Enhanced Subquery Optimizations in Oracle.

Article *in* Proceedings of the VLDB Endowment · August 2009

DOI: 10.14778/1687553.1687563 · Source: DBLP

Rafi Ahmed
Oracle Corporation

Andrew Witkowski
Oracle Corporation

Mohamed Zaït
Oracle Corporation

Enhanced Subquery Optimizations in Oracle

Srikanth Bellamkonda
Oracle USA
500 Oracle Parkway
Redwood Shores, CA, USA
Srikanth.Bellamkonda@oracle.com

Rafi Ahmed
Oracle USA
500 Oracle Parkway
Redwood Shores, CA, USA
Rafi.Ahmed@oracle.com

Andrew Witkowski
Oracle USA
500 Oracle Parkway
Redwood Shores, CA, USA
Andrew.Witkowski@oracle.com

Angela Amor
Oracle USA
500 Oracle Parkway
Redwood Shores, CA, USA
Angela.Amor@oracle.com

Mohamed Zait
Oracle USA
500 Oracle Parkway
Redwood Shores, CA, USA
Mohamed.Zait@oracle.com

Chun-Chieh Lin
Oracle USA
500 Oracle Parkway
Redwood Shores, CA, USA
Chun-Chieh.Lin@oracle.com

ABSTRACT

This paper describes enhanced subquery optimizations in Oracle relational database system. It discusses several techniques – subquery coalescing, subquery removal using window functions, and view elimination for group-by queries. These techniques recognize and remove redundancies in query structures and convert queries into potentially more optimal forms. The paper also discusses novel parallel execution techniques, which have general applicability and are used to improve the scalability of queries that have undergone some of these transformations. It describes a new variant of antijoin for optimizing subqueries involved in the universal quantifier with columns that may have nulls. It then presents performance results of these optimizations, which show significant execution time improvements.

1. INTRODUCTION

Current relational database systems process complex SQL queries involving nested subqueries with aggregation functions, union/union-all, distinct, and group-by views, etc. Such queries are becoming increasingly important in Decision-Support Systems (DSS) and On-Line Analytical Processing (OLAP). Query transformation has been proposed as a common technique to optimize such queries.

Subqueries are a powerful component of SQL, extending its declarative and expressive capabilities. The SQL standard allows subqueries to be used in SELECT, FROM, WHERE, and HAVING clauses. Decision support benchmarks such as TPC-H [14] and TPC-DS [15] extensively use subqueries. Almost half of the 22 queries in the TPC-H benchmark contain subqueries. Most subqueries are correlated and many contain aggregate functions. Therefore, efficient execution of complex subqueries is essential for database systems.

1.1 Query Transformation in Oracle

Oracle performs a multitude of query transformations – subquery unnesting, group-by and distinct view merging, common sub-expression elimination, join predicate pushdown, join factorization, conversion of set operators intersect and minus into [anti-] join, OR expansion, star transformation, group-by and distinct placement etc. Query transformation in Oracle can be heuristic or cost based. In cost-based transformation, logical transformation and physical optimization are combined to generate an optimal execution plan.

In Oracle 10g, a general framework [8] for cost-based query transformation and several state space search strategies were introduced. During cost-based transformation, a query is copied, transformed and its cost is calculated using existing cost-based physical optimizer. This process is repeated multiple times applying a new set of transformations; and at the end, one or more transformations are selected and applied to the original query, if it results in an optimal cost. The cost-based transformation framework provides a mechanism for the exploration of the state space generated by applying one or more transformations thus enabling Oracle to select the optimal transformation in an efficient manner. The cost-based transformation framework can handle the complexity produced by the presence of multiple query blocks in a user query and by the interdependence of transformations. The availability of the general framework for cost-based transformation has made it possible for other innovative transformations to be added to the vast repertoire of Oracle's query transformation techniques. This paper introduces new transformation techniques – subquery coalescing, subquery removal, and filtering join elimination.

1.2 Subquery Unnesting

Subquery unnesting [1][2][8][9] is an important query transformation commonly performed by database systems. When a correlated subquery is not unnested, it is evaluated multiple times using tuple iteration semantics. This is akin to nested-loop

VLDB '09, August 24-28, 2009, Lyon, France.

join, and thus efficient access paths, join methods and join orders cannot be considered.

Oracle performs unnesting of almost all types of subqueries. There are two broad categories of unnesting – one that generates derived tables (inline views), and the other that merges a subquery into the body of its outer query. In Oracle, the former is applied in a cost-based manner, while the latter is done in a heuristic fashion.

Unnesting of non-scalar subqueries often results in semijoin or antijoin. Oracle can use index lookup, hash, and sort-merge *semi* or *anti* join. Oracle execution engine caches the results of *anti* or *semi* joins for the tuples from the left table so that multiple evaluations of the subquery can be avoided when there are a large number of duplicates in the joining columns of the left table. Oracle unnests subqueries appearing in existentially or universally quantified non-equality comparison (e.g., > ANY, <ALL, etc.) by using sort-merge join on non-equality predicate in the absence of relevant indexes.

Subqueries with *nullable* columns in the universally quantified comparison (e.g., <> ALL) cannot be unnested using regular antijoin. Oracle uses a variant of antijoin, called *null-aware antijoin*, to unnest such subqueries.

1.3 Window Functions

SQL 2003 [11] extends SQL with *window functions*[1] that are not only easy and elegant to express, but also can lead to efficient query optimization and execution by avoiding numerous self-joins and multiple query blocks. Window functions are used widely by a number of analytic applications. Oracle has provided support for window functions since the version Oracle 8*i*. The syntax of window functions looks like the following:

```
Window_Function ([arguments]) OVER (
    [ PARTITION BY pk1 [, pk2, …] ]
[ ORDER BY ok1 [, ok2, …] [WINDOW clause] ] )
```

Window functions are evaluated within partitions defined by the PARTITION BY (PBY) keys pk_1, pk_2, etc. with data ordered within each partition on ORDER BY (OBY) keys ok_1, ok_2, etc. The WINDOW clause defines the *window* (begin and end points) for each row. SQL aggregate functions (sum, min, count, etc.), ranking functions (rank, row_number, etc.), or reference functions (lag, lead, first_value, etc.) can be used as window functions. ANSI SQL standard [10] [11] contains the details of the syntax and semantics of window functions.

Window functions in a query block are evaluated after WHERE, GROUP-BY, and HAVING clauses. Oracle computes a window function by sorting data on PBY and OBY keys and making passes over the ordered data as needed. We call this *window sort* execution. Obviously, sorting is not required when window function has no PBY and OBY keys. Oracle buffers data in this case to compute the window function and this is called *window buffer* execution.

Oracle's cost-based optimizer eliminates sort for window computation when it chooses a plan that produces data in the order of PBY and OBY keys. In this case, *window buffer* execution is used wherein Oracle just buffers the data and makes multiple passes over it to compute the window function. However, buffering can well be avoided for window functions like *rank*, *row_number*, *cumulative* window aggregates when data comes in order. By keeping some context information (window function value and PBY key values), these functions can be computed while the input data is being processed.

1.3.1 Reporting Window Functions

Subquery optimizations presented in this paper make use of a class of window functions called *reporting window* functions. These are window functions that, by virtue of their specification, report for each row the aggregated value of all rows in the corresponding partition (as defined by PBY keys). When a window function has no OBY and WINDOW clauses, or when the *window* for each row includes all the rows of the partition it belongs to, then it is a *reporting window* function. We sometimes refer these functions as *reporting aggregates* in this paper.

