Cost-Based Optimization for Magic: Algebra and Implementation

Praveen Seshadri
Univ. of Wisconsin, Madison
praveen@cs.wisc.edu

Joseph M. Hellerstein*
Univ. of California, Berkeley
jmh@cs.berkeley.edu

Hamid Pirahesh
IBM Almaden Research Ctr.
pirahesh@almaden.ibm.com

T. Y. Cliff Leung
IBM Santa Teresa Lab.
cleung@almaden.ibm.com

Raghu Ramakrishnan
Univ. of Wisconsin, Madison
raghu@cs.wisc.edu

Divesh Srivastava
AT&T Research
divesh@research.att.com

Peter J. Stuckey
Univ. of Melbourne
pjs@cs.mu.oz.au

S. Sudarshan†
IIT, Bombay
sudarsha@cse.iitb.ernet.in

Abstract

Magic sets rewriting is a well-known optimization heuristic for complex decision-support queries. There can be many variants of this rewriting even for a single query, which differ greatly in execution performance. We propose cost-based techniques for selecting an efficient variant from the many choices.

Our first contribution is a practical scheme that models magic sets rewriting as a special join method that can be added to any cost-based query optimizer. We derive cost formulas that allow an optimizer to choose the best variant of the rewriting and to decide whether it is beneficial. The order of complexity of the optimization process is preserved by limiting the search space in a reasonable manner. We have implemented this technique in IBM's DB2 C/S V2 database system. Our performance measurements demonstrate that the cost-based magic optimization technique performs well, and that without it, several poor decisions could be made.

Our second contribution is a formal algebraic model of magic sets rewriting, based on an extension of the multiset relational algebra, which cleanly defines the search space and can be used in a rule-based optimizer. We introduce the multiset θ-semijoin operator, and derive equivalence rules involving this operator. We demonstrate that magic sets rewriting for non-recursive SQL queries can be modeled as a sequential composition of these equivalence rules.

1 Introduction

Current relational database systems process complex SQL queries involving views, table expressions and subqueries with aggregate functions. Such queries are becoming increasingly important in decision support applications (see, e.g., the TPC-D benchmark [TPCD94]). The *magic sets rewriting* technique (see, e.g., [BMSU86, RLK86, BR91, MFPR90, SS94]) has been proposed as a heuristic query transformation to optimize such queries, and can result in dramatic improvements in query execution performance [MFPR90]. There can be many possible variants of this rewriting even for a single query, based upon the decisions made with respect to binding propagation. Some of these variants can actually degrade performance. Prior to this work, there has been no demonstrated algorithm to efficiently choose a variant in a cost-based manner. This paper removes an important obstacle to the incorporation of magic sets rewriting into commercial database systems.

This paper explores two approaches to the problem. The first approach is based on modeling magic sets rewriting as a join method, and has been implemented in the DB2 C/S V2 database system. The second approach presents a model of magic sets rewriting based on algebraic transformations. The two approaches are complementary, and together explore the practical and theoretical issues involved.

The goal of the implementation is to develop an algorithm that takes into account the constraints and requirements of a full-function DBMS. Magic sets rewriting is modeled as a special join method that can be added to *any* existing cost-based query optimizer. Cost formulas are derived that allow the optimizer to choose the best variant of the rewriting and determine whether it is beneficial. An exhaustive search of all variants considerably increases the complexity of query optimization. To preserve the order of complexity of the optimization process, reasonable limits are applied on the search space. A performance study based on the implementation in DB2 C/S V2 demonstrates the low additional optimization overhead, and the stability of the algorithm as it makes cost-based choices that result in significant improvements in query execution time. Importantly, the results demonstrate that a cost-based algorithm *is* required for magic sets rewriting (algorithms that are based on heuristics vary in their relative performance as the data statistics and costs change), and that the proposed cost-based algorithm works well.

The algebraic approach to magic sets rewriting is based on equivalence rules, involving the θ-semijoin operator, on the multiset relational algebra. The algebraic approach defines the search space cleanly, and can be used (possibly in conjunction with heuristics to restrict the search space) in a rule-based optimizer. We present a representative collection of equivalence rules, and show how these rules model magic sets rewriting for non-recursive SQL queries.

