Reusing Invariants: A New Strategy for Correlated Queries

Jun Rao*
Department of Computer Science
Columbia University
junr@cs.columbia.edu

Kenneth A. Ross†
Department of Computer Science
Columbia University
kar@cs.columbia.edu

Abstract

Correlated queries are very common and important in decision support systems. Traditional nested iteration evaluation methods for such queries can be very time consuming. When they apply, query rewriting techniques have been shown to be much more efficient. But query rewriting is not always possible. When query rewriting does not apply, can we do something better than the traditional nested iteration methods? In this paper, we propose a new invariant technique to evaluate correlated queries efficiently. The basic idea is to recognize the part of the subquery that is not related to the outer references and cache the result of that part after its first execution. Later, we can reuse the result and combine it with the result of the rest of the subquery that is changing for each iteration. Our technique applies to arbitrary correlated subqueries.

This paper introduces algorithms to recognize the invariant part of a data flow tree, and to restructure the evaluation plan to reuse the stored intermediate result. We also propose an efficient method to teach an existing join optimizer to understand the invariant feature and thus allow it to be able to generate better join plans in the new context. Some other related optimization techniques are also discussed. The proposed techniques were implemented within three months on an existing real commercial database system.

We also experimentally evaluate our proposed technique. Our evaluation indicates that, when query rewriting is not possible, the invariant technique is significantly better than the traditional nested iteration method. Even when query rewriting applies, the invariant technique is sometimes better than the query rewriting technique. Our conclusion is that the invariant technique should be considered as one of the alternatives in evaluating correlated queries since it fills the gap left by rewriting techniques.

1 Introduction

Correlated queries are those queries having subqueries which use references from outer query blocks. We will refer to these references as *outer references* in this paper. Correlated queries are very important because: a) decision support systems tend to use them heavily to express complex requests, and b) correlated queries are often automatically generated by many application generators that translate queries from their native languages into standard SQL. In the TPC-D decision support benchmark [TPC95], there are three correlated queries among the seventeen queries.

Traditionally, the processing of correlated queries is usually done in a nested iteration fashion. All the rows from the outer query block are fetched one by one. For each of the rows fetched, the outer references in the subquery are bound to their current values. The subquery is then evaluated and the result is returned. After that, those predicates containing correlated subqueries (we'll refer to them as *nested predicates* in this paper) will be evaluated. The whole process is repeated until we have exhausted all the rows from the outer query block. Since the whole subquery has to be executed from scratch multiple times, this strategy can be very time consuming.

To overcome this problem, query decorrelation has been studied and proposed as a better solution. The basic idea is to rewrite a correlated query in such a way that outer references no longer exist. In [SPL96], the authors proposed a technique that will extract all the distinct values of outer references and materialize all the possible results from the subquery. Later, the materialized results are joined with the outer query block on the outer reference values. Although the rewritten query usually introduces extra views, joins and possibly group-bys, we can still expect to get much better performance since now the subquery only needs to be executed once. This is especially true when the number of iterations of the subquery is large.

One problem with the rewriting strategy is that query decorrelation is not always possible and in some cases, although possible, may not be efficient. As pointed

*The implementation of the techniques in this paper was performed while the author was working at Sybase IQ, Burlington, MA.

†Research at Columbia University was supported by a David and Lucile Packard Foundation Fellowship in Science and Engineering, by a Sloan Foundation Fellowship, by grants CDA-96-25374, and by an NSF Young Investigator award. Part of the work was performed while the author was visiting Sybase IQ, Burlington, MA.

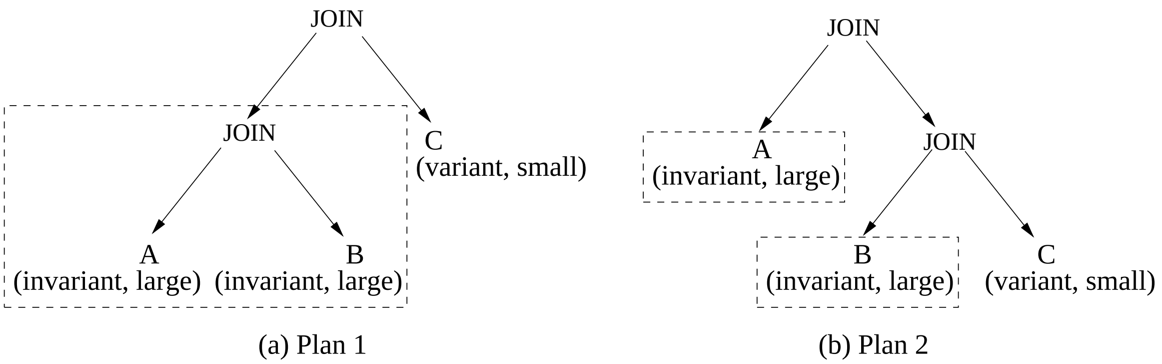


Figure 1: Plans for a correlated subquery

out in [PHH92], "queries that contain non-existential or non-Boolean factor subqueries, set operators, aggregates, or user-defined extension operators (e.g., OUTER JOIN)" do not get rewritten. Although it is shown in later work ([SPL96]) that some of the queries including aggregates can be decorrelated, there are still many queries which we don't want to rewrite for either semantic or implementation reasons. For example, correlated queries with nested predicate using a non-inclusion operator (`NOT IN`) are almost impossible to rewrite unless new types of operators are introduced as specified in [Kim82]. Another example is that an `OUTER JOIN` may be introduced in some query rewriting transformations. But this may not be desirable for systems that don't support `OUTER JOIN` directly. We enumerate several types of queries that have no existing ways (without introducing new operators) of being rewritten in Appendix A. Based on an estimation from the Sybase SQL Anywhere group [Pau97], around 40% of real world correlated queries can not be rewritten. The queries for which query rewriting fails usually involve very complex subqueries, which can be very time-consuming if executed in the traditional nested iteration fashion. The question we then want to ask is "Can we do better when rewriting techniques can't be applied?"

We can observe that although a correlated subquery can get quite complex, usually the outer references are bound to just one or two tables in the subquery. It's the correlated part of the subquery that may change its value and thus needs to executed multiple times. The remaining part, which is in fact uncorrelated, is not affected by the changing of values of the outer references. This motivates us to cache the results of the uncorrelated part after it's been executed for the first time. The cached result can be reused in subsequent executions and combined with the new results from the correlated part (which has to be regenerated). We will call the uncorrelated part *invariant* and the correlated part *variant*. Let's take a look at a possible correlated subquery plan shown in Figure 1(a). The invariant part is enclosed in the dashed box. The join between table `A` and table `B` only needs to be executed once and the join between the cached intermediate result and table `C` can be performed in subsequent executions of the subquery.

A join optimizer that does not consider the invariant feature may actually choose a different join plan (Figure 1(b)) for the subquery since table `C` is small. But here we have two smaller invariant parts instead of one large part. That means we have to perform two joins for each outer query iteration. If the number of iterations is large, the overall cost for plan 2 may be more expensive than that of plan 1. Is there an easy way to let the join optimizer choose an overall optimal plan taking invariants into consideration? This question will be addressed later in the paper.

We will refer to the technique of caching the invariant result and later reusing it as the *invariant technique*. By reducing the amount of work that has to be done for each iteration, we expect the invariant technique to give better performance for those queries that can not be rewritten. For those queries where both rewriting and invariant techniques are applicable, we have done some experimental and analytical comparisons between the two. The conclusion we have reached is that the rewriting technique is not always going to win.

