Optimizing Nested Queries with Parameter Sort Orders

Ravindra Guravannavar* Ramanujam H.S.† S Sudarshan

Indian Institute of Technology Bombay
ravig@cse.iitb.ac.in

Sybase Software India Pune
ramanujam.s@sybase.com

Indian Institute of Technology Bombay
sudarsha@cse.iitb.ac.in

Abstract

Nested iteration is an important technique for query evaluation. It is the default way of executing nested subqueries in SQL. Although decorrelation often results in cheaper non-nested plans, decorrelation is not always applicable for nested subqueries. Nested iteration, if implemented properly, can also win over decorrelation for several classes of queries. Decorrelation is also hard to apply to nested iteration in user-defined SQL procedures and functions. Recent research has proposed evaluation techniques to speed up execution of nested iteration, but does not address the optimization issue. In this paper, we address the issue of exploiting the ordering of nested iteration/procedure calls to speed up nested iteration. We propose state retention of operators as an important technique to exploit the sort order of parameters/correlation variables. We then show how to efficiently extend an optimizer to take parameter sort orders into consideration. We implemented our evaluation techniques on PostgreSQL, and present performance results that demonstrate significant benefits.

1 Introduction

Complex nested queries involving several correlation attributes, aggregates and predicates other than equality are commonly used. With support for expensive user defined functions that can appear in the WHERE clause predicates and in the projection list of the SELECT clause, and the introduction of the LATERAL construct in SQL99, users have more ways of formulating correlated nested queries.

Naïve nested iteration plans for such queries can be very inefficient as the nested subquery is evaluated for every distinct binding of the correlation attributes in the outer query. De-correlation techniques have been extensively studied [12, 6, 13, 2, 15] and applied to enable traditional optimizers generate more efficient set oriented plans for nested queries. However, decorrelation is not always applicable, and even if applicable may not be the best choice in all situations since decorrelation carries a materialization overhead.

Consider the query in Example 1. The query uses a user-defined-function in its SELECT clause. Decorrelation techniques proposed till date cannot be applied in general to queries, such as the one in the example, where the function body contains procedural statements. The only option available may be to evaluate the function repeatedly for each distinct binding of the parameters.

Example 1: Find the turn around time for high priority orders. The turn around time of an order is calculated as the maximum of the differences between the ship date and placement date of all its line items, if the order price is $<$ 2000 and it is calculated as the maximum of the differences between the commit date and placement date otherwise.

```
SELECT o_orderkey, turn_around_time(o_orderkey,
       o_totalprice, o_orderdate)
FROM   ORDERS WHERE o_orderpriority='HIGH';


DEFINE  turn_around_time(@orderkey, @totalprice, @orderdate) {
    IF (@totalprice < 2000)
        SELECT max(l_shipdate − @orderdate) FROM LINEITEM
        WHERE l_orderkey=@orderkey;
    ELSE
        SELECT max(l_commitdate−@orderdate) FROM LINEITEM
        WHERE l_orderkey=@orderkey;
}
```

Our optimization technique helps produce plans that are significantly better than naïve nested iteration for such queries. We consider the benefits of ordering the parameters and the cost of producing the required ordering by the outer query block and come up with a globally optimal plan. Note that the best parameter ordering for each query in the function body can be different and also there can be multiple functions invoked from the same outer query block. Further, the cost of the plan for the outer query block can vary significantly based on the sort order it needs to guarantee on the parameters. Our optimization algorithm can currently handle queries invoking multiple functions where each function can in turn have multiple parameter-

*Work partly done while at Aztec Software, India

†Work done while at Indian Institute of Technology, Bombay

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

ized queries, but without procedural iterations and nested procedure calls. [1]

Nested iteration is also a natural way of executing SQL/XML and XQuery queries, which generate nested structures; SQL/XML (see e.g., [3]) is a recent extension to SQL which allows XML output to be created directly using SQL, uses nested queries in the SELECT clause.

Recently, Graefe [8] argued for the importance of nested iteration plans and outlined several techniques to improve the efficiency of nested iteration. These techniques were at the evaluation engine level, and include novel ways of handling prefetching from disk. However, [8] does not consider how to extend query optimizers to effectively handle nested iteration.

Sorting of calls can reduce per-call costs of nested iteration significantly. For example, if a function uses a clustering index to fetch records matching the parameter values, calling the function with the parameters sorted in the clustering order will eliminate or greatly reduce random I/O. Such sorting has been used earlier in the context of nested loops join: a hybrid indexed-nested-loops join sorts the record ids returned by a secondary index before fetching the actual records. In this case the nested action is straightforward, but in general it can be much more complex. For example, a function or subquery may use multiple indices on different relations, or invoke other functions or subqueries itself, making the task of finding an overall optimal plan harder.

To our knowledge, the issue of finding an optimal plan taking into account sort orders for parameters of subqueries or procedures has not been addressed in the past. Having a sort order of the parameters (across calls) that matches the sort order of the inner query gives an effect similar to merge join. However, the problem of optimizing nested queries considering parameter sort orders is significantly different from the problem of finding the optimal sort orders for merge joins. The nesting of subqueries makes certain orderings impossible, whereas merge join is at liberty to sort the inputs as it sees fit. For example, if variable B is bound by a nested iteration that takes as parameter variable A, it is not possible to get a sort order of B, A, while A, B is possible. Further, when nested queries have multiple branches and multiple levels, a sort order that is suboptimal for individual query blocks may be the optimal for the overall query. We address the problem of finding an optimal plan taking parameter sort orders into account.

We make two primary technical contributions in this paper.

- First, we show how the effect of parameter sorting can be modeled by using *state retention* of operators across calls. The *state retention* techniques extend the benefits of merge join to situations where a merge join is not applicable. Based on the state retention technique, we present an efficient evaluation strategy for nested aggregate queries with equality and non-equality correlation predicates.

- We then show how a cost-based optimizer can be extended to find an optimal plan taking parameter sort orders into account. To do so, we introduce the notion of *interesting parameter sort orders*, a new physical property, termed *Containment-Differential* and cost estimation of plans for multiple evaluations.

  The naïve approach of trying every possible sort order for parameters is very expensive, since it can generate an exponential number of sort orders. We would have to not only optimize the subqueries/procedures for each sort order, but also find the best plans for the outer query that can generate the parameters in the required sort order. Most of the sort orderings would not be beneficial to the subquery or procedure. We therefore introduce a step that analyzes subqueries/procedures to find what parameter sort orders are interesting, and then try out plans that generate these sort orders. Not all sort orders are feasible for parameters due to the nesting of procedures/subqueries, and the optimizer takes this into account.

  Note that the optimizer can also consider standard optimization techniques such as decorrelation, where applicable, and choose the best plan overall.

  Our description is based on the Volcano/Cascades optimization framework of [9], but the underlying ideas can be used with System R style optimizers as well.

We have implemented the state retention techniques on the PostgreSQL database system. We present a preliminary performance study illustrating the benefits of using our techniques.