We initially motivate the problem with an example.

*This work was performed while the author was at the University of Wisconsin, Madison.

†This work was performed while the author was at AT&T Bell Laboratories, Murray Hill.

View Definition

```
CREATE VIEW DepAvgSal AS
  (SELECT E.did, AVG(E.sal) AS avgsal FROM Emp E
  GROUPBY E.did);
```

Main Query Block

```
SELECT E.eid, E.sal FROM Emp E, Dept D, DepAvgSal V
WHERE E.did = D.did AND E.did = V.did AND E.age < 30
   AND D.budget > 100,000 AND E.sal > V.avgsal
```

Figure 1: Original Query

View Definitions

```
CREATE VIEW PartialResult AS
  (SELECT E.eid, E.sal, E.did FROM Emp E, Dept D
  WHERE E.did = D.did AND E.age < 30 AND
     D.budget > 100,000)
CREATE VIEW Filter AS
  (SELECT DISTINCT P.did FROM PartialResult P );
CREATE VIEW LimitedDepAvgSal AS
  (SELECT F.did, AVG(E.sal) as avgsal FROM Filter F, Emp E
  WHERE E.did = F.did
  GROUPBY F.did);
```

Main Query Block

```
SELECT P.eid, P.sal FROM PartialResult P, LimitedDepAvgSal V
WHERE P.did = V.did AND P.sal > V.avgsal
```

Figure 2: Magic Sets Rewriting

2 Motivation

The SQL query in Figure 1 finds every young employee in a big department whose salary is higher than the average salary in that department. The query involves a relational view DepAvgSal that derives the average salary in each department, and a join between the Emp, Dept and DepAvgSal relations. Magic sets rewriting exploits the fact that the average departmental salary need not be computed for every department; it need only be computed for those departments that are big and have young employees. If there are few such departments, it is probably desirable to apply magic sets rewriting.[1] The rewritten query is shown in Figure 2. The PartialResult view represents the partial computation in the main query block at the stage where the Emp and Dept tables already have been joined together, but the view DepAvgSal has yet to be joined to them. From this PartialResult table, a duplicate-free Filter view is created, which is a set of all those departments for which the average salary needs to be computed. This filter set is now used to limit the computation in the original view.[2] The view is modified by the inclusion of an equi-join with the filter set (thereby limiting the computation in the view to the departments of interest). Finally in the main query block of Figure 2, the modified view is joined with the PartialResult table to produce the answer.

2.1 Rewriting Choices

In the rewritten query shown, the filter set contains all departments which are big and have young employees. This is the most restrictive filter set possible. A less restrictive filter set can be used instead. The filter set can contain all the big departments, or all the departments with young employees. In each of these cases, the rewritten queries will be different from that shown in Figure 2, but will have a similar overall structure, and return the same answers. While these options may result in more computation inside the view, they could be cheaper overall (because the PartialResult table or the Filter table is cheaper to compute). In general, there are many ways in which the filter set could be created, each corresponding to some subset of the tables in the FROM clause that results in the PartialResult relation. If every department is big and has young employees, rewritten queries provide no improvement over the original query, and may even be more expensive to execute. Finally, when there are multiple join attributes, a decision needs to be made if all the join attributes will contribute to the filter set, or whether only some of the attributes will be used. However, in the vast majority of queries, there is exactly one join attribute, so this is not usually an important issue.