Reporting window functions are useful in comparative analysis wherein one can compare the value of a row at a certain level with that of one at higher level. For example, to compute for a stock ticker, the ratio of each day's volume to the overall volume, each row (at day level) needs to have the aggregated SUM across all days. The window function to get the aggregated SUM reported for all rows and the output would be like:

```
Q1
SELECT ticker, day, volume,
  SUM(volume) OVER (PARTITION BY ticker)
    AS "Reporting SUM"
FROM stocks;
```

Table 1. Reporting Window SUM Example

Ticker	Day	Volume	Reporting SUM
GOOG	02-Feb-09	5	18
GOOG	03-Feb-09	6	18
GOOG	04-Feb-09	7	18
YHOO	02-Feb-09	21	62
YHOO	03-Feb-09	19	62
YHOO	04-Feb-09	22	62

When a *reporting aggregate* has no PBY keys, then the value it reports is the grand total across all rows, as there is only one implicit partition. We call such reporting aggregates *grand-total* (GT) functions. Our subquery transformations in some cases introduce GT functions into the query.

2. SUBQUERY COALESCING

Subquery coalescing is a technique where two subqueries can be coalesced into a single subquery under certain conditions, thereby reducing multiple table accesses and multiple join evaluations to a single table access and a single join evaluation. Although subquery coalescing is defined as a binary operation, it can be successively applied to any number of subqueries. Subquery coalescing is possible, because a subquery acts like a filter predicate on the tables in the outer query.

Two query blocks are said to be semantically *equivalent*, if they produce the same multi-set results. The structural or syntactic identity of two query blocks can also establish their equivalence.

[1] Referred to as '*analytic functions*' in Oracle documentation

A query block X is said to *contain* another query block Y, if the result of Y is a (not necessarily proper) subset of the result of X. X is called *container* query block and Y is called *contained* query block. In other words, X and Y satisfy the containment property, if Y contains some conjunctive filter predicates P, and X and Y become equivalent when P is not taken into account in establishing their equivalence.

Containment is an important property that allows us to incorporate the behavior of both subqueries into the coalesced subquery. When two conjunctive subqueries violate the containment property, their filter predicates cannot be combined as a conjunction in a single subquery, since the subquery will then produce only the intersecting set of rows.

Currently, Oracle performs various types of subquery coalescing where two [NOT] EXISTS subqueries appear in a conjunction or in a disjunction. Since ANY and ALL subqueries can be converted into EXISTS and NOT EXISTS subqueries respectively, we do not discuss the coalescing of ANY/ALL subqueries here. In the trivial case of two subqueries that are equivalent and are of same type (i.e., either EXISTS or NOT EXISTS), subquery coalescing results in the removal of one of the two subqueries. When equivalent subqueries are of different types, coalescing removes both the subqueries and replaces them with a FALSE/TRUE predicate depending on whether the subqueries participate in conjunction or disjunction.

2.1 Coalescing Subqueries of the Same Type

When two conjunctive EXISTS subqueries or disjunctive NOT EXISTS satisfy the containment property, they can be coalesced into a single subquery by retaining the contained subquery and removing the container subquery. For the case of disjunctive EXISTS subqueries, or conjunctive NOT EXISTS subqueries, coalescing can be done by retaining the container subquery and removing the contained subquery.

Subqueries not satisfying containment property can also be coalesced when they are equivalent except for some conjunctive filter and correlation predicates. For example, two disjunctive EXISTS subqueries differing in conjunctive filter predicates and correlation predicates, but otherwise equivalent, can be coalesced into a single EXISTS subquery with a disjunction of additional (or differing) predicates originating from the two subqueries. Two conjunctive NOT EXISTS subqueries can be coalesced in a similar fashion.

Consider query Q2 with two disjunctive EXISTS subqueries; the subqueries have same correlation predicate but differ in conjunctive filter predicate.

```
Q2
SELECT o_orderpriority, COUNT(*)
FROM orders
WHERE o_orderdate >= '1993-07-01' AND
  EXISTS (SELECT *
          FROM lineitem
          WHERE l_orderkey = o_orderkey AND
                l_returnflag = ‘R’) OR
  EXISTS (SELECT *
          FROM lineitem
          WHERE l_orderkey = o_orderkey AND
                l_receiptdate > l_commitdate)
GROUP BY o_orderpriority;
```

Our subquery coalescing combines the two EXISTS subqueries into a single EXISTS subquery with a disjunction of filter predicates yielding query Q3.

```
Q3
SELECT o_orderpriority, COUNT(*)
FROM orders
WHERE o_orderdate >= '1993-07-01' AND
  EXISTS (SELECT *
          FROM lineitem
          WHERE l_orderkey = o_orderkey AND
             (l_returnflag = ‘R’ OR
              l_receiptdate > l_commitdate))
GROUP BY o_orderpriority;
```

2.2 Coalescing Subqueries of Different Types

Coalescing of two conjunctive subqueries that satisfy the containment property and are of different types requires a different technique. Consider query Q4, which is a somewhat simplified version of TPC-H query 21.

```
Q4
SELECT s_name
FROM supplier, lineitem L1
WHERE s_suppkey = l_suppkey AND
    EXISTS (SELECT *
            FROM lineitem L2
            WHERE l_orderkey = L1.l_orderkey
              AND l_suppkey <> L1.l_suppkey)
    AND NOT EXISTS
    (SELECT *
     FROM lineitem L3
     WHERE l_orderkey = L1.l_orderkey AND
           l_suppkey <> L1.l_suppkey AND
           l_receiptdate > l_commitdate);
```

The two subqueries in Q4 are different only in their types and in the fact that the NOT EXISTS subquery has an additional filter predicate, l_receiptdate > l_commitdate. Subquery coalescing yields query Q5 with a single EXISTS subquery, thereby eliminating one instance of the lineitem table.

```
Q5
SELECT s_name
FROM supplier, lineitem L1
WHERE s_suppkey = l_suppkey AND
   EXISTS (SELECT 1
           FROM lineitem L2
           WHERE l_orderkey =
                 L1.l_orderkey AND
                 l_suppkey <> L1.l_suppkey
           HAVING SUM(CASE WHEN
                         l_receiptdate >
                         l_commitdate
                   THEN 1 ELSE 0 END) = 0);
```

The aggregate function in the HAVING clause returns the total number of rows that satisfy the subquery predicate. The HAVING clause introduced in the coalesced subquery has a new filter predicate that checks if any rows satisfy the subquery predicate, thereby simulating the behavior of the removed NOT EXISTS subquery.

For each set of correlation values, the subqueries in Q4 can be in one of the following three states:

- When the EXISTS subquery produces no rows (i.e., it evaluates to FALSE), the conjunctive result of the two subqueries is FALSE. In Q5, the HAVING clause applies on an empty set and the coalesced EXISTS subquery also evaluates to FALSE.
- When the EXISTS subquery returns some rows (i.e., it evaluates to TRUE) and the NOT EXISTS subquery also returns some rows (i.e., it evaluates to FALSE), the conjunctive result of the two subqueries is FALSE. In Q5, the HAVING clause gets applied on a non-empty set of rows that have l_receiptdate > l_commitdate, and therefore it evaluates to FALSE; thus, the coalesced subquery evaluates to FALSE.
- When the EXISTS subquery produces some rows and the NOT EXISTS subquery produces no rows because of the additional filter predicates, the conjunctive result of the two subqueries is TRUE. In Q5, the HAVING clause gets applied on a non-empty set of rows that do not have l_receiptdate > l_commitdate, and therefore it evaluates to TRUE; thus, the coalesced subquery evaluates to TRUE.

The above discussion establishes that Q4 and Q5 are equivalent. In the case where the NOT EXISTS subquery is the container query and the conjunctive EXISTS subquery is the contained query, coalescing involves removing both subqueries and replacing them with a FALSE predicate. The conditions under which the EXISTS subquery produces some rows (i.e., it evaluates to TRUE) guarantees that the NOT EXISTS subquery also produces some rows (i.e., it evaluates to FALSE). In the case where the NOT EXISTS subquery produces no rows, the EXISTS subquery also returns no rows. Hence, the conjunctive result of the two subqueries is always FALSE.

