Symmetric Difference in SQL

By Vadim Tropashko

How to compare the content of two tables is one of the most popular topics on the Ask Tom forum. In this article, we'll investigate the performance of three different table comparison techniques. Among other things, this case study is an entertaining journey into Oracle's optimizer performance arena.

The Problem

Suppose there are two tables, A and B, with the same columns, and we would like to know if there is any difference in their contents.

In the set theory a similar question is well known as a *symmetric difference* operation.

Relation Equality. A reader with an Object Oriented programming background might wonder why the relational model in general, and SQL in particular, doesn't have an equality operator. You'd think the first operator you'd implement for a data type, especially a fundamental data type, would be equality! Simple: this operator is not relationally closed, as the result is Boolean value and not a relation.

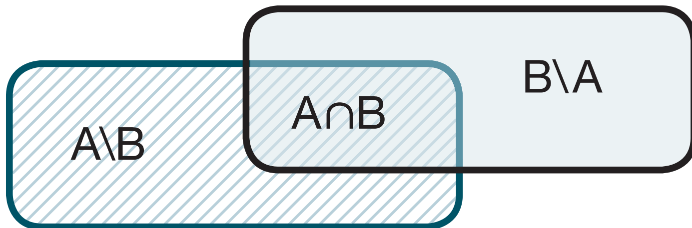

Figure 1. Symmetric difference of two sets A and B can be expressed as either $(A \setminus B) \cup (B \setminus A)$, or $(A \cup B) \setminus (A \cap B)$.

Set Solution

Figure 1 with the expressions $(A \setminus B) \cup (B \setminus A)$ and $(A \cup B) \setminus (A \cap B)$ pretty much exhausts all the theory involved. It is straightforward to translate them into SQL queries. Here is the first one:

```
(
 select * from A
 minus
 select * from B
) union all (
 select * from B
 minus
 select * from A
)
```

In practice, however, this query is a sluggish performer. With a naïve evaluation strategy, the execution flow and the operators are derived verbatim from the SQL which we have written. From the performance perspective, this approach seems to be far from optimal. First, each table has to be scanned twice. Then, four sort operators are applied in order to exclude duplicates. Next, the two set differences are computed, and, finally, the two results are combined together with the union operator.

Let's quickly verify this guess. For this test purpose any sufficiently large tables would do:

```
create table A as
select obj# id, name from sys.obj$
where rownum < 100000;
create table B as
select obj# id, name from sys.obj$
where rownum < 100010;
```

After executing the symmetric difference query, and capturing the rowsource execution statistics (from V$SQL_PLAN_STATISTICS_ALL dictionary view) we get the following result:

OPERATION	OPTIONS	OBJECT NAME	LAST OUTPUT ROWS	LAST CR BUFFER GETS	LAST ELAPSED TIME
UNION-ALL			10	1744	2411543
MINUS			0	872	1241145
SORT	UNIQUE		99999	436	463768
TABLE ACCESS	FULL	A	99999	436	100223
SORT	UNIQUE		100009	436	465849
TABLE ACCESS	FULL	B	100009	436	100060
MINUS			10	872	1170351
SORT	UNIQUE		100009	436	446301
TABLE ACCESS	FULL	B	100009	436	100064
SORT	UNIQUE		99999	436	447135
TABLE ACCESS	FULL	A	99999	436	100036

Figure 2

If confirms our intuition on the number of table scans and sorts. It also reveals that the time scanning a single table (about 100 msec) is small compared to the time spent when sorting each table (about 450 msec). Therefore we could try to aim lower than 2.4 sec of total execution time.

This is not the only possible execution plan for the symmetric difference query. Oracle has quite sophisticated query rewrite capabilities, and transforming set operation into join is one of them. In our example, both minus operators could be rewritten as anti-joins. As of version 10.2, however,

this transformation is not enabled by default, and used as part of view merging/ query unnesting framework only. It can be enabled with

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

The other alternative is to rewrite the query manually as

```
select * from A
where (col1,col2,…) not in (select col1,col2,… from B)
   union all
select * from B
where (col1,col2,…) not in (select col1,col2,… from A)
```

The new execution statistics

OPERATION	OPTIONS	OBJECT NAME	LAST OUTPUT ROWS	LAST CR BUFFER GETS	LAST ELAPSED TIME
UNION-ALL			10	1744	1656804
SORT	UNIQUE		0	872	831331
HASH JOIN	ANTI		0	872	831312
Access Predicates					
AND					
B.ID=B.ID					
B.NAME=B.NAME					
TABLE ACCESS	FULL	A	99999	436	100051
TABLE ACCESS	FULL	B	100009	436	100070
SORT	UNIQUE		10	872	825416
HASH JOIN	RIGHT ANTI		10	872	825333
Access Predicates					
AND					
B.ID=A.ID					
B.NAME=A.NAME					
TABLE ACCESS	FULL	A	99999	436	100053
TABLE ACCESS	FULL	B	100009	436	100073

Figure 3

evidence about 30 percent improvement in execution time, which is considered small by optimizer standards.

