

Tables vs. Predicates

Vadim Tropashko, vadimtro@gmail.com

The "Access Predicates" and "Filter Predicates" plan columns were introduced long time ago -- in Oracle 9.2. What initially seemed to be a pretty small feature turned out to become an important explain plan component, so that predicates are displayed unconditionally by the DBMS_XPLAN facility. In this article we'll explain this phenomenon.

Introduction to basic select-project-join optimization

Oracle optimizer consists of two large components:

- Physical optimizer, and
- Query transformation engine

Physical optimizer is responsible for optimization of a single query block, while query transformation engine orchestrates multiple local query block optimizations into a larger framework. In this article we'll focus solely on physical optimizer.

The origin of this division may be traced back to System R, which introduced many novel optimization ideas. Arguably, the most important was the invention of cost-based method for optimizing Select-Project-Join (SPJ) queries [1,2]. A typical SPJ query in SQL has a familiar "select-from-where" shape:

```
SELECT empno, dname
FROM emp, dept
WHERE emp.deptno=dept.deptno
AND sal>500
```

This query in Relational Algebra is written as:

$$\pi_{\text{empno,dname}} \sigma_{\text{sal}>500} (\text{Emp} \bowtie \text{Dept})$$

By Relational Algebra laws [2], the selection $\sigma_{\text{sal}>500}$ that affects a single table Emp can be pushed down into the join

$$\pi_{\text{empno,dname}} (\sigma_{\text{sal}>500} \text{Emp} \bowtie \text{Dept})$$

In general case, the expression inside parenthesis is a join of several relations:

$$A \bowtie B \bowtie C$$

We omitted parenthesis, because join operation is associate and commutative. From performance perspective, however, the order is significant. In our case we have 12 potentially different query executions:

```
( A ⋈ B ) ⋈ C
( A ⋈ C ) ⋈ B
( B ⋈ A ) ⋈ C
( B ⋈ C ) ⋈ A
( C ⋈ B ) ⋈ A
( C ⋈ A ) ⋈ B
A ⋈ ( B ⋈ C )
A ⋈ ( C ⋈ B )
B ⋈ ( A ⋈ C )
B ⋈ ( C ⋈ A )
C ⋈ ( B ⋈ A )
C ⋈ ( A ⋈ B )
```

Next, each alternative execution plan is evaluated from cost perspective. The cost model relies on:

- A set of statistics maintained on relations and indexes, e.g. number of blocks, number of distinct values (NDV) and so on.
- Formulas to estimate selectivity of predicates, such as single table predicate $sal = 500$, or joint predicate $emp.deptno=dept.deptno$ in our example. The size of an output of a join $\sigma_{sal>500} Emp \bowtie Dept$ is estimated to be the product of the input relations $\sigma_{sal>500} Emp$ and $Dept$ and applying the selectivity of the $emp.deptno=dept.deptno$ predicate.
- Formulas to estimate CPU and I/O costs of executions of each operator.

In our example of 3-relation join

```
( A ⋈ B ) ⋈ C
```

the cost would be evaluated in three steps:

- i. **Cost of scanning A**
which is roughly: `blocks/multiblock_read_count`
- ii. **+ Cost of joining A ⋈ B**
which is roughly: `cardinality(A)*cost_of_access(B)`
- iii. **+ Cost of joining (A ⋈ B) ⋈ C**
which is roughly: `cardinality(A ⋈ B)*cost_of_access(C)`

We can witness the best plan chosen by optimizer, by checking the content of `V$SQL_PLAN` and `V$SQL_PLAN_STATISTICS` (fig. 1).