Similar arguments can be made and coalescing can be done when the EXISTS and NOT EXISTS subqueries appear in a disjunction and satisfy the containment property.

2.3 Coalescing and Other Transformations

In [8], we discussed how various transformations interact with one another and how our cost-based transformation framework handles the possible interactions. Subquery coalescing is no exception, as the coalesced subquery may be subject to other transformations. The subquery in Q5 can undergo unnesting and yield query Q6, which contains an inline view (derived table) V.

```
Q6
SELECT s_name
FROM supplier, lineitem L1,
     (SELECT LX.rowid xrowid
      FROM lineitem L2, lineitem LX
      WHERE L1.l_suppkey <> LX.l_suppkey AND
            L1.l_orderkey = LX.l_orderkey
      GROUP BY LX.rowid
      HAVING SUM(CASE WHEN
                   L2.l_receiptdate >
                   L2.l_commitdate
                 THEN 1 ELSE 0 END) = 0) V
WHERE s_suppkey = L1.l_suppkey AND
      L1.rowid = V.xrowid;
```

After view merging, Q6 yields query Q7, which renders the table LX redundant, since in the merged query block LX and L1 are joined on the unique *rowid* column. Therefore, LX is removed, and all references to it are replaced with that of L1.

```
Q7
SELECT s_name
FROM supplier, lineitem L1, lineitem L2
WHERE s_suppkey = L1.l_suppkey AND
      L1.l_orderkey = L2.l_orderkey
GROUP BY L1.rowid, S.rowid, S.s_name
HAVING SUM(CASE WHEN L2.l_receiptdate >
                     L2.l_commitdate
                THEN 1 ELSE 0 END) = 0);
```

Here we have at least four alternative queries. In most cases, it will not be clear which one of the four alternatives provides the optimal choice. The Oracle cost-based transformation framework discussed in Section 1.1 can be used to make this decision.

2.4 Query Execution Enhancements

The HAVING clause predicate of query Q7 filters out groups that have at least one record with receipt date greater than commit date. This predicate and other predicates such as MIN(l_receiptdate) > '18-Feb-2001', COUNT(*) <= 10, SUM(amount_sold) < 2000[2], etc., when not satisfied, immediately make a group unacceptable (i.e., not a candidate in the result set) and can be pushed into group-by to short circuit aggregate processing for that group. This results in efficient execution. For example in Q7, an input record with l_receiptdate > l_commitdate makes the SUM aggregate's value 1 for that group, thus making the group a non-candidate. Similarly, when the predicate is SUM(amount_sold) < 2000 and there is a database rely constraint that specifies that amount_sold is positive, a group becomes a non-candidate as soon as SUM for that group exceeds 2000. Group-by skips processing of aggregates for non-candidate groups.

Parallel group-by execution also benefits when such predicates are used to reduce data traffic. Oracle employs cost-based parallel *group-by pushdown* (GPD) technique where in group-by evaluation is pushed to the processes producing the input (producer slaves) so as to reduce communication costs and to improve the scalability of group-by. The producer slaves distribute locally aggregated data to another set of processes (consumer slaves) by hash/range on the group-by keys. The consumer slaves then finish group-by processing and produce results. The parallel query plan with GPD for Q7 is shown in Figure 1. Reduction in data traffic is achieved when producer slaves $P_{1\ldots}$ P_N filter out groups based on the HAVING predicate during group-by processing.

Similarly, predicates that immediately make groups result-set candidates as soon as they are satisfied can be pushed into group-by processing as well. Processing of aggregates that are not part of the result set can be skipped once a group is found to be a candidate. Examples for such predicates are MIN(l_receiptdate) < '18-Feb-2001', COUNT(*) > 10, SUM(amount_sold) > 2000, when amount_sold is known to be positive.

[2] When *amount_sold* is known to be positive; for example, in the presence of a database rely constraint.

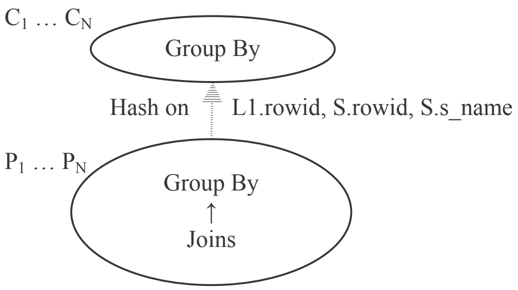

Figure 1. Parallel Group-By Pushdown

3. GROUP-BY VIEW ELIMINATION

In this section, we discuss a technique called *filtering table elimination*, which is based on the idea of filtering joins. A *filtering join* is either a semijoin or an equi-inner join that takes place on a unique column of one of the two tables involved in the join.

Here we denote a unique column by underlining it and represent equi-semijoin in a non-standard notation, θ=. R is a table and T1 and T2 are two instances of the same base or derived table T. T1 and T2 either have identical set of filters predicates, if any, or the filter predicates of T1 are more restrictive than that of T2. In the following cases, T2 and the filtering join can be eliminated.

$$R.X = T1.\underline{Y} \text{ and } R.X = T2.\underline{Y} \equiv R.X = T1.\underline{Y}$$

$$R.X = T1.Y \text{ and } R.X\ \theta{=}\ T2.Y \equiv R.X = T1.Y$$

$$R.X\ \theta{=}\ T1.Y \text{ and } R.X\ \theta{=}\ T2.Y \equiv R.X\ \theta{=}\ T1.Y$$

Assume that the non-filtering join, if any, takes place first. The filtering join then retains all the resulting rows of R, since a filtering join can only filter out rows of R in contrast with inner join that can duplicate as well as filter out rows. The filtering join renders table T2 redundant and therefore T2 can be removed. Although, this technique bears a close resemblance to that of coalescing of conjunctive EXISTS subqueries, in the following we present a different application of this technique.

3.1 View Elimination

Consider Q8, which is a simplified version of TPC-H query 18.

```
Q8
SELECT o_orderkey,c_custkey, SUM(l_quantity)
FROM orders, lineitem L1, customers
WHERE o_orderkey = l_orderkey AND
      c_custkey = o_custkey AND
      o_orderkey IN
        (SELECT l_orderkey
         FROM lineitem L2
         GROUP BY l_orderkey
         HAVING SUM(l_quantity) > 30)
GROUP BY o_orderkey, o_totalprice;
```

The subquery in Q8 undergoes unnesting yielding Q9. The inline view (derived table) V2 generated by unnesting in Q9 need not be semi-joined, since it is an equi-join and the joining column of V2 is unique as a result of being the only grouping column of V2.

```
Q9
SELECT o_orderkey,c_custkey, SUM(l_quantity)
FROM orders, lineitem L1, customers,
  (SELECT l_orderkey
   FROM lineitem L2
   GROUP BY l_orderkey
   HAVING SUM(l_quantity) > 30) V2
WHERE o_orderkey = V2.l_orderkey AND
      o_orderkey = L1.l_orderkey AND
      c_custkey = o_custkey
GROUP BY o_orderkey, c_custkey;
```

Using group-by and join permutation (i.e., group-by placement) [5][6][8], another view V1, which contains table L1, can be generated, as shown Q10; SUM(l_quantity) is added to V2's SELECT list without changing the semantics of Q9.

```
Q10
SELECT o_orderkey, c_custkey, SUM(V1.qty)
FROM orders, customers,
  (SELECT l_orderkey, SUM(l_quantity) qty
   FROM lineitem L2
   GROUP BY l_orderkey
   HAVING SUM(l_quantity) > 30) V2,
  (SELECT l_orderkey, SUM(l_quantity) qty
   FROM lineitem L1
   GROUP BY l_orderkey) V1
WHERE o_orderkey = V1.l_orderkey AND
      o_orderkey = V2.l_orderkey AND
      c_custkey = o_custkey
GROUP BY o_orderkey, c_custkey;
```

As can be seen, V1 and V2 are different instances of the same view, except that the filter predicates in V2 are more restrictive than that of V1, because of the presence of a HAVING clause in V2. Furthermore, the equi-joins of V1 and V2 with orders are on the unique column o_orderkey, since it is the only grouping column in the views; thus, these two joins are filtering joins. Therefore V1 can be eliminated and references to V1 can be replaced with that of V2. Elimination of the filtering view in Q10 results in Q11.

```
Q11
SELECT o_orderkey, c_custkey, SUM(V2.qty)
FROM orders, customers,
     (SELECT o_orderkey, SUM(l_quantity)
      FROM lineitem
      GROUP BY l_orderkey
      HAVING SUM(l_quantity) > 30) V2,
WHERE o_orderkey = V2.l_orderkey AND
      c_custkey = o_custkey
GROUP BY o_orderkey, c_custkey;
```

If the view V2 in query Q9 was merged, a different line of argument using filtering join can be proffered with the same result of eliminating the table lineitem from the outer query.

