Redundancy Awareness in SQL Queries

Bin Cao and Antonio Badia
Computer Engineering and Computer Science Department
University of Louisville
{bin.cao,abadia}@louisville.edu

Abstract

In this paper, we study SQL queries with aggregate subqueries that share common tables and conditions with the outer query. While several approaches can deal with such queries, they have limited applicability. We propose the redundancy awareness method to detect the largest common part shared by query and subquery, compute it once, and determine what operations are needed to finish evaluation of the original query. Our approach can deal with redundancy in all types of subqueries. We offer the possibility for the optimizer to choose the most efficient plan for a given query. We have implemented our approach on top of a commercial DBMS; our experiments show that our approach compares favorably to existing optimization techniques.

1. Introduction

One powerful feature of SQL is the use of subqueries with aggregation, so that whenever possible, aggregates are computed and used within the same query. However, one of the limitations is that such queries usually show a great deal of redundancy, that is, the outer query and the subquery share common tables and conditions. It is important to point out that redundancy is present because of the structure of SQL. In this paper, we study the general idea of optimizing queries having redundancy by focusing on the problem of queries with aggregate subqueries. Although other approaches have addressed such queries [3, 9], they have some limitations that we aim to overcome. In particular, such approaches only apply under limited circumstances, while we present a general and efficient approach, called the redundancy awareness method. We attack the redundancy problem directly by identifying the largest common part shared by query and subquery and executing this common part only once. However, this does not, per se, guarantee improved performance. Thus, we propose a new operator, called the for-loop, that allows efficient computation of aggregates and conditions involving them with one pass over the common part. Our approach 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.

The nested relational approach aims to detect the largest redundancy present in the outer query and the subquery. However, sharing the largest common part can not always yield the most efficient plan. Sometimes, executing none or part of redundancy may have better performance. What we do here is to offer the possibility for the optimizer to choose the best plan. We present experimental evidence that our approach, when applicable, performs better than other optimizations proposed in the literature.

2. Related Work and Motivation

A typical query having redundancy is shown below:

Query 1:

\[
\begin{align*}
\text{select } & \text{s_acctbal, s_name, n_name, } \\
& \text{p_partkey, p_mfgr, s_address, } \\
& \text{s_phone, s_comment } \\
\text{from } & \text{part, supplier, partsupp, nation, region } \\
\text{where } & \text{p_partkey=ps_partkey and } \\
& \text{s_suppkey=ps_suppkey and } \\
& \text{p_size=15 and } \\
& \text{p_type like } `%\text{BRASS}' \text{ and } \\
& \text{s_nationkey=n_nationkey and } \\
& \text{n_regionkey=r_regionkey and } \\
& \text{r_name='EUROPE' and } \\
& \text{ps_supplycost = } \\
& (\text{select min(ps_supplycost) } \\
& \text{from partsupp, supplier, nation, region } \\
& \text{where p_partkey=ps_partkey and } \\
& \text{s_suppkey=ps_suppkey and } \\
& \text{s_nationkey=n_nationkey and } \\
& \text{n_regionkey=r_regionkey and } \\
& \text{r_name='EUROPE'})
\end{align*}
\]

The most noticeable feature of the above query is redundancy: the tables and conditions in the subquery are totally
included in those in the outer query. As a query having a correlated aggregate subquery, Query 1 can be unnested by most approaches proposed in the literature (e.g. [6]). However, such approaches would not recognize the redundancy. Therefore, common tables and common conditions have to be accessed and computed more than once.

Optimization of nested queries has received significant attention since the 1980’s. However, the redundancy problem has not received attention until recently [5, 3, 9]. Rao and Ross [5] 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 [3] presented the decorrelation technique used in Microsoft SQL Server which solves the redundancy problem using the SegmentApply operator. Zuzarte et al. [9] introduced the WinMagic technique to evaluate queries having redundancy by making use of extended window aggregation capabilities. While these techniques [3, 9] can have much better performance than traditional unnesting approaches (e.g.[6]), both techniques only consider queries with some "restricted" redundancy. As a more complex example, consider the following query:

