Parameterized Queries and Nesting Equivalences

César A. Galindo-Legaria
Microsoft Corp.
cesarg@microsoft.com

April 26, 2000

Technical Report
MSR-TR-2000-31

Microsoft Research
Advanced Technology Division
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Parameterized Queries and Nesting Equivalences

César A. Galindo-Legaria
Microsoft Corp.
cesarg@microsoft.com

April 26, 2000

Abstract

Two difficulties found in a number of papers dealing with subquery evaluation and optimization are: How do they exactly relate to SQL subqueries, and how extensive is the machinery required to understand and prove the results. This is important for database implementors, who need to decide when and how to use the techniques, on *arbitrary*SQL queries. In this paper we desribe how to represent SQL queries algebraically. The mapping is comprehensive, in the sense that it covers all SQL subqueries; the target algebra is the standard relational algebra augmented by a new operator, Apply, that abstracts parameterized execution. To deal with SQL, duplicate semantics are considered.

Properties of Apply can be derived relatively easily from known properties of relational operators. Out of Apply properties follow some known subquery optimizations, along with tight, but easy to understand preconditions.

Based on the mapping into the semantically clean relational algebra, we show that arbitrary SQL subqueries can always be unnested; that is, we show how to rewrite any query containing subqueries into one without them.

1 Introduction

There is ample literature on optimization of subqueries, e. g. [Kim82, GW87, Day87, Mur89, Mur92, SPL96, Feg98], to name a few. Two difficulties somewhat common in papers dealing with subqueries are: How do they exactly relate to SQL subqueries, and how extensive is the machinery required to understand and prove the results. This is important for database implementors, who need to decide when and how to use the techniques, on *arbitrary* SQL queries.

This paper addresses those issues by developing a simple link between SQL queries that contain correlations and relational algebra, augmented with an operator that abstracts parameterized execution. Subquery execution (and optimizations) are easier to understand because the framework is a very minor extension on the semantically clean and unversally known relational algebra. The focus of this paper is not to develop particular optimizations to deal with subqueries (although some do fall out) but rather on developing the framework.

Our algebraic treatment is based on a relational construct we will call *Apply* ($\mathcal{A}$), because it performs functional application like the APPLY and MAPCAR operators of LISP.

The intuitive idea of Apply has been used often in the past. Relational database practitioners will call it simply nested loops with correlations, and sometimes use it, e. g. [RR98]. Object-oriented researchers might employ lambda-calculus concepts and notation directly, e. g. [SZ89, CD95]. Here, we study its properties.

Apply takes a relational input R and a parameterized expression $E(\bar{x})$; and it evaluates expression E for each row $r \in R$, using r to set the values of parameters $\bar{x}$. Call each such evaluation $E(r)$. The evaluation result is added to each row r. Formally,

$$R \mathcal{A}^{\times} E = \bigcup_{r \in R} (\{r\} \times E(r)).$$

Apply does not preserve the cardinality of R. A row r will appear in the result as many times as there are rows in $E(r)$, including zero. This definition is naturally seen with duplicate-preserving union. We come back to the issue of duplicates later.

The main contribution of this paper is to abstract out parameterized execution in a single operator, Apply, whose properties can be derived in a relatively simple manner. Those properties are the basis of a number of subquery optimization strategies suggested in the literature. We show how to convert SQL queries with arbitrary subqueries into relational algebra extended with Apply. The translation is, again, somewhat direct, and it allows us to deal with any subquery optimization at the algebraic level. On this clean algebraic framework we then prove that subqueries do not add expressive power to SQL, in a sense formalized in the paper.

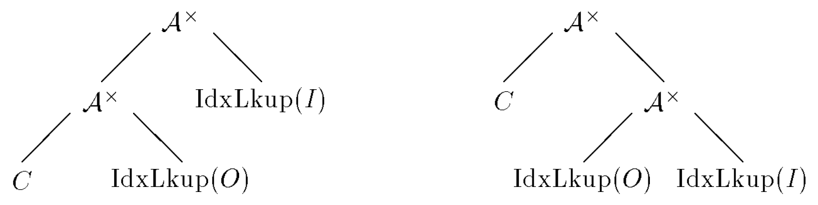
2 Parameterization in query execution

Parameterization is commonplace in query execution, and "index-lookup join" is a typical example. Index lookup join is not a binary relational operator, because it doesn't take two relational inputs. It takes one relational input, and operates directly on some physical structures of a stored table. It can be expressed algebraically as repeated execution of a correlated query. For a join predicate p on tables R, S, index lookup join can be better represented as $(R \mathcal{A}^{\times} \text{Idx-lkup}(S, p))$, where the expression Idx-lkup(S, p) is a parameterized query.[1]

Why is index-lookup join such an effective strategy? Intuitively, because the cost of executing the parameterized query depends basically on the number of qualifying rows, not the size of base tables. For index lookup, the cost of executing $E(r)$ is a (linear) function of its output size; it doesn't matter how big is R, but how many rows satisfy the condition.[2] This intuition can be extended beyond single-table selection, and it is used by the optimizer to decide when it makes sense to consider a correlated execution. Efficient correlated execution does not guarantee optimality, but only suggest consideration of the plan. Lookup cost functions often involve random I/O, which introduces a steep slope in

[1]We ommit the formal introduction of new explicit parameters in the parameterized query. Expressions are intuitively understood without them.

[2]For B-trees, there is also a $\log |R|$ search component, but it is usually negligible. In practice, the base of the logarithm is large, and inner pages of the B-tree are less likely to cause page faults due to earlier, or concurrent queries, as opposed to data pages.