4. SUBQUERY REMOVAL USING WINDOW FUNCTIONS

This technique replaces subqueries with window functions [11] thereby reducing the number of table accesses and join evaluations and improving the query performance. Some of the techniques of subquery removal discussed here were introduced in Oracle 9i and some were published in literature [13]. In its simpler form *subsumed* aggregation subqueries are removed using window functions.

An outer query block is said to *subsume* a subquery when it contains all the tables and predicates appearing in the subquery. The outer query block may have additional tables and predicates. Clearly, the property of *subsumption* is different from that of *containment* discussed in Section 2. In addition, this technique uses *lossless* join property and *algebraic* aggregates (e.g., SUM, MIN, MAX, COUNT, AVG etc.).

Q12 represents a form of a query with subsumed aggregation subquery for which subquery removal is applicable. T1 and T2 are base or derived tables or they may represent join of multiple tables. The aggregate AGG in the subquery participates in a relational comparison (relop) with a column T2.z from the outer query and the correlation is on column T1.y.

```
Q12
SELECT T1.x
FROM T1, T2
WHERE T1.y = T2.y and
      T2.z relop (SELECT AGG(T2.w)
                  FROM T2
                  WHERE T2.y = T1.y);
```

Assuming that the join T1 and T2 is a lossless join where T2.y is the foreign key that refers to the primary key T1.y, the subquery can be removed by introducing a window function that has the correlation column as the partition-by key. This is shown in Q13.

```
Q13
SELECT V.x
FROM (SELECT T1.x, T2.z,
        AGG (T2.w) OVER (PARTITION BY T2.y)
          AS win_agg
      FROM T1, T2
      WHERE T1.y = T2.y) V
WHERE V.z relop win_agg;
```

The join between T1 and T2 is not *required* to be lossless to be able to do subquery removal using window functions. However, lossless join leads to a transformation that allows more join permutations to be considered by the optimizer.

Variations to the above form Q12 where subquery is uncorrelated, or has additional tables and predicates, or doesn't have aggregates, or when the subquery and outer query have group-by can be transformed using subquery removal technique. We will give examples for these in subsequent sections.

4.1 Correlated Subsumed Subquery

Consider query Q14, which is a simplified version of TPC-H query 2. The outer query has an additional table PARTS and a filter predicate on that table. The subquery is correlated to the PARTS table and is subsumed by the outer query.

```
Q14
SELECT s_name, n_name, p_partkey
FROM parts P, supplier, partsupp,
     nation, region
WHERE p_partkey = ps_partkey AND
      s_suppkey = ps_suppkey AND
      s_nationkey = n_nationkey AND
      n_regionkey = r_regionkey AND
      p_size = 36 AND
      r_name = 'ASIA' AND
      ps_supplycost IN
     (SELECT MIN (ps_supplycost)
     FROM partsupp, supplier, nation,
          region
     WHERE P.p_partkey = ps_partkey AND
           s_suppkey = ps_suppkey AND
           s_nationkey = n_nationkey AND
           n_regionkey = r_regionkey AND
           r_name = 'ASIA');
```

Subquery removal technique transforms the query Q14 into Q15:

```
Q15
SELECT s_name, n_name, p_partkey
FROM (SELECT ps_supplycost,
            MIN (ps_supplycost) OVER
              (PARTITION BY ps_partkey)
                AS min_ps,
            s_name, n_name, p_partkey
      FROM parts, supplier, partsupp,
           nation, region
      WHERE p_partkey = ps_partkey AND
            s_suppkey = ps_suppkey AND
            s_nationkey = n_nationkey AND
            n_regionkey = r_regionkey AND
            p_size = 36 AND
            r_name = 'ASIA') V
WHERE  V.ps_supplycost = V.min_ps;
```

Duplicate rows, if any, generated by the join of PARTSUPP and PARTS tables in Q15 are not relevant as the aggregate function is MIN. If the aggregate function were not MIN/MAX, or if the join with the additional table (PARTS in this case) was not *lossless*, then the window function computation must be done within a view, which then is joined with the additional table. This is the case for TPC-H query 17, for which subquery removal transformation yields Q16:

```
Q16
SELECT SUM(V.avg_extprice)/7 AS avg_yearly
FROM parts,
    (SELECT (CASE WHEN l_quantity < (1.2 *
                  AVG (l_quantity) OVER
                  (PARTITION BY l_partkey))
                  THEN l_extprice ELSE NULL
                  END) avg_extprice,
           l_partkey
     FROM lineitem) V
 WHERE p_partkey = V.l_partkey AND
       V.avg_extprice IS NOT NULL AND
       P_brand = 'Brand#23' AND
       p_container = 'MED BOX';
```

4.2 Uncorrelated Subsumed Subquery

Consider query Q17, which is a simplified version of TPC-H query 15. Q17 has an uncorrelated aggregation subquery with the outer query and the subquery referring to an identical group-by view (derived table) V.

```
Q17
WITH V AS (SELECT l_suppkey,
             SUM(l_extprice) revenue
           FROM lineitem
           WHERE l_shipdate >= '1996-01-01'
           GROUP BY l_suppkey)
SELECT s_suppkey, s_name, V.revenue
FROM supplier, V
WHERE s_suppkey = V.s_suppkey AND
      V.revenue = (SELECT MAX(V.revenue)
                   FROM V);
```

The above query can be transformed as Q18, where a window function has been introduced and the subquery has been eliminated.

```
Q18
SELECT s_suppkey, s_name, V.revenue
FROM supplier,
     (SELECT l_suppkey,
        SUM(l_extprice) revenue
        MAX(SUM(l_extprice)) OVER() gt_rev
      FROM lineitem
      WHERE l_shipdate >= '1996-01-01'
      GROUP BY l_suppkey) V
WHERE s_suppkey = V.l_suppkey AND
      V.revenue = V.gt_rev;
```

In this case a *grand-total* window function MAX (specified using empty OVER ( ) clause) on the aggregate SUM(l_extprice) is introduced to remove the subquery. There was no need to have PBY for the window function as the subquery in Q17 is uncorrelated and needs to be applied on the entire set of rows. We employ a novel parallelization technique for *grand-total* window functions, as described in Section 5, so that the transformed query Q18 can perform efficiently and in scalable fashion.

