Complex Query Decorrelation

Praveen Seshadri
Computer Sciences Department
University of Wisconsin, Madison WI 53706, USA
praveen@cs.wisc.edu

Hamid Pirahesh
IBM Almaden Research Center
San Jose, CA 95120, USA
pirahesh@almaden.ibm.com

T.Y. Cliff Leung
IBM Santa Teresa Laboratory
San Jose, CA 95141, USA
cleung@almaden.ibm.com

Abstract

Complex queries used in decision support applications use multiple correlated subqueries and table expressions, possibly across several levels of nesting. It is usually inefficient to directly execute a correlated query; consequently, algorithms have been proposed to decorrelate the query, i.e. to eliminate the correlation by rewriting the query. This paper explains the issues involved in decorrelation, and surveys existing algorithms. It presents an efficient and flexible algorithm called **magic decorrelation** *which is superior to existing algorithms both in terms of the generality of application, and the efficiency of the rewritten query. The algorithm is described in the context of its implementation in the Starburst Extensible Database System, and its performance is compared with other decorrelation techniques. The paper also explains why magic decorrelation is not merely applicable, but crucial in a parallel database system.*

1 Introduction

The concept of *correlation* in SQL is similar to the use of non-local variables in block-structured programming languages. The processing of correlated queries is important because (a) many decision support applications use correlation, (b) correlation is a convenient programming idiom for many SQL programmers (closely mimicking a function invocation paradigm), and (c) correlated queries are often created "automatically" by application generators that translate queries from application domain-specific languages into SQL. The TPC-D decision support benchmark [TPC-D94] of seventeen queries includes two correlated queries, recognizing the importance of correlation.

In an early relational DBMS like System R [SACLP79], a correlated sub-query was executed in a tuple-at-a-time fashion (nested iteration). The same approach is still used in current database systems. Since an equivalent set-oriented execution strategy might perform orders of magnitude better, there has been more than a decade of research that aims to "decorrelate" queries, i.e. to eliminate the correlations by rewriting the queries into a form that permits set-oriented execution. However, existing decorrelation algorithms work only on specific kinds of correlated queries, and the rewritten queries are sometimes inefficient. Some algorithms can even produce incorrect results. A practical decorrelation algorithm needs to work correctly on arbitrarily complex queries, and the resulting query should be efficient to execute. To the best of our knowledge, ours is the first algorithm to satisfy these criteria. Such an algorithm has become all the more crucial due to the recent interest in using parallel database systems for complex decision support applications. In fact, processing correlated queries is considered one of the most challenging current problems in parallel query processing[Gra95].

1.1 Contributions

We explain the issues involved in decorrelation, and present a survey of other proposed decorrelation methods. We develop a query rewrite algorithm framework that decorrelates arbitrary SQL queries. The algorithm is similar to the magic sets rewriting transformation, as applied to *non*-recursive relational queries [MFPR90]. Consequently, our algorithm is called *magic decorrelation*. The algorithm framework is extensible, and permits various implementations to provide various "degrees" of decorrelation as required by different database system environments (especially parallel environments). We describe a specific implementation of magic decorrelation in the Starburst Extensible Database System [HCL$^+$90]. We compare the performance of magic decorrelation with other known techniques for evaluating a correlated query. Finally, we discuss why our decorrelation algorithm is not merely applicable, but is crucial in a parallel database environment.

2 Explaining Decorrelation

In a correlated SQL query, values from an outer query block are accessed inside a nested subquery block. Consider various evaluation strategies for an example based on the familiar EMP and DEPT relations. Each employee is assigned to a building in which he/she works. Each department is situated in a building, but may have employees in other buildings as well. The query finds those departments of low budget that have more employees than there are employees working in the building in which the department is located. Note that the correlated value *DEPT.building* is used inside its subquery.

```
Select D.name From Dept D
Where D.budget < 10000 and D.num_emps >
    (Select Count(*) From Emp E Where D.building = E.building)
```

Nested Iteration: The subquery is invoked once for every DEPT tuple (whose budget is less than 10000) in the outer query block. The table EMP may not have an index on the building column; an entire table scan access will be required for every low-budget department tuple. Further, if there are duplicate values of DEPT.building, the subquery invocations will perform redundant work.

A nested iteration execution is not set-oriented, because there is a "coupling" between each value from the outer block and the execution of the correlated subquery block. This strategy is efficient only in cases where there are few duplicates in the correlation attribute, and independent executions of the subquery perform little common work. *The aim of decorrelation is to "decouple" the execution of the subquery block from that of the outer block, by rewriting the query.*

Kim's Method: Kim's method [Kim82] produces the following rewritten query:

```
Select D.name From Dept D, Temp(empcount, bldg) AS
   (Select Count(*), E.building From Emp E GroupBy E.building)
Where D.budget < 10000 and D.num_emps > Temp.empcount
   and D.building = Temp.bldg
```

The subquery is converted into a table expression with a GROUPBY clause, and the correlation predicate is moved to the outer block. There are three problems with this approach:

- The transformation works only if the correlated predicate (on "building") is a simple equality predicate.
- The computation in the subquery is no longer restricted by the correlated predicate, and this may lead to poor performance. The COUNT computation must be done for all buildings with employees, not just for those buildings assigned to low budget departments.
- The rewritten query may be semantically different from the original query! If a department D with budget $= 500$ and num_emps $= 1$ is located in a building B that has no employees assigned to it, then department D's name is a desired answer to the query. In the rewritten query, the Temp table expression will not have a tuple in it corresponding to (0, B); consequently, D's name will not be generated as an answer to the query. This is called the COUNT bug [Kie84]. If the COUNT in the subquery were replaced by some other aggregate function like MAX, MIN, AVG, SUM, etc, the correlated subquery should return a NULL value. If the subquery is involved in a predicate like IS NULL, then a similar problem arises.

Dayal's Method: The solution to the COUNT bug requires the introduction of an outer-join operator. Dayal's method[Day87] merges the two query blocks using the left outer-join (LOJ) operator to produce a transformed query of the form:

```
Select D.name
From DEPT D LOJ EMP E On (D.building = E.building)
Where D.budget < 10000 GroupBy D.[key]
Having D.num_emps > Count(E.[key])
```

There are three problems with this transformation:

- To preserve the duplicate semantics of the query result, the transformed query is grouped by some key of the Dept relation. If there are several department tuples with the same value for the building column, there may be a repetition of aggregate computation. In other words, whenever the correlated column (in this case, Dept.building) is not a key, there may be repeated computation.
- Since the join/outer-join of all involved relations is performed first, the size of the set to be grouped might be much larger than in the case of Kim's strategy, potentially leading to a significant performance degradation.
- The strategy works only for linearly structured queries with SELECT and GROUPBY constructs.