Our technique also works when the outer reference takes the form of host variables (variables coming from the client program), or bind variables (variables generated internally) [Bel96]. Without loss of generality, we will talk about only outer references in this paper. We'd also like to point out that our technique differs from the traditional *memoization* method [Mic68], which only helps when there are duplicates in the outer references. Our invariant technique is useful whether there are duplicates or not, although we can get additional benefit from memoization when there are duplicates in the outer references.

We will base our discussion on the Volcano style query execution engine [Gra94], in which each query plan is a tree of operators. All operators are implemented as iterators and support a simple open-next-close interface, where open() does most of the preparation work, next() fetches the rows one by one and close() cleans up everything. Most relational database systems have analogous operator-based execution engines. We'll call this type of query engine a data flow engine and the execution plan a data flow tree. We implemented the invariant optimization technique on Sybase IQ's [Syb97b] data flow engine and join optimizer. It took only 3 man-months to have our technique seamlessly incorporated into the existing system. The technique will be present in an upcoming commercial version of Sybase IQ.

The rest of the paper is organized as follows: In Section 2, we describe the algorithm of marking each node in the data flow tree as variant or invariant. In Section 3, we introduce the method of caching and reusing the invariant results. In Section 4, we describe the techniques of incorporating the invariant information into an existing join optimizer. We also propose some post optimization techniques here. We summarize in

Section 5 how everything can be put together. In Section 6, we show our experimental results and analytical analysis comparing our invariant technique and the rewriting technique. We also point out that the amount of available RAM plays an important role in correlated query evaluation. We describe related work in Section 7 and conclude in Section 8.

2 Finding the Invariant Subplan

The Volcano execution engine didn't cover the implementation of a subquery. Here we choose Sybase IQ's approach [Lea96].[1] We assume that a subquery will be implemented through a subquery node (operator) in a data flow tree. The left child of the subquery node corresponds to the outer query block while the right child corresponds to the inner query block, i.e., the subquery. If the corresponding subquery is correlated, an outer reference vector will be generated in the subquery node. The nested predicate is also put in the subquery node for later evaluation. An example of a correlated query and its corresponding data flow tree is shown in Example 2.1 and Figure 2 respectively. Predicates are pushed down in the tree as deep as possible (listed in parentheses). The outer reference vector is represented in italic font. When evaluating a subquery node, next() will first be called on its left child to get one row. Then the values of the outer references are bound to all their occurrences in the right subtree and next() will be called on the right child to retrieve the result from the subquery. The whole process is repeated for all the iterations. For nested queries of level two or more, there will be more than one subquery node in the data flow tree. Outer references are usually referred to in a predicate or the output vector in a particular data flow node. We also assume that we have typical data flow nodes such as table scan node, join node, filter node and group-by node.

Example 2.1:

```
select  o_orderpriority,
        count(*) as order_count
from orders
where o_orderdate >= "1993-07-01"
    and o_orderdate <  "1993-10-01"
    and exists (select *
                  from lineitem
                  where l_orderkey  = o_orderkey
                    and l_commitdate < l_receiptdate)
group by o_orderpriority
```

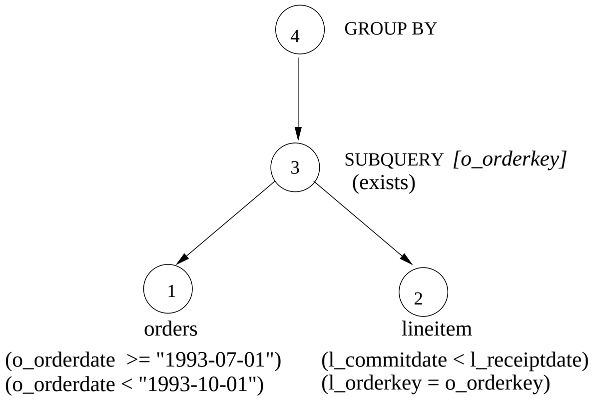


Figure 2: Example 2.1

We now give a formal definition of an invariant subtree.

Definition 2.1: An invariant subtree in a data flow tree is a subtree T where none of the nodes in T contains any outer references that are generated outside of T.

As a general rule, a node in a data flow tree will be marked as invariant if all of its children are invariant and no outer references are referred to in the node itself. But we need to be careful: notice that outer references are scoped in a data flow tree, i.e., they will be generated at a particular subquery node and are meaningful only to the right subtree. A subtree can be invariant even though there is an outer reference being used within the subtree, as long as the outer reference does not refer outside the subtree. So, we should keep track of the outer references being used in the subtree rooted at each subquery node. If they are all generated at the root, i.e., a particular subquery node, then the subquery node should be marked as invariant since the subtree doesn't use any reference generated outside.

We implemented an algorithm that checks and marks a data flow tree. The algorithm walks through the data flow tree in preorder. We pass along an outer reference vector during the traversal. Every time a subquery node is reached, we add the newly generated outer references to the vector before traversing the right subtree. Each node (except the subquery node) checks if any outer references in the vector is being used in the node locally or in one of its children. If so, the node will be marked as variant. Otherwise, the node is invariant. For the subquery node, we do a little bit more. We also check whether all the occurrences of the outer references in its right subtree are generated in the subquery node itself. If such is the case, we mark the subquery node as an invariant. In the same traversal, we also mark each node according to whether it's in a correlated subquery or not. This information is useful because a data flow node will be executed more than once only if it's in a correlated subquery.

In cases where we have a multi-level correlated subquery, it might be the case that a node in a data flow tree uses only outer references generated in the query block two or more levels up (e.g., division-style queries). The node will be marked as variant by the above algorithm. But in fact, for each inner subquery iteration, the result of that node remains the same. The result only changes when the outer references change in the outer iteration. What we can do is to store, for each outer reference in a subquery node, a list of nodes using that outer reference in a data flow tree. Then we can mark each node as invariant or not dynamically at preparation time during the execution. But this would complicate the implementation and it's not clear how many real world queries can benefit from it. For these reasons, we haven't implemented the refinements needed for multi-level correlated queries.

3 Reusing the Invariant Result

After we know about whether a data flow node is invariant or not and also whether it's in a correlated subquery or not, we can try to cache the invariant result and reuse

[1] However, the idea presented in this paper could be adapted to other implementations.

it in subsequent executions. The first thing to notice is that there may already exist some forms of storage in various types of data flow nodes. For example, in a hash join node, we may have a hash table associated with it, which will typically be used to hash the smaller operand (the rows from the larger operand will be used to do the probing). Another example will be a group-by node, where either a hash table or a sorting storage element may exist depending on the implementation. Normally, the existing storage element will be reset (the contents cleaned) during each open() call. But they can be tuned to be capable of rewinding[2] their contents. Each storage element has to be notified whether rewinding is indeed necessary at preparation time based on the invariant feature. The reason why we want to distinguish a rewindable storage element from an un-rewindable one is that the implementation without the need for rewinding may have some benefit, such as being able to release some of the resource (memory) allocated after partial retrieval. So we only want to rewind an existing storage element when it's beneficial. We assume we can achieve this by calling a SetExpectRewind() function on the storage element. If there is no existing storage element associated with an invariant data flow node such as a filter node, we may want to add a new one ourselves. There are various ways of doing this. In our implementation, we simply designed a new type of data flow node, namely a storage node, which also supports the open-next-close interface. Its only functionality is to retrieve all the rows from its (only) child and save the result internally. Later, all the rows can be retrieved from the storage node without having to execute the subtree below it. However, the adding of a storage node should be done on a cost basis. In cases where we have a Cartesian product or a many to many join in the subtree, the cost of retrieving the result may be higher than reexecuting the subtree. If so, we'd rather not cache the result.