4.3 Subsumed Subquery in Having Clause

Subquery removal technique can also be employed when the outer query has group-by. For example consider Q19, a simplified version of TPC-H query 11. The subquery in Q19 is uncorrelated and is subsumed by the outer query. In fact, the subquery and the outer query have an identical set of tables and predicates.

```
Q19
SELECT ps_partkey,
  SUM(ps_supplycost * ps_availqty) AS value
FROM  partsupp, supplier, nation
WHERE ps_suppkey = s_suppkey AND
      s_nationkey = n_nationkey AND
      n_name = 'FRANCE'
GROUP BY  ps_partkey
HAVING SUM(ps_supplycost * ps_availqty) >
      (SELECT SUM(ps_supplycost *
                  ps_availqty) * 0.0001
       FROM partsupp, supplier, nation
       WHERE ps_suppkey = s_suppkey AND
             s_nationkey = n_nationkey AND
             n_name = 'FRANCE');
```

Q19 can be transformed into Q20. As with Q17, the window function introduced was a *grand-total* without PBY keys as the subquery in Q19 is uncorrelated.

```
Q20
SELECT V.ps_partkey, V.gb_sum
FROM (SELECT ps_partkey,
      SUM(ps_supplycost*ps_availqty) value,
      SUM(SUM(ps_supplycost*ps_availqty))
           OVER () gt_value
      FROM partsupp, supplier, nation
      WHERE ps_suppkey = s_suppkey AND
            s_nationkey = n_nationkey AND
            n_name = 'FRANCE'
      GROUP BY ps_partkey) V
WHERE V.value > V.gt_value * 0.0001;
```

4.4 Subquery Producing a Multi-Set

Subquery need not necessarily have an aggregate and produce a singleton set to be able to perform subquery removal using window functions. Consider the query Q21 in which the subquery produces a multiset and participates in an "ALL" subquery predicate.

```
Q21
SELECT ps_partkey, s_name,
   SUM(ps_supplycost * ps_availqty) as VALUE
FROM partsupp, supplier, nation
WHERE ps_suppkey = s_suppkey AND
      s_nationkey = n_nationkey AND
      n_name = ‘GERMANY’
GROUP BY s_name, ps_partkey
HAVING
   SUM(ps_supplycost * ps_availqty) > ALL
     (SELECT
        ps_supplycost*ps_availqty * 0.01
      FROM partsupp, supplier, nation
      WHERE n_name = ‘GERMANY’ AND
            ps_suppkey = s_suppkey AND
            s_nationkey = n_nationkey);
```

We transform this query into Q22:

```
Q22
SELECT ps_partkey, s_name, VALUE
FROM (SELECT ps_partkey, s_name, VALUE,
        SUM(ps_supplycost * ps_availqty)
          as VALUE,
        MAX(MAX(ps_supplycost*ps_availqty))
          OVER ( ) VAL_pkey
      FROM partsupp, supplier, nation
      WHERE n_name = ‘GERMANY’ AND
            ps_suppkey = s_suppkey AND
            s_nationkey = n_nationkey
      GROUP BY s_name, ps_partkey) V
WHERE V.VALUE > V.VAL_pkey * 0.01;
```

If the subquery predicate instead was "> ANY", then the window function would be MIN (MIN (ps_supplycost * ps_availqty)) OVER ( ). Using multiple window functions, "= ALL" and "= ANY" predicates can be handled as well.

5. SCALABLE PARALLEL EXECUTION

Oracle parallelization of window functions has been enhanced for more scalable query execution. Normally, window functions are parallelized in Oracle by distributing the data, either by hash or range, to multiple processes based on the partition-by (PBY) keys. Similar parallelization is used for SQL model clause [7]. Each process works independently of other processes to compute the window function on the partitions it receives. For example, window function introduced in Q16 by the subquery removal technique is parallelized this way. Parallel query plan for Q16 would look like:

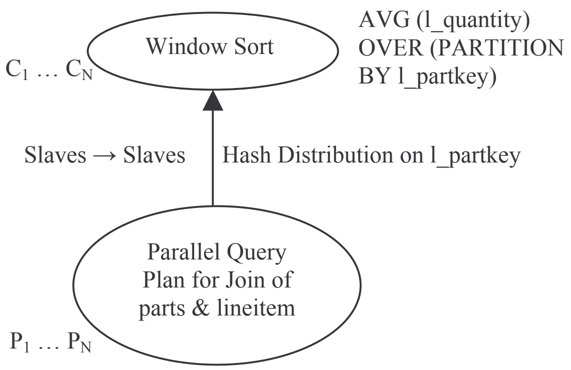
AVG (l_quantity) OVER (PARTITION BY l_partkey)
Window Sort
$C_1 \ldots C_N$
Slaves → Slaves
Hash Distribution on l_partkey
Parallel Query Plan for Join of parts & lineitem
$P_1 \ldots P_N$

Figure 2. Typical Parallelization of Window Function

Producer slave processes P_1 through P_N distribute, by hash on *l_partkey,* the result of the join between *parts* and *lineitem* tables to the consumer slave processes C_1 through C_N performing the *window sort.* Each consumer slave would compute the AVG window function on the partitions (defined by the PBY key *l_partkey*) it receives. Observe that the scalability of this regular parallelization of window functions is governed by the cardinality of the PBY keys. For window functions with low-cardinality PBY keys (e.g., *region*, *gender*) and ones with no PBY keys, scalability is either limited or nonexistent. Scalability of such window functions became critical for the subquery removal with window function transformation of Section 4 to pay high dividends. To that end, we came up with a novel parallelization technique, which we outline now.

Consider the transformed query Q20 of Section 4.3 that has a window function without PBY keys SUM(SUM(ps_supplycost * ps_availqty)) OVER(). This is a *grand-total* (GT) reporting window function as it operates on the entire dataset and reports the grand total for each row. GT functions are not intrinsically parallelizable as there are no PBY keys on which the work can be split among slave processes. If this GT function is not parallelized, it reduces the overall benefit of the subquery removal transformation. Normal parallel query plan for GT window functions is given in Figure 3.

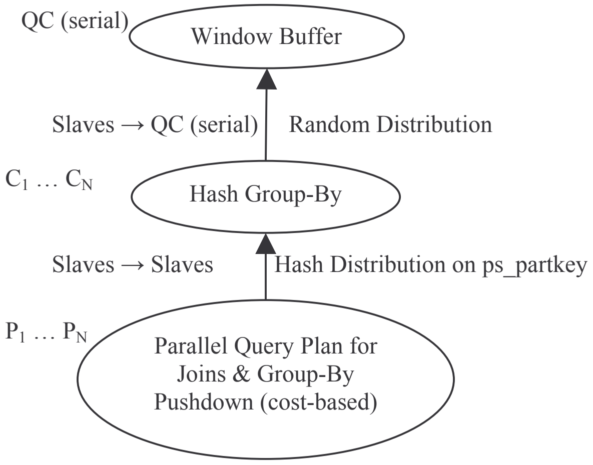

Figure 3. Not-So-Parallel Plan for GT Functions

Obviously, the parallel plan of Figure 3 will not scale well as the GT window function computation is done by the Query Coordinator (QC) process. Slaves C_1 through C_N send the result of the group-by operation to the QC, which alone computes the GT function using the "*window buffer*" execution. As mentioned in Section 1.3, Oracle picks "*window buffer*" in this case since it just has to buffer the rows to compute the GT window function. The window aggregate will be incrementally computed as rows are being buffered. When the input is exhausted i.e., when all input rows are buffered, we would have computed the *grand-total* window function value. The buffered rows are then output with the *grand-total* value.

In our new scheme to parallelize GT window function, a major portion of the GT window computation is pushed into slave processes rather than being done by the QC. It has a small coordination step between the slaves and the QC to finish the GT computation. With this model, GT evaluation becomes highly scalable. The new parallelization plan for GT window functions is shown in Figure 4.