To our knowledge, the optimization issues we tackle have not been addressed earlier. The most closely related work is the optimization algorithm used in Microsoft SQL Server, which has some extensions to better handle nested iteration [4]; however, it does not attempt to find the optimal plan taking parameter sort orders into consideration, and does not consider state retention.[2] Work on optimization of generation of nested XML structures [16] is also related; however, the earlier work has concentrated on transformation to outerjoins, which is a form of decorrelation. See Section 7 for more details on how our technique differs from earlier work.

The optimization techniques we propose have wide applicability: they can be applied to optimize correlated nested queries in SQL WHERE and SELECT clauses, and invocations of stored procedures/functions with parameter bindings being generated from an outer query block. We

[1] We are working on more accurate costing by analyzing the function body to derive the expected number of times a query in the function body gets executed each time the function is invoked. Currently we make an assumption that each query in the function body gets executed exactly once.

[2] For instance, SQL Server chooses plain nested iteration for the query in Example 5 of Section 3

address the issue of set-valued functions in the SELECT clause, used for example to generate XML, in Section 6.

The rest of this paper is organized as follows. Section 2 describes the logical representation we adopt for nested queries and queries that invoke functions. In Section 3, we describe our new evaluation strategies for nested queries that are based on state retention of operators. In Section 4, we illustrate how a Volcano style cost-based optimizer [9] can be extended to consider the proposed techniques. In Section 5, we present our experimental results and analysis. Section 6 describes some of the possible extensions to our work and Section 7 details related work. Finally, we present our conclusions in Section 8.

2 Logical Representation of Nested Iteration

In a typical nested iteration plan, the inner subquery returns a set of tuples for each binding of the correlation variable(s) produced by the outer query block. The inner subquery can be thought of as a function parameterized on the correlation variables and executed repeatedly (inside an outer loop).

Example 2: This query uses a subquery in its WHERE clause.

```
SELECT PO.order_id
FROM   PURCHASEORDER PO
WHERE  PO.order_date > '2004-01-01' AND
       PO.default_ship_to
       IN
       (SELECT  ship_to
       FROM     ORDERITEM OI
       WHERE    OI.order_id = PO.order_id );
```

Queries that invoke expensive user-defined functions as part of their WHERE clause predicates (Example 1) or SELECT expression list are similar to nested queries [8]. There can be any number of queries inside a function body and these queries can use any number of parameters bound by the query that invokes the function.

Example 3: Functions can have more than one query in their body along with several control flow statements, as illustrated below.

```
DEFINE fn(p1, p2, ... pn) RETURNS INTEGER AS
BEGIN
    fnQ1 < p1, p2 >;
    fnQ2 < p1, p2, p3 >;

    OPEN CURSOR ON fnQ3 < p2, p3 >
    // Assume v1, v2 are bound from the records of the cursor
    LOOP
        fnQ4 < p1, p2, v1, v2 >;
    END LOOP
    ...
    RETURN retval;
END
```

We adopt a variant of the *Apply* operator proposed in [5] for representing correlated nested subqueries and queries involving functions. Figure 1 depicts the *Apply* operator pictorially. The *Apply* operator evaluates its right subexpression for every tuple in the result of its left subexpression. The *Apply* operator then evaluates a predicate involving the tuple from the left subexpression and the result of the right subexpression. The predicate can be IN or NOT IN that check for set membership, EXISTS or NOT EXISTS that check for set cardinality, a scalar comparison ($=, \neq, >, \geq, <, \leq$), or a comparison of a scalar with members of a set *relop* ANY or *relop* ALL. The *Apply* operator, instead of evaluating a predicate, can evaluate a scalar valued user-defined function in the select clause. It is also possible to use the results of the right subexpression for each tuple in the left subexpression and output a nested relation. We return to this issue in Section 6.

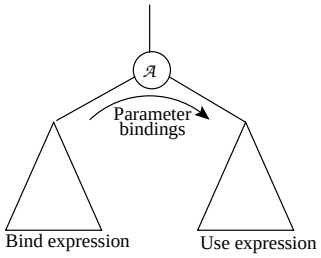


Figure 1: *Apply* Operator with a single *use* subexpression

We refer to the left subexpression of the *Apply* operator as the *bind* expression since it binds the parameters (correlation variables) and the right subexpression as the *use* expression. In general, an *Apply* operator can have multiple *use* expressions that represent multiple subqueries/functions nested at the same level. In a complex multi-level nested query an expression E may use some variables and may bind other variables. The variables that E binds may be passed on to the use expressions of parent or ancestor *Apply* operators; E must be in the left-most subtree of such *Apply* operators. The variables that E uses must be defined at parent or ancestor *Apply* operators; E must be in a use-subtree, i.e., non-left-most subtree, of such *Apply* operators.

Figure 2 shows the logical representation of the query given in Example 2 and Figure 3 shows the representation of a query block Q_i containing a call to the function fn() (of Example 3) in its WHERE clause or the SELECT list.

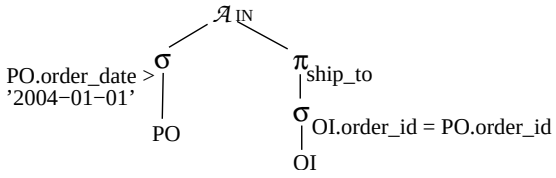


Figure 2: Representing Nested Queries

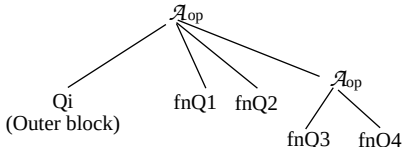


Figure 3: Query with Functions

3 Exploiting Parameter Sort Order

Graefe [8] describes how sorting of tuples produced by the outer query block on the correlation attributes helps in speeding up the inner query execution by providing advantageous buffer effects. A second advantage of sorting, as proposed in the System R paper [14], is that if the results of the inner query are cached for each distinct value of the correlation variable(s), then sorting the outer tuples (parameters) gives space savings by allowing us to cache at most one result at any given time.

In this section we propose additional techniques for exploiting the sort order of correlation bindings by retaining the state of the inner query execution across multiple bindings of the correlation variables. The cost of evaluating inner query block can vary significantly depending on the parameter sort order guaranteed by the outer query block. Correspondingly, the cost of the outer query block can vary significantly depending on the sort order it needs to guarantee on the tuples produced. A cost-based optimizer can consider the various options available and decide on the overall best plan. Section 4 addresses this issue.

3.1 Restartable Segment Scan

If a relation referenced in the inner query is physically sorted on the column that appears in an equality predicate with the correlation variable, sorted correlation bindings allow us to scan the inner relation exactly once across all iterations. We call this idea as *Restartable Segment Scan* and illustrate it with Example 4.

