Execution Strategies for SQL Subqueries

Mostafa Elhemali, César A. Galindo-Legaria, Torsten Grabs, Milind M. Joshi
Microsoft Corp., One Microsoft Way, Redmond, WA 98052
{mostafae, cesarg, torsteng, milindj}@microsoft.com

ABSTRACT

Optimizing SQL subqueries has been an active area in database research and the database industry throughout the last decades. Previous work has already identified some approaches to efficiently execute relational subqueries. For satisfactory performance, proper choice of subquery execution strategies becomes even more essential today with the increase in decision support systems and automatically generated SQL, e.g., with ad-hoc reporting tools. This goes hand in hand with increasing query complexity and growing data volumes – which all pose challenges for an industrial-strength query optimizer.

This current paper explores the basic building blocks that Microsoft SQL Server utilizes to optimize and execute relational subqueries. We start with indispensable prerequisites such as detection and removal of correlations for subqueries. We identify a full spectrum of fundamental subquery execution strategies such as forward and reverse lookup as well as set-based approaches, explain the different execution strategies for subqueries implemented in SQL Server, and relate them to the current state of the art. To the best of our knowledge, several strategies discussed in this paper have not been published before.

An experimental evaluation complements the paper. It quantifies the performance characteristics of the different approaches and shows that indeed alternative execution strategies are needed in different circumstances, which make a cost-based query optimizer indispensable for adequate query performance.

Categories and Subject Descriptors

H.2.4 [**Database Management Systems**]: **Subjects:** Query processing, Relational databases. **Nouns:** Microsoft SQL Server

General Terms

Algorithms, Performance.

Keywords

Relational database systems, Query optimization, Subqueries, Microsoft SQL Server.

1. INTRODUCTION

Subqueries are a powerful improvement of SQL which has been further extended to scalar subqueries by the SQL/92 standard [1]. Most of its value derives from the ability to use subqueries orthogonally in all SQL clauses SELECT, FROM, WHERE, and HAVING. This facilitates query formulation in important application domains such as decision support, applications which automatically generate SQL queries, and new query languages.

SIGMOD'07, June 12–14, 2007, Beijing, China.

1.1 Application Scenarios

Decision Support. Decision support benchmarks such as TPC-H and TPC-DS make extensive use of subqueries: out of the 22 queries in the TPC-H benchmark, 10 queries use subqueries. Subqueries occur in the WHERE clause for 9 of the TPC-H queries and one query uses subqueries in the HAVING clause. A common practice is to relate a nested subquery to the outer query such that the subquery only processes values relevant to the rows of the outer query. This introduces a so-called *correlation* between the outer query and the subquery. As we will discuss in more detail, correlations make it challenging to find well-performing execution plans for queries with subqueries.

Automatically Generated Queries. Today, end users oftentimes use graphical interfaces to compose ad-hoc queries or to define reports against an underlying data store. The application behind the user interface then automatically generates SQL queries based on the interactive input provided by the end user. Microsoft SQL Server 2005 Reporting Services is an example of such an application: customers can define a model on top of a data store and interactively query the data behind the model. These queries are compiled into SQL if the underlying data store is a relational database system. The nesting capabilities and the orthogonality of the SQL language simplify automatic SQL code generation. The implementation of SQL Server Reporting Services relies on these characteristics of the SQL language and – depending on the user query against the model – introduces subqueries in the SELECT or the WHERE clause.

"Nested loops" languages. Unlike SQL, some query languages incorporate the notions of *iteration* and *parameterization* as a basic mechanism. XQuery is an example [2]. With XQuery, queries are formulated using the clauses FOR, LET, WHERE and RETURN which yields so-called FLWR-expressions. Nested FOR clauses are a common programming pattern, and they are used to iterate over XML elements at a given level in the document hierarchy and then descend deeper. The concepts and techniques presented in this paper are applicable to those scenarios as well. In fact, they are used in our implementation of XQuery.

1.2 Subquery Processing Overview

Our overall compilation logic, as it pertains to subquery processing, is shown in Figure 1. The framework shown separates distinct steps for the generation of efficient execution plans, and it aims to maximally leverage the various optimizations.

A first step of SQL compilation is the *subquery removal* process, which is covered in detail in Section 3 of this paper. The result is a relational expression that uses the *Apply* operator to abstract param-

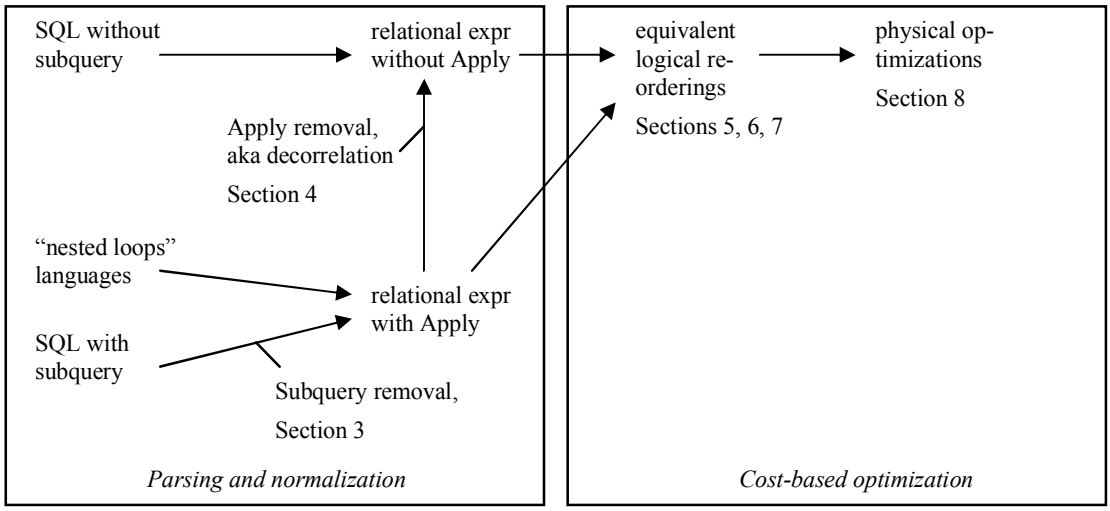


Figure 1: Overview of plan generation for subqueries in SQL Server and structure of the paper

eterized execution. A *nested loop* language is likely to be parseable directly into this algebraic form.

The *apply removal* process is covered in Section 4 of this paper. It eliminates the use of parameterization, which we identify with *decorrelating*. The result is a relational expression that uses different types of joins.

The optimizer then *explores* the decorrelated relational expression and generates various alternatives with identical query semantics. We call this the *logical level* of query optimization. This step aims at producing enough interesting plan alternatives such that a good query plan is among them. The optimizer computes cost estimates for plans and picks the most cost-effective one. A sufficiently broad search space obviously is a prerequisite for good performance. Logical optimizations for subqueries are covered by Section 5 to 7 where we discuss strategies for logical exploration of semijoin and antijoin, techniques to deal with subqueries in disjunctions, and some Group By scenarios.

After exploration of the relational expression is complete, additional optimizations at the *physical level* help to further improve performance. These techniques such as batch sort or prefetch are discussed in Section 8.

1.3 Subquery Execution Strategies

The query optimizer explores different physical execution strategies for the various logical query plan alternatives. This step generates execution plan alternatives with different performance characteristics. In the context of subquery optimization, the following execution strategies are possible: (1) navigational strategies, or (2) set-oriented strategies. *Navigational strategies* rely on nested loop joins for implementation while two interesting classes of navigational strategies are conceivable, namely forward lookup and reverse lookup. *Forward lookup* starts processing the outer query and, as outer rows are being generated, invokes the subquery one outer row at a time. *Reverse lookup* in turn starts with the subquery and processes the outer query one subquery row at a time. Set-oriented processing finally requires that the query could be successfully decorrelated. If this is the case, set operations such as hash and merge join can implement the query.

