Orthogonal Optimization of Subqueries and Aggregation

César A. Galindo-Legaria   Milind M. Joshi

{cesarg,milindj}@microsoft.com
Microsoft Corp.
One Microsoft Way
Redmond, WA 98052

ABSTRACT

There is considerable overlap between strategies proposed for subquery evaluation, and those for grouping and aggregation. In this paper we show how a number of small, independent primitives generate a rich set of efficient execution strategies —covering standard proposals for subquery evaluation suggested in earlier literature. These small primitives fall into two main, orthogonal areas: Correlation removal, and efficient processing of outerjoins and GroupBy. An optimization approach based on these pieces provides syntax-independence of query processing with respect to subqueries, i. e. equivalent queries written with or without subquery produce the same efficient plan.

We describe techniques implemented in Microsoft SQL Server (releases 7.0 and 8.0) for queries containing subqueries and/or aggregations, based on a number of orthogonal optimizations. We concentrate separately on removing correlated subqueries, also called "query flattening," and on efficient execution of queries with aggregations. The end result is a modular, flexible implementation, which produces very efficient execution plans. To demonstrate the validity of our approach, we present results for some queries from the TPC-H benchmark. From all published TPC-H results in the 300GB scale, at the time of writing (November 2000), SQL Server has the fastest results on those queries, even on a fraction of the processors used by other systems.

1. INTRODUCTION

Subqueries are a convenient and succinct construct of the SQL language, which has been implemented by commercial database systems for a number of years. They are a standard mechanism used frequently by real applications, and researchers have worked on their efficient evaluation, proposing powerful techniques. Here, we make the observation that there is significant overlap between techniques proposed for subquery execution and others such as GroupBy evaluation. Therefore we take the approach of identifying and implementing more primitive, independent optimizations that collectively generate efficient execution plans.

In this paper, we present subquery and aggregation techniques implemented in Microsoft SQL Server. It is organized as follows. First, we review standard proposals for subquery evaluation, and describe how they reduce to other primitive optimizations. Then we review the algebraic representation of correlation, or use of parameterized subexpressions. In Section 2 we focus on representation and normalization of subqueries, with the goal of replacing correlations by standard relational algebra operators. In Section 3 we describe techniques for efficient execution of queries with aggregation. In Section 4 we give an overview of how techniques described earlier fit into the architecture of our query processor. We present performance results of our approach in Section 5, with data from current published TPC-H results. Section 6 concludes the paper.

1.1 Standard subquery execution strategies

Before describing subquery strategies in detail, it is important to clarify the two forms of aggregation in SQL, whose behavior diverges on an empty input. "Vector" aggregation specifies grouping columns as well as aggregates to compute.[1] For example, obtaining the total sales per date:

```
SELECT O_ORDERDATE, SUM(O_TOTALPRICE)
FROM ORDERS
GROUP BY O_ORDERDATE
```

If ORDERS is empty, the result of the query is also empty. "Scalar" aggregation on the other hand, does not specify grouping columns. For example, get the total sales in the table:

```
SELECT SUM(O_TOTALPRICE) FROM ORDERS
```

This second query *always returns exactly one row.* The result value on an empty input depends on the aggregate; for SUM it is NULL, while for COUNT it is 0 [13]. In algebraic expressions we denote vector aggregate as $\mathcal{G}_{A,F}$, where A are the grouping columns and F are the aggregates to compute; and denote scalar aggregate as $\mathcal{G}^1_F$.

We review standard subquery execution strategies using the following SQL query, which finds customers who have ordered more than \$1,000,000. The subquery uses a scalar

[1]The use of DISTINCT for duplicate removal is a special case of "vector aggregate," collapsing groups with equal values into a single row, but without actual aggregate functions to compute. We normalize DISTINCT as GroupBy.

aggregate to compute the total ordered by a customer. It is called a “correlated subquery” because it uses parameters resolved from a table outside of the subquery, in this case column C_CUSTKEY. Our examples use the straightforward data schema of the TPC-H benchmark.

```
Q1: SELECT C_CUSTKEY
    FROM CUSTOMER
    WHERE 1000000 <
          (SELECT SUM(O_TOTALPRICE)
          FROM ORDERS
          WHERE O_CUSTKEY = C_CUSTKEY)
```

Correlated execution. The execution that is closest to the SQL formulation is to take each customer and compute a total amount, as specified in the subquery, then filter out customers who have ordered less than the specified amount. This is commonly considered an inferior strategy, as it involves per-row processing of customers, instead of a set oriented execution —but it can actually be the best strategy, if the outer table is small, and appropriate indices exist.

Outerjoin, then aggregate. This execution strategy was original proposed by Dayal [5]. To use set-oriented algorithms, we can first collect all orders for each customer, then aggregate grouping by customer, and finally filter based on the aggregate result. The corresponding SQL formulation is as follows.

```
SELECT C_CUSTKEY
FROM CUSTOMER LEFT OUTER JOIN
      ORDERS ON O_CUSTKEY = C_CUSTKEY
GROUP BY C_CUSTKEY
HAVING 1000000 < SUM(O_TOTALPRICE)
```

The use of outerjoin is faithful to the correlated execution semantics: The correlated subquery uses scalar aggregation and therefore returns exactly one row for each customer, even if there are no qualifying orders. Using outerjoin, customers with no matching orders are preserved, and the aggregation result on such non-matching row is NULL.

Aggregate, then join. This execution strategy was original proposed by Kim [11]. It is possible to aggregate directly over the ORDERS table to obtain the total sales per customer and later join with the CUSTOMER table. This alse allows pushing the aggregate condition below the join. The SQL formulation uses a *derived table*, not a subquery.

```
SELECT C_CUSTKEY
FROM CUSTOMER,
      (SELECT O_CUSTKEY FROM ORDERS
      GROUP BY C_CUSTKEY
      HAVING 1000000 < SUM(O_TOTALPRICE))
      AS AggResult
WHERE O_CUSTKEY = C_CUSTKEY
```

1.2 Our technique: Use primitive, orthogonal pieces

From those listed above, the most efficient strategy depends on size of tables, selectivity of conditions, and reduction factor of aggregation. Rather than going directly from a subquery usage to either of the above strategies, we use orthogonal, reusable primitives that lead to those listed above, and more. And use cost-estimation to choose between them. Figure 1 shows how primitive optimizations lead to different “subquery strategies.” The above standard strategies are only two of the possible boxes in the diagram; the other strategies shown are also feasible, and will sometimes be superior, depending on data distribution and indices available. There are a number of optimizations that can be done on GroupBy queries that are not shown in this figure, but are discussed later in Section 3. By implementing all these orthogonal techniques, the query processor should then produce the same efficient execution plan for the various equivalent SQL formulations we have listed above, achieving a degree of syntax-independence. The basic transformations shown in the figure are described next.