(a) Shallow correlation. (b) Deep correlation.

Figure 1: Irrelevant reordering.

the cost function, compared with serial I/O. Detailed costing is needed to compare with other algorithms, if available.

2.1 Deep correlations

In the case of index-lookup join, the parameterized expression is basically a single operator. Large parameterized expressions are sometimes very useful. The following example was provided by Pedro Celis [Cel96].

Consider a query that reports a higly filtered set of customers (CUSTOMER′) along with their orders and parts: CUSTOMER′ LOJ (ORDERS $\bowtie$ PARTS), with the obvious join predicates. Outerjoin is used to keep customers without orders. Outerjoin does not naturally reorder with join, so how can we reduce ORDERS early, to access only the relevant rows? The problem is studied in [GLR97], but the solution provided there requires the use of an additional Generalized Outerjoin operator.

Parameterization provides a new dimension in the solution space. If no supporting indices exist, then we are forced to read entire tables. For the example query, however, indices are likely to exist, so an efficient plan is obtained through parameterized execution of the join subexpression, as follows:[3]

$$\text{CUSTOMER}' \; \mathcal{A}^{\text{LOJ}} \; (\sigma_{\text{c.id = o.cid}}\text{ORDERS} \bowtie \text{PARTS}).$$

Outerjoin-Apply $\mathcal{A}^{\text{LOJ}}$ is a small variation of the original, in which parameterized results are combined using outerjoin. For each outer row r, output $\{r\}$ LOJ[TRUE] $E(r)$, instead of $\{r\} \times E(r)$. Databases that execute outerjoins through index-lookup must have all the pieces to execute this. While solutions provided in [GLR97] are more comprehensive, permit sequential I/O, and address a larger number of scenarios, they require additional implementation work. For small outer tables, correlated execution can be a very efficient strategy.

An interesting property of correlated execution is that it makes operator order less relevant. Consider a query that returns all orders for a particular customer. A likely strategy is, lookup the customer (C), from customer id lookup its orders (O), then lookup

[3]For convenience, we will use different apply operators that combine its inputs based on outerjoin, semi-join, or antijoin.

its parts (P). Figure 1 shows two operator trees that implement this plan. The first one is the more likely to be produced on current systems. But the second one, using deep correlations, has the same data access pattern and performance of the first.

For a chain of tables on which an index access order has been determined, say $R_1 R_2 \ldots R_n$, parenthesization within the sequence has no effect on the execution performance. This property can be used by the optimizer to reduce search space.

This parenthesis-insensitivity worked to our advantage in the earlier outerjoin example. The "desired" execution order is to join first CUSTOMER$'$ with ORDERS to reduce data early, through an index if CUSTOMER$'$ is small enough. But since outerjoin is involved, this reordering requires adjustments on later joins. Without changing the tree topology, we achieved performance similar to the desired join order through a deep correlation.

3 Algebraic Representation of SQL Subqueries

In SQL, subqueries allow using relational expressions to compute scalar values. Since we want to cover *all* subquery cases, it is convenient to follow a syntax-based approach. There are three types of subqueries in SQL92:

- Scalar-valued. A relational query that outpus a single-column table. At run-time it can return zero, one, or more rows. For zero rows, the corresponding scalar value is NULL; for one-row, the scalar is the column value; for more than one row, a run-time error is generated. Common scalar-valued queries compute a scalar-aggregate and are therefore guaranteed to return on row.

- Existential test. An arbitrary query is enclosed as **exists**(Q). The result is a boolean indicating if the result is non-empty.

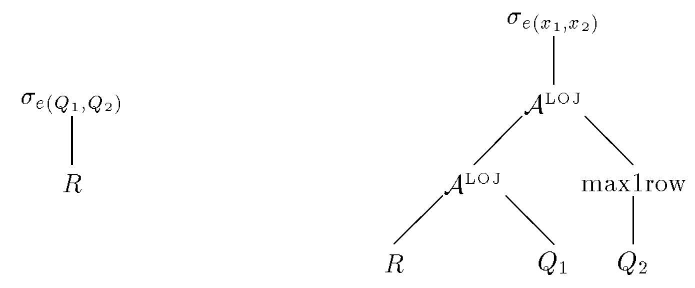
- Quantified comparison. Check a scalar expression against a set of values returned by a single-column table. The form is ⟨ expr ⟩ ⟨ cmp ⟩ ALL|SOME Q. Comparison may be either existential (SOME) or universal (ALL).[4] The result is a boolean value indicating the result of the quantified comparison.

For example, the following query returns the number of orders placed by customers in countries where the company has no direct salespeople:

```
select  CUSTOMERS.name,
        (select count(*) from ORDERS
                where CUSTOMERS.cust# = ORDERS.cust#)
from    CUSTOMERS
where   CUSTOMERS.country <> all (select SALESP.country from SALESP)
```

Anywhere scalar expressions are allowed, subqueries can be used, as long as data types are correct. The result of parsing subqueries directly yields relational operators that use scalar expressions with subqueries, e. .g. a select or project node. The general scheme to remove subqueries out of scalar expressions is shown in Fig 2. Subqueries Q_1, Q_2 are

[4]Subqueries of the form ⟨ expr ⟩ **in** Q are defined in terms of quantified comparisons.



(a) Scalar expression with subqueries. (b) Subqueries removed.