1.4 Challenges and Contributions

Depending on the cardinalities of the outer query and the subquery as well as the physical design of the database, different subquery execution strategies may differ greatly in their performance characteristics. This makes it a challenging topic for query optimization. This is reflected by the attention the area has received in previous work, e.g., [3, 4, 5, 6, 7, 8]. While [9] has specifically focused on optimization of subqueries with grouping at the logical level, this current paper takes a broader perspective on the problem. We do not limit the discussion to grouping, and we cover both the logical level and the physical level of plan generation. In particular, we investigate subqueries introduced by existential or universal quantification. Throughout the paper, we discuss rewrite strategies for subqueries at the logical level and explain how this facilitates finding a good execution plan among navigational and set-oriented alternatives at the physical level. In Section 10, our experimental evaluation of different subquery execution strategies with Microsoft SQL Server 2005 investigates the different plan choices in quantitative terms. The experiments also assess the effectiveness of a cost-based optimization approach in the presence of subqueries.

2. ALGEBRAIC REPRESENTATION OF SUBQUERIES

In this section we describe the algebraic representation of SQL subqueries. Having an algebraic representation is beneficial because it abstracts the semantics of operations, making them independent of query language, data structures or specific execution strategies. It also allows algebraic analysis and reasoning, including capturing reordering properties in the form of algebraic identities. Finally, it fits well into algebraic query processors like that of SQL Server.

2.1 Terminology

The basics of our algebraic model were discussed in [9]. Here, we briefly review this formulation and go over a number of optimizations and issues not covered in our earlier work. Since we deal with SQL, all operators in this paper are bag-oriented and we assume no automatic removal of duplicates. In particular, the union operator for most of the remainder of the paper is *UNION ALL* and we represent it using $\cup A$. Distinct union in turn is denoted as $\cup$. Besides union, we rely on the standard relational operators for grouping,

selection (filter) and projection: $\mathcal{G}_{A,F}R$ stands for a *GROUP BY* over relation R with a list A of grouping columns and a set F of aggregate functions that are applied to each group. $\pi[S]R$ denotes a *projection* of relation R onto the set of columns in S. $\sigma[p]R$ in turn represents a *selection* on relation R where p is used as a predicate to filter qualifying rows from R. For ease of presentation, we use CT(1) as a shorthand for a constant table which returns one row and no columns.Our algebraic formulation of subqueries is based on the general idea of a *parameterized relational expression* (*PRE*) and the *Apply* operator, as described in [9]. A PRE is simply a relational expression that has free variables or parameters, so it yields a relational result when values for those parameters are provided. It is effectively a function. The *Apply* operator repeatedly invokes a PRE with a series of parameters values and collects the results of these multiple invocations. Formally,

$$R\ \mathcal{A}pply_{\mathcal{JN}} E(\boldsymbol{r}) = \mathcal{UA}_{r \in R}\ (\{r\}\ \mathcal{JN} E(\boldsymbol{r})).$$

Note that *Apply* does not take two relational inputs, but only one relational input *R* that provides the set of parameter values, on which PRE *E(r)* is applied. *Apply* can use different logics to combine each row *r* with the result of *E(r)*, specified by $\mathcal{JN}$ above. It supports the common join types such as inner, outer, and semijoins which we briefly review here:

An *inner join* ($R\ \mathcal{JN}_p\ S$) is defined as the subset of the Cartesian product of R and S where all tuples that do not satisfy the predicate *p* are filtered out.

A *left outer join* ($R\ \mathcal{LOJ}_p\ S$) includes the result of an inner join between R and S with join condition *p*, plus all the unmatched tuples of R extended with NULL values for columns of S. A *right outer join* on the other hand contains the unmatched tuples of S along with the result of the inner join.

A *semijoin* ($R\ \mathcal{SJ}_p\ S$) is defined as all the tuples of R that match at least one tuple of S on the predicate *p*, while an *antijoin* ($R\ \mathcal{ASJ}_p\ S$) is defined as the tuples of R that match no tuples of S on the predicate *p*. Naturally, $(R\ \mathcal{SJ}_p\ S)\ \mathcal{UA}\ (R\ \mathcal{ASJ}_p\ S) = R$.

For example, an Apply operator can use antijoin logic if it wants to preserve row *r* when the result of *E(r)* is empty.

The *Apply* operator maps well to the nested loops execution strategy with correlated parameters, but we treat it here as a logical operators with the semantic definition described above.

We illustrate the use of *Apply* with a simple SQL subquery example. Say you want to list all your ORDERS, and include the CUSTOMER name. It is convenient to have a function that takes a customer key and returns the name of the CUSTOMER. Such function can be written as follows

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

where the free variable O_CUSTKEY is the argument of the function – there is no explicit syntax to bind free variables in SQL, so binding variables in subqueries is done simply by name. Free variables will be shown in **bold** throughout the paper. We can use this "name extraction function" to report all ORDERS with the name of the CUSTOMER as follows

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

An additional issue to note here is that we are crossing a bridge between relational expressions and scalar domains. The subquery is a relational expression, but it is used in a context that expects a scalar value, i.e. the SQL SELECT clause. The rules to bridge this relational/scalar divide are the following:

- If the relational result of the subquery is empty, then its scalar value is NULL.
- If the relational result is a single row $\{a\}$, then the scalar value is *a*.
- If the relational result has more than one row, then its scalar value is undefined and a run-time error is raised.

For the sake of this example, assume that C_CUSTKEY is a key of CUSTOMER, but O_CUSTKEY is nullable, or there is no declared foreign-key constraint. Then the subquery can return at most one row. We represent this query algebraically as:

ORDERS $\mathcal{A}pply_{OJ}$ (π [C_NAME] σ [C_CUSTKEY = **O_CUSTKEY**] CUSTOMER)

Note that this expression outputs exactly the rows from ORDERS, adding an extra column for each row, with the result of the scalar value of the subquery.

2.2 Language surface

In the early days, the SQL block with its SELECT, FROM, and WHERE clauses was central to the language and there were many syntactic restrictions around the use of multiple SQL blocks in a single query, including subqueries. Current SQL implementations allow the use of "sub-selects" in a fully composable way. There are two cases to distinguish:

- A SQL block is used where a relational value such as a table is expected, in the FROM clause. Such a "sub-select" is called a *derived table*. This is simply about composability of relational expressions and we don't consider it further in this paper.
- A SQL block is used where a scalar expression is expected, such as the SELECT or the WHERE clause. Such "sub-select" is called a *subquery*. This subquery is called *correlated* if it has free variables that are provided by the enclosing query. Unlike derived tables, subqueries require going across relational and scalar domains.

Subqueries are introduced in scalar expressions in SQL in the following ways:

- *Existential test*. These use keywords EXISTS and NOT EXISTS and test whether the result of a subquery is empty. The result is of type Boolean, either TRUE or FALSE. For example:

```
EXISTS(SELECT * FROM ORDERS
WHERE L_SHIPDATE < O_ORDERDATE).
```

- *Quantified comparison*. These test whether a particular comparison *cmp* holds for values returned by a subquery *subq*. The forms are <*cmp*> ALL <*subq*>, and <*cmp*> ANY <*subq*>. The result is again of type Boolean, but unlike existential subqueries, quantified comparisons can return TRUE, FALSE or UNKNOWN (when null values are involved in the comparison). For example:

```
L_SHIPDATE > ANY(
SELECT O_ORDERDATE
FROM ORDERS
WHERE L_ORDERKEY = O_ORDERKEY).
```

- *IN / NOT IN*. This is a shorthand for quantified comparison. <expr> IN <subq> is equivalent to <expr> =ANY <subq>. <expr> NOT IN <subq> is equivalent to <expr> <>ALL <subq>.
- *Scalar-valued*. These return non-Boolean scalar values. For example:

```
(SELECT C_NAME FROM CUSTOMER
WHERE C_CUSTKEY = O_CUSTKEY).
```

In addition to its internal use in query processing, the *Apply* operator is also available in the surface syntax of SQL Server. The common usage scenario is the invocation of parameterized table-valued functions, which are a particular case of PREs. For example, suppose you have a table-valued function that takes a string and chops it up into words, outputting one row per word. You can use the following to invoke this function on the values of column COL from MYTABLE:

```
SELECT *
FROM MYTABLE
OUTER APPLY CHOP_WORDS(MYTABLE.COL)
```

Each row of MYTABLE will be repeated as many times as rows returned by the function – but if the function result is empty then the row is still preserved, due to the use of OUTER.

Some implementations of SQL incorporated the ability to pass parameters across the "comma operator" of the FROM clause. We adopted explicit syntax for parameter passing for conceptual clarity, and also because "comma" doesn't lend itself to clarifying what to do when the PRE returns an empty set, i.e. preserve or reject the row from the left relational input.