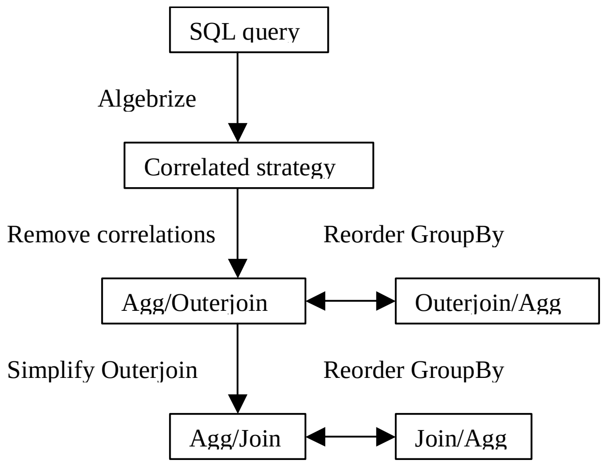


Figure 1: Primitives connecting different execution strategies.

Algebrize into initial operator tree. Our algebraic representation of correlations is based on the *Apply* operator. *Apply* is a second-order relational construct that abstracts parameterized execution of subexpressions. We elaborate more on this operator below, in Section 1.3.

Remove correlations. Removal of correlation consists of rewriting Apply into other operators such as outerjoin. In Section 2.3 we go into the details of how this is achieved. The result of correlational removal, on the example query we used early is exactly the strategy proposed by Dayal.

Simplify outerjoin. Simplification of outerjoin into join, under null-rejecting conditions, is elaborated in [7]. We use the same framework, but add derivation of null-rejection in GroupBy operators, which is not covered in that work. Correlation removal typically results in outerjoins, which are then simplified into joins, when possible. In our earlier example, the condition $1000000 < X$ rejects a NULL value of x, which triggers simplification of outerjoin into join.

Reorder GroupBy. Reordering GroupBy around joins was presented in [3, 18]. In our implementation, we build on that

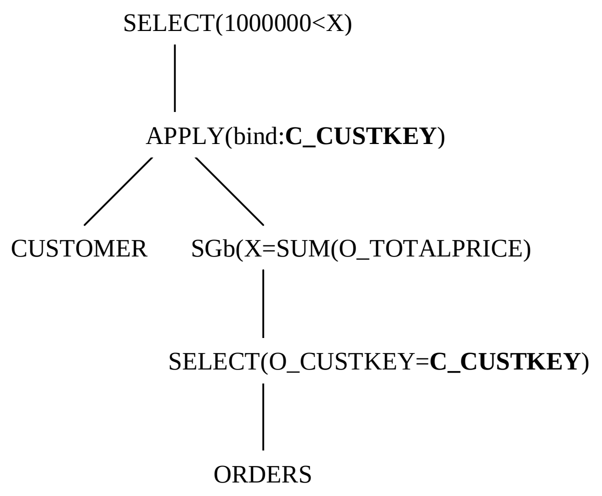


Figure 2: Subquery execution using Apply.

work by adding reordering around outerjoins, and operating based on abstract properties of aggregate functions, rather than considering the five standard SQL aggregates.

1.3 A useful tool: Represent parameterized execution algebraically

A convenient tool in the development of our orthogonal optimizations is the algebraic representation of parameterized subquery execution. The basic idea is similar to the APPLY or MAPCAR operators of LISP: Evaluate an expression on a collection of items, and gather the results. The first construct we use is *Apply*. Relational database practitioners will see this as nested loops with correlations, while researchers on object-oriented databases might employ lambda-calculus notation directly, e. g. [16, 4, 12]. *Apply* takes a relational input R and a parameterized expression $E(r)$; it evaluates expression E for each row $r \in R$, and collects the results. Formally,

$$R\,\mathcal{A}^{\otimes}\,E = \bigcup_{r \in R} (\{r\} \otimes E(r))\,,$$

where $\otimes$ is either cross product, left outerjoin, left semijoin, or left antijoin. The most primitive form is $\mathcal{A}^{\times}$, and cross product is assumed if no join variant is specified. Since we deal with SQL, all operators used in this paper are bag-oriented, and we assume no automatic removal of duplicates. In particular, the union operator above is UNION ALL. Duplicates are removed explicitly using DISTINCT, which as mentioned earlier is just a special case of GroupBy.

For example, the correlated execution strategy for the subquery example **Q1** shown earlier is represented algebraically in Figure 2. In this case, each invocation of the parameterized expression returns exactly one row, so the cardinality of CUSTOMER is preserved through Apply. In general, the number of output rows depends on the cardinality of $E(r)$, as well as the join variant used to combine the result with r.

The role of Apply in (sub-)query processing has been independently explored by at least three research/development groups: At Tandem, where it was called *tuple substitution join* [1]; at Oregon Graduate Institue and Portland State University, called *d-join* [17]; and at Microsoft [6]. It is interesting to note that those three groups were all working with an optimizer developed from the Cascades query optimizer of Goetz Graefe [8].

Apply works on expressions that take scalar (or row-valued) parameters. A second useful construct is *SegmentApply*, which deals with expressions using *table*-valued parameters. It takes a relational input R, a parameterized expression $E(S)$, and a set of segmenting columns A from R. It creates segments of R using columns A, much like GroupBy does, and for each such segment S it executes $E(S)$. Formally,

$$R\ \mathcal{SA}_A\ E = \bigcup_{a} (\{a\} \times E(\sigma_{A=a} R))\,,$$

where a takes all values in the domain of A.

These higher order constructs do not add expressive power to the standard relational operators $\times, \sigma, \pi, -, \cup$, plus operator GroupBy required for SQL. SegmentApply can be rewritten in terms of Apply, and any expression containing standard operators plus Apply can be rewritten in terms of standard operators only [6]. The constructs do remain a useful tool for query processing, facilitating query representation (Apply maps directly to a tuple-at-a-time formulation style) and significantly enhancing the space of alternatives considered for execution. This is achieved in an algebraic fashion, allowing both formal algebraic manipulation as well as smooth integration in algebraic query processors.

2. REPRESENTING AND NORMALIZING SUBQUERIES

In this section we go into the detail of taking an SQL subquery and generating an equivalent operator tree that does not make use of correlated execution. The result is a normal form corresponding to a query formulation without subqueries. We consider *all* SQL subqueries, and describe what makes their replacement by standard operators easy, or hard, leading to broad subquery classes.

2.1 Direct algebraic representation with mutual recursion

As a first step, the parser/algebrizer takes the SQL formulation and generates an operator tree, which contains both relational and scalar operators. For example, an SQL WHERE clause is translated into relational select, which has two subexpressions in the operator tree. The first subexpression has a relational operator as root, which computes the relational input to be filtered. The second subexpression has a scalar operator as root, and it represents the predicate to be use for filtering. Figure 3 shows the operator tree generated by the algebrizer for the example query **Q1** of Section 1.1. Relational nodes are shown in bold.

On this representation, scalar operators may have relational subexpressions as children, as shown in the figure. Straightforward execution of this operator tree implies a "nested loops style" of execution for the subquery, as well as mutual recursion between the relational and the scalar execution components —per-row execution of relational select calls scalar evaluation of a predicate, which in turns calls relational execution of a subquery.