Example 4

```
SELECT o_orderkey
FROM   ORDERS
WHERE  o_orderdate
       NOT IN
       (SELECT  l_shipdate
       FROM     LINEITEM
       WHERE    l_orderkey = o_orderkey );
```

Suppose the LINEITEM table is stored sorted on l_orderkey column and the plan for the outer query-block guarantees to produce the bindings for the correlation variable, o_orderkey, in sorted order. The clustered table scan can stop as soon as a value greater than the value of the correlation variable is found and restart from this point on the next invocation, thus retaining state. This query plan reads the outer and the inner relations exactly once and the reads for the inner relation will be sequential. Note that a *clustered index scan*, will have to access the non-leaf pages of the index for every tuple produced by the outer block. Further note that this plan performs better than the plan produced by magic decorrelation [15], which, if applied in this case, requires two joins; the first one is that of the LINEITEM relation with the filter set, which in this case comprises of all the distinct o_orderkeys from the ORDERS relation and the second one is an anti-join of the outer relation (ORDERS) with the result of the first join.

The effect produced by employing the *Restartable Segment Scan* is similar to that of a merge join. However, the *Restartable Scan* is applicable in situations where a merge join cannot be directly used. For example, consider a user-defined-function with several parameterized queries inside its body, interspersed with other procedural constructs. If such a function is invoked from another query that supplies the bindings for the parameters in sorted order, a merge join is not directly applicable since the other procedural statements in the function body do not permit set-oriented execution of the queries. In this case we can use a *Restartable Scan* and get the same effect as a merge join.

In general, *Restartable Segment Scan* is effective when a significant portion of the inner relation(s) is accessed to answer the query. When only a small portion of the inner relation needs to be accessed, a nested iteration plan employing index lookup for the inner relation may be the best option [8], provided an appropriate index is available.

3.2 Incremental Computation of Aggregates

The technique illustrated in the previous section is applicable only when the correlation predicate in the inner subquery is an equality predicate. We now show how the *Restartable Segment Scan* can be employed for non-equality predicates, when the inner subquery has an aggregate. Decorrelation is often very expensive for such queries. Consider the SQL query shown in Example 5. The query lists days on which the sales exceeded the sales seen on any day in the past.

Example 5

```
SELECT day, sales
FROM   DAILYSALES DS1
WHERE  sales >(SELECT  MAX(sales)
               FROM    DAILYSALES DS2
               WHERE   DS2.day < DS1.day);
```

A naïve nested iteration plan for the above query employs a *sequential scan* of the DAILYSALES table for both the outer and the inner block. Assuming the inner block scans an average of half of the table for each outer tuple, the cost of this plan would be $t_t(B_{ds} + N_{ds} \times B_{ds}/2) + t_s(1 + N_{ds})$, where B_{ds} is the number of blocks occupied by DAILYSALES table, N_{ds} is the number of tuples in the same table, and t_t and t_s are the block transfer time and seek time respectively.

Now, suppose the DAILYSALES relation (materialized view) is stored, sorted on the *day* column. If the plan for the outer query block generates the bindings for the correlation variable (DAILYSALES.day) in non-decreasing order, we can see that the tuples that qualify for the aggregate (MAX) operator's input in the i^{th} iteration will be a superset of the tuples that qualified in the $(i-1)^{th}$ iteration. The MAX operator, in its state, can retain the maximum value seen so far and use it for computing the maximum value for the next iteration by looking at only the delta tuples. So, the scan needs to return only those additional tuples that qualify the predicate since its previous evaluation.

The maximum cost of this plan would be $2 \times B_{ds} \times t_t + 2 \times t_s$, which is significantly lesser than the cost of the naïve nested iteration plan.

The technique described above is applicable for $<, \leq, >$ and $\geq$ predicates and the aggregate operators MIN, MAX, SUM, AVG and COUNT.

When there are GROUP BY columns specified along with the aggregate, the aggregate operator has to maintain one result for each group. The aggregate operator can maintain its state in a hash table; the key for the hash table being the values for the GROUP BY columns and the value against each key being the aggregate computed so far for the corresponding group.

3.3 Index Scan

Clustered Index Scan: In many practical correlated queries a clustered index is expected to exist for the inner relation on the column that is involved in the correlation predicate (e.g, the TPC-H min cost supplier query in Section 5 and the queries of Examples 1 and 2). Performance of clustered index lookups in the evaluation of correlated nested queries can be greatly improved by producing the outer tuples in sorted order [8]. Sorting ensures sequential I/O and therefore permits prefetching. Also if more than one record from the same data page are needed it is guaranteed that the page is accessed exactly once irrespective of the buffer replacement policy.

Unclustered Index Scan: When the parameter values are sorted, but access to the inner relation is through an unclustered index on columns used in correlation predicates, a *Restartable Unclustered Indexed Scan* can be employed. The Restartable Unclustered Indexed Scan remembers the leaf page at which the previous scan stopped. In this case the benefit is restricted to index access, random I/O is still required for accessing the actual data.

However, this type of indexed scan will be beneficial only when the predicates in the outer block do not create many gaps in the correlation bindings produced.

4 Extensions to a Cost-Based Optimizer

In this section, we describe how a Volcano style cost-based optimizer can be enhanced to take the interesting parameter properties and state retention of physical operators into consideration.

A correlated query block can use one or more parameters whose values are bound in any of its ancestor blocks. Further, more than one query block can be nested under the same parent query block. For a given nested query block, several execution plans are possible, each having its own *required parameter sort order* and cost. Correspondingly, the cost of the outer (parent) query block can vary significantly depending on the sort order it needs to guarantee on the tuples produced. A cost-based optimizer can consider the various *interesting sort orders* and decide on the overall best plan.

In this section, we describe how a Volcano style cost-based optimizer can be extended to consider (a) the sort order of correlation bindings and (b) the new physical operators that exploit the ordering of correlation bindings.

4.1 Overview of the Proposed Extensions

The Volcano optimizer [9] takes an initial query (expression), a set of physical properties (such as sort order) required to be satisfied by the result of the expression and a cost limit (the upper bound) as its inputs and returns a single (best) execution plan for the given query. The following method-signature summarizes the Volcano optimizer's input and output.

Plan **FindBestPlan** (Expr *e*, PhyProp *p*, CostLimit c);

The optimizer makes two implied assumptions:

1. The expression *e* does not contain any unbound parameters.
2. If the expression is evaluated multiple times the cost gets multiplied accordingly.

The first of these assumptions will not be valid for correlated queries and queries with parameter bindings being generated by the calling program. The second assumption does not take into account buffer effects due to sorting and the state retention techniques proposed in the previous section. With these techniques the cost of evaluating an expression *n* times can be significantly lesser than *n* times the cost of evaluating the expression once.

In order to consider these factors we propose a new form of the `FindBestPlan` method. Our description considers parameter sort orders, but alternatives to full sorting, such as batched bindings are considered later in Section 6. The following method-signature summarizes the new form of the `FindBestPlan` method.

Plan **FindBestPlan** (Expr *e*, PhysProp *p*, CostLimit *c*
ParameterSortOrder *s*,int *callCount*);

The new *FindBestPlan* procedure takes two additional parameters. The first of these, termed the *parameter sort order* is a vector $(a_1, a_2, \ldots a_n)$, where $a_i, i = 1, \ldots n$ are the parameters (correlation variables) used inside *e* and bound by an outer query block.