Figure 2: Removing subqueries from scalar expressions (Q_1 is known to return at most one row).

replaced in the scalar expression by new columns x_1, x_2. Those columns are computed prior to their usage. In general, $\mathcal{A}^{\mathrm{LOJ}}$ will be used, so that a NULL value is passed up when the correlated subquery returns an empty value. Unless it is determined at compile time that a subquery returns at most one row —e g. due to selection on keys— a new operator max-1-row is added on top of scalar-valued subqueries, to raise a run-time error if the subquery returns more than one row. For Fig 2, assume that Q_1 is known to return at most one row. Details on how to translate each type of subquery are relatively straightforward and we skip them for brevity.

In a direct implementation of subquery semantics, the scalar evaluator would recursively call into the relational execution engine, providing parameter values each time. In our scheme, calls are only in one direction, from the relational engine into the scalar evaluator, so there is no need to support recursive calls between the two components.

There are a number of obvious optimizations on the above scheme. For example, we have mentioned that subquery analysis can tell if at most one row will be produced, and then there is no need to check for max-1-row at run time. If exactly one row is returned, e. g. the root of a subquery is a scalar aggregate, then we can use $\mathcal{A}^{\times}$ instead of $\mathcal{A}^{\mathrm{LOJ}}$.

The frequent case where the subquery computes a new boolean column x_1, say from an Exists subquery, and the parent is a select with predicate $x_1 \wedge p$, then Apply itself can filter rows from R, instead of passing a boolean column. Semijoin-Apply, $\mathcal{A}^{\exists}$, filters out rows on which $E(r)$ is empty. Its definition is, again, a straightforward modification of $\mathcal{A}^{\times}$. Anti-semijoin-Apply, $\mathcal{A}^{\nexists}$ is also convenient. These various forms of Apply have been introduced for convenience, and they contribute to a practical, efficient implementation, but they can all be expressed based on $\mathcal{A}^{\times}$, whose purpose is to abstract parameterization. We deal with minimal operator sets later, in Section 5.

An additional practical note is needed on "strictness," which refers to whether or not it is valid to execute expressions eagerly. As described, the above scheme eagerly computes the result of all subqueries before they are needed. This is not correct for the CASE WHEN operator of SQL, which is similar to the conditional evaluation (c ? e_1 : e_2) in C —only *one* of the expressions has to be evaluated; the other could produce a run-time error. This is an uninteresting but necessary case that has to be implemented for completeness. It can be addressed by conditioning execution of the subquery at the Apply operator, or carrying the error silently until the result is actually required.

Two aspects in the algebraic representation of subqueries appear very specific to turning relations into scalar values: non-strict evaluation, and max-1-row verification. There are no run-time errors in relational algebra. Such subquery situations resemble run-time verification of types in languages like LISP. Having to rely on dynamic type checking suggests that there might be a problem in the application design, because relational databases and SQL are statically checked. An example of the problem is a scalar-valued subquery with lookup on an *undeclared* key; the subquery returns at most one row, but the compiler has no way of knowing.

4 Typical forms

Typical subquery cases that have been described in the literature include Exists, Not Exists, and comparison with a scalar aggregate query, in which the subquery does not in turn use additional subqueries. Depending on the type and position of the subquery, the algebraic representation of such query uses either $\mathcal{A}^{\times}$, $\mathcal{A}^{\mathrm{LOJ}}$, $\mathcal{A}^{\exists}$, or $\mathcal{A}^{\nexists}$. And the parameterized expression underneath each Apply is of the form Group By, followed by Select, followed by cross products. "Subquery unnesting" or "correlation removal" consists of rewritting the query without Apply operators. We examine this process more formally later on; for now, we show how various transformations lead to different alternatives.

The first obvious case is when the parameterized expression E doesn't contain a Group By node at the root. Then the filter can be "absorbed" into the Apply operator, which becomes a regular join, outerjoin, semijoin, or antisemijoin. For example, the following query obtains the customers in Mexico who have placed some order:

```
select  *
from    CUSTOMERS
where   CUSTOMERS.country = "Mexico" and
        exists(select * from ORDERS
                 where CUSTOMERS.cust# = ORDERS.cust#)
```

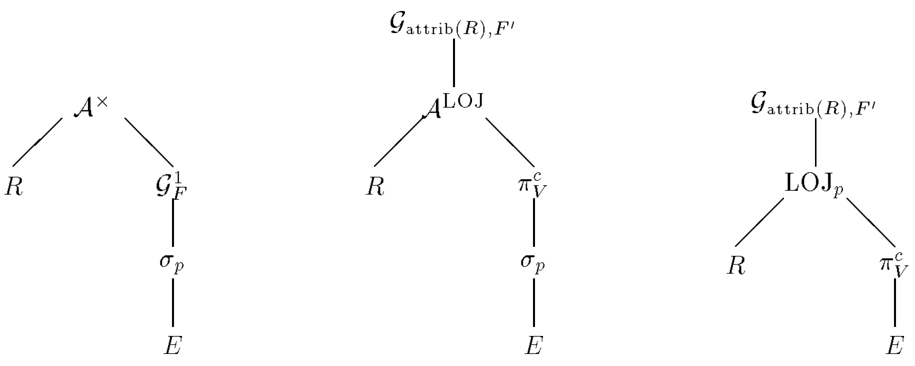
Using p_1 for the condition on country, p_2 for the key comparison, the query is directly mapped to the following, which can be unnested immediately using left semijoin:

$$(\sigma_{p_1}\mathrm{CUSTOMER})\ \mathcal{A}^{\exists}\ (\sigma_{p_2}\mathrm{ORDERS}) = (\sigma_{p_1}\mathrm{CUSTOMER})\ \mathrm{LSJN}_{p_2}\mathrm{ORDERS}.$$

The join, outerjoin, and anti-semijoin cases are similar. Once the correlation has been removed, orthogonal transformations will specify how to reorder, free of parameterization or "nesting" issues.