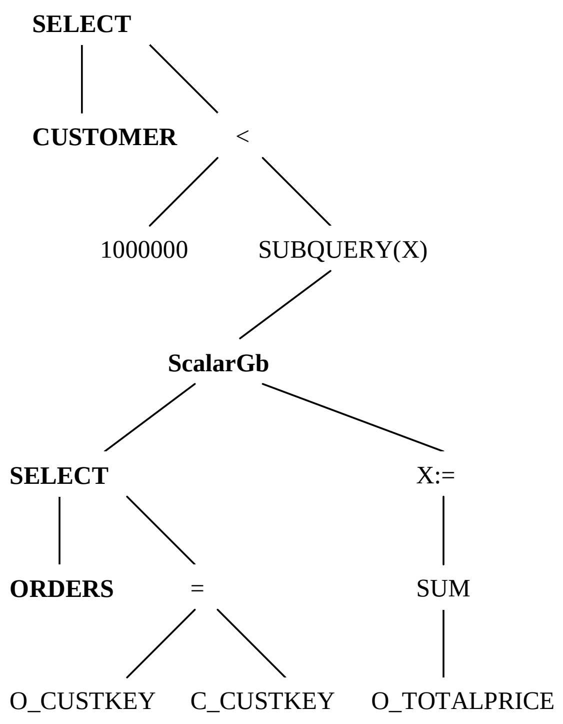


Figure 3: Direct algebraic representation of subquery.

2.2 Algebraic representation with Apply

Mutual recursion between scalar and relation nodes in the algebrizer output is removed by introducing Apply operators. The general scheme is to evaluate the subquery explicitly before the operator whose scalar expressions requires the subquery result. Say there is a relational operator $\odot$ on input R, with a scalar argument e using a subquery Q. We execute the subquery first using Apply, such that the subquery result is available as a (new) column q; then replace the subquery utilization by such variable:

$$\odot_{e(Q)} R \ \rightsquigarrow \ \odot_{e(q)}(R \ \mathcal{A}^{\otimes} \ Q).$$

As an example, removing mutual recursion from the operator tree in Figure 3 results in the tree shown in Figure 2. An operator $\mathcal{A}^{\times}$ is introduced below the relational select to compute the subquery, whose result is stored in column X. Figure 2 no longer shows the expanded operator trees for scalar expressions.

We showed how to remove one subquery from a scalar expression, but the technique naturally applies to multiple subqueries, in which case a sequence of Apply operators compute the various subqueries over the relational input.

Straightforward execution at this point is still based on nested loops. However, recursive calls between scalar and relational execution are removed, since scalar evaluation never needs to call back into the relational engine. Removing mutual recursion not only can have an impact on performance, but it also simplifies implementation.

2.3 Removal of Apply

Given a relational expression with Apply operators, it is possible to obtain an equivalent expression that does not use Apply. The process consists of pushing down Apply in the operator tree, towards the leaves, until the right child of Apply is no longer parameterized off the left child. Figure 4 describes the properties that allow this pushing —see [17, 6] for additional details, and discussion on these properties.

$$\begin{aligned}
R \ \mathcal{A}^{\otimes} \ E &= R \otimes_{\text{true}} E, & (1)\\
&\quad \text{if no parameters in } E \text{ resolved from } R\\
R \ \mathcal{A}^{\otimes} \ (\sigma_p E) &= R \otimes_p E, & (2)\\
&\quad \text{if no parameters in } E \text{ resolved from } R\\
R \, \mathcal{A}^{\times} \ (\sigma_p E) &= \sigma_p(R \, \mathcal{A}^{\times} \ E) & (3)\\
R \, \mathcal{A}^{\times} \ (\pi_v E) &= \pi_{v \,\cup\, \text{columns}(R)}(R \, \mathcal{A}^{\times} \ E) & (4)\\
R \, \mathcal{A}^{\times} \ (E_1 \cup E_2) &= (R \, \mathcal{A}^{\times} \ E_1) \cup (R \, \mathcal{A}^{\times} \ E_2) & (5)\\
R \, \mathcal{A}^{\times} \ (E_1 - E_2) &= (R \, \mathcal{A}^{\times} \ E_1) - (R \, \mathcal{A}^{\times} \ E_2) & (6)\\
R \, \mathcal{A}^{\times} \ (E_1 \times E_2) &= (R \, \mathcal{A}^{\times} \ E_1) \bowtie_{R.key} (R \, \mathcal{A}^{\times} \ E_2) & (7)\\
R \, \mathcal{A}^{\times} \ (\mathcal{G}_{A,F} E) &= \mathcal{G}_{A \,\cup\, \text{columns}(R),F}(R \, \mathcal{A}^{\times} \ E) & (8)\\
R \, \mathcal{A}^{\times} \ (\mathcal{G}^1_F E) &= \mathcal{G}_{\text{columns}(R),F'}(R \, \mathcal{A}^{\text{LOJ}} \ E) & (9)
\end{aligned}$$

Identities 7 through 9 require that R contain a key $R.key$. In identity (7), Join on $R.key$ is used as a shorthand for the obvious predicate. In identity (9), F' contains aggregates in F expressed over a single-column —for example, if F is COUNT(*), then F' is COUNT(C) for some not-nullable column C from E. Identity (9) is valid for all aggregates such that agg($\emptyset$) = agg({null}), which is true for SQL aggregates.

Figure 4: Rules to remove correlations.

For example, consider the expression shown in Figure 2. On the right child of Apply there is a scalar aggregation, then a select, and below that point there are no more outer references. Apply removal is shown in Fig 5. Identity (9) is used to push Apply below Scalar GroupBy; then Identity (2) is used to absorb the last parameterized select and remove the Apply. This results in the strategy of outerjoin followed by aggregate. In addition, due to the predicate on the aggregate result, the left outerjoin (denoted LOJ) can be simplified to join.

2.4 All SQL subqueries

So far we have described transformations that normalize subqueries into standard relational operators, and exemplified the procedure using a scalar aggregate subquery. We now describe additional SQL subquery scenarios, and how our scheme is affected on those.

For boolean-valued subqueries, i. e. EXISTS, NOT EXISTS, IN subquery, and quantified comparisons, the subquery can be rewritten as a scalar COUNT aggregate. From the utilization context of the aggregate result, either equal to zero or greater than zero, it is possible for the aggregate operator to stop requesting rows as soon as one has been found, since additional rows do not affect the result of the comparison.