The second parameter, termed *callCount*, tells the number of times the expression is expected to be evaluated. The cost of the returned plan is the estimated cost for *callCount* invocations.

Note that the original Volcano algorithm is a special case of this enhancement with the expression *e* having no unbound references (parameters), *callCount* being 1 and the parameter sort order being empty.

In the remainder of this section we elaborate on the algorithm for the new optimizer method. In Section 4.2 we briefly describe the optimizer framework, and give details in subsequent sections.

4.2 The Optimizer Framework

In this section we briefly describe the cost-based optimizer framework over which we propose extensions to consider the parameter sort orders.

A Volcano-style optimizer performs three main tasks.

1. Logical Plan Space Generation
   In the first step the optimizer, by applying logical

transformations (such as join associativity and pushing down of selections through joins), generates all the semantically equivalent rewritings of the input query.

2. Physical Plan Space Generation
   This step generates several possible execution plans for each rewriting produced in the first step. An execution plan specifies the exact algorithm to be used for evaluating each logical operator in the query. Apart from selecting algorithms for each logical operation this step also considers enforcers that help in producing required physical properties (such as sort order) on the output. The algorithms and enforcers are collectively referred to as physical operators as against the logical operators of the logical plan space.
3. Finding the Best Plan
   Given the cost estimates of different algorithms that implement the logical operations and the enforcers, the cost of each execution plan is estimated. The goal of this step is to find the plan with minimum cost.

An AND-OR graph representation called Logical Query DAG (LQDAG) is used to represent the logical plan space (all the semantically equivalent rewritings of a given query). The LQDAG is a directed acyclic graph whose nodes can be divided into *equivalence nodes* and *operation nodes*; the equivalence nodes have only operation nodes as children and the operation nodes have only equivalence nodes as children. An operation node in the LQDAG corresponds to an algebraic operation, such as join ($\bowtie$), select (σ) etc. It represents the expression defined by the operation and its inputs. An equivalence node in the LQDAG represents the equivalence class of logical expressions (rewritings) that generate the same result set, each expression being defined by a child operation node of the equivalence node and its inputs. An example LQDAG is shown in Figure 4.

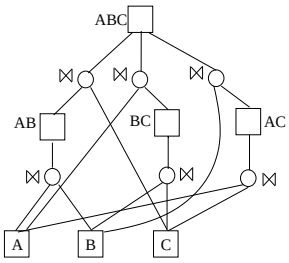


Figure 4: A Logical Query DAG for A $\bowtie$ B $\bowtie$ C

Once all the semantically equivalent rewritings of the query are generated the Volcano optimizer generates the physical plan space by considering different algorithms for each logical operation and considering enforcers that guarantee physical properties such as sort order (the logical and physical plan space generation stages are intermixed in the Cascades optimizer [7] and in the SQL Server optimizer [4] which is based on Cascades). The physical plan space is represented by an AND-OR graph called PQDAG which is a refinement of the LQDAG. Given an equivalence node *e* in the LQDAG, and a physical property *p* required on the result of *e*, there exists an equivalence node in the PQDAG representing the set of physical plans for computing the result of *e* with the physical property *p*. A physical plan in this set is identified by a child operation node of the equivalence node and its input equivalence nodes. The equivalence nodes in a PQDAG are called *physical equivalence nodes* to distinguish them from the *logical equivalence nodes* of the LQDAG. Similarly the operation nodes in a PQDAG are called *physical operation nodes* to distinguish them from the *logical operation nodes* of the LQDAG.

The optimizer framework we used models each of the logical operators, physical operators and transformations as separate classes and this design permits the extensions we propose to be easily incorporated.

4.3 Refinement of the Physical Plan Space

We redefine the equivalence of physical plans to include the parameter sort orders required by the plans. Two plans *p1* and *p2* belong to the same equivalence class *iff p1* and *p2* correspond to the same logical expression, guarantee the same physical properties on their output and require the same sort order on their (input) parameters. Thus, for a given logical expression *e* and physical property *p*, there exists a set of physical equivalence nodes. Each equivalence node in this set corresponds to a distinct *required sort order* on the parameters used in *e*. Note that not all the possible sort orders on the parameters may be of interest. Our algorithm to generate the physical plan space creates physical equivalence nodes for only *interesting* parameter sort orders.

4.4 Physical Plan Space Generation

Given a logical equivalence node *e*, a set of physical properties *p* required on the result of *e* and a sort order *s* known to be guaranteed on the unbound parameters in *e* this step finds all the evaluation plans and represents them as an AND-OR graph (the PQDAG). The search step then takes the PQDAG and a *call count* as its inputs and finds the best plan.

4.4.1 Alternatives for Generating Interesting Orders

We considered three alternatives for generating interesting parameter sort orders and corresponding plans that exploit the sort orders.

Top-Down Exhaustive Generation

Consider a query block *q* that uses parameters $p_1, \ldots p_n$ that are bound external to *q*. In this approach all possible sort orders of the parameters are enumerated exhaustively. For each sort order the best plan for the outer block is produced first and then the inner query block(s) are optimized with the given sort order. This approach leads to a very large plan space as illustrated below:

Let b_i be a query block at level *l* referencing parameters $p_1, p_2, \ldots p_n$ bound in its ancestor blocks. We consider the outermost block to be at level 0. Of the *n* parameters used

Plan for q_1	Plan for q_2	Effective Required Sort Order
pq_1	pq_3	(p_1, p_2)
pq_1	pq_4	(p_1)
pq_2	pq_3	(p_1, p_2)
pq_2	pq_4	*null*

Figure 5: Combining Plans in Bottom-up Approach

inside b_i assume an average of $k = n/l$ parameters are bound at each level 0 to $l-1$. Now, block b_i will be optimized $d(k)^l$ times, where $d(k) = {}^kP_0 + {}^kP_1 + \ldots {}^kP_k$, ${}^kP_i$ denoting k *permute* i. This is a prohibitively large number for query blocks that use large number of correlation variables. Though the possible *valid* sort orders on a set of parameters are many, we expect only a few of these orders to be of interest to the nested query blocks and hence the top-down exhaustive approach produces many redundant plans.

Bottom-Up One Pass Approach

In order to avoid optimization of subexpressions for sort orders not of interest the bottom-up approach first optimizes the inner most query block producing a *set* of plans each corresponding to an interesting order. The bottom-up approach can be understood by the following signature of the Optimizer method.

PlanSet **FindBestPlanSet** (Expr *e*, PhysProp *p*,
CostLimit *c*, int *callCount*);

If there are multiple query blocks $q_1, q_2, \ldots q_n$ nested under the same parent, plans that are compatible with each other in their required parameter sort order are combined. For example, assume query block q_1 has two plans pq_1 and pq_2 requiring sort orders (p_1) and *null* respectively. Further assume query block q_2 nested under the same parent as q_1 has two plans pq_3 and pq_4 requiring sorts (p_1, p_2) and *null* respectively. Now, the compatible combinations of plans and the effective parameter sort order they require from the parent block are as shown in Figure 5.

This approach avoids generation of unwanted sort orders and corresponding plans. In the above example, sort orders such as (p_2) or (p_2, p_1) are never generated.