The specific combination of choices made with respect to computing the filter set has been called the "sideways information passing strategy"(SIPS), so named because the filter set passes the join attributes "sideways" into the view definition. One specific SIPS results in the best execution plan for a query. However, this depends on the tables and predicates involved in the query, and the characteristics of the execution environment. No practical solution currently exists for choosing the SIPS in a cost-based manner. Instead, existing systems that perform magic sets rewriting have chosen one of two approaches:

- Use magic sets rewriting with a default SIPS (typically "left-to-right") and allow the user to specify a different SIPS or disable magic sets rewriting. This approach is used in CORAL [RSSS94].
- Independently optimize the query with and without magic sets rewriting and choose the cheaper plan. For magic sets rewriting, choose a SIPS based on some heuristic. This approach is used in Starburst [MP94]. The SIPS chosen "corresponds" to the join order that arises from optimizing the original query without magic rewriting. No cost-based justification has been presented for this heuristic, nor is there any guarantee that a good plan is chosen. In fact, we show in Section 6 that this heuristic can make some poor choices. However, since the original query is also independently optimized, one can ensure that performance is not degraded due to magic sets rewriting.

2.2 A Cost-Based Solution

This paper presents a cost-based solution to the problem of choosing an appropriate SIPS. We implemented our solution in the DB2 C/S V2 database system (which is based on the Starburst system [HCL+90]) which has support for magic sets

[1] While there have been many flavors of magic sets rewriting proposed, the most practical one in the RDBMS context is the supplementary magic sets rewriting used in this paper.

[2] In the literature, the filter set and the PartialResult have been called the "magic" set and the "supplementary" respectively.

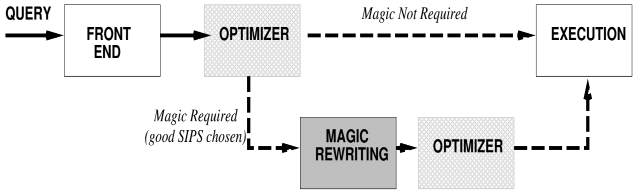

Figure 3: Optimization Architecture

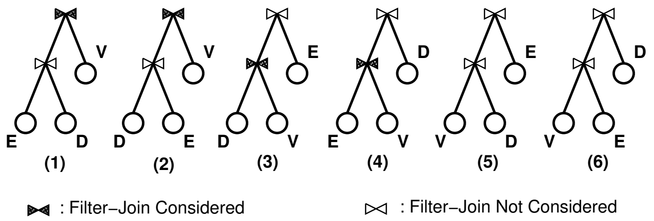

Figure 4: Some Possible Join Orders

rewriting as a query-to-query transformation. The system architecture used is shown in Figure 3. The user's query is fed directly into the cost-based query optimizer, which decides whether or not to perform magic rewriting. While the optimizer explores the space of possible join orders and methods, it also explores the space of possible options for magic sets rewriting. If the decision is that no rewriting is needed, the optimizer generates an execution plan as usual and sends it on to the execution engine. On the other hand, if the optimizer decides that magic rewriting is needed, it also chooses one specific SIPS for the rewriting, which guides the application of the magic sets transformation. Once the query is rewritten, it has to be optimized again to generate an execution plan. The Starburst solution discussed above [MP94] was the first to suggest the two-pass architecture, and we use the same idea since we would like to avoid major changes to the existing system components. An alternative approach, which we discuss in Section 7, is to implement magic rewriting via algebraic transformations (instead of as an SQL-to-SQL rewriting).

3 Magic Sets and Join Optimization

In this section, we describe how the magic sets rewriting choices can be explored as part of the join optimization phase of a query optimizer.

3.1 Primer on Join Optimization

A query optimizer determines an efficient order in which to execute the joins of N relations, and the actual join method to use for each join. Since joins are associative and commutative, there is a large space of $O((2(N-1))!/(N-1)!)$ possible join orders [GHK92]. Since this is a prohibitively large space to explore for even a small value of N, most practical join optimization algorithms [SAC+79, IK84, KBZ86] explore limited regions of it. All the algorithms have one common feature: at each step, they consider various two-way joins, and for each join, they consider the cost of applying various join methods.

3.2 Join Ordering and SIPS

We now explain the correspondence between magic sets rewriting and join ordering. Let us consider the original query of Figure 1. Six possible choices of join order for joining Emp E, Dept D, and DepAvgSal V are shown pictorially in Figure 4; selection predicates have been omitted for conciseness.

