Use Constraints to Improve the Performance

Joze Senegacnik,
Oracle ACE Director, Member of OakTable
DbProf d.o.o.
Ljubljana, Slovenia

Keywords:

constraints, cost based optimizer, transformation, performance

Introduction

A brief history of constraints in Oracle database

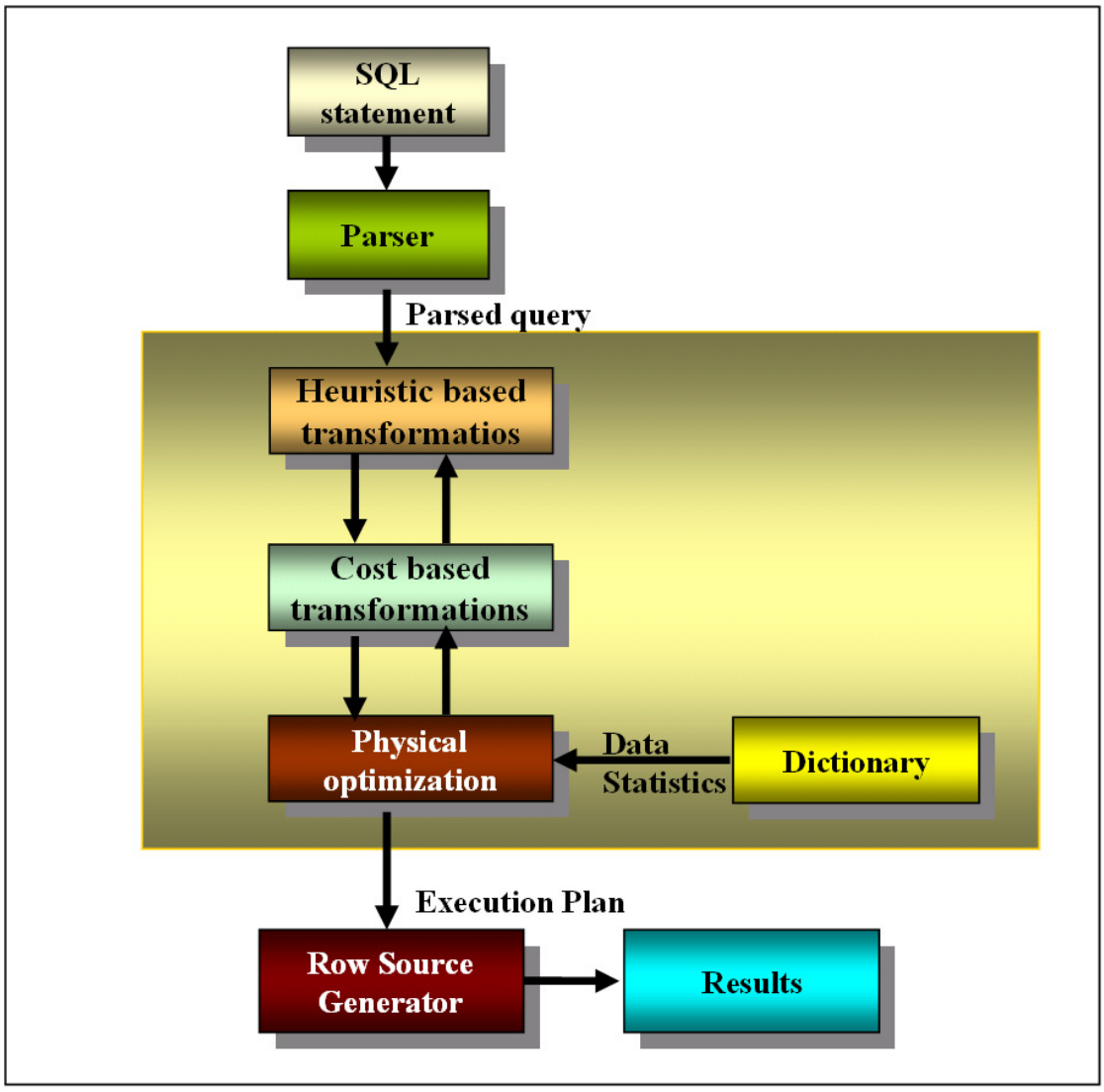
A brief check of the history of development of Oracle Database will reveal that the constraints definitions were present already in Oracle version 6 more than 20 years ago but only as definitions and were not used or enforced. The first version that was able to enforce constraints as additional rules was Oracle version 7. Oracle also introduced new statistical optimizer called Cost Based Optimizer in version 7 but in that version the CBO has many teething problems and was really not a useable product, so everybody was still using old Rule Based Optimizer (RBO). Only later in version 8.0 and especially 8i CBO got more mature. In the latest versions Oracle 9i, 10g and 11g CBO made tremendous improvement and became practically the only possible optimizer although there are still some databases using old RBO which is still there and can be invoked with RULE optimizer hint.
The development of the optimizer resulted in new approaches in the execution optimization and each version has introduced some new features which were changed in subsequent new versions. For instance, a statement which runs perfectly on 9i can have serious performance problems on 10g or 11g, not because the latest versions are bad but usually due to the fact that the development optimized statement in 9i and due to slightly changed optimizer algorithm the statement has to be re-optimized again in 10g or 11g.
In this presentation due to the limited time we will discuss the constraints from the Oracle 11gR2 (11.2) perspective as this is the latest production version available at the moment of preparation of this presentation.