To reuse the existing storage element, there are two main places that we need to modify. The first is in the open() function of all the data flow nodes with existing storage elements. We want to set these storage elements correctly as to whether they are expected to be rewound. The second place is in the next() function. If there is an existing storage element and it's reusable, we want to get the next row from the storage element directly without calling next() on its children. We need to adapt the original functions to meet this requirement. Algorithm 3.1 and Algorithm 3.2 describe the changes in open() and next().

Algorithm 3.1: Adjusted open() function (Figure 3).

Algorithm 3.2: Adjusted next() function.

```
int next(df_node nd) {
  if (there is a storage element in nd called store
      and store has not been loaded yet) {
    Fill in store by fetching tuples from the corresponding child.
    Set store as loaded.
  }
  Continue the remaining execution by retrieving rows
  from store.
}
```

[2]By rewinding, we mean that the content of the storage can be retrieved from the very beginning again.

```
void open(df_node nd) {
  if (this is the first time) {
    if (there is a storage element in nd called store) {
      if (nd is in a correlated subquery and is an invariant)
        Call SetExpectRewind(store, TRUE).
      else
        Call SetExpectRewind(store, FALSE).
      Set store as unloaded.
    }
  }
  else { // subsequent calls
    if (there is a storage element in nd called store)
      if (nd is in a correlated subquery and is an invariant) {
        if (store has been loaded)
          Rewind store.
      }
      else {
        Reset store.
        Set store as unloaded.
      }
  }
  Prepare as before.
}
```

Figure 3: Algorithm 3.1

The algorithm for adding a storage node in the data flow tree is presented in Algorithm 3.3. The idea is to traverse through the data flow tree from the root. As soon as we reach a node marked as invariant and in a subquery, we check whether there is an existing storage element associated with it. If so, we are done (actually almost done). Otherwise, we may insert a storage node above it if that's more cost-effective than reexecuting the node. Since we are traversing the tree top-down, we will only insert a storage node above a maximal invariant subtree (i.e., no storage node will be added above any of its invariant children).

Algorithm 3.3: Algorithm for adding a storage node in the data flow tree.

```
void AddStorageNode(df_node nd) {
  if (nd is not in a correlated subquery or nd is a variant)
    for each child_i of nd
      Call AddStorageNode(child_i).
  else { // now nd is in a subquery and is an invariant
    if (there is an existing storage element with nd)
      Call MarkSubtree(nd).
    else
      if ((ECost(nd) + ICost(nd) + (n - 1) * RCost(nd))/n
          < ECost(nd)) {
        Insert a storage node above nd.
        Call MarkSubtree(nd).
      }
  }
}
```

In Algorithm 3.3, $RCost$ and $ECost$ represent retrieving cost and execution cost respectively. $RCost$ could be zero when a storage element can fit into memory. $ICost$ is the cost of inserting all the rows to the new storage element and n is the number of iterations for the subquery. (See Section 4 for a discussion of the cost formula.) After we insert a new storage node or detect that there is an existing storage element in a node, we call a function MarkSubtree(). This is because the subtree below the node will be executed only once. The storage associated with those nodes in the subtree should be treated as not expecting rewinding. MarkSubtree() addresses the issue by traversing the subtree and marking all the data flow nodes. Those nodes can simply be marked as not in a subquery. Alternatively, one can use another identifier and check it in open() and next() when deciding whether a storage element should expect rewinding or not.

4 Adapting an Existing Join Optimizer

By the time the data flow tree has been generated after various optimization techniques have been applied, the execution plan is fixed. Since the data flow tree is generated by the original join optimizer which doesn't take invariants into consideration, it's very likely that we won't get the maximal invariant subplan possible. As we have seen in Figure 1, plan 2, which has a smaller invariant part, may actually be chosen. This plan will be the most efficient if executed only once, but is suboptimal if executed multiple times, since we can cache the result of part of the plan. So we need to find a way to teach the join optimizer to take the invariant feature into consideration when choosing the join orders. There are two goals here. The first is to find the optimal plan. The second is to make the adaptation of the original optimizer convenient.

On first thought, one might want to divide all the participating tables into two parts, one invariant and one variant. And then use the original join optimizer to find the optimal join orders on both parts and combine them by introducing another join between the two parts. It seems that in this way we can get the largest invariant subplan possible. But actually this strategy has two pitfalls. Firstly, it's not guaranteed to find the optimal join plan. This is because, in cases where there is no join predicate linking some tables within each part (but there are join predicates linking those tables across the two parts), we may introduce a Cartesian product on both the invariant part and the variant part. Secondly, this requires non-trivial changes to the original join optimizer, which is the last thing we want to do.

Given that queries are becoming more and more complex, especially in decision support systems, most join optimizers in commercial database systems use cost-based searching techniques to try to find the optimal join orders.[3] Our new strategy is to adjust the cost estimation in the join optimizer. Observe that the cost of an invariant subplan is, for the first execution, the cost of executing the subplan and storing it (if a new storage node has to be inserted), and for subsequent executions, the cost of retrieving all the rows from the cache. So if we know that an invariant subplan is in a correlated subquery, we can (and should) use the average cost of the subplan, i.e., $ACost = (ECost + ICost + (n-1) * RCost)/n$, where n is the number of times the subplan will be executed and $ACost, ECost, ICost, RCost$ represent Average Cost, Execution Cost, Cost of insertion into the new storage element and Retrieving Cost respectively.[4] $ECost$, $ICost$ and $RCost$ can easily be estimated using conventional techniques. We describe how n can be estimated in Section 5.

Notice that we want to consider the maximal invariant subplan. If an invariant subplan is part of another larger invariant plan, since the smaller subplan will only be executed once (it is the result of the larger invariant plan that will be cached), we still want to use its execution cost as the cost of the subplan. This can be achieved by using the average cost for an invariant subplan only when it's going to be merged into a larger variant subplan.[5]

What's nice about our new strategy is that it solves both problems we met before. It's optimal in some cases (as we will discuss below) and only minor changes are needed to incorporate it into an existing join optimizer. Algorithm 4.1 shows how to adjust the join cost when building up a join tree in a join optimizer.