A common case that is further optimized is when a relational select has an existential subquery as its only predicate (or when such select can be created by splitting another that ANDs an existential subquery with other conditions). In this case, the complete select operator is turned into Apply-semijoin for *exists*, or Apply-antisemijoin for *not*

$$\begin{aligned}
&\sigma_{1000000<X}(\text{CUSTOMER } \mathcal{A}^{\times}\ \mathcal{G}^{1}_{X=\text{SUM(O\_PRICE)}}\sigma_{\text{O\_CUSTKEY=C\_CUSTKEY}}\text{ORDERS})\\
&\quad= \sigma_{1000000<X}\ \mathcal{G}_{\text{C\_CUSTKEY},X=\text{SUM(O\_PRICE)}}(\text{CUSTOMER } \mathcal{A}^{\text{LOJ}}\ \sigma_{\text{O\_CUSTKEY=C\_CUSTKEY}}\text{ORDERS}), \text{by identity (9)}\\
&\quad= \sigma_{1000000<X}\ \mathcal{G}_{\text{C\_CUSTKEY},X=\text{SUM(O\_PRICE)}}(\text{CUSTOMER LOJ}_{\text{O\_CUSTKEY=C\_CUSTKEY}}\text{ORDERS}), \text{by identity (2)}\\
&\quad= \sigma_{1000000<X}\ \mathcal{G}_{\text{C\_CUSTKEY},X=\text{SUM(O\_PRICE)}}(\text{CUSTOMER} \bowtie_{\text{O\_CUSTKEY=C\_CUSTKEY}} \text{ORDERS}), \text{by outerjoin simplification.}
\end{aligned}$$

Figure 5: Example of correlation removal.

exists. Such Apply is then converted into a non-correlated expression, if possible, using Identity (2). For the resulting semijoin, we consider execution as join followed by GroupBy (distincting), which follows from the definition of semijoin. This GroupBy is also subject to reordering, covering the semijoin strategies suggested in [14].

There are two scenarios where normalization into standard relational algebra operators is hindered in a fundamental way. We call those *exception subqueries* and they require scalar-specific features. Consider the following query.

```
Q2: SELECT C_NAME,
           (SELECT O_ORDERKEY FROM ORDERS
           WHERE O_CUSTKEY = C_CUSTKEY)
    FROM CUSTOMER
```

For every customer, output the customer name, and the result of a subquery that retrieves an oderkey. There are three cases: If exactly one row is returned from the subquery, then such value is used in the scalar expression; if no rows are returned, then NULL is used; finally, if more than one row is returned, then a run-time error is generated [13]. The above query is valid, but will generate a run-time error if there happens to be a customer with more two or more orders. Since there are no run-time errors in standard relational algebra, we need an additional operator to represent these subqueries. We call such operator Max1row. It takes a relational input and passes input rows unmodified, and it generates a run-time error if the input has more than one row. It is placed in the right subexpression of Apply, to verify the SQL subquery semantics.

There is some amount of reordering that can be done on Max1row, but we find this case uninteresting. In our experience, at most one row is returned in most meaningful cases, and the compiler can detect this from information about keys. There is no need for Max1row then. For example, we reverse the roles of the tables next, retrieving customer name for each order. The resulting query is more meaningful, and the compiler avoids the use of Max1row, as long as S_CUSTKEY is a declared key.

```
SELECT O_ORDERKEY,
       (SELECT C_NAME FROM CUSTOMER
       WHERE C_CUSTKEY = O_CUSTKEY)
FROM ORDERS
```

Another problematic construct is conditional scalar execution, expressed in SQL as CASE WHEN <COND> THEN <VALUE1> ELSE <VALUE2> END. The point is, <VALUE2> should not be evaluated when <COND> is true. Therefore, eager execution of a subquery, say contained in <VALUE2>, is incorrect, in particular if it happens to generate a run-time error. To deal with this scenario, we use a modified version of Apply with conditional execution of the parameterized expression, based on a predicate. Implementing this is required for completeness, but in our experience this scenario is very rare in practice.

2.5 Subquery classes

Our approach delineates three broad classes of subquery usage in SQL, which grant different treatment from the query processor.

Class 1. Subqueries that can be removed with no additional common subexpressions. In general, removing Apply requires introduction of additional common subexpressions —e. g. see Identity (5), which introduces two copies of R. Cases that do not require introducing common subexpressions are easier. In particular, the common case of subqueries that are formed by a simple select/project/join/aggregate block are easy to handle. An example query for this class is **Q1** from Section 1.1. Our normalization scheme produces an operator tree with standard relational operators and no correlations involved. Afterwards, cost-based optimization will consider a number of alternatives, including re-introduction of a correlated execution, which can be very effective if few outer rows are processed and appropriate indices exist. Earlier research on SQL subqueries has implicitly focused on a subset of this class of subqueries.

Any subquery processing strategy that applies on Class 1 subqueries must have a more primitive formulation, applicable on expressions without correlations. For example, the "magic" strategy for subquery evaluation described in [15] has a primitive formulation on join and aggregation [17].

Class 2. Subqueries that are removed by introducing additional common subexpressions. Achieving optimality and syntax-independence in this class requires an understanding of the plan space and mechanisms to generate plans of interest, for queries with common subexpressions, which we believe requires additional research. In our current implementation these subqueries are not removed during normalization, but we do still consider unnesting transformations during cost-based optimization. This can lead to improvements over the original subquery form, and cost-based decisions will be used to choose appropriate execution plans. However, no syntax-independence is provided, and there is no simple characterization of the space of interest. We are not aware of any work that has attempted to optimize this class of subqueries.

It is hard to formulate a short, meaningful query that fits in this class, using the TPCH schema. The next query is a valid (but meaningless) SQL example for the class, using UNION ALL. Removing Apply requires Identity (5), which introduces multiple copies of the outer table.

```
SELECT *,
FROM PARTSUPP
WHERE 100 >
    (SELECT SUM(S_ACCTBAL) FROM
        (SELECT S_ACCTBAL
        FROM SUPPLIER
        WHERE S_SUPPKEY = PS_SUPPKEY
        UNION ALL
        SELECT P_RETAILPRICE
        FROM PART
        WHERE P_PARTKEY = PS_PARTKEY)
        AS UNIONRESULT)
```

Class 3. Exception subqueries. These subqueries are fundamentally non-relational, as they require scalar-specific features such as generating run-time errors. We consider these cases relatively uninteresting, and rare in practice. To our knowledge, no research work has addressed queries in this class. An example query for this class is **Q2** described in Section 2.4.

3. COMPREHENSIVE OPTIMIZATION OF AGGREGATION

In this section we describe techniques for efficient processing of queries with aggregations. We extend, and formulate in algebraic terms, earlier work related to reordering of GroupBy/Aggregate, and segmented execution of queries. The result is a number of transformation rules to generate interesting execution strategies, to be used in the context of cost-based optimization.

3.1 Reordering GroupBy

Aggregating a relation reduces its cardinality. This may give us the impression that we can use the same early evaluation strategy that we use for filters. But that is not always the case because aggregation can be quite expensive and the cost depends heavily on the number of rows being aggregated. Take the case of joins. If the join predicate reduces the cardinality dramatically it may be better to perform the GroupBy after the join. Doing it later avoids unnecessary calculation of aggregates which will be thrown away by the join. Another reason can be the existence of appropriate indices, which allows the join to be performed as an index-lookup. This may not be possible when the aggregate obstructs it. Therefore, it is best to generate both the alternatives and leave the choice to the cost based optimizer.