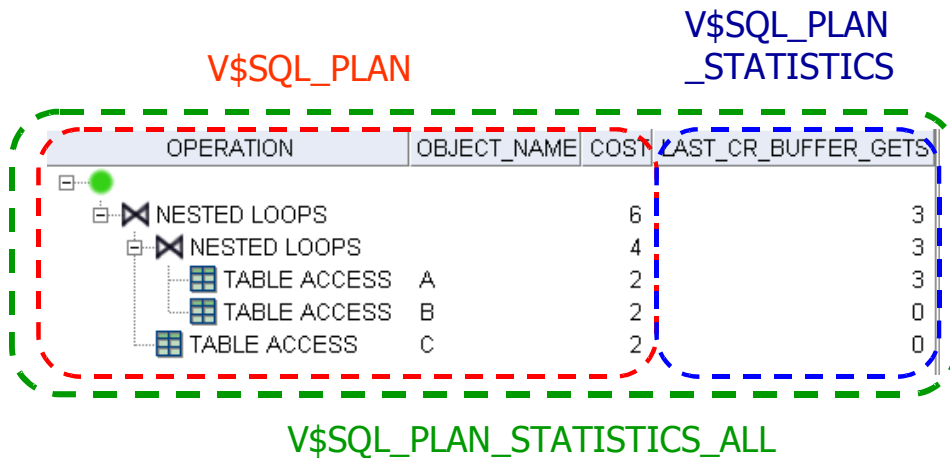


Fig. 1. SQL execution statistics can be queried via the `V$SQL_PLAN` and `V$SQL_PLAN_STATISTICS` views (which are joined conveniently into `V$SQL_PLAN_STATISTICS_ALL`). Alternatively, you can just click the “Run Autotrace” button in *SQLDeveloper* worksheet and you’ll get a similar graphical display.

The main purpose of the `V$SQL_PLAN` and `V$SQL_PLAN_STATISTICS` views is verifying that there is no significant difference between performance estimates and actual execution numbers. In our example we see that the final cost 6 is not very different from number of buffer gets, therefore optimizer estimates are OK. This is a very simple case, and it is not uncommon for optimizer estimations to be off at several orders of magnitude. Quality of optimizer estimates, however, is not our main focus in this article.

Selection

After we have learned how to optimize joins, we can reflect back and notice that selection is a special case of join. Therefore it might be possible to include selection into join optimization framework! Consider

$$(\sigma_{x=1} A) \bowtie B$$

which is the same as

$$\sigma_{x=1} (A \bowtie B)$$

Intuitively selection pushed into a join is more efficient method, but we’ll be able to suggest a more satisfactory explanation of this phenomenon.

Likewise,

$$\sigma_{x=0} \sigma_{y=1} A$$

is equivalent to

$$\sigma_{y=1} \sigma_{x=0} A$$

but, again, they might have very different performance.

It is well known that predicates can be considered as relations. For example, the $x=0$ predicate, is a finite relation (fig.2a).

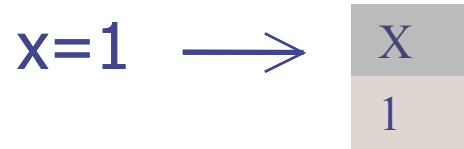


Fig. 2a. The $x=0$ predicate is unary relation that consists of a single tuple.

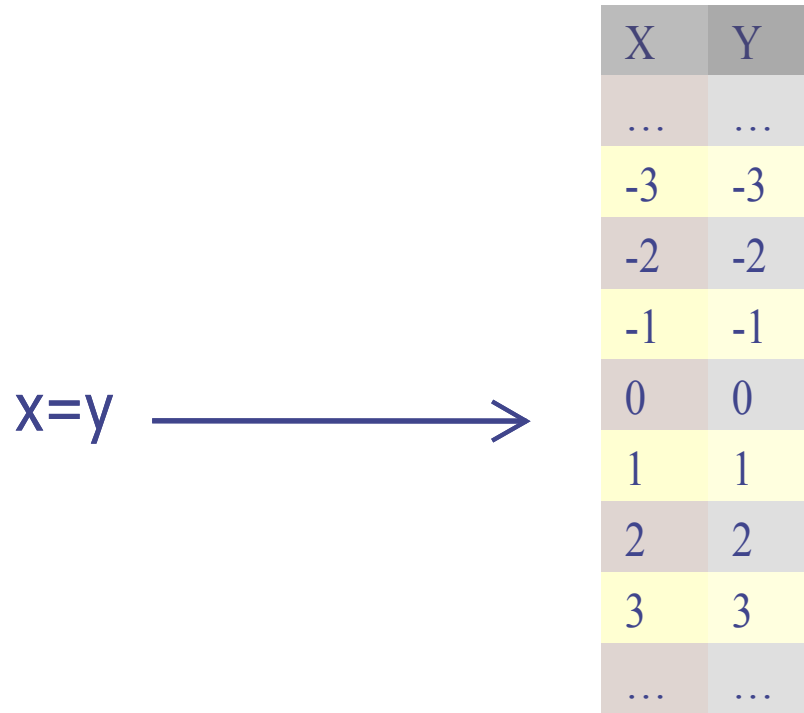
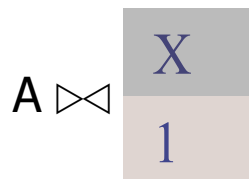


Fig. 2b. The $x=y$ predicate is an infinite binary relation.