Why are the constraints important for the Cost Based Optimizer?

The constraints definition can be used by the CBO to generate additional predicates which in turn will enable better selectivity estimation what will result in better cardinality estimation and as final result we will get an optimal execution plan.

The process of preparing the execution plan can be seen on **Fehler! Verweisquelle konnte nicht gefunden werden.**. We will walk briefly through this process. A parsed SQL statement is the input for the query transformer. The transformer rewrites the statement when this would be advantageous for better plan generation. Each new version of Oracle database introduces new transformation methods. Any combination of them can be used in the transformation process. After transformation is performed (transformation is also called logical optimization) the statement goes through physical optimization and the overall cost for the statement execution is determined. Therefore for each statement many

potential execution plans are prepared and evaluated and finally the plan with the lowest cost is the winning one. The CBO uses three measures during the physical optimization phase: selectivity, cardinality and cost.



Fehler! Verweisquelle konnte nicht gefunden werden.. *Parsing and optimizing process*

Selectivity

Selectivity is the first and the most important measure. It represents a fraction of rows from a row set where the row set can be a base table or a result of a previous 'join' or a 'group by' operator. The estimator uses statistics to determine the selectivity of a certain predicate. The selectivity is thus tied to a query predicate, such as 'id = 12445', or a combination of predicates, such as 'id = 12445' and 'status = 'A''. The purpose of query predicates is to limit the scope of the query to a certain number of rows that we are interested in. Therefore, the selectivity of a predicate indicates how many rows from a row set will pass the predicate test. Selectivity lies in a value range from 0.0 to 1.0 where a selectivity of 0.0

means that no rows will be selected from a row set and a selectivity of 1.0 means that all rows will be selected.
The CBO uses all possible information available in the process of query transformation. The constraints are one of the sources of the CBO for generation of additional predicates. As we will se later the additionally generated predicates improve the accuracy of selectivity estimation what will result in more accurate cardinality estimation.

Cardinality

Cardinality represents the number of rows in a row source (in a table or is a result of previous operations). Analyzing the table captures the base cardinality. Computed cardinality is the number of rows that are selected from a base table when predicates are applied. The computed cardinality is computed as the product of the table cardinality (base cardinality) and combined selectivity of all predicates specified in where clause. Each predicate is acting as a successive filter on the rows of the base table. CBO assumes that there is no dependency among the predicates (predicate independence assumption). The computed cardinality equals its base cardinality when there is no predicate specified. The computed cardinality for each row source (table or result of previous operations) is visible in the explain plan. The computed cardinality thus determines how many rows will be selected and is used as a measure in the subsequent calculations of cost.

Cost

The cost used by the CBO represents an estimate of the number of disk I/Os and the amount of CPU used in performing an operation. The cost represents units of work or resource used. The CBO uses disk I/O and CPU usage as units of work. The operation can be a full table scan, accessing rows by using an index, join operation, sorting a row source, a group by operation and many others. So the cost of a query plan is the number of work units that were estimated as necessary when the statement is executed.

"Cost" is the result of the "price" of the access method and the estimated cardinality of the row source. When we recall that the cardinality of a row source (i.e. table, result of previous operations) is calculated from the base cardinality of the row source and the estimated selectivity, we see how important the proper selectivity estimation is. If it is misestimated the execution plan will most likely be a sub-optimal one. Thus both factors that are used in a cardinality computation can contribute to the plan becoming sub-optimal. Incorrectly estimated selectivity and an inaccurate base cardinality of the table have same effect. The execution plan contains estimated cardinality and cost for each step and also for the final result set and we can always easily check them against the actual data.