Ganski/Wong's Method: Ganski and Wong proposed a method [GW87] that projects a unique collection of correlation values into a temporary relation. The temporary relation is then used to decorrelate the subquery using an outerjoin. However, many practical details were not considered, and the method is not applicable to non-linear correlated queries. This method is a special case of the magic decorrelation algorithm presented in this paper; consequently, we shall not elaborate further on it.

2.1 Magic Decorrelation

A general SQL decorrelation algorithm is difficult to design because of the practical details that need to be handled. Complex queries could be hierarchical (for example, a subquery/view with a UNION operator), or could involve common subexpressions. Correlations can occur not merely in simple predicates, but also within complex expressions involving multiple correlated values. Correlations can also span multiple levels of query blocks. There are also factors that could make it difficult to decorrelate parts of a query. For example, if a subquery is existential or universal (corresponding to the SQL constructs ANY and ALL respectively), it is not possible to directly convert the subquery to a table expression with join operators (as is required by the existing decorrelation methods). All the same, it may be desirable to decorrelate the query "as much as possible". Magic decorrelation deals with all these situations; some of the details have been omitted in this paper but are explained in [SPL94]).

Any correlated subquery block can be modeled as a function CS(x) whose parameters x are the correlation values. The function returns a table which is then processed at the outer block level. In our example, the correlated subquery is a function that uses the value $Dept.building$ as a parameter, and returns a table containing a single tuple. The outer query block can be represented by the following abstract pseudo-code:

```
foreach (x ∈ X) {
        SubQueryResult = CS(x);
        Process(SubQueryResult);}
```

where X represents the set of values with which the correlated subquery is invoked. The primary aim of decorrelation is to decouple the execution of CS from the execution of the outer query block. Consider some set X1, such that $X \subseteq X1$. Obviously, $(x \in X)$ implies $(x \in X1)$. Let us define a new table DS (i.e. "Decoupled Subquery") such that $DS = \{(x, y) | x \in X1 \wedge y \in CS(x)\}$. In other words, DS computes CS(x) for all values x in X1. Now consider the following version of the pseudo-code of the outer block:

```
foreach (x in X) {
        SubQueryResult = {y1|(x1, y1) ∈ DS ∧ x = x1};
        Process(SubQueryResult);}
```

The computation of DS is decoupled from that of the outer block. Note that it is important to maintain the correlating relationship between the value of x in each pass through the loop, and the values selected from DS during that pass; the condition $x = x1$ enforces this relationship. At this abstract level, three questions remain: (1) how does one compute X1?, (2) how does one compute DS using X1?, and (3) how does one enforce the correlating relationship?. The magic decorrelation algorithm is based on this abstraction. The result of applying magic decorrelation to the example query is shown below.

```
Create View Supp_Dept As (Select name, building, num_emps
    From Dept Where budget < 10000);
Create View Magic AS (Select Distinct building From Supp_Dept);
Create View Decorr_SubQuery (building, count) AS
   (Select M.building, Count(*)
    From Magic M, Emp E Where M.building = E.building
    GroupBy M.building );
Create View Complete_Decorr_SubQuery(building, count) AS
   (Select M.building, coalesce(E.count, 0)
    From Magic M LOJ Decorr_SubQuery D on (M.building = D.building)
Select S.name From Supp_Dept S, Complete_Decorr_SubQuery D
Where S.building = D.building and S.num_emps > D.count
```

The Supp_Dept table represents the computation in the outer query block until the point that the subquery invocations begin. The Magic table represents the (duplicate-free) set X1 of correlation values with which the subquery will be invoked.

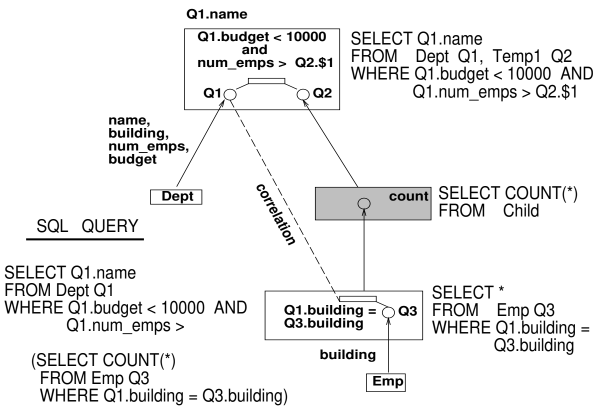

Figure 1: An Example QGM

Decorr_SubQuery is the table generated by decorrelating the subquery using the Magic table. It contains one tuple per value of M.building (i.e. one tuple per correlation value). In order to avoid the COUNT bug due to missing values of M.building, the Complete_Decorr_SubQuery table is defined. Here, the Magic table M is the outer table of a left outer-join with the decorrelated subquery D. If there is a missing value of a count attribute, it is replaced by 0 (this is the effect of the Coalesce function). This table is only needed to avoid the occurrence of the COUNT and IS-NULL bugs; it need not be introduced in the majority of queries. Finally, in the outer query block, the Supp_Dept table S is joined with the complete decorrelated subquery to produce the desired answers. The join predicate S.building = D.building enforces the correlating relationship.