First consider plans 1 and 2; the join of E and D is used as the outer relation in the final join with the view V (which is shaded in the figure, since it is the operator of interest). How does this relate to magic sets rewriting? In the example of Figure 2, the join of E and D is used as the PartialResult table from which the magic set for view V is materialized. There is a correspondence between the composite outer relation in the join plan and the PartialResult table used in the magic rewriting. Therefore, plans 1 and 2 "correspond" to this particular variant of magic sets rewriting. Similarly, plan 3 "corresponds" to a magic sets rewriting which uses only the Dept relation D as the PartialResult table. Plan 4 "corresponds" to a rewriting which uses only the Emp relation E as the PartialResult table. Finally, plans 5 and 6 "correspond" to the original query (i.e. magic sets rewriting is not performed).

We exploit this correspondence to explore the exponential space of choices of the SIPS for magic sets rewriting. We propose to piggy-back the exploration of the magic sets rewriting choices onto join optimization by modeling magic rewriting as a join method, thereby permitting our approach to be incorporated into *any* cost-based join optimization algorithm without major modifications to the optimizer code. The primary change to the optimizer is the addition of one new join method and this affects the complexity of optimization by only a constant factor.

3.3 Magic Sets Rewriting as a Join Method

We define the *Filter-join* of relations R and S as follows:

Definition 3.1 (R Filter-Join S) R is called the *outer* relation and S is called the *inner* relation in the Filter-join. A (duplicate-free) superset of values of the join attribute of R is created, and is used as a filter to restrict the tuples of S that are accessed. The restricted relation of S tuples is then joined with relation R (using any other available join algorithm). □

This join method is similar to the well-known semi-join [BGW+81] operation, and in fact, this similarity is exploited in Section 7 when modeling magic sets using a θ-semijoin operator. The important distinction is that semi-joins have usually been applied to stored relations, while magic sets rewriting works on views.

Assume that a query optimizer augmented to consider the Filter-join as a join method is invoked on a join query involving N relations. At some intermediate stage, it evaluates the cost of a particular join. The outer relation is a composite relation of the form $(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_{(k-1)})$, and the inner is a single relation R_k, for which magic sets rewriting can be applied (i.e., R_k is a view). The smallest filter set would result from the (duplicate-free) projection of the entire composite outer relation. Less restrictive filter sets could be created by using the join of any subset of the tables in the composite outer relation as the PartialResult. Once some specific choice of the PartialResult is made, the filter set itself can be represented exactly, or in a lossy fashion (i.e., some superset of the filter set can be used instead) by omitting some join attributes.[3]

There are many different choices for the PartialResult, and for the filter set. As we showed in the example of Figure 2, the filter set is used to restrict the inner relation by adding it to the FROM clause of the inner query block. Even after choosing some PartialResult and some filter set, the modified version of the inner relational view (LimitedDepAvgSal in Figure 2) needs to be planned. Clearly, if these choices are explored for every possible join involving R_k as the inner relation, we will have explored all possible SIPS combinations. However, we are unwilling to compromise on optimizer complexity for the sake of optimizing magic sets rewriting. This implies that if we propose to explore the possible choices for each Filter-join, it must be done in constant time. Therefore, our next task is to limit the search space to some tractable size.

3.4 Limiting the Search Space

The space of options for one particular Filter-join is large because of three reasons:

1. There are many possible choices for the PartialResult . In general, if there are k-1 relations joined to form the composite outer relation, any of the $2^{(k-1)}-1$ non-empty subsets of them could be used as the PartialResult .

2. Given a particular PartialResult , there are many possible choices for the filter set. In general, if there are m join attributes, any of the $2^m - 1$ non-empty subsets of them could be used as the filter set. Further, there could be several implementations of the filter set (for example, as a relation or as a Bloom filter).

3. Given a particular filter set, there is a large space of possible plans for the inner relational view modified by the addition of the filter set.

Points (1) and (2) give rise to the full range of SIPS discussed in [BR91]. We adopt two well-known and widely used optimizer techniques when faced with huge search spaces: (a) We apply heuristic limitations on the search space for Filter-joins. Hopefully, most of the search space omitted due to the heuristics is not of interest. (b) We make assumptions that allow us to use reasonably accurate cost "guess-timates" for parts of the search space, instead of actually exploring those parts and computing more accurate estimates.