4.1 Group By

There are two kinds of Group By available in SQL. One is a "regular" Group By, which we denote here $\mathcal{G}_{A,F}$, which contains grouping columns and aggregates to compute. The relational input is partitioned in groups with same values on columns A (for grouping purposes,



(a) $\mathcal{G}^1_F$ subquery. (b) Move up $\mathcal{G}^1_F$. (b) Remove correlation.

Figure 3: Correlation removal for parameter-free E.

two NULL values count as "same"), then aggregates are computed on each partition. Note that $\mathcal{G}_{A,F}\emptyset = \emptyset$, and $\mathcal{G}_{A,F}$ never evaluates aggregates on an empty set. The result of $\mathcal{G}_{\emptyset,F}$ is to evaluate the aggregate functions on the entire input relation —outputing either zero or one rows. Aggregation with columns can be expressed using parameterization from $\mathcal{G}_{\emptyset,F}$, which can be a useful formulation; for all x in the domain attributes A could ever take:

$$\mathcal{G}_{A,F}R = \bigcup_x (\{x\} \times \mathcal{G}_{\emptyset,F}(\sigma_{A=x}R)).$$

In SQL, if no grouping columns are specificed, a "scalar aggregate" is used. We denote this operator by $\mathcal{G}^1_F$. It always returns one row. If the input is not empty, then it returns the same as $\mathcal{G}_{\emptyset,F}$; otherwise, it returns the evaluation of each aggregate function on the empty set, i. e. agg($\emptyset$). Perhaps the simplest example of their distinct behavior is on COUNT(*); it can return zero in $\mathcal{G}^1$, but in $\mathcal{G}$ its value is always greater than zero.

A typical subquery form that uses scalar aggregate is shown in Figure 3, where it is transformed into a query without correlations, assuming that E contains no parameters. Moving scalar aggregates up past Apply is the most involved case. First, note that the only non-trivial Apply right above $\mathcal{G}^1$ must be $\mathcal{A}^{\times}$, as in Figure 3(a); exists checking with $\mathcal{A}^{\exists}$, or $\mathcal{A}^{\nexists}$ would be immediately simplified out, because the input generates exactly one row; for the same reason, $\mathcal{A}^{\text{LOJ}}$ reduces to simply $\mathcal{A}^{\times}$.

The difficulty arises because we are use $\mathcal{G}$ to compute $\mathcal{G}^1$, but they behave differently. Unlike $\mathcal{G}$, $\mathcal{G}^1$ might compute aggregates on empty sets. First, changing $\mathcal{A}^{\times}$ by $\mathcal{A}^{\text{LOJ}}$ keeps all outer rows, padded with nulls on empty sets. Note, however, that there is no way to distinguish the cases $E(r) = \{(\text{NULL}, \ldots, \text{NULL})\ \}$ and $E(r) = \emptyset$. This is important; the aggregate MIN(col is NULL ? 1 : col) returns 1 on an all-null tuple, but it returns NULL on the empty set. As long as agg($\emptyset$) = agg(\{ null \}), which is the case on SQL aggregates, the problem can be addressed by rewriting each agg to be over a single column, which is computed, if necessary, by the π^c_V below. F' are the single-column aggregates in Figure

3(a). It is possible to handle aggregates where agg($\emptyset$) $\neq$ agg({ null }), but it requires more involved column manipulation, and we do no go into details here.

For this transformation to be valid, R must have a key. Note that, if necessary, it is always possible to generate transient keys at run-time. We return to this point later in Section 5.

Optimizing the general case is obviously possible —i. e. the old, insightful proposal of using joins for subquery evaluation— but it depends on the utilization context. Consider the following query:

select *
from R_1
where $R_1.a = 1$ **and**
$R_1.b =$(**select** MAX($R_2.c$) **from** R_2 **where** $R_1.d = R2.f$)

The aggregate result is used by a σ operator, which will reject a row from R_1 on MAX($\emptyset$). Given this context, the scalar aggregate $\mathcal{G}^1_F$ may be replaced by a regular aggregate $\mathcal{G}_{\emptyset,F}$, which moves up the Apply without any issues related to evaluation on empty sets (R must still have a key):

$$R \; \mathcal{A}^{\times} \; \mathcal{G}_{\emptyset,F} E = \mathcal{G}_{\text{attrib}(R),F}(R \; \mathcal{A}^{\times} \; E).$$

If the SQL above used OR instead of AND, the context of usage is no longer of emptyset-rejection, and outerjoin must still be used.

In summary, the well-known "COUNT bug" [GW87] is not specific to the COUNT aggregate, and outerjoin alone does not solve it. The anomaly can occur on any aggregate function; aggregates need modification to distiguish empty set from null values;[5] and optimizing out the outerjoin depends on utilization context.[6]

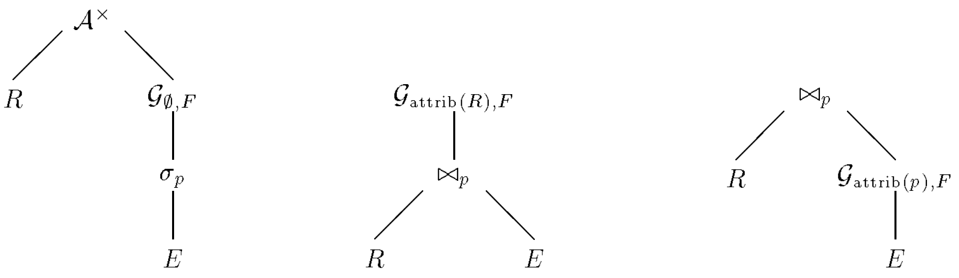
4.2 Evaluation order

Once the Apply operator is removed, the result is a "plain" Select/Join/Outerjoin/Group By query, which could have been typed directly, without subqueries. The quality of the optimizer work on such result is unrelated to any subquery issues.