3 Query Rewrite in Starburst

The query data structure used by Starburst is called the Query Graph Model (QGM) [PHH92]. Each query construct such as a Select-Project-Join(SPJ), an Aggregate, a Union, or an Intersection, corresponds to a query block in the QGM. The QGM is transformed using rewrite rules, each operating at the granularity of one query block. Each rule application should leave the QGM in a consistent state, because the query rewrite phase may be terminated at any point when the allocated resources (typically, time) are exhausted. We use visual representations of the QGM to explain magic decorrelation. Figure 1 shows the QGM of the example SQL query of Section 2. Each query block is represented as a *box* in the figure. The portion of the SQL query corresponding to each query block is shown at the side of the appropriate box. Our algorithm will treat Select-Project-Join(SPJ) boxes differently from other boxes; all non-SPJ boxes are shaded grey to help make this distinction. The input tables accessed by the operators of each box are shown by the solid lines with arrows. These lines are called *iterators* and each is a handle on an input table. All or some of the fields of a table may be accessed along an iterator on that table. The names marked along the iterator represent the columns that are being projected along it. A dotted line between two boxes indicates a correlation between them. To keep the figures uncluttered, irrelevant information is not shown. In the discussion of the algorithm, we assume for the sake of simplicity that the correlated query is hierarchical (i.e., the query is not recursive, and each subquery or table expression is used by only one parent query). At the level of the QGM, this implies that the query graph is a tree.

3.1 Correlation Terminology

Box A is a *parent* of box B (B is a *child* of A) if box A has an iterator over B. Box A is (recursively) an *ancestor* of another box B iff it is a parent of B, or one of A's children is an ancestor of B. B is a *descendant* of A iff A is an ancestor of B. Box B is *directly correlated* if it contains a correlation that references a column col_1 from a table in the From clause of an ancestor A. The column col_1 is called the *correlation column*. The ancestor A is the *source of correlation*, and the box containing the correlation (box B) is the *destination of correlation*. Box B is (recursively) said to be *correlated* to one of its ancestors A if it is directly correlated to A, or if one of its descendants is directly correlated to A. The actual values of a correlation column at the source of correlation are the *correlation bindings*.

4 Magic Decorrelation in Starburst

The magic decorrelation rewrite rule is applied to the QGM in a *top-down fashion*, transforming one box at a time. Whenever the rewrite rule is applied to a box, its ancestors in the QGM have already been processed. In all figures, CurBox corresponds to the box currently being processed. We assume that some particular order is chosen for the iterators in the CurBox(see Section 7 for a discussion of this issue). The decorrelation algorithm looks at the iterators in this order, and for each iterator over a child (subquery) box, it determines if the child box is correlated, and whether decorrelation is possible. If so, it generates the set of correlation bindings that can be used to decorrelate the box. This stage is called the *FEED* stage, because it feeds the bindings to the child box.

When the rewrite rule is applied to the subquery (i.e. when the subquery is treated as the CurBox), it decorrelates the subquery using the correlation values. This is called the *ABSORB* stage because the subquery absorbs the correlation bindings resulting in a decorrelated query.

We now separately discuss the detailed algorithm in terms of our familiar example query.

4.1 Deciding to Decorrelate

To determine if a child box is correlated, the algorithm utilizes the following information: (1) a list of its ancestors, (2) a list of its descendants, (3) which of its ancestors it is correlated to, and (4) which descendant box caused each correlation. In our implementation, this information is precomputed by a traversal of the graph, and stored for all boxes in the query graph.

If the box is correlated, the algorithm needs to decide if the box can be decorrelated. This depends on the semantics of the operators in the box, and on how the outputs of the box are used. The necessary information about the usage of the box's outputs is computed as part of a single graph traversal during preprocessing. For example, if the output column X of an Aggregate box with a COUNT aggregate is used in a predicate "X=0", naive decorrelation will lead to the COUNT bug. In this case, a left outer join with a Coalesce function will produce the count of 0 to satisfy this predicate. If the predicate were "=1", then this additional complexity is not required. Section 4.4 describes other such scenarios.

4.2 FEED Stage

In Figure 2, we illustrate the FEED stage of the magic decorrelation rewrite rule, applied to the top-level box. Figure 2[a] shows the complete initial state of the QGM. This is identical to