Heuristic 1: *The PartialResult must be the complete outer relation.*

Heuristic 2: *Some small and constant number of filter set implementations will be considered.*

Therefore, the choices for the PartialResult and the filter set can be made in constant time. Finally, we make the following assumption, which will be justified in the next section.

Assumption 1: *The cost and result cardinality of the Filter-join can be estimated in constant time.*

[3] Another way of introducing lossiness is by using a Bloom filter [Blo70] to implement the filter set.

$JoinCost_P$	Cost of performing the joins required to generate PartialResult P.
$ProductionCost_P$	Cost of materializing PartialResult P.
$ProjCost_F$	Cost of projecting P to generate the filter set F.
$FilterCost_{R_k}$	Cost of generating R_k and restricting it using the filter set F.
$FinalJoinCost$	Cost of performing the final join of the outer relation and and $R_{k'}$.

Table 1: Cost Components of a Filter-Join

If assumption 1 holds, then there is no change in the order of complexity of join optimization, although Filter-join is being considered as an option. For each particular join considered, the Filter-join method examines only one PartialResult, a small constant number of filter sets and determines the cost of the Filter-join in constant time. The entire query is optimized, and the cheapest complete plan is examined. If it contains no Filter-join, then magic sets rewriting should not be applied; otherwise it should be applied using the SIPS specified by the composite outer relation of the Filter-join.

4 Cost and Cardinality Estimation

We now derive a formula to capture the costs of the Filter-join method. Note that this cost formula must be evaluated in constant time. Assume that the Filter-join whose cost is being estimated has $(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_{(k-1)})$ as the outer relation and relation R_k as the inner relation. Because of the limits imposed on the search space, the PartialResult is simply the outer relation. The join evaluation cost may be broken up into the components shown in Table 1 and explained below. The total cost of the Filter-join is the sum of these cost components.

$$JoinCost_P + ProductionCost_P + ProjCost_F + FilterCost_{R_k} + FinalJoinCost$$

$JoinCost_P$**:** This is the cost of the outer relation, which is already computed as part of the optimization algorithm.

$ProductionCost_P$**:** P needs to be materialized because it is used both in the generation of the filter set, and also in the top-level join. The cost of materializing the PartialResult is a simple function of the cardinality of P. Since this cardinality is known (i.e., already estimated by the optimizer), the materialization cost may be computed. Instead of creating a temporary relation, P could also be recomputed. The cost of recomputation of P is the same as $JoinCost_P$. Whichever cost is lower (materialization or recomputation) is chosen as $ProductionCost_P$.

$ProjCost_F$**:** The cost of performing a (duplicate-free) projection of P to generate filter set F depends on the cardinality of P. It also can depend on physical properties of the plan for P (for example, whether P is sorted or not) and whether the projection can be combined with the generation of P. The optimizer has all the necessary information to make an estimate of the $ProjCost_F$.

$FilterCost_{R_k}$**:** This is the cost of generating the filtered version of R_k using the filter set F (let the filtered relation be called $R_{k'}$). The method for computing the cost of $R_{k'}$ and its cardinality is left for discussion at the end of these definitions.

$FinalJoinCost$: This is easy to compute. The cardinality of the outer relation is known. The cardinality of the filtered inner relation has just been computed. The cost formulas of other available join methods can be applied to determine the cheapest way to execute the final join.

Note that all (except one) of these cost components can be computed in constant time using well-known cost formulas that existing cost-based optimizers already implement. The next section shows how the cost of the filtered R_k relation($FilterCost_{R_k}$) and its cardinality can be estimated in constant time.