3. SUBQUERY REMOVAL

A straightforward implementation of subqueries requires tuple-at-a-time processing in a very specific order – evaluate the PRE for each row that requires evaluation of the scalar expression. It also introduces mutual recursion between the scalar and relational execution sub-systems. Conceptually, relational execution needs to make calls to some scalar evaluation sub-system for predicates and other scalar computations (there are multiple ways to implement scalar evaluation, as they could be compiled in-place instead of having an actual separate component). If scalar expressions contain subqueries, then the scalar subsystem needs to bind the free variables and make a recursive call back to relational execution. Subquery removal is about eliminating this mutual recursion between the scalar and relational execution sub-components.

The general subquery removal algorithm takes three arguments: a relational operator, a relational expressions and a scalar expression with subqueries; and it returns new expressions to compute the result without the need of subqueries. For example, say you have a selection of the form σ[p]R, and predicate p has subqueries. We invoke SQREM(σ,p,R) to get (p', R'), p' does not use subqueries and σ[p]R = σ[p']R'.

Algorithm SQREM is implemented through a simple tree traversal of scalar expression p, which moves all the subquery computation from p over to relational expression R. For each subquery PRE(**r**) found in p, we add a computation *Apply* PRE(**r**) on R and replace the subquery in p by a scalar computation. A more detailed example is found in [9].

For a correct and efficient translation, there are a number of special cases to incorporate in the basic algorithm outlined above. They are listed next.

3.1 Mapping multi-row relational results to a single scalar value

This issue was brought up already in the example query in Section 2. For a scalar-valued subquery *E(**r**)*, the subquery is computed in general as

$$R\ \mathcal{A}pply_{OJ}\ max1row(E(\mathbf{r})).$$

max1row is a special relational operator whose output is the same as its input, but it raises a run-time exception if its input has more than one row. Through static analysis, it is sometimes possible to determine at compile time that *E(**r**)* will return at most one row, regardless of the parameter value and database content – no *max1row* operator is required then. This is a common case in actual applications.

3.2 Filtering through existential test

If an existential test on *E(**r**)* is used in the context of directly filtering rows, then we incorporate the filtering operation with the evaluation of the subquery. The computation of EXISTS and NOT EXISTS subqueries is done as follows:

$$R\ \mathcal{A}pply_{SJ}\ E(\mathbf{r})$$

$$R\ \mathcal{A}pply_{ASJ}\ E(\mathbf{r})$$

In terms of the general rewrite procedure described above, the subquery occurrence in the original scalar expression S is replaced by the constant TRUE and the result simplified to obtain S'. This is the path followed when existential subqueries are ANDed together with other conditions in the SQL WHERE clause.

Existential subqueries are also used in a context that does not directly filter rows. In general, they need to be treated like scalar-valued subqueries, as described in the next scenario.

3.3 Conditional scalar execution

SQL provides a construct for conditional evaluation of scalar expressions, and subqueries can be used there as well. Implementing this semantics properly require the incorporation of *probe* and *pass-through* functionality in the *Apply* operator. Suppose your expression is of the form

```
CASE WHEN EXISTS(E1(r))
THEN E2(r) ELSE 0 END.
```

Note that the EXISTS subquery here is not used to directly filter rows, but to determine the result of a scalar expression. The subqueries will be computed by the following expression:

$$(R\ \mathcal{A}pply[semijoin, probe\ as\ b]\ E_1(\mathbf{r}))\ \mathcal{A}pply[outerjoin, pass\text{-}through\ b=1]\ max1row(E_2(\mathbf{r})).$$

Apply with probe preserves the rows from R and adds a new column b, which is 1 whenever $E_1(\mathbf{r})$ is non-empty. *Apply* with pass-through has a guard predicate and only executes its subquery if the guard is TRUE. This implements the required conditional evaluation.

Assuming the result of the scalar-valued subquery $E_2(\mathbf{r})$ is left in column e_2, the original scalar expression is replaced to be:

```
CASE WHEN p = 1 THEN e2 ELSE 0 END.
```

3.4 Disjunctions of subqueries

When subqueries are used in disjunctions, it is not possible to filter directly as we did in Sec. 3.2 with *Apply-semijoin* or *Apply-antijoin*. *Apply* with probe can be used to collect the subquery results and evaluate the entire disjunction afterwards, and pass-through can be

used to implement OR-shortcircuiting. But then it is difficult to convert *Apply* into join, which, as we shall see later, is a major tool for efficient execution.

Our mapping of subqueries in disjunctions is based on unions. Suppose you have a scalar filter condition of the form *p(r)* OR EXISTS(E_1*(r)*) OR EXISTS(E_2*(r)*), where *p(r)* is a scalar predicate over ***r*** without subqueries. We map to a filtering relational expression of the form:

$$R\ \mathcal{A}pply_{SJ}((\sigma_{p(r)}\ CT(1))\ \mathcal{U}\mathcal{A}\ E_1(r)\ \mathcal{U}\mathcal{A}\ E_2(r))$$

The scalar predicate *p* is evaluated on top of the constant table CT(1) (which returns one row and no columns). A row from *R* is output when any of the relational expressions underneath the UnionAll returns a non-empty result.

3.5 Dealing with quantified comparisons

Semantics. We deal with quantified comparisons by mapping them to existential subqueries, which we have already discussed. But we need to be particularly careful with universal quantification, whose semantics in the presence of NULL values is illustrated through an example. Say you have a predicate p of the form 5 NOT IN S, which is equivalent to <>ALL. The result of this predicate is as follows, for various cases of set S:

1. If S = {} then p is TRUE.
2. If S = {1} then p is TRUE.
3. If S = {5} then p is FALSE.
4. If S = {NULL, 5} then p is FALSE.
5. If S = {NULL, 1} then p is UNKNOWN.

Also, NULL NOT IN S is UNKNOWN for any S <> {}. Case 5 is particularly counter-intuitive (for database implementers as well as users), but it results from the rules of three-valued logic in SQL: value <cmp> NULL is UNKNOWN; UNKNOWN AND TRUE is UNKNOWN.

This quantified comparison can return values TRUE, FALSE and UNKNOWN, and we want to transform it into an existential test, which only returns values TRUE and FALSE. How can we do this mapping?

Utilization context. We should note that FALSE and UNKNOWN are undistinguishable for the purpose of selecting rows – i.e. if we filter rows on predicate *p*, then we discard any rows for which *p* is either FALSE or UNKNOWN. FALSE and UNKNOWN are also undistinguishable in the predicate of conditional CASE WHEN expressions.

Also, suppose you have a Boolean expression P(X1, X2, …) = Y, using only logical connectives AND and OR. The impact of changing the value of any Xi from UNKNOWN to FALSE, is either Y remains unchanged, or else Y changes from UNKNOWN to FALSE.

From the above, it is valid to change the result of quantified comparisons from three-valued to two-valued, for Boolean expressions used to filter rows or in CASE WHEN.

The mapping. To compute universal quantification we use antijoin, which evaluates a NOT EXISTS predicate. We map through the conventional equation:

(FOR ALL s ∈ S: p) = (NOT EXISTS s ∈ S: NOT p)

However, this equation holds only in two-valued logic, not in the three-valued logic of SQL (the basic reason is that NOT UNKNOWN is UNKNOWN). So, we complete the mapping in two steps: (1) change the universal quantification σ expression so it does not involve UNKNOWN values; (2) negate the predicate.
A universal quantification predicate of the form p = A <cmp> B can return UNKNOWN when either A or B are NULL. So we replace it by a two-value predicate p' = A <cmp> B AND A IS NOT NULL AND B IS NOT NULL. Predicate p' returns TRUE whenever p returns TRUE, and p' returns FALSE whenever p returns either FALSE or UNKNOWN. As argued earlier, this mapping preserves correctness under filtering and CASE WHEN contexts.

We then negate p' to create the NOT EXISTS predicate, to obtain:

A <cmp'> B OR A IS NULL OR B IS NULL,

where <cmp'> is the comparison opposite <cmp>. Of course, if A or B are not nullable, the expression can be simplified at compilation time.

Example. Suppose you start out with a subquery of the form A NOT IN S. This is first mapped to universal quantification A <>ALL S. Then it gets mapped to a NOT EXISTS subquery of the form

NOT EXISTS(σ[A = s OR A IS NULL OR s IS NULL] S).