The drawback of this approach is that it requires significant changes to the structure of any existing Volcano-style optimizer due to the need for propagating multiple plans for the same expression and then combining them suitably.

Top-Down Multi-Pass Approach

In this approach we first traverse all the blocks nested under a given query block and identify the set of all interesting parameter sort orders. For each sort order, we optimize the outer query block and then all the nested blocks. While generating the plans for the nested blocks we consider only those plans that require a parameter sort order no stronger than the one guaranteed by the outer block. This approach combines the benefits of both the top-down exhaustive approach and the bottom-up approach. This approach considers all the interesting sort orders without exhaustive enumeration and requires minimal changes to any Volcano-style optimizer. We describe this approach in subsequent subsections.

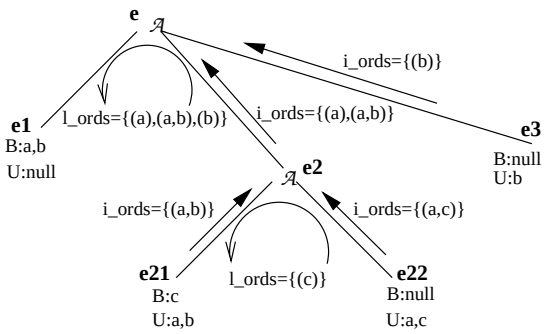


Figure 6: Sort Order Propagation for a Multi-Level Multi-Branch Expression

4.4.2 Generating Interesting Orders

Consider a query block q under which blocks $q_1, q_2, \ldots q_n$ are nested. Let the parameters bound by q be $p_1, p_2, \ldots p_m$. This is represented by an *Apply* expression with q as the *bind* input and $q_1, q_2, \ldots q_n$ as *use* inputs. We first traverse all the *use* inputs of the *Apply* operator and identify the set *i_ords* of interesting parameter sort orders by considering the available physical operators (algorithms) for each logical operator in the subexpression. We consider only those interesting orders that are *valid* under the given nesting structure of *Apply* operators. The necessary and sufficient condition for a sort order to be valid under a given nesting structure is as follows:

Definition 4.1 Valid Parameter Sort Orders
A parameter sort order $(a_1, a_2, \ldots a_n)$ *is* valid iff $level(a_i) \leq level(a_j)$ *for all* i, j $s.t.$ $i < j$, *where* $level(a_i)$ *is the level of the query block in which* a_i *is bound. The level of the outer most query block is considered as 0 and all the query blocks nested under a* level-i *query block have the level* $i+1$.

From the set *i_ords* we then derive a set *l_ords* consisting of sort orders that are relevant to the *bind* input q of the *Apply* operator. Note that *i_ords* can contain some of the parameters bound higher up in the complete query structure. Deriving *l_ords* from *i_ords* involves extracting the suffix of each order *ord* $\in$ *i_ords* such that the suffix contains only those parameters that are bound in q. Figure 6 illustrates how the parameter sort orders are propagated. We indicate the parameters *bound* and parameters *used* at each expression with the convention B: and U: respectively. Consider the *Apply* operator at the root of *e2*. The set *i_ords* of interesting orders for *e22* has a single element (a, c), i.e., *i_ords*=$\{(a, c)\}$. From this set we derive the set *l_ords* as $\{(c)\}$ since c is the only parameter bound by the expression *e21* that is the *bind* input for the *Apply* operator in consideration.

A procedure to generate interesting orders, **GetInterestingOrders**, is shown in Figure 7. The procedure makes use of an auxiliary procedure **GetAncestorOrders** which takes a set of interesting orders and a node, and returns sort orders defined by ancestors of the specified node. Procedure **GetLocalOrders**, shown in Figure 8, takes a set

```
Set<Order> GetInterestingOrders(LogEqNode e,
                    Map plm, int l)
    If the set of interesting orders i_ords for (e,plm)
    is already found
        return i_ords
    Create an empty set result of sort orders for (e,plm)
    For each logical operation node o under e
        For each algorithm a for o
            Let s_a be the sort order of interest
            to a on the unbound parameters in e
            If s_a is a valid order under plm and s_a ∉ result
                Add s_a to result
            For each input logical equivalence node
            e_i of a
                If (o is an Apply operator AND
                    e_i is a use input)
                    newLevelMap = clone(plm)
                    For each variable v bound by o.bindInput
                        newLevelMap.add(v, l)
                    childOrd = GetInterestingOrders(e_i,
                            newLevelMap, l + 1)
                    childOrd = GetAncestorOrders(
                            childOrd, o.bindInput)
                Else
                    childOrd = GetInterestingOrders(e_i,
                            plm, l)
                result = result ∪ childOrd
    return result
end proc

Set<Order> GetAncestorOrders(Set<Order> i_ords,
                        LogEqNode e)
    Initialize a_ords to be an empty set of sort orders
    For each ord ∈ i_ords
        newOrd = Empty vector;
        For (i = 1; i <=length(ord); i = i + 1)
            If ord[i] is NOT bound by e
                append(ord[i], newOrd)
            Else
                break;
        add newOrd to a_ords
    return a_ords
end proc
```

Figure 7: Get Interesting Sort Orders

of interesting orders and a node (the left child of an apply operator) and returns the interesting order suffixes that are defined by the specified node.

4.4.3 Generating Plans at an Apply Operator

For each sort order $o \in$ *l_ords* and empty (*null*) sort order we generate plans for the *bind* input making *o* as the required physical property on the result (output) and then generate plans for all the *use* expressions. We create a physical operation node a for the *Apply* operation depending on the type of the *Apply* node. The type of the *Apply* node can be IN, NOT IN, EXISTS, NOT EXISTS, <*relop*> ANY or <*relop*> ALL depending on the predicate relating the subquery and the parent query block. The plans generated for the *bind* and *use* expressions are added as the child plans for a. Procedure *ProcApplyNode* (Figure 9) shows the algorithm for plan generation at an *Apply* operator.

4.4.4 Generating Plans at a Non-Apply Operator

At each logical operator the original Volcano algorithm considers all the available algorithms (physical operators)

```
Set<Order> GetLocalOrders(Set<Order> i_ords,
                        LogEqNode e)
    Initialize l_ords to be an empty set of sort orders
    For each ord ∈ i_ords
        newOrd = Empty vector;
        For (i =length(ord); i > 0; i = i − 1)
            If ord[i] is bound by e
                prepend(ord[i], newOrd)
            Else
                break;
        add newOrd to l_ords
    return l_ords
end proc
```