Figure 4 shows three basic forms that are obtained from a typical salar aggregate subquery. The conversion between forms (a) and (b) follows from Apply properties. Depending on the size of R, a correlated execution for a query similar to (b) should be considered by the optimizer. Join / Group By reordering [CS94, YL95] takes us from (b) to (c) —as would an operator reversal followed by correlation removal from (a). The transformation to (c) requires p to be equality comparisons with columns from E. For an arbitrary predicate p, some aggregation can be pushed below the filter, but the later aggregation has to stay in place (a local/global aggregation strategy). That is the general form of alternative (c).

[5]This point is considered in [Mur92].

[6]This appears a flavor of outerjoin simplification, when a later operator discards outerjoin-introduced nulls [RGL90].



$\mathcal{A}^{\times}$
R
$\mathcal{G}_{\emptyset,F}$
σ_p
E
$\mathcal{G}_{\mathrm{attrib}(R),F}$
$\bowtie_p$
R
E
$\bowtie_p$
R
$\mathcal{G}_{\mathrm{attrib}(p),F}$
E

(a) Original correlation. (b) Late aggregation. (c) Early aggregation.

Figure 4: Execution alternatives.

A possible optimization for (c) is to employ semijoin reductions on E [SSS95, SHP+96]. This reduction can be made exact, or approximate, using something like hash filters [CHY93]. Cost estimation is required to select one of the alternatives, for a given database instance.

Depending on the aggregate utilization, the form in Figure 4(b) may require an outerjoin, instead of join. We point out that the form in Figure 4(c) can still be achieved in the outerjoin case. It requires a π^c project above the top outerjoin, to fix up aggregates that would have been computed on null rows. The details are outside the focus of the current paper.

5 Correlations can always be removed

Are SQL subqueries simply a convenient shorthand, or do they increase the class of queries that can be posed? Can we remove correlations on *arbitrary* subqueries? It is difficult to answer this question for "SQL," because there is no one SQL. The standard language evolves, and there are as many SQLs as products implementing it, because there are always product-specific variants.

Intuitively, we shold be able to handle SQL that is *deterministic, strict, and statically safe*. Some language variants include non-deterministic features, such as retrieving the first few rows (e. g. MS Access, MS SQL Server 7.0, [CK97]). Section 3 showed that some subquery utilization require non-strict (on-demand) evaluation. It also showed that a run-time check of max 1 row is sometimes necesssary. This suggest a kind of dynamis type-checking alien to relational algebra. We do not consider those features in this section.

Define a *relational-consistent* query Q as one that can be expressed algebraically using *relational-consistent* operators $\{\times, \sigma_p, \pi_A, \pi^c_V, -, \cup, \bowtie_p$, outerjoin, union all, except all, $\mathcal{G}_{A,F}, \mathcal{G}^1_F, \mathcal{A}^{\times}, \mathcal{A}^{\mathrm{LOJ}}, \mathcal{A}^{\exists}, \mathcal{A}^{\nexists}\}$. Duplicates are of course allowed in the query; π_A is duplicate-preserving column removal; and π^c_V is used to introduce new computed columns. Operator $\mathcal{G}_{A,F}$ does duplicate elimination as a special case of aggregate computation. Both duplicate-preserving and set-based union and difference are included. A language $\mathcal{L}$ is *relational-consistent* if it only allow specification of relational-consistent queries.

The scope of our proofs is relational-consistent queries. It should be relatively easy to check if our results hold for your garden variety of SQL. Examine a syntax-directed translation of your input into relational operators, and consider the rules of Section 3 for placement of Apply operators, to decide if the target expression is limited to relational-consistent operators.

We further define the set of *base SQL operators* to be $\{\times, \sigma, \pi, \pi^c, -, \cup, \mathcal{G}_{A,F}, \mathcal{G}^1_F\}$. Our main result is that any relational-consistent SQL query can be represented with base SQL operators only. As it turns out, handling EXCEPT ALL requires a special strategy. We will first consider queries without UNION ALL, EXCEPT ALL.

First, observe that duplicate preserving π can be moved up past other operators like select, or join, which do not mind the extra columns. Eventually, it would end up right below a Group By, union, difference, or at the root of the tree. Make Group By and set operators "absorb" π —for Group By, π can be ignored, for set operators it can be made part of the operator arguments, as is done with the CORRESPONDING modifier in SQL [MS93]. Also, join can always be rewritten as select over cross product, without duplicate issues. Thus, in the sequel we assume, without loss of generality, that any π operator is at the root of the tree, and there are no (inner) joins.