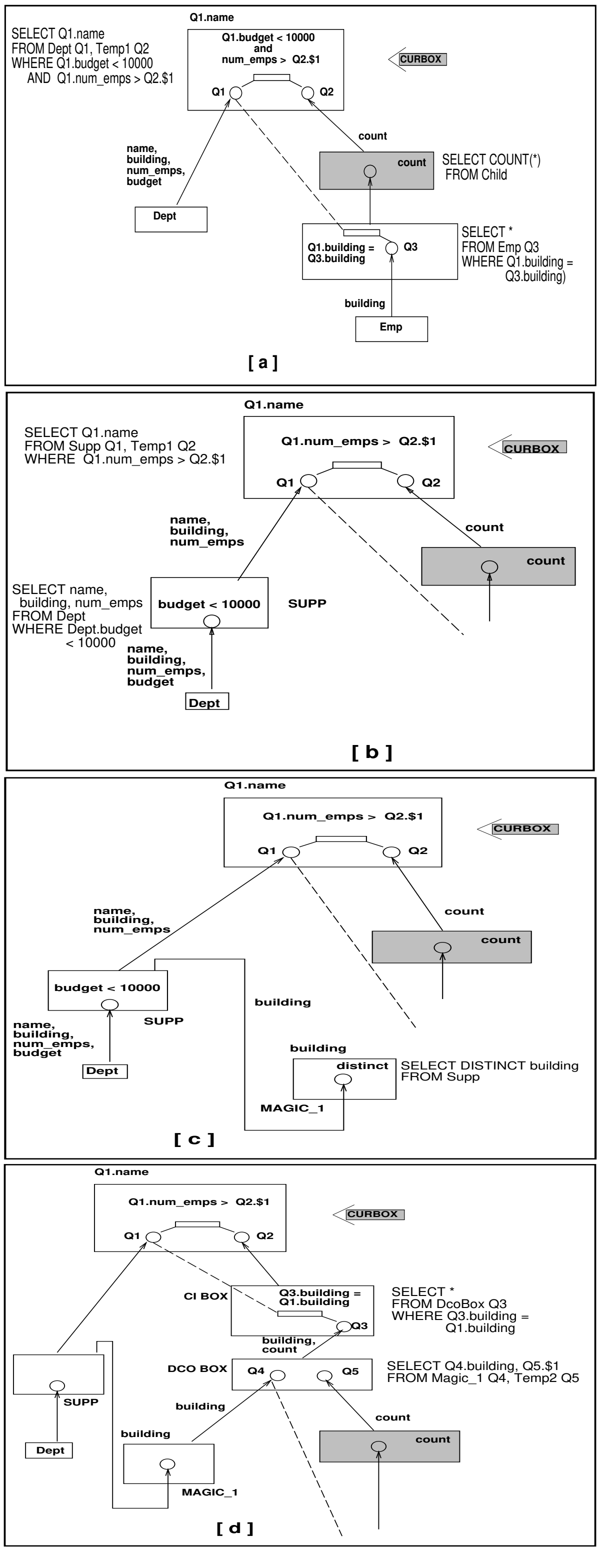

Figure 2: Decorrelation FEED Stage

Figure 1. The other figures concentrate on the relevant portion that is being rewritten. The first step in magic decorrelation determines which bindings need to be passed to the child box. In the example, the child box is correlated on the *building* attribute, and the algorithm determines that it can be decorrelated. The next step is to collect the portion of the computation ahead of the subquery into a single "supplementary" table SUPP; the rest of the query remains unchanged. This step is illustrated in Figure 2[b]. A unique set of correlation bindings is then projected into a "magic" table for the child, as shown in Figure 2[c]. The final step of the FEED stage is to decouple the CurBox from the child box. This is accomplished as shown in Figure 2[d]. A new SPJ box called the Decorrelated Output (DCO) box is introduced immediately above the child, to provide a *decorrelated view* of the child to the parent. The DCO box has an iterator Q4 over the magic table of the child and an iterator Q5 over the child, and computes the cross product of the two. The destination of correlation in the descendant is modified so that it gets its bindings from Q4 instead of Q1. In this manner, the child box and the rest of the QGM below it are decoupled from the CurBox. The CurBox, however, needs a *correlated view* of the subquery to retain the relationship between each correlation value and the corresponding answer from the decorrelated subquery. A Correlated Input (CI) box is introduced immediately above the DCO box, with a correlated predicate that provides this view to the CurBox. This last stage is essential for correctness, since otherwise the correspondence enforced by the correlation in the original query is lost. It is important to note that the query graph is *consistent* at this stage, preserving the incremental nature of the algorithm. While we have succeeded in decoupling the query blocks, we have also introduced an additional correlation between the CurBox and the CI box. In many cases, it is possible to merge the CI box into the CurBox converting the correlation predicate into an equi-join predicate. This is done by existing rewrite rules that merge query blocks

4.3 ABSORB Stage

It is usually possible to eliminate the Decorrelated Output (DCO) box entirely; this happens when the rewrite rule is applied to the child box (which is now treated as the CurBox). There is a DCO box immediately above the CurBox with an iterator over its magic table. During the *ABSORB* stage, the CurBox needs to absorb the correlation bindings that are available in the magic table. This portion of the algorithm has two variants depending on whether the CurBox is an SPJ box or not. We now look at each of these cases separately, in the context of our example.

4.3.1 non-SPJ Box

If the CurBox is not an SPJ box, for example an Aggregate box, the actual correlation is usually contained in some descendant of the CurBox(some exceptions are described in [SPL94]). Therefore, the correlation bindings in the magic table should be fed to the children of the CurBox, so that they can be decorrelated. Once the children have been decorrelated, the CurBox can absorb the correlation bindings from the children. The decorrelation of a non-SPJ box is performed, therefore, **after** the FEED stage for its children.

The Figure 3[a] shows the relevant portion of the QGM for our example query when the rewrite rule is about to be applied to the Aggregate box. Note that Figure 3[a] is the same as Figure 2[d], which is the result of applying the rewrite to the

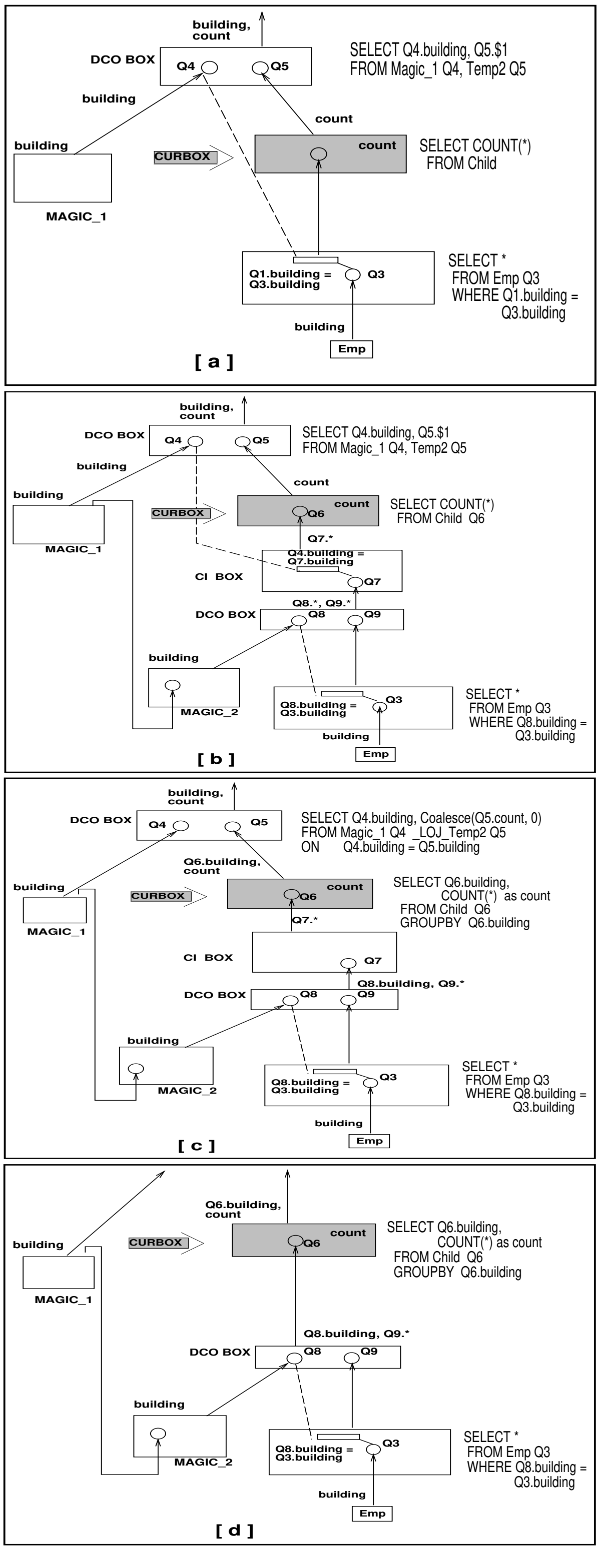

Figure 3: Decorrelation ABSORB Stage(non-SPJ)