This will then get mapped to an antijoin with a predicate that has disjuctions. This is a common form in antijoin predicates, and its efficient execution is considered in section 6.2.

An alternative to the above is to introduce a new, special aggregate that has a Boolean input and it computes universal quantification with three-valued logic. This provides a faithful implementation of quantified comparisons in SQL (and a requirement if three-valued Boolean results are allowed as parameters to opaque functions). But this aggregate approach limits the set of execution strategies. Mapping to existential tests allows efficient use of index lookups and navigational plans.

4. REMOVING APPLY

In the previous section we outline subquery removal, i.e. eliminating the use of relational expressions in scalar expressions. The result is, in general, a relational expression with *Apply* operators and parameterized expressions (PREs). In many cases, it is possible to do further transformations and eliminate the use of *Apply* and parameterization. This is commonly known as a *decorrelated* form of the query and it enables the use of a choice of join algorithms. It was the main intuition behind efficient subquery processing in the first papers on the subject, e.g., [3].

We don't view the decorrelated form of subqueries as a preferable execution strategy, but rather as a useful *normal form*. Different surface formulations can end up in the same normal form, including queries originally written with or without subqueries. Starting from this normal form, cost-based optimization will consider multiple execution strategies, including different evaluation orders and (re-)introduction of *Apply*.

4.1 Categories for Apply Removal

Apply (without pass-through, which implement conditional execution and is foreign to relational algebra) does not add expressive power to the five basic relational operators (select, project, cross product, union, difference) [9, 13]. However, it does add conciseness. Removing *Apply* from some relational expression *E* may yield an expression *E'* that is exponentially larger than *E*. This is an im-

portant distinction for the three subquery removal categories presented in [9]. We briefly review those categories here and add a fourth:

1. **Apply removal that preserves the size of the expression**. For example, the *Apply* expression from the example query in Section 2 can be rewritten as:
   ORDERS $\mathcal{A}pply_{OJ}$ (σ[C_CUSTKEY = **O_CUSTKEY**] CUSTOMER) = ORDERS OJ [C_CUSTKEY = O_CUSTKEY] CUSTOMER
2. **Apply removal that duplicates subexpressions**. The size of an expression can be increased exponentially as a result of *Apply* removal, in particular when dealing with parameterized union and difference. For example, the following expression can result from the use of subqueries in disjunctions, and its decorrelated form duplicates R:

   $$R\ \mathcal{A}pply_{JN}(\sigma_{\mathbf{R.a} = S.a}\ S)\ \mathcal{UA}\ \sigma_{\mathbf{R.b} = T.b}\ T)$$

   $$= R\ \mathcal{JN}_{R.a = S.a}\ S\ \mathcal{UA}\ R\ \mathcal{JN}_{R.b = T.b}\ T$$

   One could also write the above as a join with a complex OR condition, on top of a union of S and T with a column that tags the source of each union output row, thus preserving the size of the original expression. However, we find little use in such representation from a query processing perspective.

   Note that if the same free variable, say ***R.a***, were used in both branches of the union, then the predicate can be factored out to remove *Apply* without duplication. SQL Server identifies and handles this case as category 1. In general, pulling up predicates with free variables is part of the normalization process that removes *Apply* for expressions in category 1.
3. **Apply removal unfeasible due to special scalar-relational bridge semantics**. There are two cases here: (1) checking for *max1row* for scalar-valued subqueries and (2) conditional evaluation using *pass-through* predicates for CASE WHEN expressions. SQL Server does not remove *Apply* for these cases.
4. **Opaque functions**. This category was not called out in [9], but it is a distinct case. *Apply* is not removed when dealing with opaque functions like our earlier example of table-valued function CHOP_WORDS:
   MYTABLE *Apply* CHOP_WORDS(**MYTABLE.COL**)

   Table-valued functions written in languages such as C++ are always opaque. For functions defined through the SQL language, we also support *inlininig,* in which case the function definition simply gets expanded, as a view with parameters. This brings back the expression to one of the three earlier categories.

4.2 Tradeoffs and Query Processing Strategy

All the categories outlined above are found in practice, and their effective processing requires slightly different approaches. To describe the differences, we need to take into account the processing flow of the optimizer in SQL Server:

- The compiler front-end parses the query text and resolves symbolic references. It produces a tree representation of the statement based on logical data operators.
- This operator tree is simplified and normalized, including contradiction detection and removal of redundant operations. The output is a simplified logical operator tree representing the query.
- The simplified operator tree goes into cost-based optimization, where an execution plan is selected based on available access paths, data volume and distribution.

Query simplification / normalization is done using the same infrastructure of tree transformations, so it is possible to utilize a particular tree transformation either as simplification or as a cost-based alternative.

For expressions that fall in categories 3 and 4 above, *Apply* cannot be removed. There is basically a single physical execution strategy for the logical *Apply* operator: Use nested loops joins to repeatedly execute the parameterized expression, in a straightforward implementation of the *Apply* definition. SQL Server considers a number of optimizations on these parameterized nested loops, which are discussed later in this paper.

For expressions that fall in category 1, the query is normalized to the decorrelated form, which is fed to the cost-based optimization. This process considers a number of logical execution orders and implementation algorithms off of this decorrelated form. Going through a normal form provides syntax independence to our optimization process. There are many cases of subqueries that can also be written naturally without those in SQL, and the query optimizer has the same behavior for both forms. For example, it is easy to see that our example query in Section 2 could have been written using outerjoin directly – both the subquery or the outerjoin formulation will map to the same normal form.

Particular optimizations for these decorrelated forms are covered later in this paper, but it is worth pointing out here that one of the alternatives considered during cost-based optimization is the introduction (or re-introduction) of *Apply*, to target navigation strategies that are very efficient in some cases.

For expressions that fall in category 2, we have a tradeoff to make: Remove *Apply* but duplicate sub-expressions (as in category 1), or else keep the original (as in category 3). We do not normalize to the decorrelated form in this case for two reasons: (1) The explosion on the size of the expression; (2) the added complication faced when re-introducing *Apply* in cost-based optimization, which now requires common subexpression detection to eliminate the redundancy introduced.

We are still interested in resolving this tradeoff effectively in a cost-based way, because there are instances where the decorrelated form can perform much more efficiently, even with the duplicate subexpression. For this reason, we consider *Apply* removal for this category of queries during cost-based optimization, for a number of important special cases. Further details of this are covered later in this paper.

5. OPTIMIZING SUBQUERIES USING MAGIC SETS

In this section we briefly review the "complex query decorrelation" approach presented in [8], and reformulate it using our framework, to the best of our understanding.

5.1 The magic set technique

Magic set optimization in [8] starts out with a presentation of subqueries as functions, as we do in this paper with the *Apply* operator. A key observation is that a function can be represented extensionally as a table, for a finite set of argument values. This extensional representation has additional columns that hold the argument values

for which the function has been pre-computed. Function application then turns into join:

$$R \mathcal{A}pply E(\mathbf{r}) = R \mathcal{JN}_{R.r=R'.r'} DS,$$

assuming that DS stores the result of $E(\mathbf{r})$ for all values of r from R. The table DS is called the "decoupled query." To obtain DS, you need to compute the function for enough parameter values:

$$DS = R' \mathcal{A}pply E(r'),$$

where R' is called the "magic set," and it consists of (a superset of) the set of parameter values to be used from R. A possible choice is $R' = distinct[R.r]$ R. Of course, the two equations can be put together and there is no need to explicitly store the value of DS. The result can be seen as a generalization of the semijoin reduction strategy used in distributed systems.

The intent of [8] is that DS has a decorrelated form, which when plugged above yields a fully decorrelated expression. However, this does not really address decorrelation, in the sense we use in this paper. Removing *Apply* in the original expression is the same algebraic problem as removing it in the computation of DS.