The prevalent perception in the performance optimization field is that *Hash Join* (and *Sort-Merge Join* as well, for that matter) is almost always inferior to indexed *Nested Loops*. Would nested loops be better than hash join in this case? In order to enable indexed nested loops access path we have to create indexes, first:

```
CREATE INDEX A_id_name ON A(id, name);
CREATE INDEX B_id_name ON B(id, name);
```

Note that in a more realistic case, tables would probably have more columns, but most likely they might already have an index associated with primary key.

Unfortunately, index appearance doesn't affect the execution plan. We have to enforce it. Quick and dirty fix to the plan is disabling hash and merge join altogether:

```
alter session set "_hash_join_enabled" = false;
alter session set "_optimizer_sortmerge_join_enabled" = false;
```

The less intrusive way to influence an access path is hinting. But how could we hint the join method when there is no join in the query as written in the first place? OK, this might be a convenient opportunity to discuss hint enhancements in 10*g*.

Let's get a bird's-eye view into query optimization architecture in Oracle first. The basic query structure in the RDBMS engine is the query block. A subquery, an inner view, an outermost select statement that contain those, all are query blocks. The particular query that we are studying in this section has five query blocks: two pairs of selects from the tables A and B, and one set operation. After this query is transformed, it would consist of only three query blocks which we might witness from the QBLOCK_NAME column of the PLAN_TABLE.

OPERATION	OBJECT_NAME	QBLOCK_NAME
UNION-ALL		SET$D8486D66
SORT		SEL$C9AE5379
HASH JOIN		
Access Predicates		
AND		
C.ID=D.ID		
C.NAME=D.NAME		
TABLE ACCESS	A	SEL$C9AE5379
TABLE ACCESS	B	SEL$C9AE5379
SORT		SEL$74086987
HASH JOIN		
Access Predicates		
AND		
B.ID=A.ID		
B.NAME=A.NAME		
TABLE ACCESS	A	SEL$74086987
TABLE	B	SEL$74086987

Figure 4

Those query block names allow directing the hints to the query blocks where they are supposed to be applied. In our case, the joint method can be hinted as

```
/*+ use_nl(@"SEL$74086987" A)
    use_nl(@"SET$D8486D66" B)*/
```

which produces the following rowsource level execution statistics:

continued on page 18

Symmetric Difference in SQL continued from page 17

OPERATION	OPTIONS	OBJECT NAME	LAST STARTS	LAST OUTPUT ROWS	LAST CR BUFFER GETS	LAST ELAPSED TIME
UNION-ALL			1	10	201922	2523641
SORT	UNIQUE		1	0	100956	1257233
NESTED LOOPS	ANTI		1	0	100956	1257209
TABLE ACCESS	FULL	A	1	99999	436	100045
INDEX	RANGE SCAN	B_ID_NAME	99999	99999	100520	778132
Access Predicates						
AND						
B.ID=B.ID						
B.NAME=B.NAME						
SORT	UNIQUE		1	10	100966	1266327
NESTED LOOPS	ANTI		1	10	100966	1266249
TABLE ACCESS	FULL	B	1	100009	436	100042
INDEX	RANGE SCAN	A_ID_NAME	100009	99999	100530	780688
Access Predicates						
AND						
B.ID=A.ID						

Figure 5

I included the LAST_STARTS column into result set in order to double-check that index rowsource node has been started exactly once per each row scan in the outer table. Specifically, B_ID_NAME index node has been started 99999 times and processed 99999 rows in total, therefore each index scan probed 99999/99999=1 row in the inner table B only, as if the index were unique.

From the statistics output we see significant increase in buffer gets, which is not offset by any noticeable improvement in the execution time. It is fair to conclude that indexed nested loops didn't meet our performance expectations in this case.