Algorithm 4.1: Estimate Join Cost

Input: Two subplans subplan1 and subplan2, their associated costs cost1 and cost2, their cardinalities size1 and size2 and the number of iterations of the subquery n.
Output: The combined plan, its cost and cardinality.
Method:

```
Decide the optimal join algorithm to combine the two parts.
Estimate ICost1 and ICost2 for each part based on the join
algorithm chosen.
if (the combined plan is variant) {
  if (subplan1 is invariant)
    Let cost1 be the smaller of cost1 and
    (cost1 + ICost1 + (n-1)*RCost(size1))/n.
  if (subplan2 is invariant)
    Let cost2 be the smaller of cost2 and
    (cost2 + ICost2 + (n-1)*RCost(size2))/n.
}
else
  cost1, cost2 remain unchanged.

Use the new values of cost1 and cost2 to estimate the cost
of the combined plan.
Output the cardinality as before.
```

∎

Pruning is heavily used in commercial join optimizers to narrow the search space. But we need to be more careful about pruning in the adjusted join optimizer. For example, a simple pruning rule can be "if the cost of a subplan is greater than the best cost we have so far, that subplan is ignored." If that subplan happens to be invariant, since it hasn't been merged with a variant part, its cost is still the execution cost. If we use this cost to do the pruning, we may miss a better plan because when the invariant subplan is eventually being merged into a variant plan, the average cost (if smaller) could be used. One way to solve the problem is to always keep invariant subplans, i.e., without pruning at all. Alternatively, we can use the smaller of the average cost and the execution cost to do the pruning, a technique we call *conservative* pruning. This means that we may do less pruning than before and thus increase the optimization time. But this overhead only occurs in correlated subquery optimization. For outer query blocks and uncorrelated subqueries, the optimization cost remains the same. Given the fact that we can do significantly better (as shown in the experiments later) by using the invariant technique, the extra optimization time is unlikely to be a problem.[6]

Observation 4.1: For correlated queries involving joins in the subquery, if the original join optimizer searches

[3] We assume the join optimizer will be invoked on each query block separately.

[4] Ideally, the estimation of those costs should take into consideration the amount of memory available.

[5] If $(ECost + ICost - RCost)/n \ll RCost$, $ACost$ can be reduced to $RCost$.

[6] In our experiments, we can hardly notice the increase in optimization time, which is dominated by the execution time of the query.

all possible join orders with the conservative pruning rule illustrated above and we have exact estimation of n (the number of iterations), then the adjusted join optimizer will give the optimal join order with respect to our revised cost metric.

Reason: We can multiply the optimal join cost by n. The optimal cost can then be divided into two parts, the term with a factor n and the term without. The term with a factor n corresponds to the cost of variant part of the join plan and the term without n corresponds to the cost of invariant part. Notice how we adjusted the pruning technique so that an invariant subplan will still be considered unless both its execution cost and retrieving cost are very high. So we will consider all possible combinations of join orders that may be the best plan. Further, the total cost estimation of each plan is exactly the cost when the plan will be executed using our invariant technique. So we are guaranteed that the final join plan is the best plan we can choose. ∎

There is another possible optimization technique that is orthogonal to the invariant optimization. We can avoid invoking the subquery on duplicate outer reference values by using the subquery memoization technique [Mic68], which has already been implemented in several commercial systems [Syb97a, Syb97b]. There are two possible ways to do that. The first is to build a main-memory hash table on the subquery node. Every time we evaluate a subquery, we first probe the hash table based on the values of the outer references for that iteration. If the outer reference don't exist, we insert the values of the outer references and the corresponding subquery result into the hash table. If the probe succeeds, we can reuse the cached result instead of reexecuting the subquery. Recently, a new *hybrid cache* method has been suggested in [HN96] to avoid hash table thrashing. The second approach, described in [SAC+79], is to presort all the rows from the outer query block on the outer references before executing the subquery. Then in the subquery node, we simply cache the result of the last execution of the subquery. Since outer references are bound in sorted order, the cached result can be used whenever the outer reference values are repeated. Both methods have some overhead. The first one needs to build a hash table and the second one needs to presort some rows. But if the number of duplicated values is large enough, this overhead will be compensated for by the saving in the number of reexecutions of the subquery. In practice, we chose the second approach. The presorting was implemented by adding above the left subtree of a subquery node an order-by node, whose only functionality is to retrieve all the rows from its child and sort them on designated keys (outer references). We use a heuristic rule to determine when to add an order-by node. The rule simply calculates the ratio of the number of rows from the outer query and the number of possible distinct values of the outer references. If the ratio exceeds a threshold, we will add the order-by node. The join optimizer should also take the above technique into consideration. In cases when we do want to presort the outer references, the average cost should now be $min((ECost + ICost + (n' - 1) * RCost)/n', ECost)$, where n' is number of *distinct* values in the outer references.

Other post optimization techniques related to invariant subplans still exist. One of them is that when doing a hash join, we find that it's more efficient to build the hash table on the invariant operand even though its size may be larger than the variant operand (assuming the invariant operand can still fit into memory). The reason is that now the hash table only needs to be built once.

5 Query Processing using the Invariant Technique

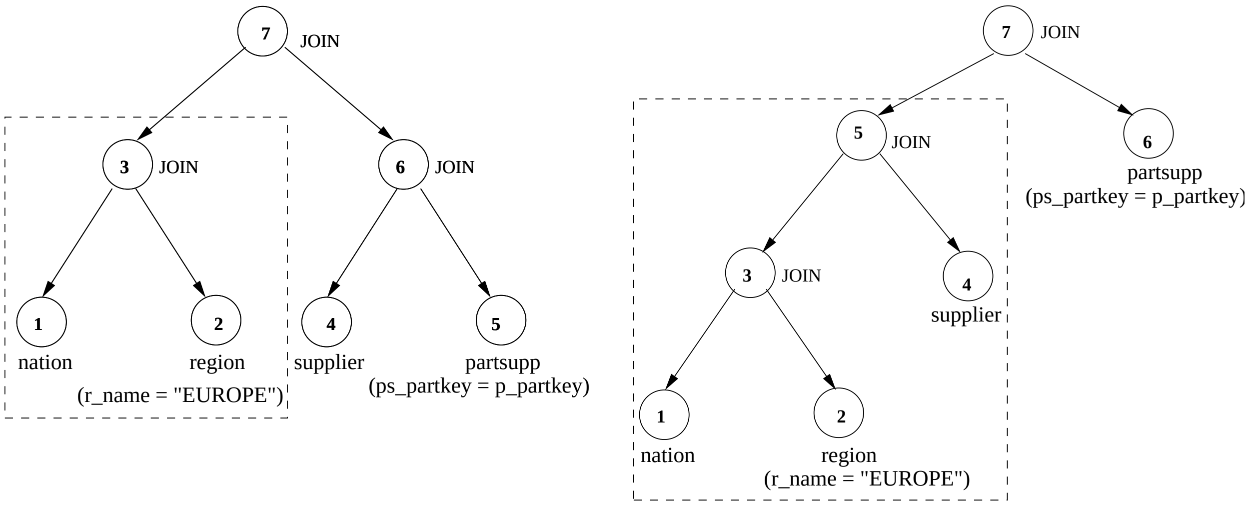
In this section, we describe how everything can be put together. After a query has been parsed, it will be sent to a query optimizer which consists of our adjusted join optimizer. The query optimizer will generate the data flow tree for the query. The plan for correlated subqueries will be optimized with respect to the invariant feature using the technique described in Section 4. Then we call AddStorageNode() described in Section 3 on the root of the data flow tree to add necessary new storage elements. After that, we mark all the nodes in the data flow tree as variants or invariants using the algorithm introduced in Section 2. Finally the data flow tree will be evaluated by calling the modified open() and next() on each data flow node. If a query is non-correlated, it will be handled as usual. So, by using the algorithms described in previous sections, an existing query optimizer and query execution engine can be adapted to incorporate the invariant technique without significant changes.

Usually the outer query block is optimized after the inner query block. Thus an important issue is how to get n, the number of iterations of a subquery before it's needed. In fact, what we need is just the number of distinct values of the outer reference since we use outer reference presorting and cache the previous result when there are duplicates. Sybase IQ stores tables vertically and each column can be treated as an index. Local selection predicates in the outer query block are pushed down into base tables before the subquery is optimized. Thus we can get the exact number of distinct values for an outer reference after filtering. This happens before we optimize the join order in the outer query block. Joins in the outer query block can only increase the derivations of the outer reference, but not the number of distinct values. When there is more than one outer reference in the same subquery, we may not have a very accurate estimation of the distinct count. But we can still get an upper bound in this situation.