Theorem 1. *A relational-consistent query Q that does not use UNION ALL, EXCEPT ALL, can be made to have a key at each intermediate result, except for, perhaps, the root. The tree topology is unchanged and no new operators are used.*

Proof sketch. Any base table can be made to have a key; if necessary, add a new column with the row-id of each row (e. g. a disk address, or whatever is used to make secondary indices identify rows). The output of operators $\{-, \cup, \mathcal{G}, \mathcal{G}^1\}$ always contains a key, regardless of the inputs. The output of operators $\{\times, \sigma, \pi^c, \mathcal{A}^\times, \mathcal{A}^{\mathrm{LOJ}}, \mathcal{A}^\exists, \mathcal{A}^\nexists$, outerjoin $\}$ contains a key, if the inputs have keys. Only the root output, if π is used, may not have a key. The lemma follows by induction arguments. ■

Lemma 2. *A relational-consistent query Q that does not use UNION ALL, EXCEPT ALL, can be rewriten using relational-consistent operators excluding UNION ALL, EXCEPT ALL, outerjoin, $\mathcal{A}^{\mathrm{LOJ}}$, $\mathcal{A}^\exists$, $\mathcal{A}^\nexists$.*

Proof sketch. First augment the base tables in Q to produce keys on all intermediate results. Then $\mathcal{A}^\exists, \mathcal{A}^\nexists, \mathcal{A}^{\mathrm{LOJ}}$ and outerjoin can be rewritten using $\mathcal{A}^\times$, $\mathcal{G}$, $-$, $\cup$. ■

Theorem 3. *A relational-consistent query Q that does not use UNION ALL, EXCEPT ALL, can be rewritten as Q' over base SQL operators.*

Proof sketch. First, by Lemma 2, Q can be rewritten into Q_1 on base SQL operators plus $\mathcal{A}^\times$. By Lemma 1, Q_1 can be rewritten into Q_2 so that there is a key on each intermediate result. On Q_2, use identities (1) through (8) in Figure 5, to move up base SQL operators up past $\mathcal{A}^\times$, and eventually convert each $\mathcal{A}^\times$ it into $\times$. Remove Apply operators recursively from bottom to top. ■

[7]If agg($\emptyset$) $\neq$ agg({ null }), moving up the $\mathcal{G}^1$ can still be achieved using base SQL operators, but it needs a more involved column manipulation.

$$
\begin{aligned}
R\,\mathcal{A}^{\times}\;E &= R \times E, \text{if no parameters in } E \text{ resolved from } R && (1)\\
R\,\mathcal{A}^{\times}\;(\sigma_p E) &= \sigma_p(R\,\mathcal{A}^{\times}\;E), && (2)\\
R\,\mathcal{A}^{\times}\;(\pi^c_v E) &= \pi^c_v(R\,\mathcal{A}^{\times}\;E), && (3)\\
R\,\mathcal{A}^{\times}\;(E_1 \cup E_2) &= (R\,\mathcal{A}^{\times}\;E_1) \cup (R\,\mathcal{A}^{\times}\;E_2) && (4)\\
R\,\mathcal{A}^{\times}\;(E_1 \times E_2) &= (R\,\mathcal{A}^{\times}\;E_1) \bowtie_{R.key} (R\,\mathcal{A}^{\times}\;E_2) && (5)\\
R\,\mathcal{A}^{\times}\;(E_1 - E_2) &= (R\,\mathcal{A}^{\times}\;E_1) - (R\,\mathcal{A}^{\times}\;E_2), && (6)\\
R\,\mathcal{A}^{\times}\;(\mathcal{G}_{A,F} E_2) &= \mathcal{G}_{A \,\cup\, \text{attrib}(R),F}(R\,\mathcal{A}^{\times}\;E_1), && (7)\\
R\,\mathcal{A}^{\times}\;(\mathcal{G}^1_F E_2) &= \mathcal{G}_{\text{attrib}(R),F'}(R\,\mathcal{A}^{\text{LOJ}}\;(\pi^c_V)). && (8)
\end{aligned}
$$

Identities 4 through 8 require that R contains a key $R.key$. In identity (8), F' contains aggregates in F expressed over a single-column, which is computed if necessary by the π^c_V below. Icentity (8) is valid for all aggregates such that $\text{agg}(\emptyset) = \text{agg}(\{\text{ null }\})$, which is true for SQL aggregates.[7]In identity (5), Join on $R.key$ is used as a shorthand for the obvious predicate.

Figure 5: Basic properties of $\mathcal{A}^{\times}$.

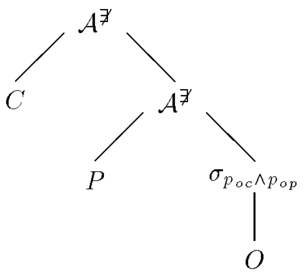
As an example, Figure 6 shows how correlations are removed from a universal quantification query: Select customers that have ordered all products. This is typically expressed in SQL through double negation: There is no product that the customer hasn't ordered. Not-exists Apply is converted directly to difference over $\mathcal{A}^{\times}$, insted of semijoin, because the duplicate count does not matter on the right side of set difference. The columns on which difference is performed are obvious and are not included, to avoid clutter. The resulting query is basically the definition of relational division, which also expresses the universal quantification query.