In this section we study the conditions necessary to move a GroupBy around filters, joins, semijoins etc. As we mentioned earlier some of these ideas were developed in [3, 18]. We formulate them here in a way that they can be implemented as primitive optimization rules. In the next section we do the same for outerjoins. In our discussion we will formally denote a GroupBy as $\mathcal{G}_{A,F}$ where A is the set of grouping columns and F are the aggregate functions.

Let us begin with the discussion of a primitive to reorder a filter and an aggregate. For a GroupBy operator, rows in the input relation that have same values for the grouping columns generate exactly one output row. If a filter above an aggregate rejects a row, it needs to reject the whole group that generated this row after it is pushed below the aggregate. The only characteristic shared by this group of rows is the values of the grouping columns. Therefore we can move a filter around a GroupBy if and only if all the columns used in the filter are functionally determined by the grouping columns in the input relation.

Moving aggregates around a join is a little more complicated. A GroupBy can be pushed below a join if the grouping columns, the aggregate calculations and the join predicate each satisfy certain conditions. Suppose we have a GroupBy above a join of two relations, i.e. $\mathcal{G}_{A,F}(S \bowtie_p R)$, and we want to push the GroupBy below the join so that the relation R is aggregated before it is joined i.e. $S \bowtie_p (\mathcal{G}_{A \cup columns(p) - columns(S),F} R)$. This is feasible if and only if the following three conditions are met -

1. If a column used in the join predicate p is defined by the relation R then it is part of the grouping columns.
2. The key of the relation S is part of the grouping columns.
3. The aggregate expressions only use columns defined by the relation R.

To see why this is correct, think of a join as a cross product followed by a filter. The first two condition ensure that all the columns of the predicate are functionally determined by the grouping columns. This allows us to push the GroupBy below the filter. The second condition implies that no two rows from the relation S are included in the same group during aggregation. The last condition ensures that the aggregate expressions can be calculated with just the relation R. These two conditions together enable us to push the GroupBy below the cross product.

Pulling a GroupBy above a join is a lot easier. All that is required is that the relation being joined has a key and that the join predicate does not use the results of the aggregate functions. These two restrictions follow directly from the discussion above. Formally,

$$S \bowtie_p (\mathcal{G}_{A,F} R) = \mathcal{G}_{A \cup columns(S),F}(S \bowtie_p R)$$

Even these two conditions are not as restrictive as they appear at first sight. If the relation S does not have a key, one can always be manufactured during execution. As for the second condition, conjuncts of a join predicate can always be separated out into a filter performed after the join. We can apply this strategy to the predicate p if it uses results of the aggregate functions.

One can think of semijoins and antisemijoins as filters since they include or exclude rows of a relation based on the column values. The conditions necessary to reorder these operators around a GroupBy can therefore be easily deduced from those for a filter. Suppose we have an aggregate followed by a semijoin i.e. $(\mathcal{G}_{A,F} R) \rhd\!\!<_p S$ we can push the semijoin below if and only if p does not use the result of any aggregate expressions and every column of predicate p, say c, satisfies the condition that if $c \notin columns(S)$ then c is functionally determined by the grouping columns(i.e. the set A). The condition for antisemijoin is exactly the same.

3.2 Moving GroupBy around an outerjoin

Removing correlations for scalar valued subqueries results in an outerjoin followed by a GroupBy. Therefore it is especially important for our optimizer to have primitives that allow reordering of these two operators. None of the literature cited above discusses this issue.

In order to push a GroupBy below an outer join the join predicate, grouping columns and the aggregate calculations once again have to meet the three conditions mentioned above. The only difference is that an extra project may have to be added above the outerjoin if the aggregate expressions do not meet a certain condition.

Let us see why this works and why a project may be necessary. The result of an outerjoin has two types of rows viz. those that match, and those that do not and are padded with NULLs. We know that our grouping columns include a key of the outer relation. This implies that a group can never have both matched and unmatched rows. For the rows that match, correctness can be proved by using the same argument as join. For the rows that do not match, early aggregation means that they will not be aggregated at all! This is where the extra condition and the optional project comes in.

In the result of an outerjoin, an unmatched row appears exactly once. Therefore given that our grouping columns include the key of the outer relation, a group that has an unmatched row cannot have any other row. The aggregate functions use only the columns from the non-outer relation. For an unmatched row all these columns are NULL. Therefore the property of aggregate expressions that is important to us is the result of applying it to NULLs. If the result is NULL as it is for most simple aggregate expressions, we need do nothing more. The outerjoin will automatically provide the NULLs we need. For the aggregate expressions which do not result in a NULL, we need to add a project which for each unmatched row sets the aggregate result to the appropriate constant value.[2] Note that this constant can be calculated at compile time. For COUNT(*), the value of this constant is zero.

Formally we have

$$\mathcal{G}_{A,F}(S \text{ LOJ}_p R) = \pi_c(S \text{ LOJ}_p(\mathcal{G}_{A-columns(S),F} R))$$

where the computing project π_c introduces the non-NULL results if necessary.

As an example, in the outerjoin/aggregate strategy shown earlier in Section 1.1, the aggregation can be pushed down below the outerjoin.

```
SELECT C_CUSTKEY
FROM CUSTOMER LEFT OUTER JOIN
        (SELECT O_CUSTKEY,
                SUM(O_TOTALPRICE) AS TOTALORDER
        FROM ORDERS
        GROUP BY C_CUSTKEY) AS AggResult
ON O_CUSTKEY = C_CUSTKEY
WHERE 1000000 < TOTALORDER
```

No computing projects are required here as the aggregate expression $sum(o\_totalprice)$ does result in NULL when calculated on a singleton NULL.

3.3 Local Aggregates

The restrictions on the join predicate, grouping column etc. mean that it is not always possible to push a GroupBy below a join. But sometimes it may be possible to do part of the aggregations before the join and then combine these partially aggregated rows afterwards to get the final result. This can be efficient because it reduces the cardinality of the join inputs. Some of these ideas are discussed in [3, 18]. Here we introduce a new operator called LocalGroupBy (formally $\mathcal{LG}$) and develop primitives that allow us to push it below other relational operators giving us this ability to aggregate early.

In order to introduce a LocalGroupBy, the aggregate function has to be split into two new functions - one which does the early partial aggregation, called a local aggregate in this paper, and another which combines these aggregates to generate the final result, called a global aggregate in this paper. Formally if we have an aggregate function f, we need a local aggregate function f_l and a global aggregate function f_g such that for any set S and for any partition of it $\{S_1, S_2, \ldots, S_n\}$ we have

$$f(\bigcup_{i=1}^{n} S_i) = f_g(\bigcup_{i=1}^{n} f_l(S_i))$$

Note that the implementation, whether hash based or sort based, of aggregate functions in a query execution engine requires this ability of splitting an aggregate into local and global components, if it has to spill data to disk and then recombine it.[3]

If all the aggregate functions used in a GroupBy can be split this way, which should almost always be true, we can replace a "standard" GroupBy with a LocalGroupBy followed by a "global" GroupBy. Formally we have

$$\mathcal{G}_{A,F} R = \mathcal{G}_{A,F_g} \mathcal{LG}_{A,F_l} R$$

where F_l and F_g are the local and global aggregate expressions corresponding to F.