6 Experiments and Performance Analysis

To verify the efficiency of the invariant technique, we have done experiments on different kinds of queries. In this section, we compare the performance of the original nested iteration implementation (without considering invariants), our invariant technique, and query rewriting. We call them NL, NLI and QR respectively. We use the query rewriting technique introduced in [SPL96] since it gives better performance than other rewriting techniques in most cases. The machine we used was a dedicated Ultra Sparc II machine (200MHz and 128M of RAM), running SunOS Release 5.5.1. One disk was used to hold the databases, another for Sybase IQ's binaries, and a third was used for paging. The experiments were run on TPC-D [TPC95] databases at scale factor 1 and 10 with size of 1GB and 10 GB respectively[7]. We used

[7] To save space, we populated only those tables needed in our experiment.



(a) Plan for NL

(b) plan for NLI

Figure 4: Plans for the correlated subquery in Query 1(a)

the improved Sybase IQ version 11.2.1 for all the tests (with invariant techniques turned on and off). The cost we measured is the query response time (scaled within the range of [0..1]). The needed data set can fit into memory at scale factor 1 but can't be totally held in memory at scale factor 10. The impact of memory size will be discussed below. We will base our discussion on the following two queries and their variations.

Query 1: This is a TPC-D [TPC95] query that lists those suppliers that offer the desired type and size of parts in a particular region at the minimum cost.

```
select  s_acctbal, s_name, n_name, p_partkey, p_mfgr,
       s_address, s_phone, s_comment
from part, supplier, partsupp, nation, region
where p_partkey     = ps_partkey
   and s_suppkey     = ps_suppkey
   and p_size        = 15
   and p_type like "%BRASS%"
   and s_nationkey   = n_nationkey
   and n_regionkey   = r_regionkey
   and r_name        = "EUROPE"
   and ps_supplycost = (select min(ps_supplycost)
          from partsupp, supplier, nation, region
          where p_partkey   = ps_partkey
           and s_suppkey   = ps_suppkey
           and s_nationkey = n_nationkey
           and n_regionkey = r_regionkey
           and r_name      = "EUROPE")
order by s_acctbal desc, n_name, s_name, p_partkey
```

Query 2: This query lists those parts with available quantity less than 2,000 whose cost is equal to some cost of the same part offered by any country except one.

```
select  ps_partkey, ps_availqty
from partsupp ps
where ps_partkey <= [partkey]
  and ps_availqty < 2000
  and ps_supplycost in (select ps_supplycost
                      from partsupp, supplier
                      where ps.ps_partkey = ps_partkey
                        and s_suppkey = ps_suppkey
                        and s_nationkey <> 13)
order by ps_availqty
```

Both Query 1 and Query 2 are correlated queries. The outer references are `p_partkey` in Query 1 and `ps.ps_partkey` in Query 2. Each column has a fast projection index, which is a compressed list of column values in `TID` order. Every key column has a high non-group index, which consists of a traditional B^+ tree and a Bit-Sliced index described in [OQ97]. There is also a high non-group index on `ps_availqty`. Other columns in the local selection predicates have low fast indexes on them. A low fast index is basically a Bitmap index ([OQ97]).

Our first experiment was done on Query 1(a), which is derived by changing the nested predicate in Query 1 to `ps_supplycost not in <SUBQUERY>` and omitting the `MIN` in the `SELECT` clause of the subquery. This is a non-rewritable query. There is a four way join in the correlated subquery. NL chooses a bushy join plan (Figure 4(a)) while NLI chooses a left deep tree (Figure 4(b)) which has a larger invariant part. Since no existing QR techniques are available for this query, we are comparing only NL and NLI. The results are shown in Figure 5.[8] In both graphs, NLI gives better performance than NL. At scale factor 10, the data required is too large to fit into memory and thus causes thrashing when evaluating the variant part. The cost of evaluating the variant part becomes more significant in the total cost. So we get less improvement at scale factor 10 than at scale factor 1.

Our second experiment was done on Query 2(a), which is designed by changing the nested predicate in Query 2 to `ps_supplycost not in <SUBQUERY>`. Again, this is a non-rewritable query. We chose different values for `[partkey]` within the range of `ps_partkey`. The predicate `ps_availqty < 2000` is used to reduce the number of duplicate outer reference values (`ps_availqty` ranges from 1 to 9,999). The result at scale factor 1 is shown in Figure 6. We can see our invariant technique significantly improves the performance across the board even though the invariant part contains only one table. Although not shown here, NL and NLI again choose different plans for the subquery. NLI will retrieve all the rows from `supplier` only once to build a hash table

[8] At scale factor 1, there are 469 invocations of the subquery.

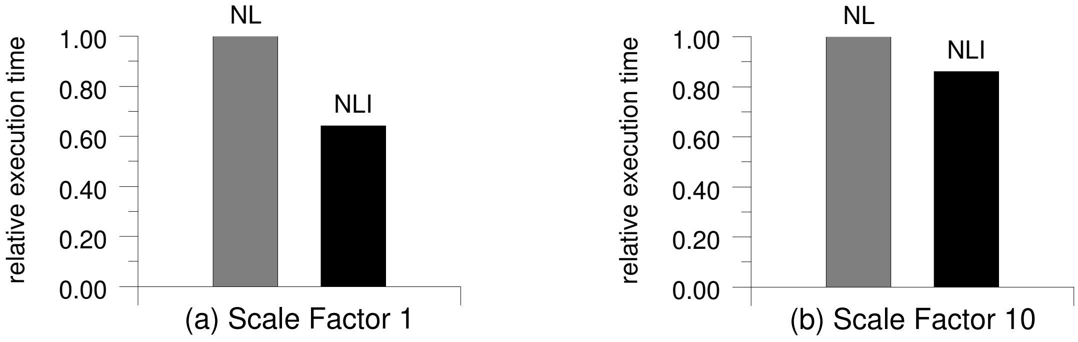


Figure 5: Query 1(a)

and probes will be performed on the in-memory hash table for all the iterations. NL, on the other hand, has to do table lookups on `supplier` for each iteration. So the larger the number of iterations, the worse its performance.

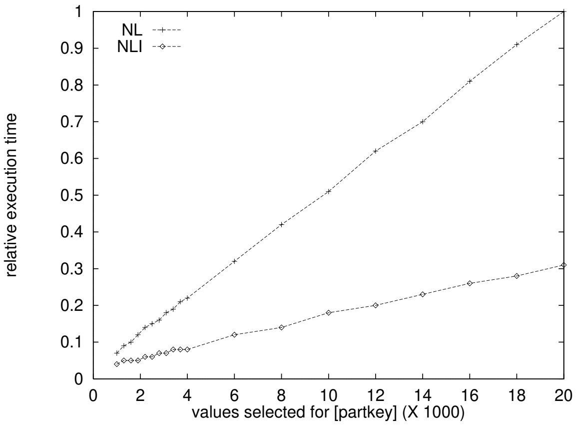


Figure 6: Query 2(a) (scale factor 1)

We now try to compare NLI with QR. The first test is done on Query 1. The syntax for the rewritten query is shown in Figure 7. Notice that V0 is mentioned twice, once in evaluating V1 and once in evaluating the final query. We pre-materialize V1 in our tests to avoid V0 being reexecuted. Since we didn't count the cost of writing V1 to disk, our setup favors QR a little bit.