4.1 Estimating $FilterCost_{R_k}$

In order to model magic sets rewriting as a Filter-join and obtain an estimate of its cost, we do not need to explicitly plan the modified inner view. We only need to estimate the cardinality of the modified view and the *cost* of the best plan to generate it. The actual plan is not needed at this stage, because the optimization architecture presented in Section 2.2 requires a second pass through the optimizer after magic rewriting is performed. The extent of the filtering effect of the filter set (i.e., its selectivity) depends on its cardinality. While it is difficult to estimate the cardinality of projections accurately [LNSS93], existing optimizers do use some assumptions to estimate projection cardinality [Yao77].

What is needed is a *parameterized* plan for the restriction of R_k, whose parameter is the filter set. Further, we would like to be able to generate the parameterized plan just once. Each specific plan is obtained by instantiating the parameterized plan with a specific filter set. The plan instance would provide the cost as well as the cardinality of the result. While parameterized query optimization has been the topic of recent research [INSS92], the results are too preliminary to apply to our problem. We need a concrete technique to deal with parameterization in our specific context. A trivial solution is to perform a nested invocation of the query optimizer for each plan instantiation. However, the time for each instantiation needs to be a small constant, and since R_k can be a complex expression involving several joins, this is not feasible.

We observe that the cardinality of the filtered inner relation $R_{k'}$ is a function of only the *selectivity* of the filter set (which is known), and does not depend on the filter set's physical size or implementation. Once the cardinality of $R_{k'}$ has been computed for a few values of the selectivity of F, a line can be fitted to them, thereby defining the $R_{k'}$ cardinality function for all filter sets. In our implementation, in fact, we chose to compute exactly two points and perform a straight line interpolation. The chosen points had selectivities of 0 (where the result cardinality is obviously 0) and 1 (where the result cardinality is that of the unmodified view R_k).

Estimating the cost of $R_{k'}$ is more complicated, because join cost functions may be non-linear especially at size boundaries when a relation no longer fits in the memory available for some operation. Therefore, a simple straight line approximation method may not be accurate. One approach is to identify a few equivalence classes based on the size of the magic set (for example, "smaller than buffer" and "larger than buffer") and to use a straight line approximation within each class. We should note that in practice, filter sets are typically small, because they contain a (duplicate-free) projection of only the join column. A simple straight line approximation may therefore perform adequately; in fact, our prototype implementation used exactly this technique.

4.2 Space Complexity of Optimization

It should be evident why space complexity is not an issue in this work. The optimization in the main query block now has to try an extra join method for every join considered. However, it does not require that any extra plans be *stored*. Therefore, there is no change to the order of space complexity due to considering an extra join method. With respect to the "parameterized" planning of the complex view, our approach is to optimize the filtered version of the complex view for a small constant number of equivalence classes. Therefore, this causes no change in the order of space complexity either.

4.3 How did the Complexity Disappear?

While there are an exponential number of possible rewritings, we managed to combine the search for the best plan into one invocation of the optimizer to find the best SIPS and another invocation after the rewriting to optimize the query rewritten based on that SIPS. It is important to understand how the original complexity disappeared; there is no "magic" involved! Section 3.4 imposed certain limits on the search space explored. For instance, based on our example in Section 2, the cost of the following magic rewriting choice will not be estimated: Dept is used as the PartialResult table, and the filtered version of the view LimitedDepAvgSal is first joined with Emp before being joined with Dept. The other technique used was to make a "good estimate" of the cost of the parameterized plans, rather than compute them explicitly. While our estimates are admittedly approximate, they are far superior to no estimate at all (which is the current state of the art with respect to algorithms like magic sets). In a nutshell, we claim that using these techniques is very likely to improve the results of today's query optimizers. An implementation in the DB2 C/S V2 database system provides the empirical evidence to back up this claim.

5 Implementation in DB2 C/S V2

An implementation of the algorithm presented in the earlier sections is needed to answer the following questions: (1) How feasible is it to incorporate such an algorithm into a real database system? (2) Does the algorithm really find good plans (i.e., does it perform as expected)?