5.2 Magic on join / group by scenarios

In addition to the general definition of magic set reduction, [8] also shows a specific strategy that can be very efficient to deal with some Join and GroupBy queries. It transforms expression **A** below to an expression **M** with magic-set reduction:

A: $R \mathcal{JN}_{p(r,s) \text{ and } p(r,x)} \mathcal{G}_{s,x=agg} S$

M: $R \mathcal{JN}_{r = r' \text{ and } p(r,x)} (\mathcal{G}_{s,x=agg} R' \mathcal{JN}_{p(r',s)} S)$

with R' being the distinct values of R used in the join with S. There are at least two other possible expressions to execute **A** [9]: Call **B** the result of moving up the GroupBy operation above the Join; call **C** the segmented execution over R, which is possible when *R* and *S* are common subexpressions. Magic set strategy **M** can be much better than the other alternatives when all the following conditions hold (which is not a very common scenario, in practice):

- Many of the values in R.r do not appear in S.s. Otherwise, **A** is probably quite effective, as it is not computing unnecessary groups.
- R.r is a low selectivity column (i.e. there are relatively few distinct values of r in R) and also S.s is a low selectivity column. Otherwise, **B** is probably quite effective, since the early join will filter out any unnecessary rows and no extra work is created for the final aggregation.
- R and S are not common subexpressions. Otherwise, strategy **C** requires a single pass and can be very efficient.

From the point of view of our framework, the magic set strategy is an alternative for queries with Join and GroupBy, and it needs to be considered in a cost-based way along with other choices.

As with *Apply* removal over parameterized unions, magic set reduction requires the duplication of subexpressions, which introduces additional complications during optimization.

6. OPTIMIZING SEMIJOIN AND ANTIJOIN

The process of removing correlation generates trees which contain semijoins, antijoins and outer joins. Optimization of semijoins, antijoins and outer joins is therefore an important part of handling subqueries. The paper [11] discusses outer join optimization in detail. In this section we will concentrate on semijoins and antijoins. Note that SQL does not expose semijoins and antijoins as language constructs. Therefore for SQL, subquery removal is the only way these operators make an appearance in the query tree.

6.1 Reordering semijoins and antijoins

From the reordering perspective, semijoins and antijoin are very similar to filters. They can be pushed or pulled through an operator whenever a filter can be pushed or pulled. E.g. a filter can be pushed through a GroupBy operator whenever the predicate does not use the results of the aggregate expressions. Similarly a semijoin or an antijoin can be pushed through a GroupBy as long as the join predicate does not use the results of the aggregate expression. i.e.

$$(\mathcal{G}_{A,F} R) \mathcal{SJ}_{p(A,S)} S = \mathcal{G}_{A,F} (R \mathcal{SJ}_{p(A,S)} S)$$

$$(\mathcal{G}_{A,F} R) \mathcal{ASJ}_{p(A,S)} S = \mathcal{G}_{A,F} (R \mathcal{ASJ}_{p(A,S)} S)$$

Reordering semijoins and antijoins as though they were filters is a powerful tool but it still keeps the tables in relation S together as a block. If we want to reorder individual tables of S, we have to be careful about the number of duplicate rows in the result and the columns that are visible. An identity which gives us a simple and general solution for this problem is:

$$R \mathcal{SJ}_{p(R,S)} S = \mathcal{G}_{key(R),Any(R)} (R \mathcal{J}oin_{p(R,S)} S)$$

This transformation converts a semijoin into a join. Reordering of tables around a join is a well understood problem.

Here is an example that illustrates the benefits of this ability to freely reorder tables. Consider the query that tries to find the number of orders placed on the New Year's Day in 1995 for which at least one of the suppliers is in the same country as the customer and the item was shipped within seven days of the order. The SQL for this query looks something like this:

```
SELECT COUNT(*)
FROM ORDERS
WHERE  O_ORDERDATE = '1995-01-01'
    AND EXISTS(SELECT *
    FROM CUSTOMER, SUPPLIER, LINEITEM
    WHERE
        L_ORDERKEY = O_ORDERKEY
        AND S_SUPPKEY = L_SUPPKEY
        AND C_CUSTKEY = O_CUSTKEY
        AND C_NATIONKEY = S_NATIONKEY
        AND L_SHIPDATE BETWEEN '1995-01-01'
        AND dateadd(dd, 7, '1995-01-01')
```

The EXISTS clause of the subquery is transformed into a semijoin. The transformed query looks like:

ORDERS $\mathcal{SJ}$(CUSTOMER $\mathcal{JN}$ SUPPLIER $\mathcal{JN}$ LINEITEM)

The inner side of the semijoin contains an inner join of three tables. These three tables cannot be reordered freely with the ORDERS table as otherwise the count(*) may see incorrect number of rows. We are therefore forced to do one of the following. Either join the CUSTOMER, SUPPLIER and LINEITEM table before we join the result with ORDERS (if we choose to do the semijoin as a hash or merge join) or join the three tables for every row of ORDERS that qualifies the date predicate (if we choose to do the semijoin as a nested loop or an index lookup join). Both these alternatives are quite slow. Since the predicates on both ORDERS and LINEITEMS are quite selective, it is better to join the two tables first and then join the small result with the remaining tables. This is exactly what converting a semijoin to a join lets us do.

This ability to freely reorder tables may tempt us to convert all semijoins to inner joins as a normalization step. The problem is that it creates a GroupBy(distinct) operator for every semijoin that is converted. We need to find an optimal place for these GroupBys in the final plan. GroupBy placement is not an easy task and can increase compilation time dramatically. Therefore converting semijoins to inner joins as a normalization step may not be a good idea and it is preferable to do this on a case-by-case basis.

The tables from the two sides of an antijoin cannot be mixed this way since there is no simple identity to convert them to inner joins. Converting antijoins to inner joins requires introduction of a subtraction since inner joins don't allow us to do universal quantification. Evaluation of subtraction requires an antijoin on keys taking us back to where we started. Therefore the tables used on the inner side of an antijoin are forever separated from the rest of the query and can only be reordered amongst themselves.

6.2 Evaluating semijoins and antijoins efficiently

Semijoins and antijoins can be evaluated using all the standard join implementations viz. nested loop join, hash join, merge join and index lookup join. Nothing special is required when evaluating a semijoin using any of these join algorithms. Evaluating antijoins as a hash join requires some special considerations. For the antijoin (R $\mathcal{ASJ}_p$ S), if the relation R is used to build the hash table, we have to mark and rescan the hash table to decide which rows to output [14]. Implementing the antijoin (R $\mathcal{ASJ}_p$ S) as nested loop join requires that the relation R be on the outer side.

Evaluating the semijoin (R $\mathcal{SJ}_p$ S) or the antijoin (R $\mathcal{ASJ}_p$ S) by using the relation R as the source of the lookup in an index-lookup join is also easy. We take a row of relation R and look it up in the index on the relation S. For a semijoin, we output the row if the index lookup yields a matching row. For an antijoin, we output the row if the index lookup fails to find a match.

Doing index lookups from the relation S into the relation R requires a lot more care. Suppose we want to find out the number of urgent orders that had at least one item that was shipped on January 1, 1995 but was committed for December 31, 1994. The query would look like:

Example 6.1:

```
SELECT COUNT(*)
FROM ORDERS
WHERE O_ORDERPRIORITY = '1-URGENT'
AND EXISTS (SELECT *
    FROM LINEITEM
    WHERE L_SHIPDATE = '1995-01-01'
        AND L_COMMITDATE = '1994-12-31'
        AND L_ORDERKEY = O_ORDERKEY
    )
```

The query does a semijoin between urgent ORDERS and LINEITEMs shipped on a specific date and committed for a specific date (ORDERS $\mathcal{SJ}$ LINEITEM). Since the predicate on LINEITEM is a lot more selective than the predicate on ORDERS and can be solved with an index, a natural strategy would be to use an index seek to get the qualifying LINEITEMs and then look them up in the ORDERS table using the index on O_ORDERKEY. This of course does not work for semijoins. There are multiple LINEITEMs per ORDER. Therefore if two items in an ORDER shipped on the New Year's Day in 1995 but were committed for the New Year's Eve, that ORDER will get looked up twice, giving us an incorrect count of orders.

The identity to convert semijoins to joins described in the previous section comes to our rescue. For an inner join it is legal to do lookups in any direction. This is the second benefit of converting semijoins to inner joins. It gives us the freedom to choose either table as a lookup table. If the predicate on ORDERS is more selective and easily solvable we can first find the ORDERS and then look up matching LINEITEMs. If on the other hand if the predicate on LINEITEMs is more selective we can do a reverse lookup.