Symmetric Difference via Aggregation

Symmetric difference can be expressed via aggregation:

```
select * from (
  select id, name,
    sum(case when src=1 then 1 else 0 end) cnt1,
    sum(case when src=2 then 1 else 0 end) cnt2
  from (
    select id, name, 1 src from A
    union all
    select id, name, 2 src from B
  )
  group by id, name
)
where cnt1 <> cnt2;
```

This is appeared as rather elegant solution where each table has to be scanned once only, until we capture execution statistics:

OPERATION	OPTIONS	OBJECT NAME	LAST OUTPUT ROWS	LAST CR BUFFER GETS	LAST ELAPSED TIME
FILTER			10	872	1992602
Filter Predicates					
SUM(CASE SRC WHEN 1 THEN 1 ELSE 0 END)<>SUM(CASE SRC WHEN 2 THEN 1 ELSE 0 END)					
HASH	GROUP BY		100009	872	1880665
VIEW			200008	872	1200120
UNION-ALL			200008	872	1000105
TABLE ACCESS	FULL	A	99999	436	100051
TABLE ACCESS	FULL	B	100009	436	100052

Figure 6

Although we succeeded decreasing buffer gets count in half, the execution time still remained around two seconds. Now, let us see how this can be done efficiently – this is also the main point of this article.

Hash Value-based Table Comparison

When comparing data in two tables with identical signatures there actually are two questions that one might want to ask:

- Is there any difference? The expected answer is Boolean.
- What are the rows that one table contains and the other doesn't?

Answering the second question implies that we can answer the first, yet it is possible that the first query might have better performance. We'll approach it with the hash-based technique in mind.

The standard disclaimer of any hash-based method is that it is theoretically possible to get a wrong result. The rhetorical question, however, is how often did the reader experience a problem due to hash value collision? I never did. Would a user accept a tradeoff: significant performance boost of the table difference query at the expense of remote possibility of getting an incorrect result?

The idea of hash-based method is associating a single hash value with a table. This hash value behaves like aggregate, and therefore, it can be calculated incrementally: if a row is added into a table, then a new hash value is a function of the old hash value and the added row. Incremental evaluation means good performance, and the two hash values comparison is done momentarily.

Figure 7: A method to "weight" a table into a single value involves concatenating all the fields per each row first. Then a concatenated string is mapped into a number—hash value. Finally all hash values are aggregated into a single one. Had a single field in a single row changed, the table weight would change as well.

Almost any hash function would work for our purposes, and there are at least two hash functions that we can use:

1. The legacy PL/SQL function in the package DBMS_UTILITY.
2. The newer native SQL function orahash.

Since we’ll invoke hash function from a SQL statement, we’d better use orahash and, therefore, avoid expensive overhead of SQL <-> PL/SQL context switch.

A naive implementation of our table hash based comparison method is as simple as

```
select sum(orahash(id||’|’||name,
                                        1,
                                        POWER(2,16)-1))
from A
 union all
select sum(orahash(id||’|’||name,
                                        1,
                                        POWER(2,16)-1))
from B;
```

We concatenate all the row fields (with a distinct separator, in order to guarantee uniqueness), then translate this string into a hash value. Alternatively, we could have calculated hash values for each field, and then apply some asymmetric function in order for the resulting hash value to be sensitive to column permutations:

```
select sum(orahash(id
                                        1,
                                        POWER(2,16)-1)
       + 2*orahash(name
                                        1,
                                        POWER(2,16)-1))
from A
 union all
select sum(orahash(id
                                        1,
                                        POWER(2,16)-1)
       + 2*orahash(name
                                        1,
                                        POWER(2,16)-1))
from B;
```

Row hash values are added together with ordinary sum aggregate, but we could have written a modulo 2^{16}-1 user-defined aggregate hash_sum in the spirit of CRC (Cyclic Redundancy Check) technique.

Acknowledgements

Many thanks to Rafi Ahmed for patiently answering my numerous questions about query transformation in Oracle RDBMS. The aggregation solution has been suggested by Marco Stefanetti in an exchange at Ask Tom forum.

About the Author

Vadim Tropashko graduated from Moscow Institute of Physics and Technology in 1984. Tropashko researched Petri Nets for five years following graduation. In the early ’90s, his interests switched to OOP. Tropashko translated The *C++ Programming Language* by B. Stroustrup into Russian. In the mid ’90s, Tropashko’s interest switched from OOP to databases. He published numerous research and programming articles and authored the book *SQL Design Patterns* (Rampant Tech Press).