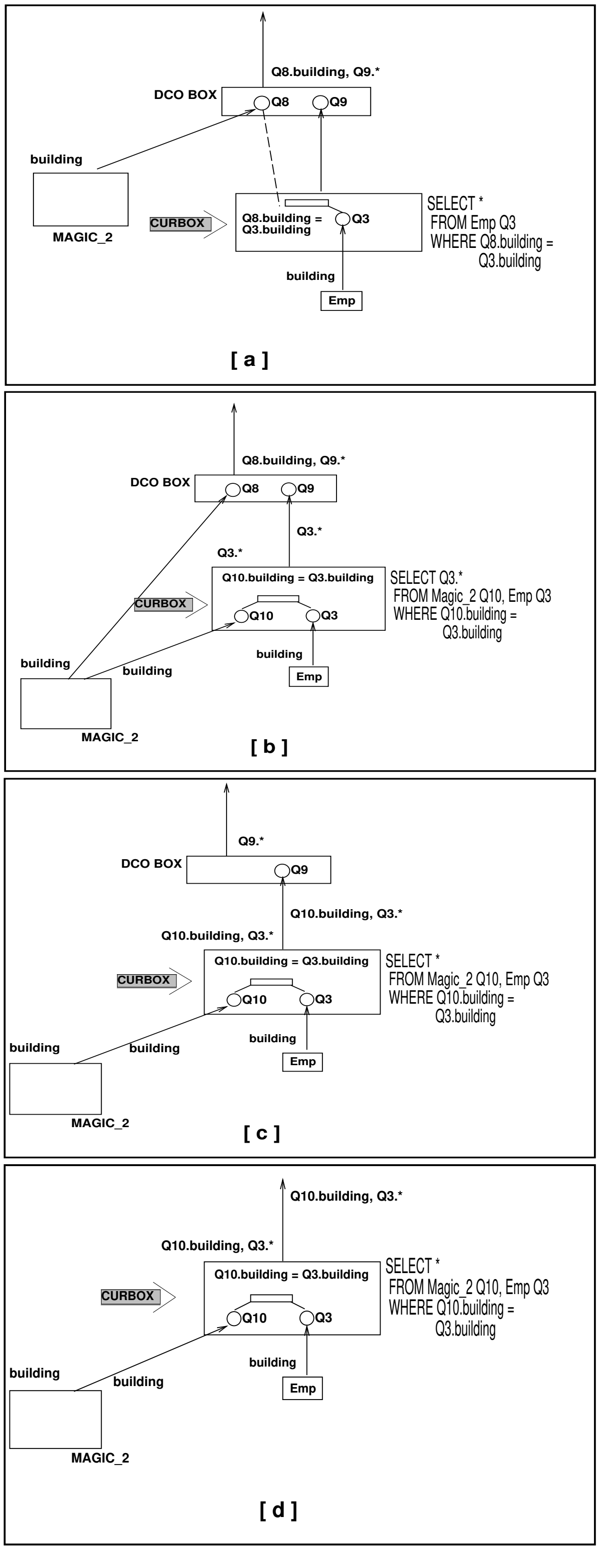

Figure 4: Decorrelation ABSORB Stage(SPJ) – correlation is totally eliminated.

immediate parent of the CurBox. The FEED stage provides it with correlation bindings in a Magic table. Since the CurBox is a non-SPJ box, the bindings are drawn directly from the magic table of the CurBox. Apart from this variation, the rest of the FEED stage proceeds as described earlier. The result of the FEED stage is shown in Figure 3[b].

Once the FEED stage is complete, the CurBox can be decorrelated because it can now access the correlation bindings from its child. Figure 3[c] shows the stage after the decorrelation of the CurBox. Decorrelation is effected by adding the *building* attribute to the output, and grouping by that attribute. Now, the correlated predicate in the CI box below can be removed. The Aggregate box will not produce any tuple where before the count = 0 was produced (the problem that lead to the COUNT bug). To reproduce this tuple and ensure that the semantics of the query is not altered, we simply convert the DCO box to an outer join box. Figure 3[d] shows the simplified query after the redundant CI box is removed (by other rewrite rules). Note that if the problem of the COUNT bug is not present (as determined by a semantic analysis of the query – see Section 4.1), the entire DCO box above the CurBox can also be removed.

4.3.2 SPJ Box

If the CurBox is an SPJ box, it can add the magic table to its From clause.The destination of the correlation is modified to reference the columns from the magic table, instead of the original source of correlation. The columns from the magic table are added to the output of the CurBox (i.e. they are added to the list of attributes in the Select clause), thereby completing the decorrelation.

The Figure 4[a] shows the relevant portion of the QGM for our example query when the rewrite rule is about to be applied to the lowest SPJ box. Note that Figure 4[a] is the same as Figure 3[d]. In the first step in Figure 4[b], the CurBox adds the magic table to its From clause. The correlation predicate is changed so that the source is now the magic table iterator in the CurBox. As the next step, the correlation bindings from the magic table iterator are added to the output of the CurBox. In this example, the attribute "Q10.building" is added to the output. The iterator over the magic table in the DCO box is now redundant and can be removed, leaving the CurBox decorrelated as in Figure 4[c]. The Figure 4[d] shows how the simplified query looks when the redundant DCO box is eliminated (by other existing rewrite rules). If the query were more complex, and this SPJ box itself had children that needed to receive correlation bindings, the rewrite rule would also have to perform the FEED stage of the algorithm on this box. Unlike the non-SPJ boxes, however, the ABSORB stage can be performed **before** the FEED stage for its children.

4.4 Algorithmic Details

We have presented a simplified description of magic decorrelation, and demonstrated its execution on an example. We have ignored issues of how it interacts with magic sets rewriting, how common sub-expressions are handled and how recursion is handled. We have also glossed over some of the tricky details that arise in dealing with non-SPJ boxes. Further, while we have simplified the presentation by speaking in terms of SPJ and non-SPJ boxes, the actual Starburst implementation allows for extensibility of SQL constructs by classifying each kind of box as either capable of accepting a magic table (AM) or incapable of it (NM). The behavior of each box with respect to the magic decorrelation algorithm is captured by a box *encapsulator*, The details of these aspects of the algorithm are described in [SPL94].