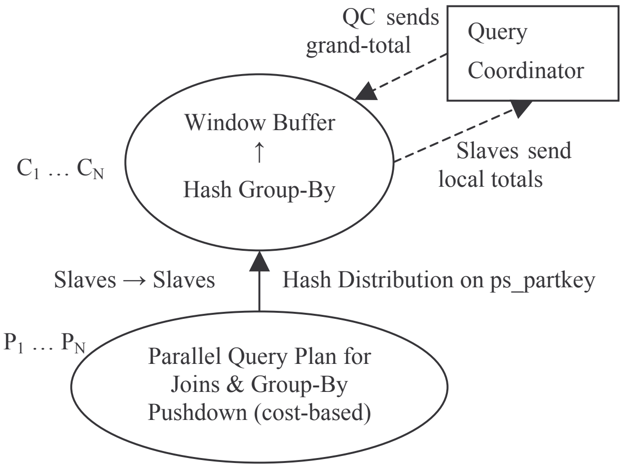

Figure 4. Parallelization of GT Functions

Window buffer processing is now pushed into slave processes C_1 through C_N. Each of these slaves computes a local total and communicates it to the query coordinator process. The QC consolidates the results it receives from all slaves and sends the *grand-total* value back to the slaves. The slaves then output their buffered rows with the *grand-total* values. With this parallel plan, the slaves perform the majority of row processing concurrently. Serialization (or QC consolidation) will be unnoticeable since the number of values QC processes would be small (at a maximum of 1000) as dictated by the degree of parallelism (number of processes concurrently working on a task).

This window pushdown parallelization technique can be extended to non-GT reporting window functions as well. Specifically, it can be used to improve the scalability of window functions with low cardinality PBY keys. Though the concept is the same, more information needs to be exchanged between slaves and QC, and more processing is needed on both sides.

The slave processes C_1 through C_N compute locally the reporting window aggregates for each partition and communicate to the QC, an array of local window aggregates and the corresponding PBY key values. The QC finishes the computation of reporting aggregates for each partition and communicates results (final window aggregates and PBY key values) back to the slaves. To produce results, window execution inside slaves does a join of the local data with the data sent by the QC. As slave local data as well as the data sent by QC are ordered by PBY keys, this join is like a sort-merge join. We present results of our experiments to showcase window function scalability in Section 7.

6. NULL-AWARE ANTI JOIN (NAAJ)

In this section, we discuss a variant of antijoin, called *null-aware antijoin* (NAAJ), introduced in Oracle 11g. Most applications issue <>ALL (i.e., NOT IN) subqueries quite commonly as application developers find the <>ALL syntax more intuitive than its *near* equivalent NOT EXISTS. Subqueries with NOT EXISTS are unnested into antijoin. The semantics of antijoin is exactly the opposite of inner join since a row from the left table is returned only if it does not join with any row from the right table. We call this *regular antijoin*. In general, commercial databases can employ regular antijoin to unnest <>ALL subqueries only when all the columns in the quantified comparison are guaranteed to have *non-null* values. Another strategy used in [9] involves introducing a duplicate table and an additional antijoin to deal with NULLs.

In SQL, the <> ALL operator can be treated as a conjunction of inequalities. The operators < ALL, <= ALL, > ALL and >= ALL can be similarly expressed. The SQL Standard supports ternary logic and hence any relational comparison with null values always evaluates to UNKNOWN. For example, the predicates 7 = NULL, 7 <> NULL, NULL = NULL, NULL <> NULL, evaluate to UNKNOWN, which is different from FALSE in that its negation is also UNKNOWN. If the final result of the WHERE clause is either FALSE or UNKNOWN, the row is filtered out. Consider query, Q23 which has a <> ALL subquery.

```
Q23
SELECT T1.c
FROM T1
WHERE T1.x <> ALL (SELECT T2.y
                   FROM T2
                   WHERE T2.z > 10);
```

Suppose the subquery returns the following set of values {7, 11, NULL} and T1.x has the following set of values: {NULL, 5, 11}. The <> ALL operation can be expressed as T1.x <> 7 AND T1.x <> 11 AND T1.x <> NULL. This evaluates to UNKNOWN, since T1.x <> NULL always evaluates to UNKNOWN irrespective of the value of T1.x. Thus, for this set of values, Q23 will return no rows. Regular antijoin, if used in this case, will incorrectly return {NULL, 5}.

6.1 Null-Aware Antijoin Algorithm

The subquery in Q23 can be unnested using NAAJ as shown in the query Q24. We use the following non-standard notation to represent NAAJ: T1.x NA= T2.y, where T1 and T2 respectively are tables on the left and right of the null-aware antijoin.

```
Q24
SELECT T1.c
FROM T1, T2
WHERE T1.x NA= T2.y and T2.z > 10;
```

We explain the semantics of NAAJ using Q24 as an example. NAAJ is performed after the evaluation of all filter predicates on the right table. So, when we say T2 in the following explanation, it implies the dataset obtained by applying the predicate T2.z > 10 on the original table T2.

1. If T2 contains no rows, then return all the rows of T1 and terminate.
2. If any row of T2 contains null values in all the columns involved in the NAAJ condition, then return no rows and terminate.
3. If a row of T1 contains null values in all the columns involved in the NAAJ condition, then do not return the row.
4. For each row of T1, if the NAAJ condition evaluates to TRUE or UNKNOWN using any row of T2, then do not return the T1 row; else return the T1 row.

Step 1 is identical to that of regular antijoin; that is, if the right side is empty, then all of the rows of the left side, including those having null values in the antijoin condition, are returned. Observe that steps 2 and 3 are subsumed by the step 4, but they provide efficient execution when *all* the join columns on the left or right row contain nulls. Step 4 essentially distinguishes NAAJ from regular antijoin. While in regular antijoin, a row on the left is returned if the antijoin condition evaluates to UNKNOWN, in NAAJ it is not. Next we present execution strategies for NAAJ. The strategies are complex for antijoin involving multiple columns and we illustrate them using this query:

```
Q25
SELECT c1, c2, c3
FROM L
WHERE (c1, c2, c3) <> ALL (SELECT c1, c2, c3
                           FROM R);
```

6.2 Execution Strategies for NAAJ

In NAAJ semantics, a row from the left side can join with or match several different rows on the right side. For example, a row from the left that has a null value in one of the join columns will match rows from the right that have any value in that particular key column. In this case the NAAJ condition evaluates to UNKNOWN and hence the row is not returned. Consider row (null, 3, null) from left table L in Q25. Assume that right table R has two rows R={(1, 3, 1), (2, 3, 2)}. Although there is no (null, 3, null) row in R, the row in L matches both rows of R because the non-null key column c2 has value 3, and consequently (null, 3, null) is not returned.

The regular sort-merge and hash antijoin methods are extended wherein they gather information about which join columns contain NULL values as the data structure (sort or hash table) is being built.

Following that we perform steps 1 and 2 given in Section 6.1 for early termination of the join by either returning all rows or no rows. Otherwise, for each row from the left side, we do the following to find a matching row from the right unless step 3 given in Section 6.1 eliminates the row. If a match is found using any of the following three steps, then the left side row is discarded as in a regular antijoin:

1. Search the sort or hash access structure on the right for an exact match.
2. Using the gathered information about nulls, search for other possible matches. For example, suppose there are three join columns c1, c2, and c3 but only c1and c2 have null values on the right side. If the incoming row from the left has values (1, 2, 3), then we search for (1, null, 3), (null, 2, 3), and (null, null, 3) in the access structure on the right.
3. If the row from the left has a null value in one or more of the key columns, then we build a secondary access structure, if it has not already been built, on the non-null key columns. For example, suppose the row from the left has values (null, 2, 3). We build a secondary sort or hash table using columns c2 and c3 on the right side, and then we search for (x, 2, 3), (x, 2, null), (x, null, 3), and (x, null, null) in the new access structure.

Further optimizations are possible if we keep track of all the possible patterns of nulls on the right side. Using the example given in step 3 above, suppose we know that there is a row on the right side that has null values in both c2 and c3. This row will match any row on the left side where c1 is null. In this case, the row (null, 2, 3) from the left side can be eliminated immediately, and there is no need to build the extra access structure. This information can also be used to eliminate some of the accesses done in step 2.

Single-Key Column Optimization: If there is only one column involved in the NAAJ condition (e.g., Q24), the execution strategy is much simpler. Rows from the left side that have a null value in the join column can be skipped or returned depending on whether the right side has any rows or not, without actually looking for matches.