As we discussed earlier, antijoins do not have a corresponding identity to convert them to joins and therefore do not yield to reverse lookups. Even worse, subquery removal sometimes generates antijoins that are hard to do even as regular lookups. The reason for this is NULL semantics in SQL. Consider the query that gets the count of all the LINEITEMS that were shipped on a day on which no order was placed. Assume we have a modified version of TPCH where the unshipped LINEITEMs are identified with L_SHIPDATE as NULL.

```
SELECT COUNT(*)
FROM LINEITEM
WHERE L_SHIPDATE != ALL(SELECT O_ORDERDATE
from ORDERS)
```

As we saw in Section 2 this generates an antijoin with the predicate L_SHIPDATE = O_ORDERDATE OR L_SHIPDATE IS NULL. (We assume here that O_ORDERDATE cannot be NULL.) If we don't include the L_SHIPDATE is NULL predicate, we will count unshipped orders. The reason is that the predicate (L_SHIPDATE = O_ORDERDATE) evaluates to unknown for every ORDER when L_SHIPDATE is NULL and an antijoin outputs a row if the predicate does not evaluate to true for any of the rows of the inner relation.

The set based join algorithms viz. hash and merge cannot be used for this query, since the predicate for the antijoin is (L_SHIPDATE = O_ORDERDATE OR L_SHIPDATE IS NULL). Hash or merge join require at least one conjunct that is a simple equality so that it has columns that can be used to hash or sort the relations. Since we are evaluating (LINEITEM $\mathcal{ASJ}_p$ ORDERS), ORDERS has to be the lookup table for the reasons mentioned earlier. But it is very hard to do lookups into ORDERS with the predicate (L_SHIPDATE = O_ORDERDATE OR L_SHIPDATE IS NULL). Therefore the only choice we are left with is nested loop join. A nested loop join is going to be excruciatingly slow given the size of LINEITEMS and ORDERS. The solution is to use the identity

$$(R\ \mathcal{ASJ}_{(p1\ OR\ p2)}\ S) = ((R\ \mathcal{ASJ}_{p1}\ S)\ \mathcal{ASJ}_{p2}\ S)$$

This gives us one antijoin with the predicate L_SHIPDATE = O_ORDERDATE which can be implemented either as a set based join or as a lookup join and another antijoin with a predicate only on LINEITEMS which can be evaluated without looking at all rows of ORDERS.

Quantified comparisons frequently contain inequalities of the form >ANY, <=ALL etc. Removing correlations for such queries gives us expressions of the form (R $\mathcal{SJ}_{(R.a\ .cmp.\ S.a)}$ S) or (R $\mathcal{ASJ}_{(R.a\ .cmp.\ S.a)}$ S). Consider the case when .cmp. is one of (<, >, <=, >=). A semijoin or an antijoin with such a predicate cannot use hash joins. It may be very expensive to implement it using a merge or an index-lookup join if the proper indices do not exist. We may therefore have to implement it as a nested loop join. An interesting strategy that can be used to improve performance in this case is to use a max or a min

aggregate on the relation S. e.g. ($R \mathcal{SJ}_{(R.a > S.a)} S$) can be transformed to ($R \mathcal{SJ}_{(R.a > x)} (\mathcal{G}_{\Phi, x=min(S.a)} S)$). Here is the intuition behind this transformation. ($R \mathcal{SJ}_{(R.a > S.a)} S$) outputs a row of R if there is at least one row of S for which R.a > S.a. But in that case it is enough to check if R.a > min(S.a). The min/max aggregate can be calculated once and saved so that the relation S does not have to be scanned multiple times. The same strategy works for antijoins.

7. OPTIMIZING PARAMETERIZED UNIONS (SUBQUERY DISJUNCTIONS)

In the previous section we discussed cases where the semijoin or antijoin predicate was a disjunction. A more complicated case is when the disjunctions of the original query are subqueries.

Consider a query where we try to get a count of LINEITEMs which were either shipped after the commit date or were shipped more than 90 days after the order date.

Example 7.1:

```
SELECT COUNT(*)
FROM LINEITEM
WHERE L_SHIPDATE > L_COMMITDATE
    OR EXISTS (SELECT *
        FROM ORDERS
        WHERE DATEADD(DD, 90, O_ORDERDATE) <
        L_SHIPDATE AND O_ORDERKEY =
        L_ORDERKEY)
```

As we saw earlier, whenever we have subqueries in a disjunction, we end up with a tree of the form

$$R \mathcal{Apply}_{SJ} (E_1(\mathbf{r_1}) \mathcal{UA} E_2(\mathbf{r_2}) \mathcal{UA} E_3(\mathbf{r_3}) \ldots)$$

where r_i are the correlated columns of R and E_i are the transformed subqueries. When we try to remove correlations from a tree of this form, we encounter two possibilities.

7.1 Simple correlation

The simple correlation case is when we can transform

$$E_1(\mathbf{r_1}) \mathcal{UA} E_2(\mathbf{r_2}) \mathcal{UA} E_3(\mathbf{r_3}) \ldots \rightarrow \sigma_{p(r)} (F_1 \mathcal{UA} F_2 \mathcal{UA} F_3 \ldots)$$

where the expressions F_i are free of correlation from R. This is possible if and only if the following two conditions are met: (1) The correlated predicates in each branch are identical in form except for the columns of E_i that they use. (2) The columns of E_i that they use are such that the ones appearing in the same position in the correlated predicates are mapped to the same resulting column by the union-all. If we manage to do this transformation our expression becomes

$$R \mathcal{SJ}_{p(r)} (F_1 \mathcal{UA} F_2 \mathcal{UA} F_3 \ldots)$$

and we have reduced our problem back to the optimization of semijoins.

7.2 Complex correlation

Given the severe restrictions on the predicates in the previous section, such a transformation is possible only in very few cases. For example it does not work for our query because the correlated predicate on one branch is L_SHIPDATE > L_COMMITDATE while that on the other branch is DATEADD(DD, 90, O_ORDERDATE) < L_SHIPDATE AND O_ORDERKEY = L_ORDERKEY. The two predicates are not even close in form to each other and the only way that they can be pulled above the union-all is by introducing new columns to distinguish the two branches. This method of introducing columns to remove correlation generates a complicated disjunction which can neither be used for set based implementations like hash and merge join, nor for index lookups. Therefore, we will not discuss it further in this paper.

A more common case, like our example, is when we cannot pull the correlations above the union-all, but we can remove correlations for some (or maybe all) of the expressions $R \mathcal{Apply}_{SJ} E_i(\mathbf{r_i})$. In that case we can use the distributivity of Apply over union-all to transform

$$R \mathcal{Apply}_{SJ} (E_1(\mathbf{r_1}) \mathcal{UA} E_2(\mathbf{r_2}) \mathcal{UA} E_3(\mathbf{r_3}) \ldots) \rightarrow$$

$$(R \mathcal{Apply}_{SJ} E_1(\mathbf{r}_1)) \mathcal{U} (R \mathcal{Apply}_{SJ} E_2(\mathbf{r_2})) \mathcal{U} (R \mathcal{Apply}_{SJ} E_3(\mathbf{r_3})) \ldots$$

and then remove correlations from individual expressions. Note that distributing an $\mathcal{Apply}_{SJ}$ requires that we change a union-all to a distinct-union to avoid the duplicates generated by the multiple occurrences of R. The advantage of this transformation is that for the subexpressions for which we can remove correlations, all the optimization strategies in our repertoire, like set based implementation, reverse lookup etc. are available. The disadvantage of course is that the relation R gets duplicated many times and therefore has to be either evaluated multiple times or spooled.

The paper [12] suggests an interesting way of avoiding the duplication of relation R by using a *bypass* operator. The query is converted into a DAG where the bypass operator dispatches the rows to the branches of the DAG and a union-all at the top combines them together again. The bypass operator is nice in its generality, but we think that there are two issues with it. The first issue relates to query execution engines that use the *pull* model i.e. each operator asks its child for a row (*pulls a row*) whenever it needs one. The union-all which combines the rows from the two streams that are created by the bypass operator cannot be implemented as a pull operator unless all but one of its streams is spooled away. Otherwise it has to be clairvoyant about which of its child will produce the next row. This means that in reality it has to spool data and cannot avoid duplication. The other issue is that the bypass operator makes reverse lookups impossible. It blocks the access to the original relation into which we can do reverse lookups.

In case both strategies described in this section, that allow us to remove correlations from disjuncts, do not work, we can still do some minor optimization. The $\mathcal{Apply}_{SJ}$ operator can be implemented such that it stops reading its right child as soon as it gets a row back. This allows us to shortcut evaluation of unnecessary expressions. If we reorder the children of the union-all such that either cheaper subexpressions or the sub-expressions that are most likely to return rows are evaluated first, we can save the cost of evaluating the expensive sub-expressions in many cases.