Identities in Figure 5 can be proved relatively easily from the definition of Apply, and the known properties of base operators. The one involved identity is (8), which was explained earlier in Section 4.1. As an example of the proof style, here is how to derive identity (2):

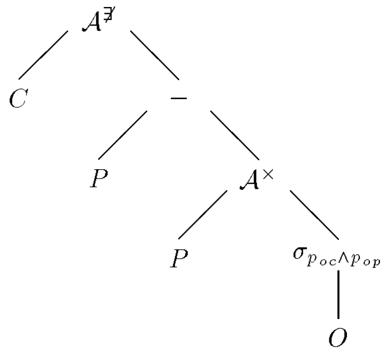
$$
\begin{aligned}
R\,\mathcal{A}^{\times}\;(\sigma_p E) &= \bigcup_{r \in R} (\{r\} \times (\sigma_p E(r)))\\
&= \bigcup_{r \in R} \sigma_p(\{r\} \times E(r)), \text{select moves up past product}\\
&= \sigma_p\left(\bigcup_{r \in R} \{r\} \times E(r)\right), \text{distributivity of select over union}\\
&= \sigma_p(R\,\mathcal{A}^{\times}\;E).
\end{aligned}
$$

Now we show how to handle UNION ALL and EXCEPT ALL.

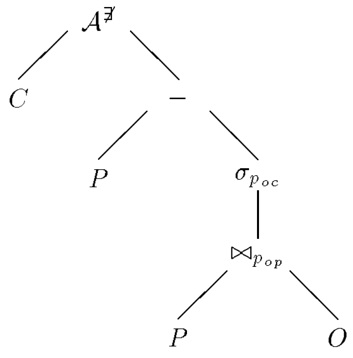
Lemma 4. *A relational-consistent query Q can be rewritten such that UNION ALL, EXCEPT ALL are locally replaced by $\mathcal{G}, \bowtie$ operators, and an additional join with an extra*



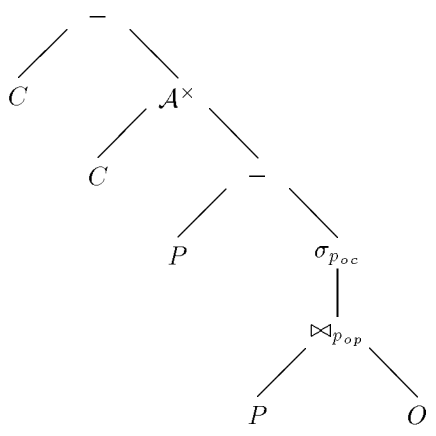
(a) Double not exists.



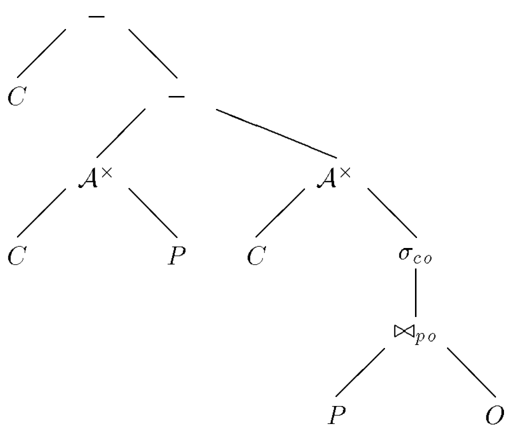
(b) $\nexists$ as difference.



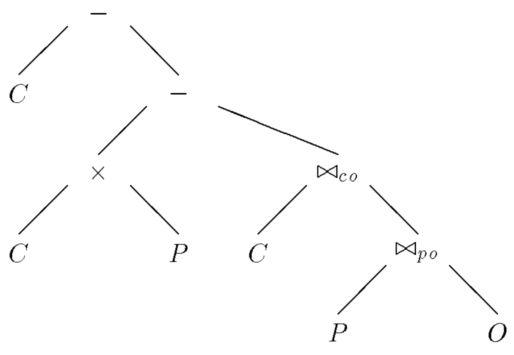
(c) Remove Apply.



(d) $\nexists$ as difference.



(e) Move $-$ up past $\mathcal{A}^{\times}$.



(f) Remove Apply.

Figure 6: Removal of correlations in a universal quantification query.

table of integer values, whose size is bounded by the largest intermediate result. The tree topology and all other operators remain unchanged.

Proof sketch. Duplicate counts can be manipulated directly to execute SET ALL operators, through Group By and Join. To evaluate R_1 EXCEPT ALL R_2 on common columns A, compute $R'_i = \mathcal{G}_{A,x_i=\text{count}(*)}$, then $R' = R'_1 \bowtie_A R'_2$. R' contains each group of values with duplicate counts. A "duplicate expansion" operator $\text{Dup}_{R'.x_1-R'.x_2}$ could generate $R'.x_1-R'.x_2$ copies of each row, appending an integer column y with values $1,\ldots,R'.x_1 - R'.x_2$. The Dup operator introduces keys by appending a counter to its output.
The operator Dup cannot be expressed directly in SQL, but it can be implemented as a join on condition $I.y \leq R'.x_1 - R'.x_2$ on a table of integers I. Note, however, that the size of the table I is potentially unbounded. ■

The goal of Lemma 4 is to allow having keys on each intermediat result of the query. UNION ALL could be handled without the Dup strategy, by column manipulation that reuses input keys. We do not get into the extra complexity because EXCEPT ALL does seem to require using Dup. Observe that when EXCEPT ALL chooses a subset out of a set of rows with equal-valued columns, the input keys to be reused would need to be chosen non-deterministically. The final project would discard any extra key columns and make the result deterministic, but intermediate results are not.

The following Theorems follow from Theorem 1, Theorem 3, and Lemma 4:

Theorem 5. *A relational-consistent query Q can be made to have a key at each intermediate result, except for, perhaps, the root. There can be minor, local modifications to the tree topology, that use an extra table of integer values, whose size is bounded by the largest intermediate result. Operators used are all relational-consistent.*

Theorem 6. *A relational-consistent query Q can be rewritten as Q' over base SQL operators, using an extra table of integer values whose size is bounded by the largest intermediate result.*

From Theorems 3 and 6, subqueries do not add expressive power to a relational-consistent language $\mathcal{L}$ that is rich enough to implement the base SQL operators.