7. PERFORMANCE STUDY

We conducted performance experiments on a 30G TPC-H schema. We used a Linux machine with 8 dual-core 400MHz processors and 32GB of main memory. The machine was connected to a shared storage managed by the Oracle Automated Storage Manager. The I/O bandwidth to the shared storage was somewhat limited compared to the power of CPUs and it showed in our parallelization experiments. All queries unless otherwise noted use parallelism provided by all CPUs.

7.1 Subquery Coalescing

Consider our Q4 that is a simplified version of TPC-H query 21. Compared to Q4, the original TPC-H query 21 has two additional tables orders and nation and a selection predicate that restricts data to a given nation. There are 25 nations in the schema and their data is uniformly distributed. Subquery coalescing of Section 2.2 transforms Q4 into Q5. Figure 5 shows the elapsed times for the transformed (Q5) and untransformed (Q4) versions as a function of the number of nations. On an average, the performance improvement was 27%.

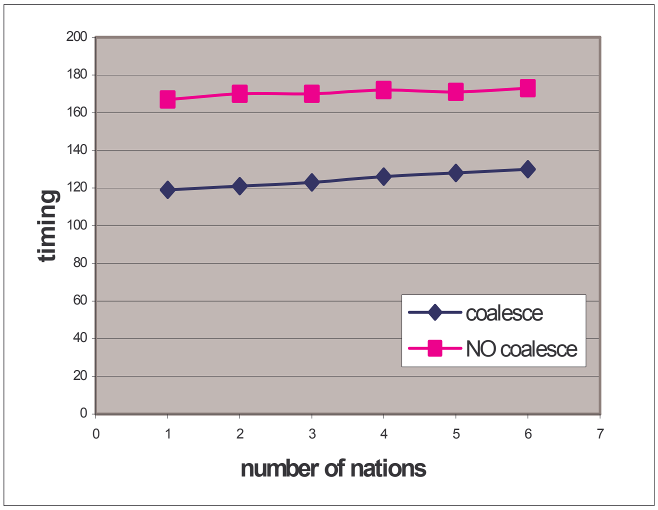

Figure 5. TPC-H q21, Subquery Coalescing

7.2 Group-By View Elimination

For this experiment, we used Q8, a simplified version of TPC-H query 18. Our Group-By view elimination technique of Section 3.1 eliminated the Group-By view, avoided unnecessary references to the lineitem table and resulted in Q11. We varied the HAVING clause predicate from 30 to 300 so that the subquery returns about 2,000 to 34,000,000 rows (as depicted on X-axis). Figure 6 shows the result of this experiment.

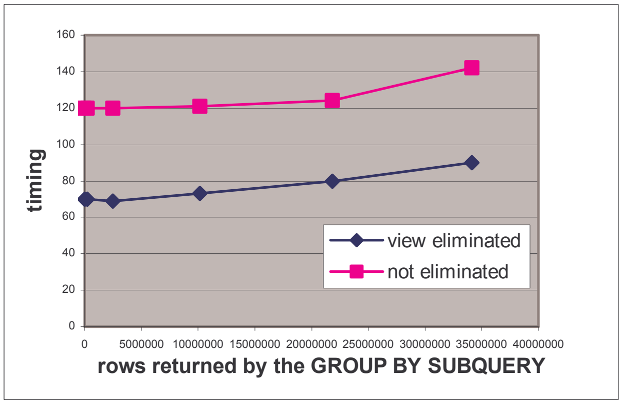

Figure 6. TPC-H q18, View Elimination

The transformation into Q11 can be challenging for the optimizer as the subquery HAVING clause predicate is on an aggregate and it is difficult for the optimizer to estimate the cardinality. If it is under-estimated, the optimizer may choose nested loops rather than hash join, degrading the performance.

7.3 Subquery Removal with Window Function

To illustrate this optimization we executed TPC-H queries 2, 11, 15 and 17. Figure 7 shows the elapsed time for optimized (marked with "*opt*") and non-optimized queries. The transformed queries Q15 and Q16 of TPC-H queries 2 and 17, showed pronounced improvement. For TPC-H query 2, the elapsed time got reduced by 24 times, the reason being that multiple table accesses and join evaluations have been eliminated by window transformation.

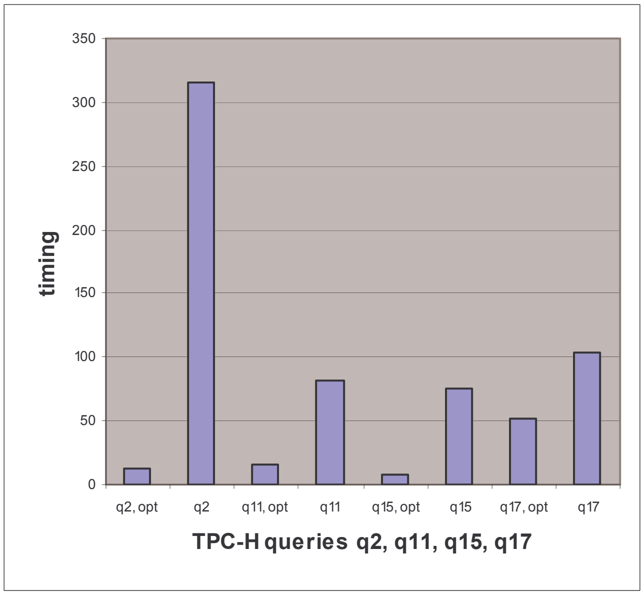

Figure 7. Subquery Removal Using Window Functions

Figure 8 shows the benefit of removing the uncorrelated subquery with a window transformation for the TPC-H query 15 as a function of the number of months scanned from the lineitem table. The optimization is explained in Section 4.2 as a transition from Q17 (original query) to Q18 (transformed query). The average benefit in Figure 8 is reduction of execution time by 8.4 times.

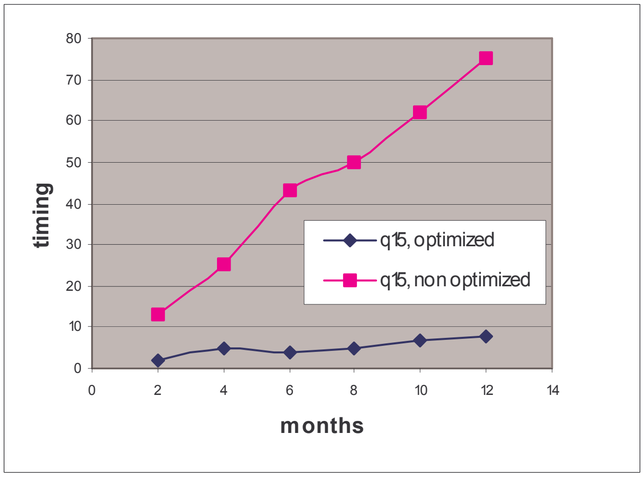

Figure 8. TPC-H q15, Removing Uncorrelated Subquery

It is noted that subquery removal using window function is very effective for the TPC-H queries. The average gain for TPC-H queries q2, q11, q15, and q17 is over 10 times.

7.4 Scalable Execution of Window Functions

To illustrate the benefit of parallelizing window functions without PBY clause, we used the following query on lineitem table.

```
SELECT SUM(l_extprice)) OVER() W
FROM lineitem;
```

The query is executed with and without the parallelization enhancement of Section 5, varying the degree of parallelism (i.e., DOP) from 2 to 16. Figure 9 shows the result of this experiment.