Figure 8: Get Local Orders

```
void ProcApplyNode(LogOpNode o, ParamSortOrder s,
                    PhysEqNode n_p, Map plm, int l)
    Initialize i_ords to be an empty set of sort orders
    // Augment the parameter-level map
    newLevelMap = clone(plm)
    For each variable v bound by o.bindInput
        newLevelMap.add(v, l)
    For each use expression u under o
        uOrds = GetInterestingOrders(u,
                    newLevelMap, l + 1)
        i_ords = i_ords ∪ uOrds
    l_ords = GetLocalOrders(i_ords, o.bindInput)
    For each order ord in l_ords and null
        l_eq = PhysDAGGen(o.bindInput, ord, s, plm, l)
        Let newOrd = concat(s, ord)
        applyOp = create new applyPhysOp(o.TYPE)
        applyOp.lchild = l_eq
        For each use expression u of o
            u_eq = PhysDAGGen(u, null, newOrd,
                        newLevelMap, l + 1)
            Add u_eq as a child node of applyOp
        n_p.addChild(applyOp)
end proc
```

Figure 9: Process Apply Node

that implement the logical operation and guarantee the required physical properties on the result. The only change we require to this algorithm is, while considering possible physical operators (algorithms) for a logical operation the parameter sort order guaranteed on the unbound parameters must be taken into account. Only those algorithms that require a parameter sort order *no stronger* than the guaranteed sort order are considered. As an example, consider a *Select* logical operator $\sigma_{R1.a=p_1}(R1)$, where p_1 is a correlation (outer) variable. Assume two algorithms, a plain table scan requiring no sort order and a state retaining scan requiring a sort order $\{p_1\}$ are available. Now, if the parameter sort order guaranteed by the parent block is stronger than or equal to $\{p_1\}$, both the algorithms (physical operators) are possible candidates. However, if the parameter sort order guaranteed by the parent block is weaker (e.g., *null*), then only the plain table scan is a possible candidate.

To reduce the number of candidate plans we can adopt a heuristic of considering only the physical operator(s) that requires the strongest parameter sort order less than the guaranteed sort order. If this heuristic is adopted in the above example, when the parameter sort order guaranteed from the parent block is $\{p_1\}$ only the state retaining scan is considered and the plain table scan is dropped. Proce-

```
void ProcLogOpNode(LogOpNode o, PhysProp p,
                   ParamSortOrder s, PhysEqNode n_p,
                   Map plm, int l)
    For each algorithm a for o that guarantees p and
    requires no stronger sort order than s
        Create an algorithm node o_a under n_p
        For each input i of o_a
            Let o_i be the i^th input of o_a
            Let p_i be the physical property required
            from input i by algorithm a
            Set input i of o_a = PhysDAGGen(o_i, p_i, s,
                                 plm, l)
end proc
```

Figure 10: Process Logical Op Node

dure *ProcLogOpNode* (Figure 10) shows the algorithm for plan generation at a *Non-Apply* logical operator.

4.4.5 Extended Optimization Algorithm

The top level procedure for generating the physical plan space is given in Figure 11, and it calls the procedures described earlier. For simplicity we omit the cost based pruning from our description and return to this issue later. As a result the *callCount* parameter does not appear in the algorithm. We assume that physical operator(s) corresponding to the *Apply* operator do not guarantee any sort orders on the output[3]. Therefore if any sort order needs to be guaranteed on the output of the *Apply* operator an enforcer plan is generated.

In the logical query DAG (LQDAG), due to the sharing of common subexpressions, the mapping of parameters to the level of the query block that binds it cannot be fixed statically for each logical equivalence node. In fact, a single logical equivalence node can get different level numbers because of the *level altering* transformations such as:

$$R \bowtie_{R.c1=S.c2} S \Longleftrightarrow R\, \mathcal{A}^{\oplus}(\sigma_{S.c2=R.c1} S)$$

where $\oplus$ represents concatenation followed by dropping of common columns. In the LHS of the above equivalence rule relation S gets the same level as R, where as, in the RHS S gets a level number higher than R. If a subexpression E is nested below $R \bowtie S$, it sees a different mapping of parameters to levels depending on which of the two forms is chosen for $R \bowtie S$. This is the reason why the the parameter-level-map and the level number are passed as arguments while generating the physical plan space for a given logical equivalence node.

4.5 Search for Best Plan and Cost-Based Pruning

At the end of physical plan space generation we will have a physical query DAG with a single root physical equivalence node. The best plan for the PQDAG is computed recursively by adding the cost of each physical operator to the cost of the best plans for its inputs and retaining the cheapest combination.

[3]Though the *Apply* operator passes through the physical properties of its *bind* input to its output, for simplicity we do not consider it here

```
PhysEqNode PhysDAGGen(LogEQNode e, PhyProp p,
                      ParamSortOrder s, Map plm, int l)
    If a physical equivalence node n_p exists for e, p, s
        return n_p
    Create an equivalence noce n_p for e, p, s
    For each logical operation node o below e
        If(o is an instance of ApplyOp)
            ProcApplyNode(o, s, n_p, plm, l)
        else
            ProcLogOpNode(o, p, s, n_p, plm, l)
    For each enforcer f that generates property p
        Create an enforcer node o_f under n_p
        Set the input of o_f = PhysDAGGen(e, null,
                              s, plm, l)
    return n_p
end proc
```

Figure 11: Physical Plan Space Generation

While computing the cost we take into account the fact that the *use* subexpressions of the *Apply* operator are evaluated as many times as the cardinality[4] of the *bind* subexpression of the *Apply* operator. Each physical operator's cost function is enhanced to take an integer *'n'* as the parameter and return its cost for *'n'* invocations of the operator.

Memoization of the best plan is done against {expression, output physical properties, input parameter sort order, call count} This is required since the best plan may be different for different call counts. Call counts do not affect the physical DAG generation except by pruning parts of the DAG.

Memoization taking call counts into account can potentially increase the cost of optimization. However, if the plan is the same for two different call counts, we can assume that it would be the same for all intermediate call counts. The same plan can then be reused for all calls with an intermediate call count, with no further memoization required. Indeed results from parametric query optimization [11] indicate that the number of different plans can be expected to be quite small. We can thus reduce both the number of plans stored and the number of calls which differ in just the call count.

We apply all simple (non-nested) predicates before the nested predicate is applied. This further reduces the number of distinct call counts with which we optimize an expression. And finally, if as a heuristic, nested calls are made only after all other join predicates have been applied, we will have exactly one call count for the nested expression. Adopting this heuristic, however, may not always be a good strategy [10, 1].

Cost-Based Pruning

We ignored cost-based pruning for simplicity in our earlier presentation and split the physical DAG generation and the search for the best plan into different phases. In our actual implementation, the generation of the physical plan space and search for the best plan take place in a single phase. While generating the physical plan space the cost of each

[4]If the implementation of *Apply* operator is capable of caching results of the *use* subexpression then, the number of distinct correlation bindings will be used in place of cardinality

plan (with parameter sort order and call count) is calculated and the best plan seen so far is memoized. We can then do cost-based pruning in the same fashion as in [9].

4.6 Containment Differential

Section 3.2 showed how nested aggregate queries can benefit from sorted parameter bindings. In Example 5, we showed how a state-retaining aggregate can work in conjunction with a restartable scan to efficiently evaluate an aggregate subquery having a non-equality predicate. The aggregate values computed are incrementally updated in each invocation of the subquery, thus avoiding multiple scans of the relation referenced in the subquery. In general, the input to the aggregate can be the result of a parametrized subexpression.