At scale factor 1, the time for QR remains the same whether V1 is materialized or not since all the data can fit into memory. Figure 8(a) shows the result. NL is about 25% worse than QR, which is in turn, about 15% worse than NLI. It's not surprising that QR is better than NL. But in this case, NLI can be better than QR. We can take a close look at the query plans of NLI and QR shown in Figure 9. The join columns and the grouping columns are listed next to the corresponding nodes. The plans for the outer query and the invariant part enclosed in triangles are the same for both techniques. In Figure 9(a), QR first joins the variant part (node 5) with all the values of the outer reference. Next, the intermediate result is joined with the invariant part. Then a group-by is performed on the

```
create view V0 as
select  s_acctbal, s_name, n_name, p_partkey, p_mfgr,
        s_address, s_phone, s_comment, ps_supplycost
from part, partsupp, supplier, nation, region
where p_partkey     = ps_partkey
   and s_suppkey     = ps_suppkey
   and p_size        = 15
   and p_type like "%BRASS%"
   and s_nationkey   = n_nationkey
   and n_regionkey   = r_regionkey
   and r_name        = "EUROPE"

create view V1 as
select distinct p_partkey
from V0

create view V2(ps_partkey, mincost) as
select V1.p_partkey, min(ps_supplycost)
from partsupp, supplier, nation, region, V1
where V1.p_partkey   = ps_partkey
  and s_suppkey   = ps_suppkey
  and s_nationkey = n_nationkey
  and r_name      = "EUROPE"
group by V1.p_partkey

select  s_acctbal, s_name, n_name, p_partkey, p_mfgr,
        s_address, s_phone, s_comment
from V0, V2
where V0.ps_supplycost = V2.mincost
  and V0.p_partkey = V2.ps_partkey
order by s_acctbal desc, n_name, s_name, p_partkey
```

Figure 7: Rewritten Query 1

outer reference. Finally the group-by result is joined with the outer query block. It's important to notice that some join columns and the grouping column are the same, namely, the outer reference. The query optimizer chooses to implement node 10 with a hash join and node 8 with a sort-merge join on a column different from the outer reference, which cuts the possible liaison between the join in node 7 and the group-by in node 9. The way that NLI evaluates the query can be visualized as partitioning the subquery on the outer reference. Each partition is then evaluated separately and results are combined at the subquery node. The evaluation plan for each partition is shown in Figure 9(b). A nice feature of this plan is that partitioning is shared by more than one node (namely, nodes 1, 3, 4). Potentially, the optimizer can choose an indexed join for node 10 in the QR plan and make it as good as NLI. But this is

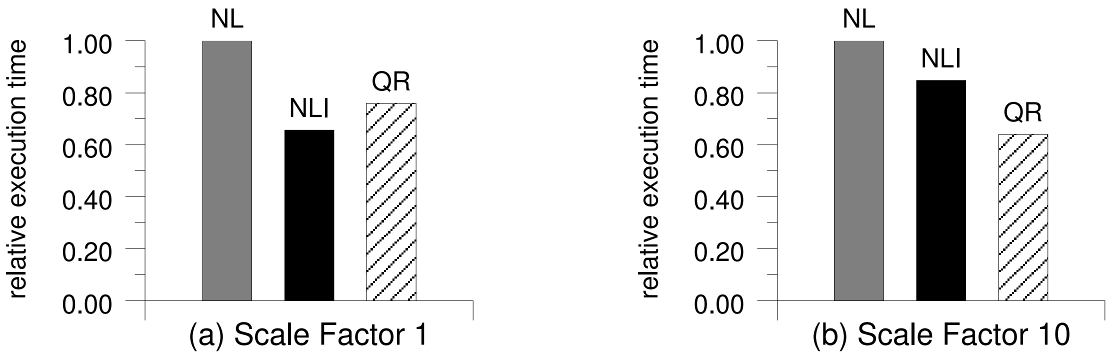


Figure 8: Query 1

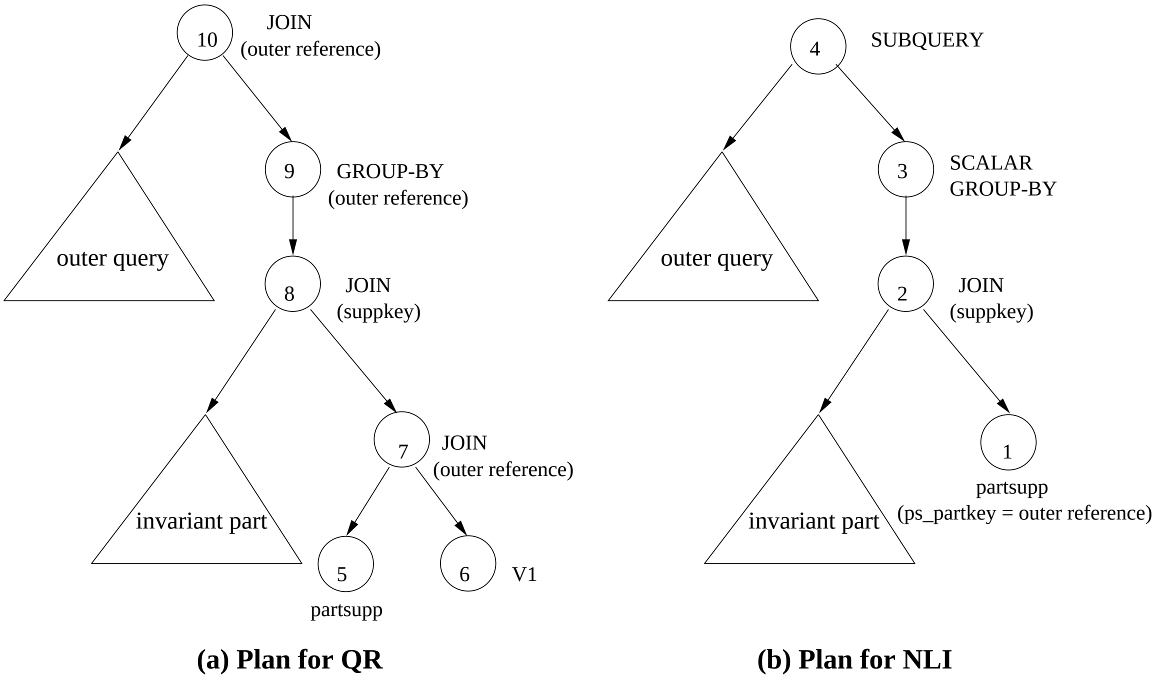


Figure 9: Plans for Q1

very challenging since the optimizer has to include all the join columns and group-by columns as "interesting orders" [SAC+79] (possibly from different query blocks). Techniques of pushing down sorts in joins have been proposed in [SSM96]. Unfortunately, the process of pushing down sort-ahead orders increases the complexity of join enumeration [OL90], possibly by a factor of $O(n^2)$ for n sort-ahead orders. This may become a problem when n is large. To summarize, at scale factor 1, data can fit into memory and the query is CPU cost dominated. The most CPU intensive operations are partitioning operations such as sorting and hashing. So avoiding extra partitioning can make a big difference.