LocalGroupBy has the interesting property that its grouping columns can be extended without affecting the final result. Adding a new column to the set of grouping columns just partitions the groups further. But since we have a final "global" aggregate to combine these partial results, the final result remains unchanged.

This ability to extend grouping columns gives us infinite freedom. Take the case of a join. It is now trivially easy to satisfy the first two restrictions we mentioned in our discussion about reordering aggregates with a join. We just add the necessary columns. That leaves us with only the third restriction about aggregate functions. The solution once again is to extend the grouping columns. An aggregate function computation can be removed from a LocalGroupBy using the following steps: First extend the grouping columns by adding the aggregate input column (or expression, in general); at this point, the aggregate function is operating on a set of COUNT(*) identical values. Now, replace the aggregate computation by a later project that, in general, computes the result based on COUNT(*) and the original aggregate input, which is a constant for the group. For example, suppose the grouping columns include the column a and one of the aggregates being calculated is SUM(a). We can get the same answer with the expression ($a\times$COUNT(\*)) calculated after the aggregation. This property of aggregate functions allows

[2]Detection of unmatched rows requires a non-nullable column from the inners side, which can always be manufactured, or else changes to outerjoin to provide a *match* column.

[3]There can be *composite* aggregates, such as AVG, which do not have local/global versions. But they are computed based on primitive aggregates that do, since we want to be able to compute them using algorithms that spill to disk.

us to replace any aggregate function used in a LocalGroupBy with a COUNT(*), thus addressing the third restriction. We can push a LocalGroupBy below any join and to any side of the join. More details about reordering LocalGroupBy with several other operators, and proofs of correctness can be found in [10].

Note that the actual implementation of a LocalGroupBy in the query execution engine need not be different from a GroupBy. We use a separate operator only to make the job of optimizer easy since the transformations described above are only valid for the LocalGroupBy operator.

3.4 Segmented execution

Frequently the process of correlation removal for scalar valued subqueries results in two almost identical expressions joined together. The only difference is that one of them is aggregated and the other is not. A simple example of this is Query 17 of the TPC-H benchmark. After removing the correlation, the SQL representation of the query is -

```
SELECT SUM(L_EXTENDEDPRICE)/7.0 AS AVG_YEARLY
FROM LINEITEM, PART,
        (SELECT L_PARTKEY AS L2_PARTKEY,
                0.2 * AVG(L_QUANTITY) AS X
        FROM LINEITEM
        GROUP BY L_PARTKEY) AS AGGRESULT
WHERE P_PARTKEY = L_PARTKEY
      AND P_BRAND = 'BRAND#23'
      AND P_CONTAINER = 'MED BOX'
      AND P_PARTKEY = L2_PARTKEY
      AND L_QUANTITY < X
```

Here we have two instances of the LINEITEM table joined together where one of them is grouped by L_PARTKEY. The SQL representation for this join using the implied predicate is -

```
SELECT L_PARTKEY, L_EXTENDEDPRICE
FROM LINEITEM,
        (SELECT L_PARTKEY AS L2_PARTKEY,
                0.2 * AVG(L_QUANTITY) AS X
        FROM LINEITEM
        GROUP BY L_PARTKEY) AS AGGRESULT
WHERE L_PARTKEY = L2_PARTKEY
      AND L_QUANTITY < X
```

Semantically this join is trying to find all the LINEITEM rows where the quantity ordered is less than 20% of the average for that part. But this means the selection of a particular LINEITEM row does not require the whole derived table AGGRESULT. All we need is the average quantity for the part referenced in that row. That gives rise to an interesting correlated execution strategy. We can segment the LINEITEM table based on the part and calculate the join for each segment independently. A SegmentApply does exactly that.

A SegmentApply operator is very similar to the Apply operator we studied in detail. The only difference is that the parameter is a set of rows rather than a single row. The inner child of the SegmentApply is an expression that uses this set. Figure 6 shows the transformed version of the LINEITEM join. Segmented execution was discussed in [2]. Our contribution is to formulate it as an algebraic operator so that it can be used in a cost based optimizer. The formalism also allows us to introduce reordering primitives.

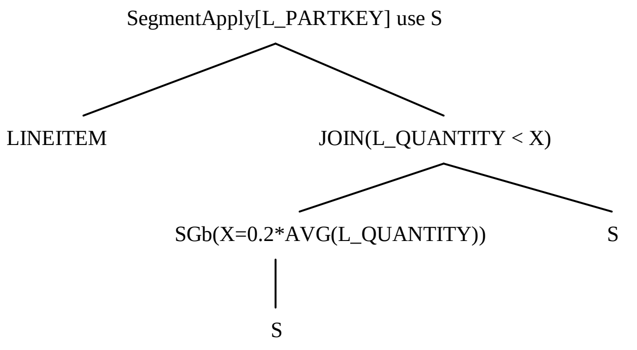


Figure 6: SegmentApply

3.4.1 *Introducing SegmentApply*

Whenever we see two instances of an expression connected by a join, where one of the expressions may optionally have an extra aggregate and/or an extra filter, we attempt to generate an alternative that uses SegmentApply. The key thing to look for is a conjunct in the join predicate that is an equality comparison between two instances of the same column from the two expressions. Such a comparison implies that rows for which this column differs will never match. We can therefore use the column to partition the relation. Note that the join predicate can have multiple columns that satisfy this criterion allowing us to have finer segments. If the join predicate does yield such segmenting columns, we introduce a correlated execution alternative that uses SegmentApply.

Removing correlations for an existential subquery generates a semijoin, or antisemijoin. The argument in the previous section is valid for those operators too. We can therefore introduce a SegmentApply in both these cases, if there are common subexpressions and the required equality conditions hold. The only difference is in the correlated expression.

3.4.2 *Moving joins around SegmentApply*

Going back to our TPC-H example, we can see that looking at all the LINEITEM rows is an overkill because the final result cares only about parts with a specific brand and a specific container. It would be more efficient to only process LINEITEM rows for these parts. This optimization is possible if we add a primitive to reorder a SegmentApply and a join that allows us to reduce the LINEITEM table earlier by joining it with the PART table. Algebraic representation of SegmentApply allows us to manipulate it like any other operator and makes it very easy to add such a primitive.