The state-retaining aggregate operator requires that, logically, its input with the i^{th} parameter value (in the specified parameter sort order) be a superset of the input with the $(i-1)^{th}$ parameter value; further, physically, the input with the ith parameter value should be just the delta *w.r.t.* to the tuples seen till the $i-1^{th}$ parameter value. In the optimizer we model this requirement on the input of the state-retaining aggregate operator as a new physical property termed *containment differential*. A state-retaining aggregate is applicable only when its input satisfies the *containment differential* property for a specified parameter sort order, that is, it returns only the differential tuples when invoked with successive parameter values.

Note that unlike the *sort order* physical property, where a plan that guarantees some sort order can be used in a place where no sort order is expected, a plan that satisfies the *containment differential* property can be used only when the containment-differential property is expected from the parent operator.

5 Experimental Results

To evaluate the benefits of our approach we measured the performance of our evaluation plans and compared them against nested iteration with indexing and decorrelated plans. Note that these plans were manually generated (hard coded). We used PostgreSQL for prototyping the new physical operators and obtaining the actual execution costs. The state retaining table scan and aggregate algorithms were newly coded and the query plans employing these were constructed with hard coding and bypassing the PostgreSQL optimizer. The experiments were run on a single CPU Intel Pentium III 850MHz machine with 256 MB of memory.

5.1 Algorithms Considered

We considered three algorithms: nested iteration(NI), magic decorrelation(MAG) [15] and nested iteration with state retention(NISR). In the case of Nested Iteration (NI) a suitable index was assumed to be present and used. Whenever a relation was assumed to be sorted the NI plan always used a clustered index. The plans employing magic decorrelation were manually composed with the supplementary table materialized. While performing the decorrelation of NOT IN queries we assumed the availability of sort-merge anti-join. When an implementation of anti-join is not available, decorrelation is not applicable and only nested iteration with indexing will be a candidate to compare with the plan generated by our algorithms.

PostgreSQL did not automatically decorrelate any of the queries we considered and it always used a simple nested iteration plan. Hence, the results noted for the Nested Iteration (NI) algorithm also act as the baseline PostgreSQL measures.

5.2 Experiments and Analysis

We used TPC-H 1GB dataset. The tables used for our experiments and the number of tuples in each table are shown in Figure 12. We used four queries for our experiments and the results are summarized in Figures 13 through 16.

Name	orders	lineitem	part
Tuples	1,500,000	600,000	200,000
Name	supplier	partsupp	dailysales
Tuples	10,000	800,000	2,500

Figure 12: Tables used for the Experiments

Query 1 : The first query lists all the orders none of whose line items have the shipping date same as the order's placement date. This query is a small variation of the query shown in Example 4 of Section 3. To be able to apply magic decorrelation for Query 1, we translated the NOT IN predicate to an IN predicate (due to the non-availability of an anti-join implementation). Figure 13 shows the performance results on Query 1. Magic decorrelation performs poorly for Query 1 because there are no outer predicates and no duplicates. This leads to a large redundant join in the plan produced by magic decorrelation. Indexed nested loops join performs significantly better but is still less efficient than nested iteration with state retention. This is due to the overhead of index lookup. This overhead is significant even though most of the index pages above the leaf level are cached in memory. A sort-merge anti-join implementation if present and used would perform exactly same as NISR and hence we have not consider it here explicitly. However, note that a sort-merge anti-join cannot be used if the correlated query block is part of a procedure or function where as NISR can be used in this case.

Query 2 : This query (not part of TPC-H) lists the days on which the sales exceeded the maximum daily sales seen in the past. This query, shown in Example 5 of Section 3, is a nested aggregate query with a non-equality correlation predicate. As Figure 14 shows, nested iteration with state retention completely outperforms magic decorrelation and plain nested iteration for this query. Due to the presence of a non-equality correlation predicate the cost of both magic decorrelation and plain nested iteration increase very rapidly with the increase in the number of outer block tuples. Nested iteration with state retention performs a single scan of the inner and the outer relations as described in Section 3.2.

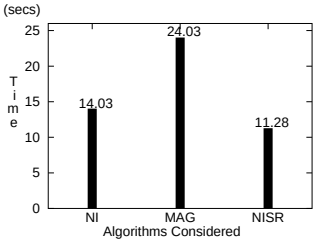


Figure 13: Query 1

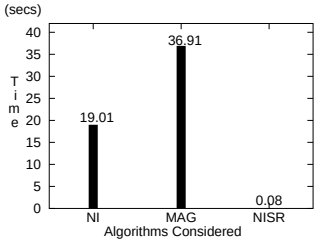
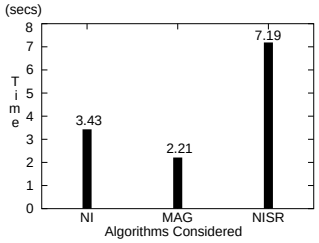
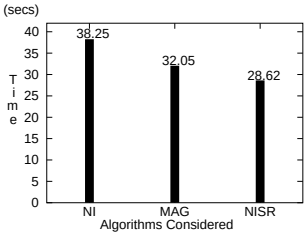


Figure 14: Query 2



(a) With all outer predicates



(b) Dropping 'p_size=15'

Figure 15: Query 3

Query 3 : The third query is a modified version of the TPC-H min cost supplier query shown below.

```
SELECT s_sname, s_acctbal, s_address, s_phone
FROM   PARTS, SUPPLIER, PARTSUPP
WHERE  s_nation='FRANCE' AND p_size=15 AND
       p_type='BRASS' AND p_partkey=ps_partkey
       AND s_suppkey=ps_suppkey AND
       ps_supplycost =
              ( SELECT min(PS1.ps_supplycost)
              FROM     PARTSUPP PS1, SUPPLIER S1
              WHERE    p_partkey=PS1.ps_partkey AND
              S1.s_suppkey=PS1.ps_suppkey AND
              S1.s_nation='FRANCE');
```

Figure 15(a) shows that for Query 3, magic decorrelation performs the best because of the high selectivity of the outer predicates. There were only 108 distinct tuples satisfying the outer predicates. Restart scan performs poorly in this case as the entire relation is scanned where only small fraction of it was required. A cost-based query optimizer would therefore choose the magic decorrelation plan for this query. When we dropped the predicate "p_size=15" NISR performs better as shown in Figure 15(b).

Query 4 : The fourth query is a query with UDF shown in Example 1 of Section 1. For this query, we compare only NI with NISR since decorrelation techniques are not directly applicable. Figure 16 shows that NISR performs significantly better than NI; this is because the inner (lineitem) relation is scanned at most 2 times with NISR, whereas NI performs an indexed lookup of the inner relation for each tuple in the outer relation.

6 Extensions

This section describes some of the possible extensions that can be implemented over the proposed techniques.