6 Conclusions

We introduced a new operator, Apply, to abstract parameterized query execution, and showed how it could be used to represent algebraically SQL queries with arbitrary subqueries. The Apply operator is likely to be already built into execution engines, so removing subqueries from scalar expressions removes also the requirement for the scalar expression evaluation to call back into the relational engine.

We showed some of the basic properties of Apply, and how they lead to known subquery optimizations. We also showed how Apply opens the door for deep correlated executions, which can be very efficient.

Our detailed mapping into an algebraic form uncovered the general form of the well-known "COUNT bug," which, as explained in Section 4.1, is not really specific to the COUNT aggregate. We described how it can be avoided, in the general case.

Our algebraic model allowed us to prove that subqueries do not add expressive power (see Section 5 for exact details). The basic idea is, if we can tell at compile time that a query Q will not generate run-time errors (due to a scalar-valued subquery returning more than one row), we can rewrite Q without using subqueries. A useful intermediate step in our construction was the introduction of keys on each intermediate result of a query. This is an important result for the use in SQL of techniques that rely on non-duplicates semantics.

The algebraic model described here is implemented in MS SQL Server 7.0, which is SQL92 compliant —this showing, in practice, that our scheme is complete for SQL.

There are a number of obvious, practical open problems. Our construction for the removal of arbitrary correlations does not necessarily lead to efficient expressions, e. g. the number of operators in the un-correlated query can grow exponentially in the number of initial binary operators (see identities (4), (5), (6)). Can we say something about un-correlated forms when *composite* operators, or "idioms," such as semijoin, or relational division are allowed?

Are there other "idioms" that are more suitable to the kind of requests currently done via subqueries (e. g. the direction taken in [CR96])? Outerjoin is an "idiom" that made it all the way to the source language; anti-semijoin hasn't, but it is very effective in execution and it can be introduced early on in the internal algebraic representation.

In principle, subqueries do not make optimization harder, since equivalente queries can be written without them so the class of input queries is not really expanded. In practice, subqueries tend to be very concise, compared to equivalent queries without them. In our view, subquery manipulation through Apply is a fundamental step in dealing with SQL subqueries, and it enables robust optimizations for a large class of common cases, but there is work left on the generation of efficient plans for arbitrary subqueries.

Acknowledgements. Pedro Celis has independently developend some of the material presented here, while at Tandem; I don't know to what extent. Arnie Rosenthal provided valuable comments on an early draft of this paper.

References

[CD95] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proceedings of ACM SIGMOD 1992*, pages 383–392, 1995.

[Cel96] P. Celis, 1996. Private communication.

[CHY93] M.-S. Chen, H.-I. Hsiao, and P. S. Yu. Applying hash filters to improving the execution of bushy trees. In *Proceedings of the Nineteenth International Conference on Very Large Databases, Dublin*, pages 505–516, 1993.

[CK97] M. Carey and D. Kossman. On saying "Enough already!" in SQL. In *Proceedings of ACM SIGMOD 1997*, pages 219–230, 1997.

[CR96] D. Chatziantoniou and K. A. Ross. Querying multiple features of groups in relational databases. In *Proceedings of the 22nd International Conference on Very Large Databases, Bombay*, pages 295–306, 1996.

[CS94] 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.

[Day87] 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.

[Feg98] L. Fegaras. Query-unnesting in object-oriented databases. In *Proceedings of ACM SIGMOD 1998*, pages 49–60, 1998.

[GLR97] C. A. Galindo-Legaria and Arnon Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Transactions on Database Systems*, 22(1):43–73, March 1997.

[GW87] R. A. Ganski and H. K. T. Wong. Optimization of nested SQL queries revisited. In *Proceedings of ACM SIGMOD 1987*, pages 23–33, 1987.

[Kim82] W. Kim. On optimizing an sql-like nested query. *ACM Transactions on Database Systems*, 7(3):443–469, September 1982.

[MS93] J. Melton and A. R. Simon. *Understanding the new SQL: A complete guide.* Morgan Kaufmann, San Francisco, 1993.

[Mur89] M. Muralikrishna. Optimization and dataflow algorithms for nested tree queries. In *Proceedings of the Fifteenth International Conference on Very Large Databases, Amsterdam*, pages 77–85, 1989.

[Mur92] M. Muralikrishna. Improved unneesting algorithms for join aggregate sql queries. In *Proceedings of the Eighteenth International Conference on Very Large Databases, Vancouver*, pages 77–85, 1992.

[RGL90] A. Rosenthal and C. A. Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. In *Proceedings of ACM SIGMOD 1990*, pages 291–299, 1990.

[RR98] J. Rao and K. A. Ross. Reusing invariants: A new strategy for correlated queries. In *Proceedings of ACM SIGMOD 1998*, pages 37–48, 1998.

[SHP$^+$96] P. Seshadri, J. M. Hellerstein, H. Pirahesh, T. Y. C. Leung, R. Ramakrishnan, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. In *Proceedings of ACM SIGMOD 1996*, pages 435–446, 1996.

[SPL96] 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.

[SSS95] D. Srivastava, P. J. Stuckey, and S. Sudarshan. The magic of theta-semijoins. Technical report, AT&T Bell Laboratories Technical Report, 1995.

[SZ89] 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.

[YL95] 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.