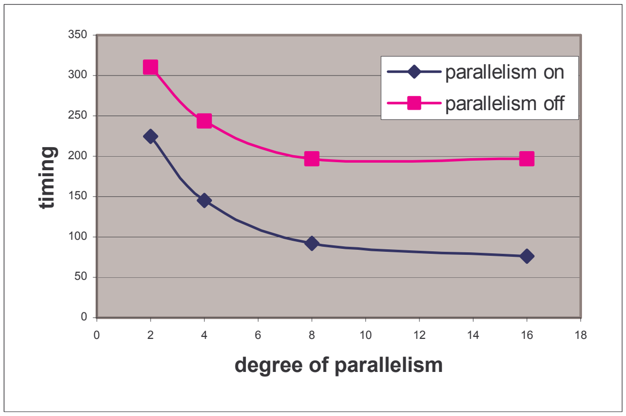

Figure 9. Parallelization of Window Function without PBY

Note that even when the window function is not parallelized, scan of the table still happens in parallel. The scanning slaves send their data to the query coordinator, which then computes the window function. Hence for degree of parallelism 2, the improvement is slightly less than 2. Figure 9 also shows that our system did not scale linearly with DOP for DOP > 8 due to limited bandwidth of our shared disk system.

7.5 Null-Aware Anti Join

We have conducted two sets of experiments to demonstrate the improvements provided by null-aware antijoin. The first uses query Q26 to find suppliers, from a given supplier list, that had no orders in a given month (January 1996). In this scheme, L_SUPPKEY may have null values, but S_SUPPKEY may not.

```
Q26
SELECT s_name FROM supplier
WHERE s_suppkey in (<supplier_list>) AND
      s_suppkey <> ALL
       (SELECT l_suppkey FROM lineitem
        WHERE l_shipdate >= '1996-01-01' AND
              l_shipdate <  '1996-02-01')
```

Without NAAJ, the subquery in Q26 cannot be unnested and this results in a correlated execution where for each row from supplier table, we have to execute the subquery. The performance is agonizingly slow since no index probe for the correlated predicate can be used. This is because Oracle converts the ALL subquery into a NOT EXISTS subquery and uses a correlation predicate that is equivalent to (l_suppkey IS NULL OR l_suppkey = s_suppkey). The only saving grace for non-NAAJ execution is the partitioning of lineitem table by l_shipdate that allows us to prune the scan to one partition. Figure 10 illustrates the performance gain for up to 40 suppliers. When unnested, the query is executed using hash null-aware antijoin between supplier and lineitem tables.

The second experiment is on a real workload – 241,000 queries issued by Oracle Applications, whose schema consists of about 14,000 tables representing Supply Chain, Human Resources, Financial, Order Entry, CRM, and many others. Most of the applications have highly normalized schemas. The number of tables in a query varies between 1 and 159, with an average of 8 tables per query.

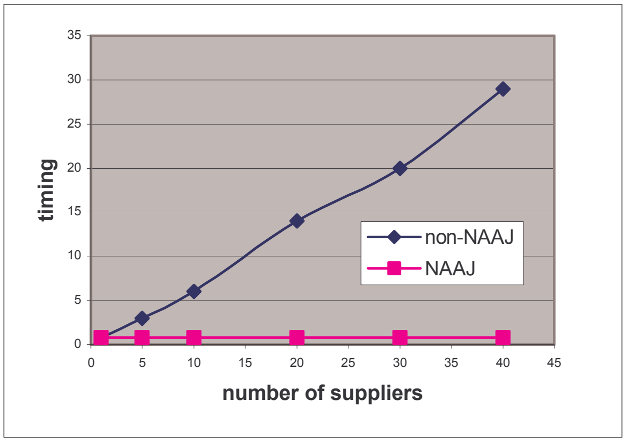

Figure 10. Non-NAAJ vs. NAAJ

There were 72 distinct queries with <> ALL subqueries, which were eligible for NAAJ execution. 68 of the queries benefited from NAAJ by an average of 736730 times measured in CPU time! One query reported 350 seconds without NAAJ and 0.001 with NAAJ. The 4 queries that degraded did that with an average of 100 times, the worst degradation was from 0.000001 to 0.016 in CPU time.

8. RELATED WORK

The unnesting of various types of subqueries has been studied extensively before [1][2][3][4][9]. Our system supports almost all of these forms of unnesting techniques employing heuristic- as well as cost-based strategies. An early work on query transformations [2] proposes a technique for pulling up aggregates and group-by before join in the operator tree. Early versions (8.1 and 9i) of Oracle applied a similar algorithm. The transformation was triggered by heuristics during the query transformation phase. In Oracle 11g, the cost-based framework [8] is used to place distinct and group-by operators and commutates them with joins [5][6]. Some techniques for removing subquery using window functions have been published in [13]. Informal citations [12] suggest that a group-by subquery elimination technique has been discussed elsewhere. We show how a similar technique can be incorporated in the Oracle optimizer. Any work related to subquery coalescing, null-aware antijoin, and scalable window function computation has not been discussed in the literature.

9. CONCLUSION

Subqueries are a powerful component of the SQL and SQL-like query languages enhancing their expressive and declarative capabilities. This paper outlined enhanced subquery optimizations in Oracle relational database, which benefit from Oracle's cost-based transformation framework. This paper makes important contributions[3] by describing a number of techniques – subquery coalescing, subquery removal using window functions, view elimination for group-by queries, null-aware antijoin and parallel execution techniques. Our performance study shows that these optimizations provide significant execution time improvements for complex queries.

10. ACKNOWLEDGMENTS

The authors wish to acknowledge Thierry Cruanes for QC-Slave communication framework, Nathan Folkert and Sankar Subramanian for parallelizing a certain class of window functions using that framework. Susy Fan deserves many thanks for her assistance in the performance experiments.

11. REFERENCES

[1] W. Kim. "On Optimizing an SQL-Like Nested Query", *ACM TODS*, September 1982.

[2] U. Dayal, "Of Nests and Trees: A Unified Approach to Processing Queries that Contain Nested Subqueries, Aggregates, and Quantifiers", *Proceedings of the 13th VLDB Conference*, Brighton, U.K., 1987.

[3] M. Muralikrishna, "Improved Unnesting Algorithms for Join Aggregate SQL Queries", *Proceedings of the 18th VLDB Conference*, Vancouver, Canada, 1992.

[4] H. Pirahesh, J.M. Hellerstein, and W. Hasan, "Extensible Rule Based Query Rewrite Optimizations in Starburst". *Proc. of ACM SIGMOD*, San Diego, U.S.A., 1992.

[5] S. Chaudhuri and K. Shim, "Including Group-By in Query Optimization", *Proceedings of the 20th VLDB Conference*, Santiago, Chile, 1994.

[6] W.P. Yan and A.P. Larson, "Eager Aggregation and Lazy Aggregation", *Proceedings of the 21th VLDB Conference,* Zurich, Switzerland, 1995.

[7] A. Witkowski, et al, "Spreadsheets in RDBMS for OLAP", *Proceedings of ACM SIGMOD,* San Diego, USA, 2003.

[8] R. Ahmed, et al, "Cost-Based Query Transformation in Oracle", *Proceedings of the 32nd VLDB Conference,* Seoul, S. Korea, 2006.

[9] M. Elhemali, C. Galindo-Legaria, et al, "Execution Strategies for SQL Subqueries", *Proceedings of ACM SIGMOD*, Beijing, China, 2007.

[10] A. Eisenberg, K. Kulkarni, et al, "SQL: 2003 Has Been Published", *SIGMOD Record,* March 2004.

[11] F. Zemke. "Rank, Moving and reporting functions for OLAP," *99/01/22 proposal for ANSI-NCITS.*

[12] C. Zuzarte et al, "Method for Aggregation Subquery Join Elimination", http://www.freepatentsonline.com/7454416.html

[13] C. Zuzarte, H. Pirahesh, et al, "WinMagic: Subquery Elimination Using Window Aggregation", *Proceedings of ACM SIGMOD*, San Diego, USA, 2003.

[14] TPC Benchmark H (Decision Support), Standard Specification Rev 2.8, http://tpc.org/tpch/spec/tpch2.8.0.pdf

[15] TPC-DS Specification Draft, Rev 32, http://www.tpc.org/tpcds/spec/tpcds32.pdf

[3] There are U.S. patents pending for some of the work discussed here.