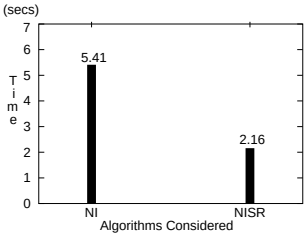


Figure 16: Query 4

Sorting in Batches: Our optimization algorithm can be easily extended to the case where the parameters (correlation bindings) are not completely sorted but are sorted in smaller batches. Sorting smaller batches avoids writing intermediate runs to the disk [8]. We can handle batched binding by creating a new physical operator *BatchedApply*. Since correlation bindings are sorted in batches, the optimizer uses the batch size as the *callCount* for the inner subquery and multiplies the cost of the inner subquery by the number of batches expected. The BatchedApply operator can sort its input bindings, and the sort order can be used when optimizing the inner subquery.

Caching of Results: Caching of results can be handled by creating a physical operator *CachedApply* which caches the results of earlier calls. The call count for the inner subquery is then the number of distinct parameter values. The amount of space for caching is determined by the sort order of the parameters, with unsorted parameters requiring all results to be cached.

Set Valued Functions in the SELECT Clause: Set valued functions and nested queries in the SELECT clause are very useful for efficiently publishing relational data as XML. Our optimization algorithm can be used unchanged for such functions and queries. However, to support such set valued functions/queries in the SELECT clause the *Apply* operator needs to be extended in the following way. For each tuple in the outer block (*bind* expression) the *Apply* operator should *collect* the results of the inner block (*use* expression) and output a nested-relational result or tagged XML.

7 Related Work

Nested queries have been studied quite extensively; however, most of the emphasis so far has been on decorrelation techniques [12, 6, 13, 2, 15]. Decorrelation techniques try to rewrite a given nested query into a form that does not use the nested subquery construct. Decorrelation techniques allow an optimizer to consider alternative set oriented plans such as merge join or hash join for evaluating a nested query and in most cases these methods perform better than the naïve nested iteration method. The techniques we proposed to speed up nested iteration are orthogonal to decorrelation and a cost-based optimizer should consider both decorrelated evaluation as well as the improved nested iteration methods while choosing the best plan. In fact, in this paper we show the cases where the improved nested iteration methods can perform significantly better than the plans generated by decorrelation techniques proposed.

Techniques for improving the performance of nested iteration have been proposed by Selinger et.al. [14] and Graefe [8]. In the System R paper [14] Selinger et.al. propose the idea of caching the inner subquery result for distinct values of correlation variables and sorting the outer tuples which allows caching of only one result of the inner query at any point in time. Graefe [8] emphasizes the importance of nested iteration plans and discusses asynchronous I/O, caching and sorting outer tuples as techniques that can improve the performance of nested iteration.

The techniques we propose in this paper for improving the nested iteration method augment the techniques proposed in [14] and [8]. Sorting in System R is purely to ensure the cached result can be kept in memory (only one cached result need be retained). Graefe [8] describes sorting of outer tuples to produce advantageous buffer effects in the inner query plan, where as we propose different evaluation strategies possible with the knowledge of the order of correlation bindings. Both [14] and [8] do not discuss the changes required in the optimizer to consider these options and generate an overall best plan. Database systems such as Microsoft SQL Server consider sorted correlation bindings and the expected number of times a query block is evaluated with the aim of efficiently caching the inner query results when duplicates are present and to appropriately estimate the cost of nested query blocks. To the best of our knowledge, the state-retention techniques and optimization of multi-branch, multi-level correlated queries considering *parameter sort orders* have not been proposed or implemented earlier.

The Microsoft SQL Server query optimizer (which is also based on Volcano), has an extension to the findBest procedure (their equivalent to our FindBestPlan procedure), which passes it "context" information [4]. The information in the context includes the parameters, their sort order, number of rows required, and number of expected executions. The parameter sort order and number of expected executions information is used to cost plans appropriately, taking cache effects in index lookups into account, and to decide what results to materialize. The number of rows required is orthogonal to nested iteration, and is used for top-K queries and for EXISTS subqueries. The same subtree may be optimized under different contexts; however, further details are not available to us. As far as we are aware they cannot infer and use interesting sort orders for parameters, and they do not handle state retention.

8 Conclusions and Future Work

We revisited nested iteration plans for correlated queries and showed how the sort order of correlation bindings can be exploited to produce plans that can be significantly faster than the corresponding set oriented plans. Our techniques, being based on the tuple iteration semantics, are more general in terms of their applicability compared to the decorrelation techniques. We showed how the proposed techniques can be extended to efficiently evaluate queries involving expensive user-defined functions. We addressed the issues involved in extending a Volcano-style cost-based optimizer to take into account the proposed evaluation techniques. We presented a performance study based on an actual implementation of our execution techniques, which demonstrates significant benefits. We believe our extensions can be added to existing execution engines and optimizers in a straightforward manner, and are thus of practical use for existing database systems.

Having shown that our proposed execution techniques can give significant execution cost benefits, we are now in the process of implementing our optimizer extensions. Future work includes studying the performance of our optimization algorithm. We expect the overheads of our algorithm to be quite small, since it considers only interesting sort orders, and the cost of finding interesting sort orders is itself quite low, linear in the size of PQDAG.

References

[1] S. Chaudhuri and K. Shim. Optimization of Queries with User-defined Predicates. In *VLDB*, 1996.

[2] U. Dayal. Of Nests and Trees: A Unified approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. In *VLDB*, 1987.

[3] A. Eisenberg and J. Melton. Advancements in SQL/XML. In *SIGMOD Record 33(3)*, 2004.

[4] C. A. Galindo-Legaria and C. Fraser, Nov. 2004. Personal Communication.

[5] C. A. Galindo-Legaria and M. M. Joshi. Orthogonal Optimization of Subqueries and Aggregation. In *ACM SIGMOD*, 2001.

[6] R. A. Ganski and H. K. T. Wong. Optimization of Nested SQL Queries Revisited. In *ACM SIGMOD*, 1987.

[7] G. Graefe. The Cascades Framework for Query Optimization. In *Data Engineering Bulletin 18 (3)*, 1995.

[8] G. Graefe. Executing Nested Queries. In *10th Conference on Database Systems for Business, Technology and the Web*, 2003.

[9] G. Graefe and W. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*, 1993.

[10] J. M. Hellerstein. Practical Predicate Placement. In *ACM SIGMOD*, 1994.

[11] A. Hulgeri and S. Sudarshan. AniPQO: Almost non-intrusive parametric query optimization for non-linear cost functions. In *VLDB*, 2003.

[12] W. Kim. On Optimizing an SQL-like Nested Query. In *ACM Transactions on Database Systems, Vol 7, No.3*, 1982.

[13] M. Muralikrishna. Optimization and Dataflow Algorithms for Nested Queries. In *VLDB*, 1989.

[14] P. G. Selinger, M.M.Astrahan, D.D.Chamberlin, R.A.Lorie, and T.G.Price. Access Path Sselection in a Relational Database Management System. In *ACM SIGMOD*, 1979.

[15] P. Seshadri, H. Pirahesh, and T. C. Leung. Complex Query Decorrelation. In *ICDE*, 1996.

[16] J. Shanmugasundaram et al. Efficiently Publishing Relational Data as XML Documents. In *VLDB*, 2000.