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<l_commitdate and
  l_shipdate>='1994-01-01' and
  p_size=15 and p_type like '%BRASS' 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_supkey=ps_suppkey and
     ps_availqty>5000)
```

In 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 [9], 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. In this paper, we propose a general approach, the redundancy awareness method, to avoid redundant computation in SQL queries.

3. Redundancy Awareness Method

The basic idea of the redundancy awareness method is straightforward: the conditions and tables in the outer query and the subquery are 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. To achieve this goal, the base relation must be correctly identified. Furthermore, we introduce the for-loop operator to compute the subquery and the outer query based on the base relation. With an efficient implementation, the for-loop operator needs only one pass over the base relation to compute the subquery and the outer query.

3.1. Generating base relations

For the purpose of our approach, we use a query pattern to represent a query. A query pattern is a schematic representation of a query using keywords SELECT, FROM, WHERE (and optional GROUP BY, HAVING), and variables over tables, aggregate functions and conditions. A typical one-level nested query having redundancy with a correlated aggregate subquery in the WHERE clause can be expressed by the query pattern shown below (we assume that the desired result is an aggregate function).

```
SELECT F_1(attr_1)
FROM T_{common}, T_{correlation}, T_{outer}
WHERE C_{common-outer} AND
  C_{correlation} AND
  C_{outer} AND
  I_{common-correlation} AND
  attr_{inputs} \theta (SELECT F_{inner}(attr_{inner})
                         FROM T_{common}, T_{inner}
                         WHERE C_{common-inner} AND
                           C_{inner} AND
                           I_{common-inner} AND
                           P_{correlated})
```

Where $F$ denotes aggregate functions, $attr$ denotes attributes, $T$ denotes tables, $C$ denotes selection conditions, $L$ and $P$ denote join conditions. Obviously, Query 2
matches this query pattern. Due to lack of space, we omit the details of how to match a query to its query pattern. Query structure detection, especially the common part detection, is similar to matching part or all of queries to materialized views [7, 4]. How to detect common subqueries and exploit them in complex query optimization has been studied by Zhu, Tao and Zuzarte [8]. The algorithm proposed in [8] can be modified and reused to implement our matching technique.

The key conditions to check when using the redundancy awareness method are: (1) the set of common tables \( T_{common} \) is not empty, and (2) correlated predicates \( P_{correlated} \) and the join conditions between common tables and correlation tables in the outer query \( L_{common-correlation} \) are the same (for correlated subqueries). Even though the above conditions are satisfied, what conditions are exactly common to the outer query and the subquery still need further consideration, because the conditions involving the common tables in the outer query \( (C_{common-outer}) \) and the conditions involving the common tables in the subquery \( (C_{common-inner}) \) might contain different conditions. We define the base relation, denoted by \( BR \), as the common part (tables and conditions) between the outer query and the subquery. To obtain the base relation, we compute:

\[
\begin{align*}
C_{base} &= C_{common-outer} \cap C_{common-inner}, \\
C_{base-outer} &= C_{common-outer} - C_{common-inner}, \\
C_{base-inner} &= C_{common-inner} - C_{common-outer}.
\end{align*}
\]

Clearly, \( C_{base} \) is used to compute the base relation; but \( C_{base-outer} \) and \( C_{base-inner} \) must be applied at the right time, to produce the right subquery result and the right final result. Thus, the base relation can be simply obtained by performing selection \( C_{base} \) on \( T_{common} \) (for non-correlated subqueries), or by performing a join of \( T_{common} \) and \( T_{correlation} \) on \( C_{base} \), \( C_{correlation} \) and \( P_{correlated} \) (for correlated subqueries).

If there are extra tables in the subquery except common tables, i.e. \( T_{inner} \) is not empty, we have to extend the base relation to the extended base relation, denoted by \( EBR \). Generally, the extended base relation is obtained by performing a left outer join of \( BR \) and \( T_{inner} \) on \( L_{common-inner} \). Note that \( C_{inner} \) can be pushed down. Performing an outer join of \( BR \) and \( T_{inner} \) might cause tuples in \( BR \) duplicated if \( L_{common-inner} \) is not a lossless join. Such duplicates are required to compute the aggregation in the subquery, but will cause an error when computing the final result. The solution to this problem is to use \( EBR \) to compute the aggregation in the subquery, and then reduce \( EBR \) to \( BR \) to compute the outer query by performing \( GB_{BR \# count(T_{inner-#})}(EBR) \), where \( GB \) denotes group-by, and \( # \) denotes the primary key. With respect to the SQL syntax, the grouping attributes might include the primary key of \( BR \) as well as other attributes required for later processing.