8. OPTIMIZING GENERAL APPLY

As we discussed earlier there are cases where the correlation cannot be removed and the only option available is to execute the correlated query as a nested loop join. Even after the correlation is removed, the join predicate may be an inequality which prevents us from using the set based join algorithms like hash and merge. In addition, in some situations it's more efficient to evaluate the query as a straightforward Apply than in a set-based fashion. In all these situations, we can still improve on a naïve nested loop implementation. In fact Microsoft SQL Server utilizes several mechanisms to optimize the IO and throughput of queries with nested loop joins.

8.1 Caching

In many situations, we can efficiently avoid a lot of IO if we cache the results for the inner side of the nested loop operator. If the subquery contains no correlations, e.g. when the query processor successfully decorrelates the subquery, then a simple caching mechanism is appropriate and will be chosen if the subquery is complex enough, and the result set small enough, that is worth caching. This caching mechanism is applicable and utilized in a wide variety of plans beyond nested loop joins and will not be discussed here.

If there are correlations, the query processor might also cache the results in a temporary index that is keyed on the correlated columns (aka "memoize" the results as described in [16], [17]). The query processor in this case takes into consideration the cost of seeking into this temporary index for each outer reference, as well as the cost of building the index itself. As hinted at in [14], these are complex decisions and need to be decided on a case-by-case basis, but this complexity is overcome by the cost-based approach of the Microsoft SQL Server query optimizer.

8.2 Asynchronous IO (Prefetch)

With modern server IO subsystem, there are often many more disk controllers than CPU's available for query processing. Utilizing these disk controllers efficiently results in big gains in query processing throughput and response time. In order to achieve that, many database products (including Microsoft SQL Server) issue asynchronous IO operations while keeping the processing thread busy processing other rows.

When implementing an Apply operator, Microsoft SQL Server utilizes this in the following way: whenever a row is obtained from the outer input to the Apply, this is used to issue a prefetch request for the required pages of the corresponding underlying table in the subquery so that the row is already in-memory by the time the subquery is ready to process it. Consider the following example:

Example 8.1:

```
SELECT S_ACCTBAL, S_NAME, P_PARTKEY, P_MFGR,
S_ADDRESS, S_PHONE, S_COMMENT
FROM PART, SUPPLIER, PARTSUPP
WHERE P_PARTKEY = PS_PARTKEY
       AND S_SUPPKEY = PS_SUPPKEY
       AND P_SIZE    = 1
       AND P_TYPE    LIKE '%TIN'
       AND PS_SUPPLYCOST    = (
       SELECT MIN(PS_SUPPLYCOST)
       FROM PARTSUPP, SUPPLIER
       WHERE P_PARTKEY = PS_PARTKEY
              AND S_SUPPKEY = PS_SUPPKEY)
```

This is a simplified version of Query 2 of TPC-H, which is just asking for the suppliers who can supply tin parts of a specific size at the lowest cost. In this case, because of the restrictive filter on the PART table, Microsoft SQL Server estimates that it is best to calculate the minimum supply cost for each qualifying part, then use the part key to seek into the index for the PARTSUPP and filter the qualifying rows on the cost. In other words, let

$$R = \mathcal{G}_{P\_PARTKEY,m=min(PS\_SUPPLYCOST)}(\text{PART } \mathcal{JN} \text{ SUPPLIER } \mathcal{JN} \text{ PARTSUPP})$$

SQL Server evaluates:

$$R \; \mathcal{A}pply_{JN} (\sigma[\textbf{R.m} = \text{PS\_SUPPLYCOST}] (Seek_{\textbf{R.P\_PARTKEY}=PS\_PARTKEY} \text{ PARTSUPP}))$$

(The plan for evaluating R is not relevant to this discussion).

To improve the efficiency of this query, the query processor uses IO prefetching as follows: for each (P_PARTKEY, m) tuple obtained for R, an asynchronous prefetch request is issued for the PARTSUPP table to fetch the suppliers for this part into memory immediately for use by the right subtree of the Apply when it is ready. So in a production server with 4 CPU's and 8 disk controllers, this plan can easily utilize all these resources efficiently to process this query by keeping all the disk controllers fetching pages from the PARTSUPP tables (assuming they are not cached before this query) while the CPU's are processing the rest of the query tree.

8.3 Improving locality of reference (Batch Sort)

Introducing a local (batch) sort operation for better performance is another mechanism to reduce the IO cost of query processing with nested iterations. With this technique, Microsoft SQL Server locally sorts portions of the outer input of Apply operators (where it feels this will be effective) in order to localize the references in the inner subtree. This technique has been shown by researches to give significant performance gains for nested loop join performance [15], and indeed we have observed such performance gains when the query processor judiciously applies this technique to select queries – for example, this technique is not applied if the inner side is so small that it would fit in memory anyway so it's not worth the effort of the extra sorts.

Continuing the example above, the query processor would apply this technique to the Apply operator as follows: for each set of tuples (P_PARTKEY, m) obtained from the outer subtree, the query processor sorts them in ascending order of P_PARTKEY before passing each row to the inner subtree (and before issuing the prefetches). This has the advantageous effect that sets of rows within a page in the PARTSUPP B-tree are read together, reducing random IO and the total number of pages accessed. The size of each sorted batch is ramped up dynamically so the query processing starts with small batches that pipeline the first few rows quickly up the query plan, then the size of each batch gets bigger to better improve locality of reference and sequential IO.

9. SUMMARY OF STRATEGIES

This paper has described a large number of strategies that are used in SQL Server to efficiently evaluate queries that contain subqueries. Here is a summary of these strategies along with the section number where they are described. In the next section we show experimental results for some of these techniques.

Operator	Strategies
Semijoin	Forward-lookup (Section 6.2) Reverse-lookup (Section 6.2) Set-based evaluation (Section 6.2) Conversion to inner join (Section 6.1) Use of max/min (Section 6.2)
Antijoin	Forward-lookup (Section 6.2) Set-based evaluation (Section 6.2) Split antijoin with disjunction (Section 6.2) Use of max/min (Section 6.2)
Join with GroupBy	See [9]
Apply	Subqueries in disjunctions. (Section 7) Caching the PRE (Section 8.1) Prefetching the PRE (Section 8.2) Sorting the outer relation (Section 8.3)

10. EXPERIMENTAL EVALUATION

10.1 Experimental Setup

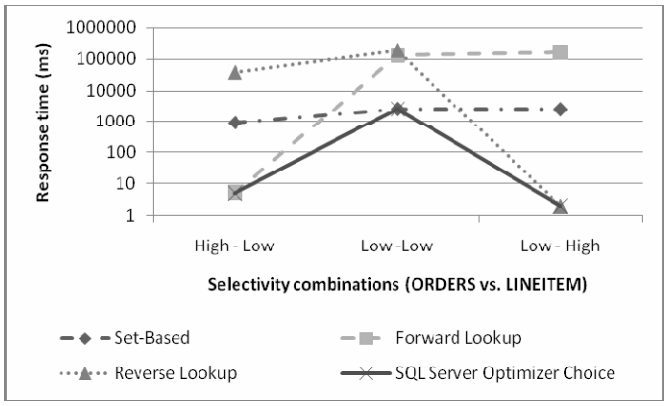
The experiments were performed on a Dell 2950 Dual core 2.66 MHz machine with 16 GB RAM running Microsoft Windows 2003 X64 SP1. We used the TPC-H benchmark schema for our experiments and populated the database with the TPC-H data at scale 30 GB. We created indexes on the tables as required by the TPC-H benchmark specification and provided additional indexes to facilitate all navigational and set-oriented subquery execution strategies relevant to this paper.

10.2 Execution strategies for semijoin

To compare the different strategies for evaluating semijoins, we relied on variants of the query in Example 6.1. The variants modified the filter predicates in the query so that their selectivity on the ORDERS and the LINEITEM tables can yield up to 20 rows (high selectivity) or up to 2,000,000 rows (low selectivity).

The different strategies compared were: forward lookup, reverse lookup and set-based join (viz. hash join). For each strategy, the response time was measured in milliseconds. Before running these queries we made sure that supporting indexes for all strategies were present. The above chart plots response times for the strategies and also indicates which strategy was automatically picked by the SQL Server query optimizer.