The key condition to check when pushing a join below a SegmentApply is preservation of the segment. The join predicate needs to be such that it either allows all rows in a segment to pass through or none of them to do so. If pushing the join removes only some of the rows, the result of the correlated expression will not be same. Suppose we have a SegmentApply expression $(R\ \mathcal{SA}_A\ E) \bowtie_p T$ where A is the set of segmenting columns and p is the join predicate. The all or none condition means that the predicate p can use only the segmenting columns or columns of the relation T and nothing else.[4]

[4]As we mentioned in case of GroupBy, this is not as restrictive as it appears. A join predicate can be split up into a

A join predicate which satisfies this condition may still change the segment. If the join is such that one row of R matches multiple rows of T all these resulting rows will be included in the segment. There is a simple solution to this however and it is to add the key of relation T to the segmenting columns.[5] This ensures that each instance of the row goes to a different segment.

Formally we have

$$(R\ \mathcal{SA}_A\ E) \bowtie_p T = (R \bowtie_p T)\ \mathcal{SA}_{A \cup columns(T)}\ E$$
$$\text{iff } columns(p) \subseteq A \cup columns(T).$$

The predicate of the join with the PART table in TPC-H Query 17 uses the column L_PARTKEY, which is our segmenting column. We can therefore push the join below the SegmentApply. We need to add the key of the PART table viz. P_PARTKEY to the segmenting columns, but since the two segmenting columns L_PARTKEY and P_PARTKEY are equal we can safely remove one of them. The final expression is shown in Figure 7.

Remember once again that these alternatives will be costed and used in the final plan only if they appear cheaper.

4. COMPILATION IN SQL SERVER

The following is a brief description of relevant compilation steps in SQL Server, and how different optimizations are incorporated. For the most part, the material of Section 2 deals with preparing a normalized operator tree, to be used as input for cost-based optimization. The material of Section 3 deals mostly with interesting alternatives that can reduce execution time dramatically, but need to be costed to determine when to use them, and are therefore part of cost-based optimization.

Parse and bind. This step is a relatively direct translation of SQL text into an operator tree containing both relational and scalar operators, in the form shown in Section 2.1. Note that current SQL allows the use of (correlated) subqueries anywhere scalar expressions are allowed, including SELECT and WHERE clauses. Any scalar expression may have a relational expression as children.

Query normalization. This step transforms an operator tree into a simplified/normalized form. Simplifications include, for example, turning outerjoins into joins, when possible, and detecting empty subexpressions. For subqueries, mutual recursion between relational and scalar execution is removed, which is always possible; and correlations are removed, which is usually possible. At the end of normalization, most common forms of subqueries have been turned into some join variant.

Cost-based optimization. Execution alternatives are generated using transformation rules, and the plan with cheapest estimated cost is selected for execution. Important classes of optimizations include: Reordering of join variants; reordering of GroupBy with join variants; considering special strategies for GroupBy; and introduction of correlated execution (the simplest and most common being index-lookup-join). The architecture of our cost-based optimizer follows the main lines of the Volcano optimizer [9], so that generation of interesting reorderings is done by means of transformation rules.

5. PERFORMANCE RESULTS

We now present the results of our optimizations on queries from the TPC-H benchmark. Normalization flattens all subqueries in the benchmark, but this does not have a direct impact on query performance —it is reordering, and GroupBy optimization techniques that do have an impact. In particular, our full set of techniques apply on Query2 and Query17.

Figure 8 lists all published TPC-H results on the 300GB scale, as of November 27th, 2000, which we include here in compliance with the TPC reporting rules. In Figure 9 we plot the published results of elapsed time for Query2 and Query17. The numbers are taken from the information available in the TPC web page (http://www.tpc.org), for all eight TPC-H results on the 300GB scale. None of those results use clusters. The x axis plots the number of processors used in each benchmark result, and the y axis plots the elapsed execution time on the power run. For example, the lower left point in the graph for Query17 corresponds to an elapsed time of 79.7 sec, obtained on 8 processors; while the lower right point on the same graph is for 210.4 sec time on 64 processors. Points are separated by DBMS, since the different query processors are likely to use different technologies. On these two queries, SQL Server has published the fastest results, even on a fraction of the processors used by other systems.

TPC-H has strict rules on what indices are allowed, reducing the relative impact of physical database design, in comparison to query processor technology and hardware. Several factors contribute to the fast SQL Server result, including both query optimization and execution. An essential component are the techniques described earlier in this paper.

6. CONCLUSIONS

In this paper we described the approach used in Microsoft SQL Server to process queries with subqueries and/or aggregations. The approach is based on several ideas:

Subqueries and aggregation should be handled by orthogonal optimizations. Earlier work has sometimes combined multiple, independent primitives to derive strategies that are suitable for some cases. What we do instead is to separate out those independent, small primitives. This allows finer granularity of their application; it generates a richer set of execution plans; it makes for more modular proofs; and it simplifies implementation.

Algebraic constructs for parameterized subexpressions are a useful query processing tool. The Apply and SegmentApply constructs do not add expressive power to relational algebra. However, they facilitate query representation for some query language constructs, and significantly enhance the space of alternatives considered for execution. This is achieved in an algebraic fashion, allowing both formal algebraic manipulation as well as smooth integration in

filter applied after the join.

[5]This is exactly what we did when pushing a join below a GroupBy, which should not be surprising since a SegmentApply is just a more general version of a GroupBy.

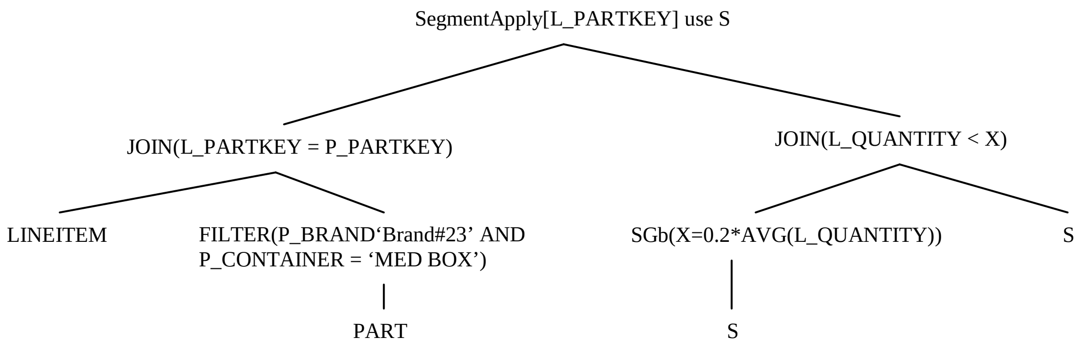
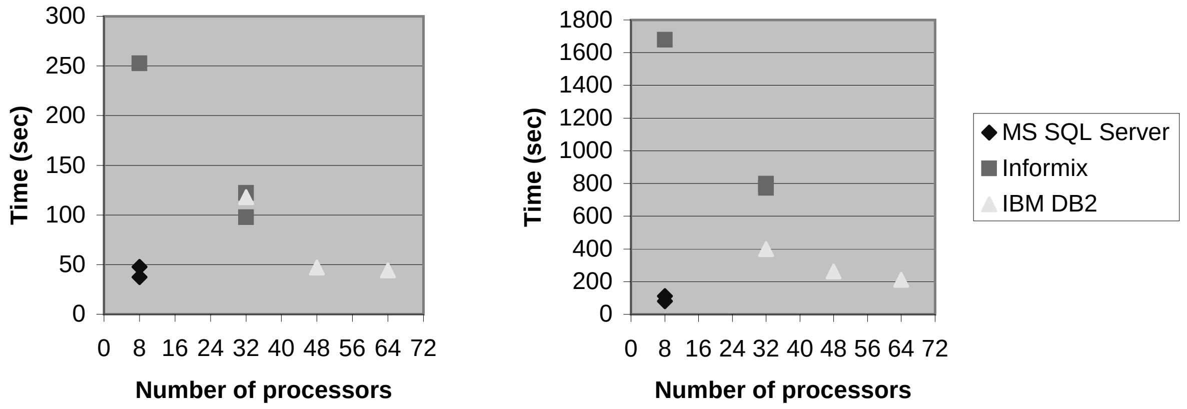