### 3.2. The for-loop operator

Once obtaining the (extended) base relation (denoted by \( [E]BR \), where the square bracket denotes optional), the subsequent computations of the aggregation in the subquery and the linking predicate between the outer query and the subquery can be done efficiently by only one pass over the (extended) base relation. In order to do so, we define a new operator, \( for-loop \), which combines several relational operators into a new one (i.e. a \emph{macro-operator}). This approach 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. In the following, \( GB \) is used to indicate a group-by-operation, and \( AGG_{F_{\alpha}}(R) \) indicates the aggregation \( F \) computed over all values of the attribute \( A \) of the relation \( R \).

**Definition 1** Let \( R \) be a relation, \( sch(R) \) the schema of \( R \), \( L \subseteq sch(R), A \in sch(R) \), \( F \) an aggregate function, \( \alpha \) a condition on \( R \) (i.e. involving only attributes of \( sch(R) \)) and \( \beta \) a condition on \( sch(R) \cup \{F(A)\} \) (i.e. possibly involving \( F(A) \)). Then the \emph{for-loop operator} is defined in two variants:

- \( FL_{L,F_{\alpha} \cdot \beta}(R) \). The meaning of the operator is given by \( \sigma_{\beta}(\forall_{R \in L, L \subseteq L}(GB_{L,F_{\alpha}}(\sigma_{\alpha}(R)))) \), where the condition of the join is understood as the pairwise equality of each attribute in \( L \). This is called a \emph{grouped for-loop}.

- \( FL_{F_{\alpha} \cdot \beta}(R) \). The meaning of the operator is given by \( \sigma_{\beta}((AGG_{F_{\alpha}}(\sigma_{\alpha}(R))) \times R) \). This is called a \emph{flat for-loop}. \( \square \)

Although algebraic expression is an important issue, it is not the only reason to define the for-loop operator. There are also significant implementation issues. To achieve the objective of computing several results at once with a single pass over the data, the operator can be implemented as an \emph{iterator} that loops over the input implementing a simple program. The basic idea is twofold: first, selections and groupings (either grouping alone or together with aggregate calculations) 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 means that results (and temporary results) are only accumulated by group, and therefore all temporary information needed is likely to fit in memory and performance may be good even for cases where the condition is hard to compute.

The for-loop operator is similar to the SegmentApply operator [3] and the window aggregation [9] in that all of these operators are used to do computation based on the common part between the outer query and the subquery. Both the for-loop operator and the SegmentApply operator are extended relational operators which have to be implemented inside DBMS. The window aggregation is one of the analytic features defined as part of the ANSI SQL 1999 standard and has been implemented in some commercial DBMS. Thus, the query can be rewritten in SQL with the redundancy clause. From the performance point of view, the for-loop operator may perform more efficient than the other two due to its efficient implementation for computing the linked aggregation, the linking predicate, and the final result, which needs only one pass over the common part.

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:

1. We create the (extended) base relation \([E]BR\). Note that standard relational optimization techniques can be applied to this part.

2. For queries with correlated subqueries, we apply a grouped for-loop operator, \(FL_{L,F(A),\alpha,\beta}(\{E\}BR)\), to \([E]BR\). For queries with non-correlated subqueries, we apply a flat for-loop operator, \(FL_{F(A),\alpha,\beta}(\{E\}BR)\), to \([E]BR\). According to the for-loop operator definition, \(L\) is \(attr_{correlating}\) (the attributes of the outer query in the correlated predicates); if the correlating attribute is not the primary key of \(T_{correlation}\), \(L\) should be replaced by the primary key. \(F(A)\) is \(F_{linked}(attr_{linked})\), \(\alpha\) is \(C_{base-inner}\), \(\beta\) is \(C_{base-outer} \land attr_{linked} \land F_{linked}(attr_{linked})\).