As soon as we allowed relations defined by predicates, it follows that we have to admit infinite relations too (fig. 2b). This drawback, however, is outweighed by the fact that selection is no longer a fundamental operator. Our earlier selection example

$$\sigma_{x=0} A$$

is transformed into a join



Likewise,

$$\sigma_{x=y} A$$

X	Y
...	...
-3	-3
-2	-2
-1	-1
0	0
1	1
2	2
3	3
...	...

$A \bowtie$

becomes a join as well

Now that we have joins instead of selections, why don't we apply the join optimization method from previous section? Let's consider infinite relation case first. For two relations we have 2 join orders.

1. Infinite relation as a leading table:

X	Y
...	...
-1	-1
0	0
1	1
...	...

$\bowtie A$

We can the infinite table, and for every tuple we find matching tuples from **A**. This is clearly unfeasible method, and it should also be rejected from cost estimation perspective! Infinite table is similar to a very large table, and for very large tables scanning them is prohibitory expensive.

2. **A** as a leading table:

$\mathbf{A} \bowtie$	X	Y

	-1	-1
	0	0
	1	1

There we scan the full relation \mathbf{A} , and for each tuple find the matching pairs $x=y$ from the infinite equality relation. This is quite feasible method, even though the inner relation is infinite. In other words, we scan \mathbf{A} and retain only those tuples that satisfy $x=y$. This is exactly the same as classic implementation of the selection operator, but we see that our new method is general enough to embrace the classic method of selection.

Let see if there finite predicate relation is any different. We evaluate the two join orders again.

1. Predicate relation, as a leading table:

X	\bowtie	\mathbf{A}
1		

Unlike the infinite relation case, now we are capable of scanning the full leading table. In fact, we can join both tables with any of the three known join methods: nested loops, sort merge join, and hash join. In case of the nested loops we might probably want to create and leverage a join index on column x of table \mathbf{A} . This access path is essentially an single table index range scan access path in classic framework!

2. \mathbf{A} as a leading table:

$\mathbf{A} \bowtie$	X
	1

There the join of both relations can be informally be viewed as scanning \mathbf{A} , and retaining only those tuples that satisfy $x=1$. But formal physical optimization method which we

studied in the previous section suggests that we approach the problem as merely a join optimization.

Which alternative is better? Unlike the previous case with infinite relation it is impossible to tell. Just run the optimizer and evaluate the cost of all the alternatives!

Now that we handled single selection, let's move on to cases with more than one predicate.

Predicate Ordering

A typical query that we analyze in this section would be

```
select * from EMP
where sal = 500
and deptno = 10
```

According to “Predicates as Relations” perspective, this query is essentially a join of three relations:



Joins queries are like nails, begging to be hammered yet again with our universal physical optimization method. Just write down all the join order permutations, evaluate their costs, and select the winner!

And what is the reality? Well, there is something that 10053 trace tries to tell us in at least 2 different sections. In the transformation section (that is outside of physical optimizer) it logged:

```
PM: Considering predicate move-around in SEL$1 (#0).
PM: Checking validity of predicate move-around in SEL$1 (#0).
```

Then, in single access predicate it dumped:

```
SINGLE TABLE ACCESS PATH
Column (#6): SAL(NUMBER) AvgLen: 4.00 NDV: 12 Nulls: 0 Density: 0.083333 Min: 800 Max: 5000 Using
prorated density: 0.077381 of col #6 as selectivity of out-of-range value pred Column (#8):
DEPTNO(NUMBER) AvgLen: 3.00 NDV: 3 Nulls: 0 Density: 0.333333 Min: 10 Max: 30 Table: EMP Alias: EMP
Card: Original: 14 Rounded: 5 Computed: 5.39 Non Adjusted: 5.39 Access Path: TableScan
```

Which predicate it thinks is more advantageous to apply first, and moreover what is selectivity/cardinality estimation after each individual predicate applied is outside of optimizer current tracing capabilities.

Next, the standard DBMS_XPLAN facility outputs:

```

-----
| Id | Operation          | Name | Rows | Bytes |
-----
|  0 | SELECT STATEMENT  |      |     1 |    32 |
|*  1 |  TABLE ACCESS FULL| EMP  |     1 |    32 |
-----
Predicate Information (identified by operation id):
-----
   1 - filter("SAL">=500 AND "DEPTNO">=10)

```

Even though the predicate is shown, and it is possible to figure out the combined predicate selectivity, it is less convenient compared to the case where the query has been split into 2 joins, so that we can know the individual predicates selectivity estimations.

As it comes up to plans, SQLDeveloper goes one step further, parses the predicate information, and displays individual predicates in-place (Fig 3).

