

Optimization of Complex Nested Queries in Relational Databases

Bin Cao

Computer Engineering and Computer Science Department
University of Louisville
bin.cao@louisville.edu

Abstract

Due to the flexible structures of SQL, no general approach works efficiently for all kinds of queries. Some special kinds of queries can be further optimized for better performance. In this dissertation, we study two kinds of such queries: one is queries having non-aggregate subqueries and the other is queries having redundancy. To deal uniformly with non-aggregate subqueries in SQL, we propose the nested relational approach based on the nested relational model. To deal with redundancy in complex SQL queries, we propose the redundancy awareness method with the introduction of the for-loop operator. The main contribution of this dissertation is to provide more efficient solutions for these two kinds of queries than existing techniques.

1. Introduction

One of the most powerful features of SQL is nested queries (queries having subqueries). Since it is usually inefficient to directly execute nested queries in their original form [12], optimization of nested queries has received significant attention since the 1980's. Many unnesting techniques (e.g. [12, 8, 6, 14, 17]) have been proposed to improve nested query performance. However, due to the flexible structures of SQL, no general approach works efficiently for all kinds of queries. Some special kinds of queries can be further optimized for better performance. In this dissertation, we study two kinds of such queries: one is *queries having non-aggregate subqueries* and the other is *queries having redundancy*.

1.1. Problem of non-aggregate subqueries

Depending on whether there is an aggregate function in the SELECT clause or not, subqueries can be simply classified into two types: *aggregate* and *non-aggregate*. A non-aggregate subquery is linked to the outer query by one of the

following operators: EXISTS, NOT EXISTS, IN, NOT IN, θ SOME/ANY, and θ ALL, where $\theta \in \{<, \leq, >, \geq, =, \neq\}$; the result is either a set of values or empty. An example query is shown below.

Query 1 :

```
select p_partkey, p_name
from part
where p_size>=x1 and p_size<=x2 and
      p_retailprice < all
      (select ps_supplycost
       from partsupp
       where ps_partkey=p_partkey and
            ps_availqty<y and
            not exists
            (select *
             from lineitem
             where p_partkey=l_partkey and
                  ps_suppkey=l_suppkey and
                  l_quantity=z))
```

Although significant research efforts have been devoted to optimization of nested queries, most proposed approaches concentrate on aggregate subqueries. The solutions proposed for non-aggregate subqueries still have some limitations, especially for queries with multiple subqueries and null values. Generally, there are two methods for evaluating non-aggregate subqueries. The first is derived from the approaches for aggregate subqueries [8, 7, 1], that is, non-aggregate subqueries are transformed into aggregate subqueries first, and then the transformed queries are evaluated by the existing approaches. However, transformation of the NOT IN or ALL subquery may not preserve semantics when null values are present. Although Akinde and Bohlen [1] introduced the transformation rule for the NOT IN or ALL subquery which can deal with null values, transformed queries might be complicated for multi-level queries and might not yield the best possible plan. The second method is proposed directly to non-aggregate subqueries [6, 3, 2]. They all involve extending the standard relational algebra. Unfortunately, unnesting non-aggregate sub-

queries, especially with certain operators, still pose problems. The common problem of these proposed approaches is twofold: first, not all queries can be unnested directly, and therefore some transformations are required; second, each kind of subquery is evaluated in a different manner. As an example, consider Query 1. It can not be unnested directly using existing techniques; instead, rewriting predicates NOT IN and ALL is required. However, rewriting such predicates may yield a very complex transformed query which can not be evaluated efficiently. Furthermore, even when rewriting is possible, the resulting query tree may have several outer joins and antijoins that can not be moved, as well as extra operations. From this example, we can see that existing approaches have difficulties in dealing with non-aggregate subqueries. What is needed is an approach which uniformly deals with all types of non-aggregate subqueries without introducing undue complexity.

1.2. Problem of redundancy

Queries having aggregate subqueries are widely used in data warehousing and decision support systems. One important thing to notice is that such queries usually show a great deal of *redundancy*, that is, the outer query and the subquery share common tables and conditions. The following query is an example.