We have seen that the magic decorrelation algorithm can introduce extra CI boxes into the query graph; this happens when correlated subqueries occur within existential(ANY,EXISTS,IN) or universal (ALL) quantification. These boxes perform repeated correlated selections on the result of the decorrelated subquery. This may be unacceptably slow in many systems that do not support indexes on temporary relations. Consequently, the box encapsulator could choose not to decorrelate in such situations. On the other hand, as we discuss in the Section 6, correlation greatly degrades performance in a parallel database system, and such a system may be willing to incur the extra overheads of decorrelation.Similarly, if a system does not implement a left outer-join operator, it may not be possible to totally decorrelate an aggregate box. All the same, the rest of the query can be decorrelated, and the remaining correlations will be localized to the aggregate box and the temporary boxes created above and below it. These decisions on whether and how to decorrelate act as knobs that can be used to adapt the magic decorrelation algorithm [SPL94].

5 Performance Results

Magic decorrelation has been prototyped in the Starburst DBMS. In this section, we present a performance comparison with other strategies for processing correlated queries.

5.1 Algorithms Investigated

We considered four basic algorithms: nested iteration(NI), Kim's method(Kim), Dayal's method(Dayal), and magic decorrelation. All Starburst query transformations that were unrelated to decorrelation were applied to all queries; i.e. we compared the "optimal" versions of each rewritten query. For Kim's and Dayal's methods, the query rewrite was manually performed and then submitted to the system. With magic decorrelation, when the correlation attributes form a key of the supplementary table, the common sub-expression formed by the supplementary table can be eliminated. We measured the performance with (OptMag) and without (Mag) this optimization. When the common sub-expression remains, it would be desirable for the system optimizer to automatically decide whether it is cheaper to materialize it or recompute it. However, the version of Starburst on which the experiments were run always recomputes common sub-expressions. Finally, as explained in Section 7, the magic decorrelation algorithm uses the join order of the nested iteration strategy to generate the correlation bindings.

5.2 Parameters

Name	customers	parts	suppliers	partsupp	lineitem
Tuples	15,000	20,000	1,000	80,000	600,000

Table 1: TPC-D Database

In our experiments, we used the TPC-D complex query benchmark database [TPC-D94] of size 120 megabytes. The experiments were run on a IBM RS6000/530H workstation, configured with 40 megabytes of database buffer and 40 megabytes of heap memory. The database tables for our experiments is shown in Table 1, along with the number of tuples in each table. We measured the relative query execution times on three

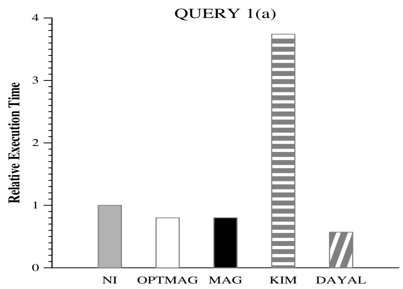

Figure 5: Query 1(a)

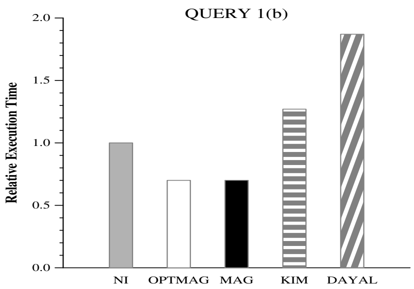

Figure 6: Query 1(b)

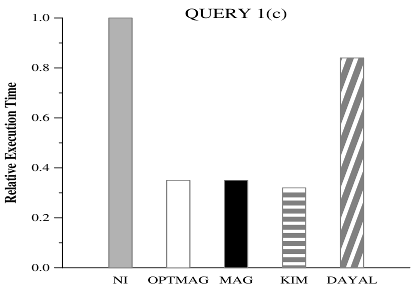

Figure 7: Query 1(c)

sample queries. Each reported measurement is the average of several consecutive runs of the query. The first two queries were directly from the TPC-D benchmark suite (actually, from the version of the TPC-D benchmark of late-1993, when this work was performed; any later modifications to these queries are minor). The third query demonstrates the decorrelation of a non-linear query with duplicates in the correlation column. Indexes were available on all the necessary attributes, except when explicitly dropped to study the stability of the algorithms. None of the queries required the use of a left outer-join during decorrelation. The results are summarized in Figures 5 through 9.

5.3 Analysis

In general, we expect decorrelation to be beneficial when there are many duplicates in the correlation bindings, or when there is considerable work performed in each subquery invocation, or both. We now analyze the three queries, and examine the reasons for the performance differences.

Query (1): This query lists those suppliers that offer the desired type and size of parts in a particular nation at the minimum cost.

```
Select s.s_name, s.s_acctbal, s.s_address, s.s_phone, s.s_comment
From   Parts p, Suppliers s, Partsupp ps
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, Suppliers s1
      Where p.p_partkey=ps1.ps_partkey and
            s1.s_suppkey=ps1.ps_suppkey and s1.s_nation='FRANCE';
```

Figure 5 corresponds to this query executed with all necessary indexes present. For nested iteration, the optimizer chooses a plan that applies the subquery *after* executing the joins in the outer block. There are only 6 invocations of the subquery, and no duplicates in the correlation column. All the same, magic decorrelation performs slightly better than nested iteration. Note that the supplementary table common sub-expression (which is the join of three relations) could not be eliminated in this case, because the correlation attribute (p_partkey) is not a key of the supplementary table. While Kim's method does poorly (it performs unnecessary subquery computation), Dayal's method performs better than magic decorrelation. The difference is because magic decorrelation causes the recomputation of the supplementary table (in this case, a join of three tables). It would be comparable to Dayal's method if the system materialized the common sub-expression instead, especially since in this case, the table contains only six tuples!

To study the sensitivity of the algorithms to variations in the query, we dropped the predicate "p_size = 15" from the outer query block, and changed the predicates on s_nation to "s_region in (AMERICA, EUROPE)". Now, there are 3954 invocations of the subquery, of which only 2138 are distinct. As Figure 6 shows, magic decorrelation continues to perform well. Kim's method starts to do relatively better, because the amount of unnecessary computation is decreased. Dayal's method now performs poorly, because it has to perform a large join before the aggregation, and also performs redundant aggregations.

