source. A variation of the transformed query from 10053 trace file is printed below. With ORDERED hint, of course, CBO must choose the unnested view as the leading row source. Use LEADING hint instead of ORDERED hint if necessary.

```
SELECT /*+ ORDERED */ "T1"."N1" "N1","T2"."N1" "N1" FROM
(SELECT MAX("T2"."N2") "MAX(N2)","T2"."N1" "ITEM_1" FROM "CBO3"."T2" "T2"
GROUP BY "T2"."N1") "VW_SQ_2",
"CBO3"."T1" "T1","CBO3"."T2" "T2" WHERE "T1"."N1"="T2"."N1" AND
"T1"."N1">"VW_SQ_2"."MAX(N2)" AND "VW_SQ_2"."ITEM_1"="T1"."N1"
```

Summary

There are many such techniques to cover in one blog entry. We will discuss these features further in upcoming blogs.