Query 2 :

```
select sum(l_extendedprice)/7.0
from lineitem, part, orders
where p_partkey=l_partkey and
      p_size=15 and p_type like '%BRASS' and
      l_shipdate>='1994-01-01' and
      l_shipdate<l_commitdate and
      l_orderkey=o_orderkey and
      o_orderdate>='1994-01-01' and
      o_orderdate<'1994-02-01' and
      l_quantity <
      (select 0.2*avg(l_quantity)
       from lineitem, partsupp
       where l_partkey=p_partkey and
            l_commitdate<l_receiptdate and
            l_shipdate<l_commitdate and
            l_suppkey=ps_suppkey and
            ps_availqty>5000)
```

The redundancy problem has not received attention until recently [15, 7, 20]. Rao and Ross [15] proposed the invariant technique which implements the nested iteration method while considering invariants in the subquery. However, the invariant technique only has better performance for the nested queries that can not be unnested by traditional unnesting strategies. Galindo-Legaria and Joshi [7] presented the decorrelation technique used in Microsoft SQL

Server which solves the redundancy problem using the *SegmentApply* operator. Zuzarte et al. [20] introduced the WinMagic technique to evaluate queries having redundancy by making use of extended window aggregation capabilities. While these techniques can have much better performance than traditional unnesting approaches, both techniques only consider queries with some “restricted” redundancy. As a more complex example, consider Query 2. The outer query and the subquery share common tables and common conditions, but extra tables and conditions are present in both the outer query and the subquery. Furthermore, the join operation between `partsupp` and `lineitem` in the subquery is not a lossless join, which is not within the scope of the WinMagic technique. Redundancy may also appear in subqueries in the HAVING clause, in non-aggregate subqueries, or in queries having multiple subqueries. In [20], the authors indicate that WinMagic can handle non-aggregate subqueries, but detailed techniques are not provided. In fact, dealing with the NOT IN or ALL subquery needs careful consideration due to null values. To the best of our knowledge, no existing technique adequately considers the redundancy present in such queries.

1.3. Contributions

The main contribution of this dissertation is to provide more efficient solutions for queries having non-aggregate subqueries and queries having redundancy than existing techniques by means of going outside the relational model and looking for other ways to implement such queries. Specifically, to efficiently and uniformly evaluate queries having non-aggregate subqueries, we propose the *nested relational approach*. Unlike traditional approaches, which evaluate relational queries using relational algebra, our approach is based on the *nested relational algebra*. The nested relational approach not only allows unnesting non-aggregate subqueries directly without transformation, but also allows each subquery to be evaluated in a uniform manner. Furthermore, it does not require indexes; only hash joins are necessary. Finally, being algebraic, the nested relational approach has clear semantics and can be further optimized. To deal with redundancy in SQL queries, we propose the *redundancy awareness method*. We attack the redundancy problem directly, by identifying tables and conditions common to query and subquery and executing this common part only once, and introducing a new operator, the *for-loop* operator, that allows efficient computation of aggregates and conditions involving them with one pass over the common part. The redundancy awareness method can deal with redundancy in WHERE clause subqueries without restrictions, and applies also to subqueries in the HAVING clause. It also can be extended to non-aggregate subqueries via a rewriting of the subquery and multiple subqueries.

2. Nested Relational Approach

The basic idea of the nested relational approach is straightforward: we unnest the query from top-down using existing techniques, and then we compute *linking predicates* (a linking predicate refers to the predicate that connects a subquery and an outer query) from bottom-up. The second step requires the subquery result to be a set, and a comparison between a single value and a set of values. Such operations are not supported by relational algebra. Thus we propose to use the nested relational algebra.

2.1. Extended nested relational algebra

The nested relational algebra has the standard operations of the relational algebra: selection, projection, Cartesian product, join, union, intersection, difference, plus the nest and unnest operators. For the purpose of the nested relational approach, we slightly extend the standard nested relational algebra, redefining nest, and modifying selection.

We define the *nest* operator as $v_{N_1, N_2}(r)$ by clearly identifying atomic attributes (N_1) and set-valued attributes (N_2), where r is a relation, N_1 and N_2 are disjoint subsets of the attributes of r . The nest operator means for tuples with the same value of N_1 , keeping N_2 as a set.