5.1 Feasibility

We prototyped the proposed cost-based optimization in an IBM internal version of the DB2 C/S V2 database system. While the query optimizer does support various search strategies, we focused on the strategy that uses the well-known left-deep dynamic programming optimization algorithm. By choosing a full-fledged database system, we were forced to face all the practical constraints that exist in a real DBMS. One of us worked on DB2 C/S V2 to prototype the optimization algorithm for magic sets rewriting. Despite being initially unfamiliar with the optimizer code, the modifications as well as performance measurements were completed within 3 months. While changes were required in many parts of the optimizer, the actual number of lines of C++ code added was well under 1000. We believe that this validates the feasibility of adding the Filter-join method to existing optimizers.

5.2 Performance

We studied the performance of our algorithm using controlled experiments described in detail in the next section. The primary conclusion is that the results firmly support our expectations. *Cost-based optimization prevents magic sets rewriting from being chosen when it should not, and chooses one of the best variants when magic sets rewriting should be chosen.* Further, these positive results are obtained without changing the order of complexity of optimization.

5.3 Practical Experience

We discovered that magic sets rewriting acts as a "stress-test" for the query optimizer. This is because it uses common subexpressions, duplicate-free projections, temporary relations and other features which are usually used only in complex decision support queries. Therefore, while cost estimation for magic sets rewriting is certainly reasonable at a logical level, it is important that the optimizer correctly model the constructs that the rewriting uses.

6 Performance Measurement

The objectives of our performance study were (1) To show that various SIPS choices exist for magic rewriting, and that no particular SIPS choice is optimal across all queries and execution environments. (2) To show that cost-based magic optimization makes a choice that is close to optimal with different queries and with different execution environments. (3) To show that the additional overhead due to cost-based magic optimization does not affect the order of complexity of the optimization process.

Unfortunately, there is no "standard" benchmark to evaluate a technique like magic sets rewriting. Instead, we had to devise an experimental methodology for this purpose. The experiments we chose had to be relatively simple to understand and explain. Further, we had to be able to explore the various dimensions of magic sets rewriting in a comprehensibly small number of experiments. The next section describes our attempt at devising such a methodology.

6.1 Experimental Methodology

The TPC benchmark D is an industry-wide standard benchmark for complex queries [TPCD94]. There are two queries in the benchmark to which magic sets rewriting can be applied (Query 2 and Query 17). We were interested in Query 2 because it has a large number of variants for magic rewriting (it involves the join of many relations), whereas the other candidate (Query 17) has only one possible variant.

In the first experiment, TPC-D Query 2 is the starting point. Selection predicates in the outer query-block were gradually varied by either removing them, modifying them to be less selective, or replacing them with less selective predicates. The effect of this is that the query answer cardinality gradually increased. However, the actual tables in the query were not changed, so that the same SIPS choices were always available. As the predicates change, different SIPS choices become optimal and sometimes it is better not to perform magic rewriting at all. The query answer cardinality is crudely related to the possible filtering effect due to magic sets; consequently we expect that magic sets rewriting should have greater benefits when the answer cardinality is smaller. The optimizer enhanced with cost-based magic optimization should always choose the right SIPS for all queries.

We repeated the entire experiment after modifying the initial query used as the starting point (by changing the contents of the inner query block). Therefore, we were able to explore the effects on a variety of queries in a controlled and understandable manner. We also repeated the experiment after dropping some of the indexes used in the original plans; this represents a change in the execution environment. Therefore, the execution costs of various portions of the queries change. Cost-based magic optimization should be able to detect this, and make the right decision in the new environment.

The experiments were run on an IBM RS/6000 workstation connected to two disks. A 100MB TPC-D database (i.e. with scale factor of 0.1 as defined by the TPC-D standard) was generated with all appropriate indexes available. The graphs of the results plot the variations in the queries on the X-axis, ordered by increasing answer size. The primary performance metric used is query execution time. Note that the graphs for execution time use a logarithmic scale, so seemingly small differences are really quite significant. We also measure query compilation time to demonstrate the overhead paid for the extra work in the optimizer. All times reported have been uniformly scaled by a fudge factor.