Finally, we dropped the index on the ps_suppkey column of PartSupp, thereby increasing the work performed in each correlated invocation. Figure 7 shows the results. Now magic decorrelation performs even better compared to nested iteration. Dayal's method is worse once again, because it has to perform a large join before the aggregation. Kim's method performs comparably with magic decorrelation. This is because the cost of recomputation of the supplementary in magic decorrelation balances the extra cost incurred by Kim's method in performing unnecessary subquery computations. If the optimizer were to consider materializing the supplementary table instead of recomputing it, magic decorrelation would perform much better than any of the other algorithms.

Query (2): This query asks for the average yearly loss in revenue if for each part, all orders with a quantity of less than 20% of the average ordered quantity were discarded.

```
Select sum(l_extendedprice*l_quantity)/5
From  Lineitem, Parts p
Wherep_partkey=l_partkey and p_brand='Brand#23' and
        p_container='6 PACK' and l_quantity <
         (Select 0.2*avg(l.l_quantity)
           From Lineitem l Where l.l_partkey=p.p_partkey)
```

When this query is evaluated using nested iteration, the plan optimizer places the subquery *before* the join between Parts and Lineitem. Since the correlation attribute is a key, there are no duplicate invocations of the subquery. In all, there are 209 subquery invocations. However, the subquery itself is very cheap to compute, since there is an index available to perform the selection predicate. This is therefore a case in which we expect decorrelation to have little impact. Note that since the correlation value is a key, the optimization of eliminating the supplementary table can be applied. Magic decorrelation with this optimization performs comparably with nested iteration as shown in Figure 8. Without the optimization, magic decorrelation performs slightly worse. Kim's and Dayal's methods are, however, orders of magnitude worse on this query!

Query (3): This query lists the European suppliers and the sum of balances of those customers who belong to two specific market segments and are in the same country as the supplier.

```
Select s.*, sumbal From Suppliers s, DT(sumbal) AS
  (Select sum(bal) From DDT(bal) AS
```

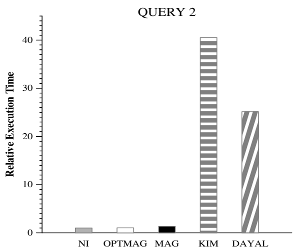

Figure 8: Query 2

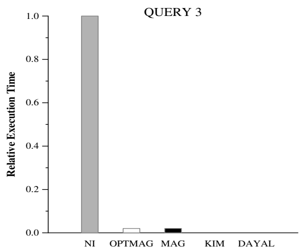

Figure 9: Query 3

((Select a.c_acctbal From Customers a
Where a.c_mktsegment='BUILDING' and
a.c_nation=s.s_nation)
Union (SELECT b.c_acctbal From Customers b
Where b.c_mktsegment='HOUSEHOLD' and
b.c_nation=s.s_nation)))
Where s.s_region='EUROPE';

In this query, the correlation column has duplicate values. Neither Kim's nor Dayal's methods can be applied, since the query is not linear(it has a UNION). Magic decorrelation is applicable, and results in a tremendous performance improvement as shown in Figure 9. The reason is that the correlation column has only 5 unique values. Of the 209 invocations of the subquery during nested iteration, most are therefore redundant. In this example, the subquery computation itself is quite simple. If the subquery had involved a larger amount of work, the effects of the duplicate elimination would be even more significant.

5.4 Performance Summary

We conclude that magic decorrelation is a stable and efficient decorrelation algorithm. In the case where the subquery has a reasonably large computation (Query 1), it performs efficiently, independent of the actual number of correlation bindings. In the case where decorrelation is expected to be unnecessary (Query 2), it does not cause significant degradation in performance. In the case where there are many duplicate values in the correlation column (Query 3), decorrelation greatly improves the execution efficiency. None of the other decorrelation algorithms has these features of stability and efficiency. We should also note that while these queries were run after creating all useful indexes, many ad-hoc queries in the real world do not necessarily have this luxury. Consequently, subquery computation can be very expensive (involving unindexed joins), and the benefits of set-orientation after decorrelation are even more noticeable (as in Query 1 (c)).

6 Decorrelation in Parallel Databases

In centralized database systems, correlated queries evaluated using nested iteration are inefficient due to tuple-at-a-time computation. In shared-nothing parallel database systems, the nested iteration approach results in an added performance penalty, since it inhibits the potential for intra-query parallelism. In this section, we explain the added problems with nested iteration in parallel databases, and the efficiency that results from applying magic decorrelation.

6.1 Inefficiency of Nested Iteration

Let us once again consider our example query of Section 2. Assume that all nodes participate in the parallel execution of the query, and that the Emp and Dept tables are partitioned across all the nodes.

Select D.name From Dept D
Where D.budget < 10000 and D.num_emps >
(Select Count(*) From Emp E Where D.building = E.building)

Consider the scenario where both the tables, Emp and Dept, are partitioned on the attributes involved in the correlation predicate (i.e. the $building$ attribute). All the nodes can execute the query in parallel, with each node computing the portion of the query corresponding to its partition. In this case, parallelism does not reveal any special inefficiency in nested iteration. There are a couple of variants of this scenario which are similar. If the Emp table is small, it can be copied to all nodes. All nodes can execute the query in parallel, as in Case 1, with each node computing the portion corresponding to the partition of the Dept table. A similar execution strategy is possible when the Dept table is small.

If these scenarios do not apply, the evaluation strategy is as follows. For each qualifying Dept tuple at each node, the $building$ attribute is sent to all nodes. Each processor computes a local count and returns it to the requesting node. Upon receiving all the local counts, the requesting node can compute the final count. When complex queries are being processed, this form of nested iteration is the common case, and is very inefficient. Note that our example has a relatively simple subquery computation. When the subquery involves joins, each subquery invocation at any node could cause join computation on all the nodes. This competes for system resources with other such subquery invocations from other nodes, as well as with the computation of the outer query block, which is in progress at all the nodes. In other words, if n is the number of nodes, nested iteration can result in $O(n^2)$ computation fragments. Much of the subquery computation is also often repeated across subquery invocations.