We express the *linking predicate* as $A \theta L \{B\}$, where A is an attribute in the outer query, B is an attribute in the subquery, $\theta \in \{<, \leq, >, \geq, =, \neq\}$, and $L \in \{\text{SOME/ANY, ALL}\}$, or $L \{B\}$, where $L \in \{\text{EXISTS, NOT EXISTS}\}$ and B as above. We call EXISTS, SOME/ANY and IN *positive linking operators*, and NOT EXISTS, ALL and NOT IN *negative linking operators*. If a query has both positive and negative linking operators, we say it has *mixed linking operators*.

In our extended nested relational algebra, we allow the condition in the standard selection (denoted by $\sigma_C(r)$) to be a linking predicate. Such a selection is called *linking selection*. For multi-level queries, we introduce *pseudo selection* (denoted by $\sigma'_{C,A}$), which not only keeps the result of the standard selection, but also keeps the tuples that fail the condition and pads null values on the attributes in A . The reason for introducing pseudo selection is that a nested tuple that does not satisfy a linking predicate cannot be discarded arbitrarily. Both standard selection and pseudo selection are called linking selection if the condition is a linking predicate.

2.2. Evaluation plan

Based on the extended nested relational algebra, we evaluate a query with non-aggregate subqueries in three steps. **First**, we reduce each query block to a single relation by doing all operations in the WHERE clause, except the linking

predicate and the correlated predicates. This is equivalent to producing a complementary set in magic decorrelation [17]; however, we do not produce a magic set. **Second**, we create a tree expression for reduced query blocks. Each query block is represented by a node, and edges between nodes are labeled by linking predicates and/or correlated predicates. **Third**, based on the tree expression, we unnest subqueries from top-down; if a query has only positive linking operators, joins are used, otherwise, we have to use outer joins; then from bottom-up, we apply a nest followed by a linking selection to compute each linking predicate.

2.3. Optimizations

To optimize the nested relational approach, it is important to study the algebraic properties of nested relational operators used in the approach, as well as the techniques used to implement those operators.

The nested relational approach involves not only the relational operators, but also the (extended) nested relational operators. Thus, algebra equivalence rules for relational operators and also for nested relational operators [16, 11, 13] are not sufficient to do the necessary transformations. Besides selections, the nested relational approach only involves (outer) join, nest, (pseudo) linking selection and projection, which can be regarded as a simplified version of the traditional nested relational algebra. Thus, algebraic optimization here only needs to concern itself with reordering these operators. Such optimizations include *pushing down nest past join*, *combining linking selection and join* and *pushing down projection*. Furthermore, we can optimize positive linking operators to semijoin or join, NOT EXISTS to antijoin (for one-level nested queries).

The choice of implementation technique for the operators present in a query tree also affects query performance. For multi-level queries, we can do all nest operations in a single step, followed by executing linking selections one by one, because only the first nest operator involves physical reordering, all others are conceptual. We can also pipeline linking selections with nest operations to further reduce the cost of such plans.

The nested relational approach could be further optimized for some special queries to gain better performance. For instance, *linear correlated queries*, in which each subquery is only correlated to its adjacent outer query, can be processed efficiently from bottom-up instead of from top-down. *Tree queries with redundancy*, in which at a certain level, the tables and the correlated predicates in several subqueries are the same, can be efficiently evaluated by the *generalized nest* operator, which allows more than one nested attributes (N_2 in the definition of nest) based on different conditions. The performance gain is obtained by computing such subqueries simultaneously.

3. Redundancy Awareness Method

To avoid redundant efforts for queries having redundancy, we must correctly identify the common part between the outer query and the subquery to guarantee it to be executed once and correctly used for later processing. Such a task can be achieved by our *query structure detection technique*. Furthermore, we introduce the *for-loop operator* to efficiently compute the subquery and the outer query based on the common part.

3.1. Query structure detection technique

For queries having redundancy, the conditions and tables in the outer query and the subquery can be roughly divided into three parts: one that is common in both the outer query and the subquery, one that belongs only to the outer query, and one that belongs only to the subquery. Based on these three parts, a query can be evaluated as follows: first, we create a *base relation* based on common tables and common conditions; second, starting from the base relation, we compute the aggregation in the subquery based on the tables and conditions belonging only to the subquery; finally, we generate the desired result based on the tables and conditions belonging only to the outer query and the subquery result. If the subquery has extra tables, the base relation should be extended to include these tables; such a base relation is called the *extended base relation*. Query structure detection, especially the common part detection, is similar to matching part or all of queries to materialized views [18, 9, 10]. However, our query structure detection technique concentrates on the common part between the subquery and the outer query. Thus, the algorithm proposed in [19], which is based on the *query graph*, a representation of the logical structure of a given query, can be slightly modified and reused to implement this technique.