Constraint Types

In Oracle database there are the following constraint types available:
- Check constraint
- Not null (check)
- Unique constraint
- Primary key constraint
- Foreign key constraint

Constraints and execution plan preparation

As we have already mentioned the CBO tries to use constraints to generate additional constraints which are later on used for selectivity estimation. We will observe the process of generating additional

predicates by observing prepared execution plans and also by analyzing the CBO trace file which is generated by using event 10053.

Check constraints

In this paper I will discuss here only the check constraint while other types will be covered in the live conference presentation.

For the start let us create a simple table:

```
CREATE TABLE JOC.T1
(
  ID  NUMBER,
  C1  VARCHAR2(30),
  C2  VARCHAR2(1),
  C3  DATE,
  C4  NUMBER,
  C5  NUMBER
);

ALTER TABLE JOC.T1 ADD ( CONSTRAINT C4_CHECK CHECK (c4 in (10,20,30))
DISABLE);

ALTER TABLE JOC.T1 ADD ( CONSTRAINT C5_CHECK CHECK (c4 < c5) DISABLE);
```

We have 1000 rows in this table and the distribution of values for column C4 is as follows:

```
SQL> select c4,count(*) from t1 group by c4;

        C4   COUNT(*)
---------- ----------
        30         20
        20         80
        10        900
```

We have disabled the check constraint.

```
ALTER TABLE JOC.T1 disable CONSTRAINT C4_CHECK;

Table altered.

explain plan for select * from t1 where c4 >= 35 and c4 <= 36;

-------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |    10 |   280 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T1   |    10 |   280 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

```
---------------------------------------------------

   1 - filter("C4">=35 AND "C4"<=36)
```

We have already gathered statistics for this table and we know that there are no values bigger than 30 in the C4 column, however the CBO cardinality estimation is 10 rows.

Now let us look what happen when we enable the check constraint:

```
ALTER TABLE JOC.T1 enable CONSTRAINT C4_CHECK;

Table altered.

explain plan for select * from t1 where c4 >= 35 and c4 <= 36;

------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    28 |     0   (0)|          |
|*  1 |  FILTER            |      |       |       |            |          |
|*  2 |   TABLE ACCESS FULL| T1   |    10 |   280 |     3   (0)| 00:00:01 |
------------------------------------------------------------------------------

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

   1 - filter(NULL IS NOT NULL)
   2 - filter("C4">=35 AND "C4"<=36)
```

As we can see from the execution plan the CBO generated a new filtering condition which is applied in step 1 named as FILTER and we can see the expression under the “Predicate information”. In this particular case the CBO added the expression “NULL IS NOT NULL” which will always evaluate to FALSE and thus prevent the execution of the subsequent execution steps in particular execution tree. We also see that the CBO estimates that there will be only 1 row returned. In fact, the CBO uses cardinality estimation of 1 row whenever the cardinality of the result set will be only 1 row or less.

If we look in the CBO trace file which we produce by setting event 10053:

```
alter session set events ‘10053 trace name context forever, level 1’;
```

and repeat the explain plan command which causes a hard parse, we can find the following in the report:

```
try to generate transitive predicate from check constraints for query block
SEL$1 (#0)
constraint: "T1"."C4"=10 OR "T1"."C4"=20 OR "T1"."C4"=30

finally: "T1"."C4">=35 AND "T1"."C4"<=36 AND NULL IS NOT NULL

FPD:   transitive predicates are generated in query block SEL$1 (#0)
```

```
"T1"."C4">=35 AND "T1"."C4"<=36 AND NULL IS NOT NULL
....
Final query after transformations:******* UNPARSED QUERY IS *******
SELECT "T1"."ID" "ID","T1"."C1" "C1","T1"."C2" "C2","T1"."C3" "C3",
   "T1"."C4" "C4"
FROM "JOC"."T1" "T1"
WHERE "T1"."C4">=35 AND "T1"."C4"<=36 AND NULL IS NOT NULL
```

We can see that the CBO takes the check constraint as the base for generating additional predicates. The text of the SQL statement is changed and additional filtering condition which in this case as it evaluates to false prevents execution of the subsequent step in the execution plan. Therefore the "TABLE ACCESS FULL T1" step is never executed and the amount of logical (and also physical) I/O is zero.

Here is a simple proof by using autotrace option in SQL*Plus:

```
select * from t1 where c4 >= 35 and c4 <= 36;

no rows selected


Execution Plan
----------------------------------------------------------
Plan hash value: 3773335649

--------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    28 |     0   (0)|          |
|*  1 |  FILTER            |      |       |       |            |          |
|*  2 |   TABLE ACCESS FULL| T1   |     1 |    28 |     3   (0)| 00:00:01 |
--------------------------------------------------------------------------------