We repeated the same query at scale factor 10 and the result is shown in Figure 8(b). This time the order of the three techniques is QR, NLI and NL, with QR the best. There are two main reasons for the different rankings at scale factor 1 and 10. First, at scale factor 10, the relevant part of data set is too large to fit into memory. So for the nested iteration methods, both NL and NLI, there is thrashing in the buffer manager which increased their actual costs. Our profile analysis indicates that a significant portion of time for NLI is spent on reading pages from disk. Appropriate clustering and a more sophisticated buffer management will be helpful in addressing this issue. We'll elaborate on that in Section 7. The second reason is that although the system using NLI chooses a plan that can share the work of partitioning, it also fixed the underlying join method to be of nested loop style. Thus when the number of iterations is large, this plan is unlikely to be optimal even though the work of partitioning is shared. We'd like to mention also that if V1 is not pre-materialized, QR is actually worse than NLI (it's still better than NL) since V0 has to be evaluated twice without sharing much I/O. We need to take this into account given

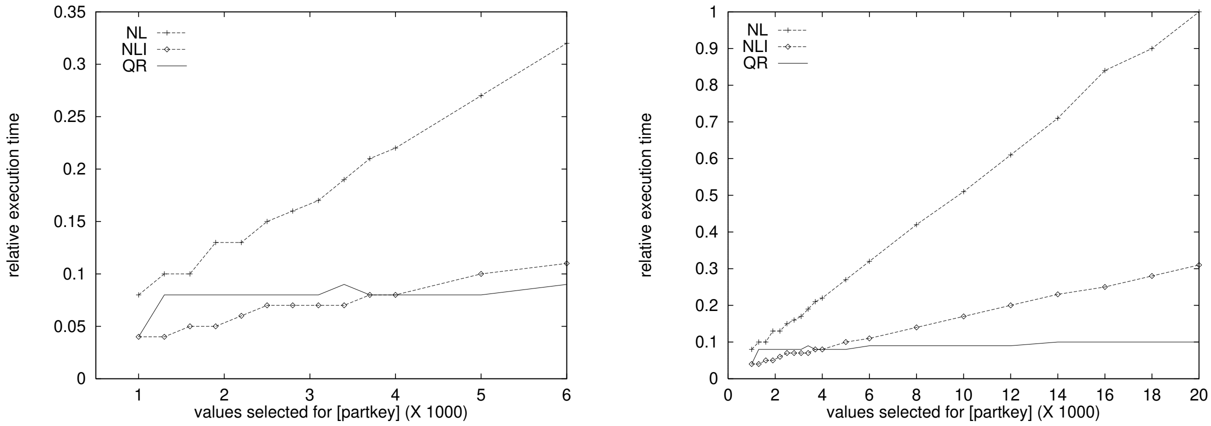


Figure 10: Query 2 (Scale Factor 1)

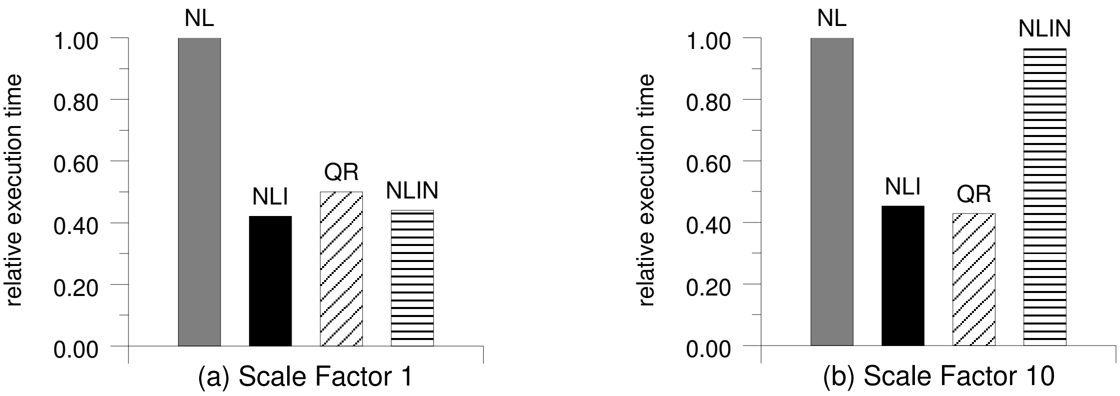


Figure 11: Query 1(b)

the fact that deciding whether to materialize common views/subexpressions could be difficult [Sel88].

To verify our analysis above, we did another experiment using Query 2. The corresponding rewritten query can be derived in a similar fashion, but without introducing a group-by clause. The test was done at scale factor 1 and the result is shown in Figure 10. We can see that when the number of iterations is not too large (less than 3,700), NLI performs better than QR. But NLI degrades quickly when there are too many iterations. As before, both NLI and QR are much better than NL.

In our final test, we want to compare the performance of various techniques when there are duplicates in the outer reference. We designed Query 1(b) by changing the predicates on `r_name` in both the outer query and the subquery of Query 1 to `r_name in ("EUROPE", "ASIA", "AFRICA")`. On average each distinct value of the outer reference will appear 2.4 times. The results are shown in Figure 11. NLI is the technique that will presort on the outer reference while NLIN won't presort. Both of them will use the invariant technique. First we can see that at both scales, NLI is better than NLIN, which proves that our post optimization technique is useful. The comparison between NLI and QR is similar to that of Query 1. NLI wins when the number of iterations is relatively small and QR wins when the number of iterations is large. Notice that, at scale factor 1, NLIN can be better than QR although it has to do more iterations than necessary.

Here, we want to point out another possible optimization for NLI. It's very common in decision support systems that only a few of the selected rows (from the top) in a query need to be returned. In such cases, we can first fetch all the rows from the outer query block and sort it based on the columns in the order-by clause. We then iterate through the sorted list to evaluate the subquery. We stop when we have enough rows to return. Since fewer iterations will be invoked, we can expect better performance when using the invariant technique. In cases when there are duplicates in outer references and the outer references are not the primary ordering keys specified in the query, we can still use the above method, but we use the alternative hashing method (or hybrid cache [HN96]) to avoid the reexecution of the subquery on repeated outer reference values. Similar ideas have been proposed in other different applications [JM96, HHW97].

To summarize our experiments, we have the following conclusion. The invariant technique helps when query rewriting is not applicable. As RAM becomes cheaper, more complex queries (possibly involving new CPU intensive operations) will fit into memory. The invariant technique we proposed here performs particularly well for this class of queries and may suggest better evaluation plans than query rewriting. Thus an optimizer should take the invariant technique into consideration even when query rewriting techniques apply. When the retrieving cost (RCost) is too high, the optimizer will decide not to cache the invariant result. In this case, NLI reduces to NL. So NLI will never perform worse than NL.

7 Related Work

In [Cha97, CR97], techniques of how to evaluate complex queries involving joins and group-bys efficiently have been proposed. The authors introduced a class of queries called "group queries". The unique feature of group queries is that a relation will participate in joins and group-bys multiple times on the same join and group-by columns. A better evaluation plan has been suggested by first partitioning the data on the common join and group-by attributes. Then each partition can be evaluated separately and the results will be combined later to form the final answer. The new evaluation method is better when the complexity of the partial computation is high and/or the number of groups is large. From our experimental results, we can see that even if a relation is used only once in a query, sharing the partitioning work among different levels of joins and group-bys can sometimes be helpful.