3.2. The for-loop operator

To efficiently compute the aggregation in the subquery and the linking predicate between the outer query and the subquery by only one pass over the (extended) base relation, we define a new operator, *for-loop*, as $FL_{L,F(A),\alpha,\beta}(r)$ (*grouped for-loop*) or $FL_{F(A),\alpha,\beta}(r)$ (*flat for-loop*), where r is a relation, L is a subset of attributes of r , A is an attribute of r , F is an aggregate function, α is a condition on r and β is a condition on $r \cup \{F(A)\}$. The main use of a for-loop operator is to compute the aggregation in the subquery and the linking predicate on the fly, possibly with additional selections. For each group of L (if it exists), the for-loop operator computes $F(A)$ based on α , then filters out tuples that satisfy β .

This definition is based on the observation that some basic operations appear frequently together and they could be more efficiently implemented as a whole, thus saving considerable disk I/O. The basic idea is twofold: first, selections and groupings can be effectively implemented in one algorithm; second, and more important, in some cases computing an aggregation and using the aggregate result in a selection can be done at the same time. This is due to the behavior of some aggregates and the semantics of the conditions involved. For instance, a comparison of the type $attr1 = \min(attr2)$, where both $attr1$ and $attr2$ are attributes of some table r , can be efficiently implemented by a sequential pass over r . Furthermore, when a subquery is correlated, we use the grouped for-loop and aggregate computation is done per group. That implies that all temporary information needed is likely to fit in memory and performance may be good.

3.3. Evaluation plan

Based on the (extended) base relation and the for-loop operator, the execution plan for queries having redundancy can be created by the following three steps. **First**, we create the (extended) base relation $[E]BR$. Note that standard relational optimization techniques can be applied to this part. **Second**, for queries with correlated subqueries, we apply a grouped for-loop operator, $FL_{L,F(A),\alpha,\beta}([E]BR)$; for queries with non-correlated subqueries, we apply a flat for-loop operator, $FL_{F(A),\alpha,\beta}([E]BR)$, where L is the attributes in correlated predicates (if L is not the primary key of the table that introduces correlation, L should be replaced by the primary key), $F(A)$ is the aggregation in the subquery, α is the conditions used to compute $F(A)$, β is the conditions involving the common tables and the linking predicate. **Third**, if there are no extra tables in the outer query, the final result can be obtained by projection on the desired attributes. Otherwise, the final result can be obtained by performing a join of $[E]BR$ and these tables followed by projection of the desired attributes.

3.4. Extensions

We consider three main extensions of the redundancy awareness method to cover subqueries in the HAVING clause, non-aggregate subqueries, and multi-level queries. For each case, the for-loop operator needs to be extended to deal with several aggregations simultaneously; each one with its own conditions.

In SQL, the HAVING clause usually occurs after the GROUP BY clause. If the subquery is correlated, the attributes of the outer query in the correlated predicates must be the same as the attributes in the GROUP BY clause. If

the subquery is not correlated, the aggregation in the subquery is computed over a whole base relation.

Non-aggregate subqueries can be rewritten as queries with aggregate (COUNT) subqueries. Such rewrites must be carefully specified for the ALL or NOT IN subquery. Our rewrites are basically equivalent to those of [1]).

When considering queries of arbitrary depth, we distinguish between *linear queries* (where there is at most one subquery in any given level) and *tree queries* (where in some level there are two or more subqueries). The redundancy awareness method can be easily extended to linear queries of any depth. Tree queries can also be taken care of, but require some additional care.