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

   1 - filter(NULL IS NOT NULL)
   2 - filter("C4">=35 AND "C4"<=36)



Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          0  consistent gets
          0  physical reads
          0  redo size
        650  bytes sent via SQL*Net to client
        508  bytes received via SQL*Net from client
          1  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
          0  rows processed
```

From the statistics section we can observe that there was no logical or physical I/O as the execution of table access was suppressed by filter operation.

Of course this is only a tiny optimization as we had very small number of rows in a table. What is really important is the fact that additional predicate was generated which caused the correct cardinality estimation. With the correct cardinality estimation we have good chances that the CBO will produce an optimal execution plan.

Check constraints and bind values

Let us look now what happen when we use bind variables instead of literal values.
Consider the following case:

```
variable a1 number
variable a2 number

begin :a1 := 30; :a2 := 30; end;
/
PL/SQL procedure successfully completed.

set autotrace on

select * from t1 where c4 >= :a1 and c5 <= :a2;

no rows selected

Execution Plan
----------------------------------------------------------
Plan hash value: 838529891

--------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |    17 |   476 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T1   |    17 |   476 |     3   (0)| 00:00:01 |
--------------------------------------------------------------------------------

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

   1 - filter("C5"<=TO_NUMBER(:A2) AND "C4">=TO_NUMBER(:A1))
```

From the autotrace output we clearly see that the CBO has not generated the additional filter and this is the right answer. Although the CBO peeked at the values of bind variables at the parse time they could not be used to generate additional predicates as they are not constants and they could and most likely would change for subsequent executions. Therefore with bind variables the CBO has tight hands and a good solution would be that for such cases one would use literal values in the SQL statement text instead of bind variables.

Cardinality misestimates

Let us look how good the CBO is in "understanding" the constraints. If we write a simple equality condition as c4 = 30 we get the exact answer.

```
explain plan for select * from t1 where c4 =  30;
```

```
-------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |    20 |   640 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T1   |    20 |   640 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - filter("C4"=30)
```

But if we slightly change the condition and request all rows where c4 >= 30 we get an inaccurate cardinality estimation of 19 rows. So it is obvious that the CBO has no good arithmetic algorithm built in for such estimation.

```
explain plan for select * from t1 where c4 >=  30;

-------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |    19 |   532 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T1   |    19 |   532 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - filter("C4">=30)
```

We can observe a similar behaviour in the following case in which we are essentially looking for the same result, for all rows where c4 = 26 but we can define this also as c4 > 25 and c4 < 27.

```
explain plan for select * from t1 where c4 = 26;

-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    32 |     0   (0)|          |
|*  1 |  FILTER            |      |       |       |            |          |
```

```
|*  2 |   TABLE ACCESS FULL| T1   |    10 |   320 |     3   (0)| 00:00:01 |
---------------------------------------------------------------------------

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

   1 - filter(NULL IS NOT NULL)
   2 - filter("C4"=26)
```

As we have equality condition the CBO correctly figures out from check constraint that the value 26 is not present in the table and therefore generates an additional condition which always evaluates to false.
However, the behavior is different in this case:

```
explain plan for select * from t1 where c4 > 25 and c4 < 27;

---------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |    10 |   320 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| T1   |    10 |   320 |     3   (0)| 00:00:01 |
---------------------------------------------------------------------------

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

   1 - filter("C4">25 AND "C4"<27)
```

Due to the lack of good algorithm the query has to execute a full table scan although no rows are returned.

Other types of constraints

In the live conference presentation we will look at the other types of constraints so the reader should consider the presentation slides as the logical continuation of this paper.

Conclusions

As we can see from the above case the CBO is able to use constraints as a base for generation of additional predicates. The CBO tries to perform all possible transformations on a particular statement. The new predicates which are result of transformation open possibilities for new access paths and different physical optimization in which the CBO tries to find the correct join order, the most suitable join method and the best data access path.

We should not overlook a very important fact that we insert data only once but query it many times and therefore it is very important that all queries are running in optimal way.

We must be aware of the limitations of the current version of the CBO, but we can expect that forthcoming versions will be event “more clever” in this area and therefore constraints should be defined.

Contact address:

Joze Senegacnik
Dbprof d.o.o.
Smrjene 153
SI-1291 Skofljica, Slovenia

Phone: +386 41 72 44 61
Fax: +386 59 92 56 25
Email joze.senegacnik@dbprof.com
Internet: www.dbprof.com; joze-senegacnik.blogspot.com