3. If \(T_{outer}\) is empty, the final result can be obtained by projection on the desired attributes. If \(T_{outer}\) is not empty, the final result can be obtained by performing a join of \([E]BR\) and \(T_{outer}\) on \(L_{common-outer}\) 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.

- **Subqueries in the HAVING clause.** In SQL, the HAVING clause usually occurs after the GROUP BY clause. If the subquery is correlated, the correlating attributes must be the same as the attributes in the GROUP BY clause. Thus, the aggregation in the outer query and the aggregation in the subquery can be computed simultaneously. If the subquery is not correlated, the aggregation in the subquery is computed over a whole base relation. The only thing that is needed is to continue the computations, without resetting, across groups. After each group is computed, we can compare the result of the global aggregate so far to the new group, and proceed as in the regular case.

- **Non-aggregate subqueries.** A non-aggregate subquery is linked to its outer query by one of the following operators: EXISTS, NOT EXISTS, IN, NOT IN, \(\theta\) SOME/ANY, and \(\theta\) ALL, where \(\theta \in \{<, \leq, >, \geq, =, \ne\}\). Such queries can be rewritten as queries with aggregate subqueries, in particular with the COUNT aggregate. Such rewrites must be carefully specified for the ALL or NOT IN subquery, though, since approaches usually thought to work fail in the presence of null values. Let the subquery be \(attr_1 \theta\) ALL (select \(attr_2\ldots\)). Then the query can be rewritten as (select \(\theta\) \(attr_2\ldots\))= (select \((\text{count}(attr_2\ldots)\ldots\)) where the first subquery is exactly as it was in the original, and the second one is also the same as the original but has the predicate \(attr_1 \theta\) \(attr_2\ldots\) added to the WHERE clause (this approach is basically equivalent to that of [1]).

- **Multi-level queries.** 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). It is easy to see that the redundancy awareness method can be extended to linear queries of any depth: the method is applied from the bottom up, starting at the innermost subquery. Once this step is done, one can proceed up the query until the outermost block. Tree queries can also be taken care of, but require some additional care. Assuming that query block \(B\) has two subqueries, \(B_1\) and \(B_2\), we consider two cases: (1) If \(B_1\) has some overlap with \(B\) and \(B_2\) does not, \(B_2\) can be processed by a traditional approach and \(B_1\) can be processed with the redundancy awareness method. (2) If both \(B_1\) and \(B_2\)
have overlap with B, we further distinguish two cases: if B_1 and B_2 are both correlated and have the same correlating attribute with B, both subqueries could be computed by a for-loop operator for further efficiency. Otherwise, it is better to treat B_1 and B_2 separately, creating a subtree for-loop operator for each one.

For each case, the for-loop operator needs to be extended to deal with several aggregations simultaneously; each one with its own conditions. The extended notation of the for-loop operator is: FL([[L_1], F_1 (A_1), a_1], [[L_2], F_2 (A_2), a_2], ..., ϑ([E]BR)), where the square bracket denotes optional. This definition is very similar to the multidimensional join (MD) operator defined in [1]. Unlike that work, though, the for-loop operator is aware of redundancy and works on the (extended) base relation, while the MD operator does not consider redundancy and have to access common tables more than once. Thus, the MD operator relies on further optimization to recognize redundancy on its operation, while we create and control such redundancy.

4. Experiments

To verify the efficiency of the redundancy awareness method, we have implemented it on top of a leading commercial DBMS, which we call “System A”. We created TPC-H databases [2] at scale factors 1 and 10 (size of 1GB and 10GB respectively) in System A. We run Query 1 and Query 2 on a buffer cache of size 32MB and 128MB respectively. The execution results are shown in figure 1 and figure 2. From these two figures, we can see that both the redundancy awareness method and WinMagic [9] perform much better than magic decorrelation [6] by avoiding redundant computations. Furthermore, the redundancy awareness method performs slightly better than WinMagic. Detailed performance analysis and the results of other test queries will be included in our full paper.

![Figure 1. Query 1](image1)

![Figure 2. Query 2](image2)

5. Conclusions

In this paper, we propose the redundancy awareness method to efficiently evaluate queries having redundancy. The results of our experiments show that our approach indeed outperforms traditional optimizations. Furthermore, our approach is potentially applicable to a wide range of queries. We are currently expanding our experiments to examine the impact of different amounts of redundancy.

References