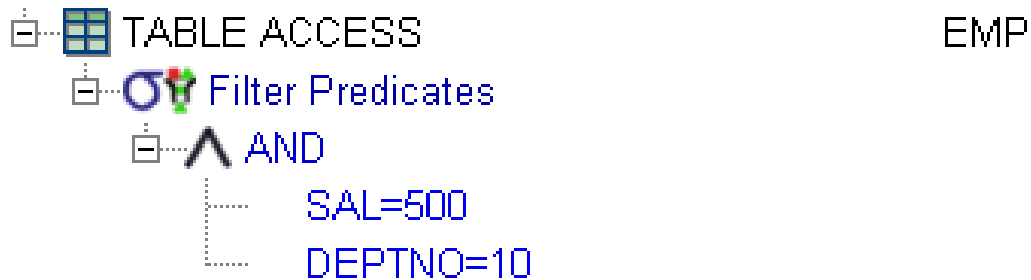


Fig 3. SQLDeveloper autotrace plan rendering. The combined predicate “sal = 500 and deptno = 10” is parsed, and the tree of Boolean operators is shown under the node where it belongs. The Boolean conjunction between the “sal = 500” and “deptno = 10” predicates is essentially the join between corresponding relations. Furthermore, the result is joined with the Emp table, although the display is a conservative selection operator.

Critique and Conclusion

This paper “rediscovered” a well-known idea, which was culminated in the LDL system [3]. Major LDL-style optimization advantage is that it provides much simpler optimization framework. Join order enumeration is a fundamental optimization method. For SQL engine developer physical optimization is comparatively easy to implement, and for SQL performance analyst it is quite easy to understand. Join order enumeration is perhaps the only well documented part of the 10053 trace facility. The more it is handled by this relatively clear functionality, the better.

LDL-style optimization has been challenged in two ways.

First, there is more joins to consider, which is a serious problem, because the optimizer search space grows exponentially with the number of joins. This is, however, something that optimization folks grudgingly accepted as unavoidable in their area. Cost based subquery unnesting, for example, increases the chances of getting better plan at the cost of exploring much larger search space. The tradeoff between having better plan quality vs. doing more optimization job is unavoidable.

A common solution for combinatorial search explosion is employing some heuristic that limits search space. Then, it is easier to troubleshoot a problem of optimizer not finding a good plan due to a known heuristics, rather than follow convoluted and often inconsistent logic of optimizer decisions.

Second objection to LDL-style optimization is that left deep join tree enumeration (which is employed by virtually any cost based optimizer today) may miss good bushy plan. Consider

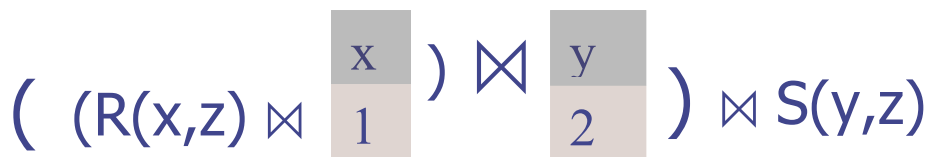


Here the best plan is to select unique tuples from relations EMP and DEPT first, then join them. Admittedly, the execution where we fetch a single record from the EMP table via unique index scan, and then join DEPT table by a unique DEPTNO key should be equally fast. Therefore, this is not really a showcase of bushy vs. left deep join optimization.

What if we don't have foreign key relation between the tables? Consider



Is this bushy plan such a good idea? First, the general objection to bushy plans applies here as well -- bushy plans are not pipelined. Second, the table S is assumed to be big, so we'd better have an index on S(y). Then, why don't we extend it to a composite index S(y,z) ? Then, left deep plan



is again a good competitor. In a word, the case where all linear execution plans are inferior to some bushy plan is quite challenging to create!

Selection operator represented as a join is only the first step. Simpler optimization engine idea can't be advanced without simpler relational algebra system. An interested reader is referred to [4,5] for more advanced study of the subject.

Literature

#	Author(s)	Title	URL
1	Surajit Chaudhuri	An Overview of Query Optimization in Relational Systems	citeseer.ist.psu.edu/chaudhuri98overview.html
2	S.Abiteboul R.Hull V.Vianu	Foundations of Databases	http://www.amazon.com/Foundations-Databases-Logical-Serge-Abiteboul/dp/0201537710
3	D. Chimenti R. Gamboa R. Krishnamurthy S. Naqvi S. Tsur C. Zaniolo	The LDL System Prototype	http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=627400
4	V. Tropashko	Relational Algebra as non-Distributive Lattice	http://arxiv.org/abs/cs/0501053
5	M. Spight, V. Tropashko	First Steps in Relational Lattice	http://arxiv.org/abs/cs.DB/0603044