A buffer management strategy has been introduced in [CD85], which takes into account the way tables will be accessed when allocating buffers to them. We believe, with the help of this strategy and some appropriate clustering, a lot of the thrashing in our invariant evaluation method (for datasets larger than memory) can be avoided. The details are left to future work.

Many access plan strategies for complex decision support queries exploit the physical ordering of data provided by indexes or sorting. A recent paper [SSM96] introduced techniques of pushing down sorts in joins, reducing the number of sorting columns and detecting whether sorting can be avoided. Predicate migration [Hel94] considers whether an expensive predicate should be applied before or after a join. Similarly, group-by push-down [YL95] considers whether a group-by should be performed before a join. The above techniques are orthogonal to the invariant technique. If they are applied in separate phases independently by the query optimizer, the best plan may be missed. We'd like to investigate in the future how to combine these techniques more wisely.

The multi-query optimization problem is to optimize a collection of queries so as to minimize their total cost. The basic approach is to reduce the overall query execution cost by sharing intermediate results. Heuristic algorithms have been introduced in [Sel88, SG90]. As we have seen, the rewritten queries usually involve views being shared. Techniques for multiple query optimization on an intra-query (as opposed to inter-query) basis are desirable to improve the performance of the rewritten queries.

Other query rewriting related work has been done in [Kim82, Day87, GW87, PHH92]. A comparison of the performance among these techniques can be found in [SPL96].

8 Conclusion

In this paper, we first discussed various ways of evaluating a correlated query. We then presented the technique of caching and reusing invariants and showed how to incorporate the invariant feature into a join optimizer smoothly. We also introduced other optimization methods to improve our invariant technique. As the most straightforward way of implementing correlated queries, the nested iteration method is the only resort when other techniques are not applicable. Our invariant techniques significantly improved the naive nested iteration method because it avoids unnecessary reexecution of the invariant part of the subquery. This can be a huge saving when the number of iterations is large. In cases when query rewriting techniques are feasible, we showed that neither query rewriting nor the invariant technique dominate one another. So there are some tradeoffs in using the two techniques. We believe that both techniques should be considered on a cost basis, especially when the number of rows required from the outer query block is not very large. The optimizer should understand the advantage of both techniques and be able to make the correct choice.

Acknowledgments

The basic idea of marking the invariant nodes in a data flow tree and replaying the content in a storage element was originally proposed in Sybase IQ [Bel96]. Roger MacNicol, manager of the database engine group in Sybase IQ, gave a lot of help both in implementation and validating the usefulness of the technique. Steve Kirk, a senior engineer in Sybase IQ, spent a lot of time explaining the architecture of the join optimizer and data flow engine in IQ, which makes the realization of the invariant technique possible. We also want to thank him for several useful suggestions in improving the technique, such as subquery memoization.

References

[Bel96] Randy Bello. Invariant subplans in dataflow. *Sybase IQ Internal Engineering Document*, 1996.

[CD85] Hong-Tai Chou and David J. Dewitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 11th VLDB Conference*, pages 127–141, 1985.

[Cha97] Damianos Chatziantoniou. *Optimization of Complex Aggregate Queries in Relational Databases*. PhD thesis, Department of Computer Science, Columbia University, 1997.

[CR97] Damianos Chatziantoniou and Kenneth A. Ross. Groupwise processing of relational queries. In *Proceedings of the 23th VLDB Conference*, pages 476–485, 1997.

[Day87] Umeshwar Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates and quantifiers. In *Proceedings of the 13th VLDB Conference*, pages 197–208, 1987.

[Gra94] Goetz Graefe. Volcano, an extensible and parallel query evaluation system. *IEEE Transactions on knowledge and data enginnering*, 6(6):934–944, 1994.

[GW87] Richard A. Ganski and Harry K.T. Wong. Optimization of nested sql queries revisited. In *Proceedings of the ACM SIGMOD Conference*, pages 23–33, 1987.

[Hel94] Joseph M. Hellerstein. Practical predicate placement. In *Proceedings of the ACM SIGMOD Conference*, pages 325–335, 1994.

[HHW97] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *Proceedings of the ACM SIGMOD Conference*, pages 171–182, 1997.

[HN96] Joseph M. Hellerstein and Jeffrey F. Naughton. Query execution techniques for caching expensive methods. In *Proceedings of the ACM SIGMOD Conference*, pages 423–433, 1996.

[JM96] Roberto J. Bayardo Jr. and Daniel P. Miranker. Processing queries for first few answers. In *Proceedings of the Fifth International Conference on Information and Knowledge Management*, pages 45–52, 1996.

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

[Lea96] Dan Leary. Dataflow operators feature specification. *Sybase IQ Internal Engineering Document*, 1996.

[Mic68] Donald Michie. "memo" functions and machine learning. *Nature*, 218:19–22, 1968.

[OL90] Kiyoshi Ono and Guy M. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proceedings of the 16th VLDB Conference*, pages 314–325, 1990.

[OQ97] Patrick O'Neil and Dallan Quas. Improved query performance with variant indexes. In *Proceedings of the ACM SIGMOD Conference*, pages 38–49, 1997.

[Pau97] Glenn Paulley, 1997. personal communication.

[PHH92] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/rule based query rewrite optimization in startburst. In *Proceedings of the ACM SIGMOD Conference*, pages 39–38, 1992.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomsa G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD Conference*, pages 23–34, 1979.

[Sel88] Timos K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.

[SG90] Timos K. Sellis and Subrata Ghosh. On the multiple-query optimization problem. *IEEE Transactions on knowledge and data enginnering*, 2(2):262–266, 1990.

[SPL96] Praveen Seshadri, Hamid Pirahesh, and T.Y.Cliff Leung. Complex query decorrelation. In *Proc. IEEE Int'l Conf. on Data Eng.*, pages 450–458, 1996.

[SSM96] David Simmen, Eugene Shekita, and Timothy Malkemus. Fundamental techniques for order optimization. In *Proceedings of the ACM SIGMOD Conference*, pages 57–67, 1996.

[Syb97a] Sybase Corporation. *Adaptive Server Enterprise 11.5*, 1997.

[Syb97b] Sybase Corporation. *Sybase IQ 11.2.1*, 1997.

[TPC95] Tpc-d benchmark standard specification (revision 1.0). May 1995.

[YL95] Weipeng P. Yan and Per-Ake Larson. Eager aggregation and lazy aggregation. In *Proceedings of the 21th VLDB Conference*, pages 345–357, 1995.

A Non-rewritable Correlated Queries

In this section, we illustrate several types of correlated queries that have no known ways of being rewritten without introducing new internal operators.

1. Nested predicate involves negation.

```
SELECT  Ri.Ck
FROM    Ri
WHERE   Ri.Ch NOT IN  (SELECT  Rj.Cm
                       FROM    Rj
                       WHERE   Ri.Cn = Rj.Cp)
```

2. Nested predicate involves `All`.

```
SELECT  Ri.Ck
FROM    Ri
WHERE   Ri.Ch <> ALL  (SELECT  Rj.Cm
                       FROM    Rj
                       WHERE   Ri.Cn = Rj.Cp)
```

3. Nested predicate involves set predicates.

```
SELECT  Ri.Ck
FROM    Ri
WHERE   (SELECT   Rj.Ch
         FROM     Rj
         WHERE    Ri.Cn = Rj.Cp)
         CONTAINS
        (SELECT   Rk.Cm
         FROM     Rk)
```