SegmentApply[L_PARTKEY] use S
JOIN(L_PARTKEY = P_PARTKEY)
JOIN(L_QUANTITY < X)
LINEITEM
FILTER(P_BRAND'Brand#23' AND P_CONTAINER = 'MED BOX')
PART
SGb(X=0.2*AVG(L_QUANTITY))
S
S

Figure 7: Join with PART pushed below SegmentApply

System	Database	QphH @300GB	Price/QphH @300GB, in US$	System Availability	Date Submitted
COMPAQ ProLiant 8000-8P	Microsoft SQL Server 2000	1506	280	11/17/00	11/17/00
HP NetServer LXr 8500	Microsoft SQL Server 2000	1402	207	08/18/00	08/18/00
HP 9000 N4000 Enterprise Server	Informix Extended Parallel Server 8.30 FC2	1592	973	05/02/00	05/02/00
COMPAQ AlphaServer GS320 Model 6/731	Informix XPS 8.30 FC3	4951	983	08/31/00	07/13/00
HP 9000 V2500 Enterprise Server	Informix Extended Parallel Server 8.30 FC2	3714	1119	12/17/99	10/21/99
IBM NUMA-Q 2000	IBM DB2 UDB 7.1	4027	652	09/05/00	09/05/00
IBM NUMA-Q 2000	IBM DB2 UDB 7.1	5923	653	09/05/00	09/05/00
IBM NUMA-Q 2000	IBM DB2 UDB 7.1	7334	616	08/15/00	05/03/00

Figure 8: Published TPC-H results for 300GB, as of November 27th, 2000. Included here in compliance with the TPC reporting rules.



Results for Query 2.

Results for Query 17.

Figure 9: Query performance reported in TPC-H results, 300GB scale.

algebraic query processors.

Parameterized expressions (i. e. correlated subqueries) should be removed during query normalization, for syntax-independence. In Section 2 we described algebraic removal of correlations. We considered *all* SQL subqueries, and described what makes their replacement by standard operators easy, or hard, leading to three broad classes. On one side of the scale, subqueries whose body consists of a simple SQL query block always can and should be normalized out. In the next level, removal of subqueries requires introducing common subexpressions, which impose additional requirements on the query processor on the resulting "flat" expressions. Finally, subqueries that need checking for errors at execution time, such as verification that a scalar-valued subquery returns at most one row, are not relational in nature. From what we know, earlier work on subqueries has implicitly focused on the simplest class.

A rich set of alternatives for GroupBy/Aggregate execution should be considered during cost-based optimization. GroupBy/Aggregate appears frequently, either writen directly in the original query, or as a result of query normalization. In Section 3 we described two powerful techniques: Reordering of GroupBy, and segmented execution. We extend, and formulate in algebraic terms, earlier work, resulting in a number of transformation rules to generate interesting execution strategies. It is these optimizations that make for the order-of-magnitude performance improvements that we report.

We showed the effectiveness of our overall approach with figures from published TPC-H results, in Section 5. In particular, our full set of techniques apply on Query2 and Query17. From all published TPC-H results in the 300GB scale, at the time of writing (November 2000), SQL Server has the fastest results on those queries, even on a fraction of the processors used by other systems.

7. REFERENCES

[1] P. Celis and H. Zeller. Subquery elimination: A complete unnesting algorithm for an extended relational algebra. In *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K*, page 321, 1997.

[2] D. Chatziantoniou and K. A. Ross. Groupwise processing of relational queries. In *Proceedings of the 23rd International Conference on Very Large Databases, Athens*, pages 476–485, 1997.

[3] S. Chaudhuri and K. Shim. Including Group-By in query optimization. In *Proceedings of the Twentieth International Conference on Very Large Databases, Santiago*, pages 354–366, 1994.

[4] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proceedings of ACM SIGMOD 1992*, pages 383–392, 1992.

[5] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proceedings of the Thirteenth International Conference on Very Large Databases, Brighton*, pages 197–208, 1987.

[6] C. A. Galindo-Legaria. Parameterized queries and nesting equivalences. Technical report, Microsoft, 2001. MSR-TR-2000-31.

[7] C. A. Galindo-Legaria and A. Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Transactions on Database Systems*, 22(1):43–73, Mar. 1997.

[8] G. Graefe. The Cascades framework for query optimization. *Data Engineering Bulletin*, 18(3):19–29, 1995.

[9] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient. In *Proceedings of the Ninth International Conference on Data Engineering, Viena, Austria*, pages 209–218, 1993.

[10] M. M. Joshi and C. A. Galindo-Legaria. Properties of the GroupBy/Aggregate relational operator. Technical report, Microsoft, 2001. MSR-TR-2001-13.

[11] W. Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems*, 7(3):443–469, Sept. 1982.

[12] T. Leung, G. Mitchell, B. Subramanian, B. Vance, S. L. Vandenberg, and S. B. Zdonik. The AQUA data model and algebra. In *DBPL*, pages 157–175, 1993.

[13] J. Melton and A. R. Simon. *Understanding the new SQL: A complete guide.* Morgan Kaufmann, San Francisco, 1993.

[14] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In *Proceedings of ACM SIGMOD 1992*, pages 39–48, 1992.

[15] P. Seshadri, H. Pirahesh, and T. Y. C. Leung. Complex query decorrelation. In *Proceedings of the Twelfth International Conference on Data Engineering, New Orleans, Luisiana*, pages 450–458, 1996.

[16] G. Shaw and S. Zdonik. An object-oriented query algebra. In *Proceedings of the Second International Workshop on Database Programming Languages*, pages 249–225, 1989.

[17] Q. Wang, D. Maier, and L. Shapiro. Algebraic unnesting of nested object queries. Technical report, Oregon Graduate Institute, 1999. CSE-99-013.

[18] Y. P. Yan and P. A. Larson. Eager aggregation and lazy aggregation. In *Proceedings of the 21st International Conference on Very Large Databases, Zurich*, pages 345–357, 1995.