6.2 Magic Decorrelation and Parallelism

Any decorrelation algorithm should prove beneficial in a parallel execution environment. Magic decorrelation is the only algorithm that applies to arbitrary correlated queries, and therefore, it should be of interest to builders of parallel databases. We now demonstrate how the magic decorrelated query (see Section 2.1) would be evaluated in parallel.

The supplementary table Supp_Dept is generated and partitioned across the nodes based on the correlation attribute *Dept.building*. The correlation bindings are also projected to form the Magic table, which is similarly partitioned across the nodes. The projection is performed locally at each node. The decorrelated subquery is then evaluated. This execution can choose a suitable efficient join order and join evaluation strategy. The results of the join are partitioned on the correlation attribute. Note that the GroupBy clause of the subquery is again on the correlation attribute; the aggregation can therefore be performed locally. Finally, the decorrelated subquery is joined with Supp_Dept to produce the answer. Both these tables are already partitioned on their join attribute (i.e. on the correlation attribute). Since there is no coupling between the query blocks, each of the joins can be executed in parallel on all nodes without interference from each other.

7 Related Work

Current database systems attempt to merge subquery blocks into the outer query block, thereby eliminating some correlations[PHH92]. There are many cases where such merging is not possible (usually when the subquery involves aggregation), and the correlation persists. A theoretical approach to decorrelation is described in [Bül87], which extends the relational algebra and calculus to express correlated aggregate subqueries. As we mentioned briefly in Section 2, a special case of magic decorrelation was proposed by Ganski/Wong [GW87]. This method considers a simple outer block consisting of a single table, and a single correlated aggregate subquery. A temporary table similar to a magic table is generated from the single outer block relation. The important step of generating a supplementary table when the outer block is more complex is not considered. The temporary table is then incorporated into the subquery via an outer-join(instead of the more efficient left outer-join). Finally, the method cannot deal with arbitrary correlated SQL queries.

Magic decorrelation is similar in flavor to the magic sets rewriting technique [BR91] which is used to propagate join bindings into subqueries to restrict computation. The primary difference between the two is that while magic rewriting propagates join bindings, magic decorrelation propagates correlation bindings. The importance of magic sets to non-recursive relational systems has been described in [MFPR90]. However that work did not address the issue of correlated queries, assuming instead that the queries would be decorrelated by the prior application of some decorrelation algorithm. Existing decorrelation algorithms deal correctly with only a limited class of queries, and can alter the structure of the query making it difficult to subsequently apply magic sets. In order to appreciate the details of our implementation in Starburst, we direct the reader to [PHH92] which describes the rewrite rule mechanism of Starburst, and [MP94] which describes the implementation of the magic sets rewriting rule.

It is only fair to note that magic decorrelation is a heuristic optimization that is not based on statistical cost estimates. Further like magic sets rewriting, it is dependent on the order of tables chosen. Our implementation simply optimizes the query once without decorrelation, and using the chosen join orders repeats the optimization with decorrelation. The better of the two optimized plans is chosen. We are currently working at a closer integration between cost-based optimization and such heuristic rewrite-based transformations [SHR94].

8 Conclusions

We discussed the issues involved in decorrelation, and the problems with existing approaches. We then presented an abstraction of decorrelation as a basis for developing a magic decorrelation algorithm which correctly handles arbitrarily structured SQL queries, and can be flexibly adapted to work in a variety of system environments. In light of the growing trend towards complex query processing, a generic decorrelation algorithm that works on arbitrary queries is extremely important in centralized as well as parallel databases. The performance improvements that result from magic decorrelation are significant, and this should encourage a commercial RDBMS to incorporate this technique.

Acknowledgments

Praveen Seshadri's research was supported by IBM Research Grant 93-F153900-000. Inderpal Mumick implemented the Extended Magic Sets transformation rule. Michelle Jou and many others at IBM Almaden provided much help understanding the QGM code. Ted Messinger helped run performance numbers. Kurt Brown, Joe Hellerstein, Jeff Naughton, Jignesh Patel, S.Sudarshan and Janet Wiener gave useful suggestions that helped improve the presentation.

References

[BR91] C. Beeri and R. Ramakrishnan. On the Power of Magic. *Journal of Logic Programming*, 10:255, 1991.

[Bül87] G. von Bültingsloewen. Translating and Optimizing SQL Queries having Aggregates. In *VLDB*, pages 235–243, 1987.

[Day87] U. Dayal. Of Nests and Trees: A Unified Approach to Processing Queries that contain Nested Subqueries, Aggregates and Quantifiers. In *VLDB*, pages 197–208, 1987.

[Gra95] J. Gray. A Survey of Parallel Database Systems. Invited Talk, *SIGMOD*, May 1995.

[GW87] R.A. Ganski and K.T. Wong. Optimization of Nested SQL Queries Revisited. In *SIGMOD*, pages 23–33, 1987.

[HCL+90] L. Haas, et al. Starburst Mid-Flight: As the dust clears. *IEEE TKDE*, March 1990.

[Kie84] W. Kiessling. SQL-like and Quel-like Correlation Queries with Aggregates Revisited. Technical Report 84/75, UCB/ERL, September 1984.

[Kim82] W. Kim. On Optimizing an SQL-like Nested Query. *ACM TODS*, 7, September 1982.

[MFPR90] I.S. Mumick, S. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is Relevant. In *SIGMOD*, 1990.

[MP94] I.S. Mumick and H. Pirahesh. Implementation of Magic-Sets in Starburst. In *SIGMOD*, 1994.

[PHH92] H. Pirahesh, J.M. Hellerstein, and W. Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *SIGMOD*, 1992.

[SACLP79] P.G. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, pages 23–34, 1979.

[SHR94] P. Seshadri, J.M. Hellerstein, and R. Ramakrishnan. Filter Joins: Cost-Based Optimization for Magic Sets. Technical Report, Computer Sciences Department, U.W.-Madison, 1995.

[SPL94] P. Seshadri, H. Pirahesh, and T.Y.C. Leung. Decorrelating Complex Queries. Research Report RJ 9846, IBM Almaden Research Center, 1994.

[SQL93] ISO_ANSI. ISO-ANSI Working Draft: Database Language SQL2 and SQL3; X3H2; ISO/IEC JTC1/SC21/WG3. 1993.

[TPC-D94] TPC benchmark group. TPC-D Draft, December 1994. Information Paradigm. Suite 7, 115 North Wahsatch Avenue, Colorado Springs, CO 80903.