4. Implementations

To verify the efficiency of the nested relational approach and the redundancy awareness method, we have implemented them on top of a leading commercial DBMS, which we call “System A”. We created the TPC-H database [5] at scale factors 1 and 10 (size of 1GB and 10GB respectively) in System A. Our implementation is designed in two stages: first, an SQL query is used to unnest subqueries (for the nested relational approach) or obtain the (extended) base relation (for the redundancy awareness method). Second, we use stored procedures in *procedural SQL* to implement the nest and linking selection operators (for the nested relational approach) or the for-loop operator (for the redundancy awareness method) which processes the data fetched from the first stage. We compare the nested relational approaches and the redundancy awareness method to the state-of-art techniques. Detailed experiment results and performance analysis of the nested relational approach have been shown in [4]. For the redundancy awareness method, we have run different kinds of queries over different buffer caches and speedups are achieved.

5. Conclusions and Future Work

This dissertation proposes the nested relational approach and redundancy awareness method to efficiently evaluate queries having non-aggregate subqueries and queries having redundancy respectively. The original work on nested relational approach has shown a great performance improvement [4]. Currently, we are studying algebraic optimizations and doing performance analysis among alternative query plans. The redundancy awareness method also has shown the potential to outperform existing optimizations. We are currently expanding our experiments to examine the impact of different amounts of redundancy.

Acknowledgment. This research is sponsored by NSF under the research grant IIS-0091928 and the NSF CAREER award IIS-0347555.

References

- [1] M. O. Akinde and M. H. Bohlen. Efficient computation of subqueries in complex olap. In *Proceedings of the ICDE Conference*, pages 163–174, 2003.
- [2] A. Badia. Computing sql subqueries with boolean aggregates. In *Proceedings of the DAWAK Conference*, 2003.
- [3] L. Baekgaard and L. Mark. Incremental computation of nested relational query expressions. *ACM Transactions on Database Systems*, 20(2):111–148, 1995.
- [4] B. Cao and A. Badia. A nested relational approach to processing sql subqueries. In *Proceedings of the SIGMOD Conference*, pages 191–202, 2005.
- [5] T. P. P. Council. The tpc-h benchmark. <http://www.tpc.org/tpch>.
- [6] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proceedings of the VLDB Conference*, pages 197–208, 1987.
- [7] C. A. Galindo-Legaria and M. M. Joshi. Orthogonal optimization of subqueries and aggregation. In *Proceedings of the ACM SIGMOD Conference*, pages 571–581, 2001.
- [8] R. A. Ganski and H. K. T. Wong. Optimization of nested sql queries revisited. In *Proceedings of the ACM SIGMOD Conference*, pages 23–33, 1987.
- [9] J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: a practical, scalable solution. In *Proceedings of the SIGMOD Conference*, pages 331–342, 2001.
- [10] A. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10:270–294, 2001.
- [11] Y. Jan. Algebraic optimization for nested relations. In *Proceedings of the ICSS Conference*, pages 278–287, 1990.
- [12] W. Kim. On optimizing an sql-like nested query. *ACM Transactions on Database Systems*, 7(3):443–469, 1982.
- [13] H.-C. Liu and J. X. Yu. Algebraic equivalences of nested relational operators. *Information Systems*, 30:167–204, 2005.
- [14] M. Muralikrishna. Optimization and dataflow algorithms for nested tree queries. In *Proceedings of the VLDB Conference*, pages 77–85, 1989.
- [15] J. Rao and K. A. Ross. Reusing invariants: a new strategy for correlated queries. In *Proceedings of the ACM SIGMOD Conference*, pages 37–48, 1998.
- [16] M. H. Scholl. Theoretical foundation of algebraic optimization utilizing unnormalized relations. In *Proceedings of the ICDD Conference*, pages 380–396, 1986.
- [17] P. Seshadri, H. Pirahesh, and T. Y. C. Leung. Complex query decorrelation. In *Proceedings of the ICDE Conference*, pages 450–458, 1996.
- [18] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex sql queries using automatic summary tables. In *Proceedings of the SIGMOD Conference*, pages 105–116, 2000.
- [19] Q. Zhu, Y. Tao, and C. Zuzarte. Optimizing complex queries based on similarities of subqueries. *Information Systems*, 8(3):350–373, 2005.
- [20] C. Zuzarte, H. Pirahesh, W. Ma, Q. Cheng, L. Liu, and K. Wong. Winmagic: Subquery elimination using window aggregation. In *Proceedings of the ACM SIGMOD Conference*, pages 652–656, 2003.