As expected, lookup from the table with the highly selective predicate into the other table yields the best performance, while set-based evaluation gives good performance when neither predicate is selective enough to do a small number of lookups.

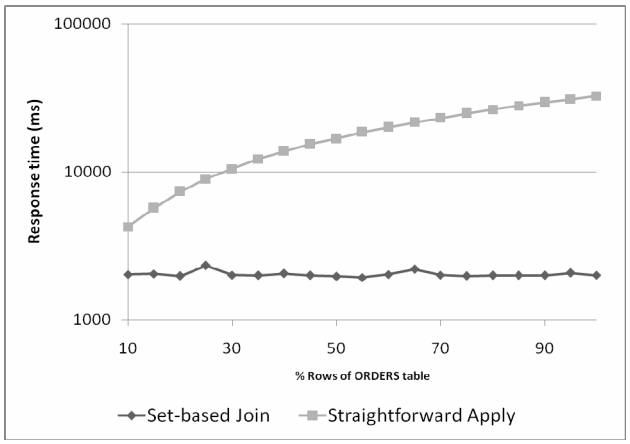


Note that the difference in performance between the best alternative and the second best strategy is dramatic: it is at least one order of magnitude for the selectivity combinations explored in the above figure.

Another important observation is that no single execution strategy provides satisfactory performance across all selectivity combinations. Cost-based optimization therefore is required to inspect selectivity estimates and pick an appropriate strategy for subquery execution. As the results show, Microsoft SQL Server succeeds at this task as it consistently chooses the best performing strategy for the selectivity ranges investigated.

10.3 Execution strategies for antijoin

We conducted this series of experiments with variants of Example 6.1. The variants again modified the filter predicates of the query so that their selectivity on the ORDERS table varies in from 10% of the rows to 100%. We compared query performance when the antijoin was applied directly versus when the optimizer splits the antijoin and does a set-based join (hash join).

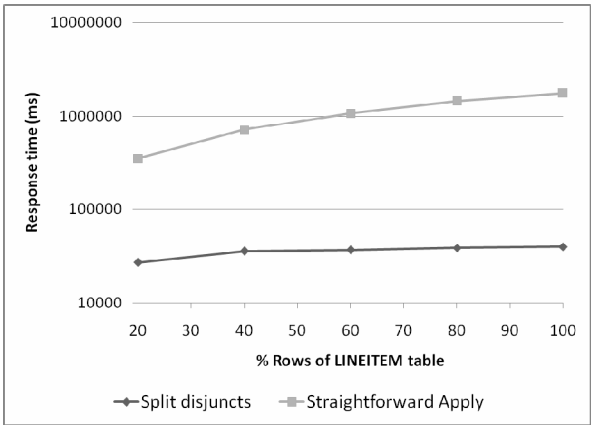


As the figure shows, splitting the antijoin in this case gives a performance gain of an order of magnitude and scales much better compared to the straightforward evaluation.

10.4 Strategies for subquery disjunctions

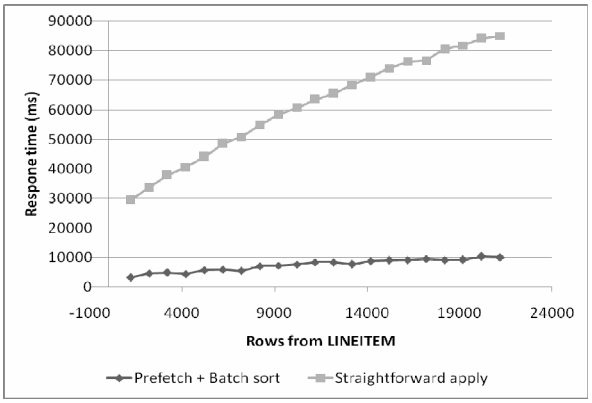
We evaluated the performance of the variants of Example 7.1 when the optimizer transformed the disjunction as explained in Section 7.2 compared with when this transformation was disabled. For this experiment we varied the percentage of qualifying rows from the LINEITEM table from 20% to 100%.

We see that in this query splitting the disjuncts gave an improvement of more than an order of magnitude and better scaling performance than the straightforward implementation.



10.5 Optimizations for general Apply

To evaluate the effect of the two optimizations described for improving the performance of Apply, we ran Example 8.1 using the reverse lookup strategy. Variants of the query modified the predicate on the LINEITEM table to range from high to low selectivity. We compared the response time of straightforward Apply to Apply with Prefetch and Batch sort.



As the figure shows, these optimization techniques yield a significant performance gain (about 88% reduction in response time).

11. CONCLUSIONS

Subqueries are a powerful addition to the SQL language. They facilitate query formulation in important application scenarios such as decision support or automatic query formulation by ad-hoc reporting tools. This paper has explained how Microsoft SQL Server 2005 represents subqueries by introducing the Apply operator to its internal algebra. We have discussed how Apply is transformed into the common join operators such as inner join, outer join, semi-join or anti-join during query decorrelation. The main contribution of the paper is our detailed discussion of (1) how alternative query plans for subqueries are generated by equivalence rewrites at the logical query optimization level, (2) which different execution strategies they can be mapped to at the physical level, and (3) which further performance optimizations they lend themselves to. Our experimental evaluation shows that proper choice of execution strategy from forward or reverse lookup plans to set-oriented processing is essential for satisfactory performance. It turns out that no single strategy yields acceptable performance across important selectivity ranges. Therefore, all aforementioned execution strategies for subqueries need to be available to an industrial-strength query processor and cost-based optimization is crucial to pick the appropriate alternative.

In the future, we expect to see even increasing interest into the topic of subquery execution as relational database vendors gain more experiences with "nested loop languages" for semi-structured data processing such as XQuery. Like Microsoft SQL Server, other vendors have also chosen to map XQuery to an (extended) relational algebra inside the database engine. With XQuery's navigational programming paradigm and its existentially quantified semantics for filter predicates, subqueries are a frequent guest in the underlying relational query algebra. The techniques and strategies for subquery optimization and execution discussed in this paper will help database vendors achieve competitive performance as their customers deploy their semi-structured offerings to more challenging application scenarios.

12. REFERENCES

[1] International Organization for Standardization (ISO): Database Language SQL, Document ISO/IEC 9075: 1992.

[2] XQuery 1.0: An XML Query Language, http://www.w3.org/TR/xquery/

[3] W. Kim. On Optimizing an SQL-like Nested Query. *ACM TODS,* 7, September 1982.

[4] W. Kießling: On Semantic Reefs and Efficient Processing of Correlation Queries with Aggregates. *VLDB 1985*: 241-250.

[5] R. A. Ganski, H. K. T. Wong: Optimization of Nested SQL Queries Revisited. *SIGMOD Conference 1987*: 23-33.

[6] G. von Bültzingsloewen: Translating and Optimizing SQL Queries Having Aggregates. *VLDB 1987*: 235-243.

[7] M. Muralikrishna: Improved Unnesting Algorithms for Join Aggregate SQL Queries. *VLDB 1992*: 91-102.

[8] P. Seshadri, H. Pirahesh, T. Y. C. Leung: Complex Query Decorrelation. *ICDE 1996*: 450-458

[9] C. A. Galindo-Legaria, M. Joshi: Orthogonal Optimization of Subqueries and Aggregation. *SIGMOD Conference 2001*: 571-581

[10] J. Zhou, P. Larson, J. Goldstein, L. Ding: Dynamic Materialized Views, to appear in: *ICDE 2007*.

[11] C. A. Galindo-Legaria, A. Rosenthal: Outerjoin Simplification and Reordering for Query Optimization. *ACM TODS 22(1)*: 43-73 (1997)

[12] M. Brantner, N. May, G. Moerkotte: Unnesting Scalar SQL Queries in the Presence of Disjunction, to appear in: *ICDE 2007*.

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

[14] Goetz Graefe: Query Evaluation Techniques for Large Databases. *ACM Comput. Surv. 25(2), 1993*: 73-17.

[15] David J. DeWitt, Jeffrey F. Naughton, Joseph Burger: Nested Loops Revisited. PDIS 1993: 230-242.

[16] D. Michie: “Memo” Functions and Machine Learning. *Nature (218), April 1968*: 19-22.

Hellerstein, J. M.,Naughton, J. F.: Query execution techniques for caching expensive methods. *SIGMOD Conference* 1996: 423-434.