6.2 Algorithms Compared

In our experiments, we compared the following algorithms: no magic sets rewriting (**nomag**), cost-based optimization based on Filter-joins (**magopt**), and some of the possible "hand-chosen" pre-determined variants of magic sets rewriting (**mag1, mag2, mag3, mag4, mag5**). The pre-determined variants represent some reasonable SIPS choices. For each query, the **magopt** algorithm should choose the best SIPS in a cost-based manner, or choose to not perform the rewriting.

Recall from Section 2.1 that there was a heuristic approach originally proposed in Starburst [MP94]. This approach first optimizes queries without any knowledge of magic sets rewriting, and derives the SIPS from the resulting join order. The drawback in this approach is that nowhere in the estimation are the true costs of magic rewriting computed. By examining the plans generated by the optimizer when magic rewriting was disabled (**nomag**), we were able to manually reconstruct how the Starburst approach would perform. We call this algorithm **sbmag**.

6.3 Overall Results

The graph in Figure 5 shows the overall performance results. The Filter-join based optimization of magic sets rewriting (i.e.**magopt**) can greatly improve execution performance when compared to **nomag**. This happens when the answer size is small, because the filter set is small and reasonably selective. This limits the computation in the complex view, and hence reduces the execution time. However, as the filter set gets larger, its advantages slowly diminish while its cost increases, so that at the last point in the graph (size 11986), the

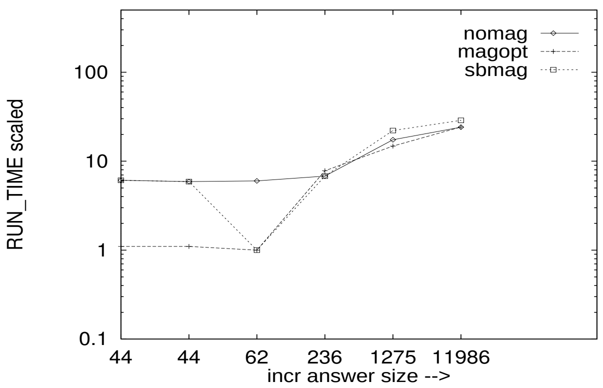

Figure 5: Overall Comparison

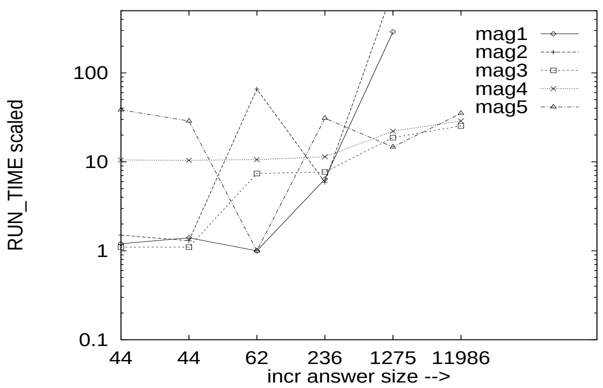

Figure 6: Magic Sets Rewriting Choices

algorithm chooses not to perform magic sets rewriting. This explains why the last point is the same for both **nomag** and **magopt**. The **sbmag** algorithm performs well for the larger queries, but performs poorly for the smallest two queries. In order to understand the performance of **sbmag**, consider the fact that the complex view is treated by the optimizer as a temporary relation without indexes. When the other joins in the query are very selective and the other relations are indexed, the cheapest plan often places the view at the beginning of the join order. Based on such plans, **sbmag** decides not to invoke magic sets. However, these are exactly the queries for which magic is expected to be most useful!

The fourth query (size 236) in the first graph presents an interesting issue for discussion. We notice that the performance of **magopt** is slightly worse than that of **nomag**. Obviously, the optimizer expected the chosen magic rewriting for **magopt** to perform better than **nomag**. This shows that the optimizer's estimate of costs does differ from the actual cost due to inaccurate statistics and/or assumptions. Algorithms like cost-based magic optimization that build on top of these errors may occasionally show sub-optimal performance.

6.4 Fixed SIPS Choices

The graph in Figure 6 shows the performance of various fixed choices of SIPS for magic sets rewriting. The existing state of the art requires the database user to pick one such choice for each query. As the graph shows, the best choice

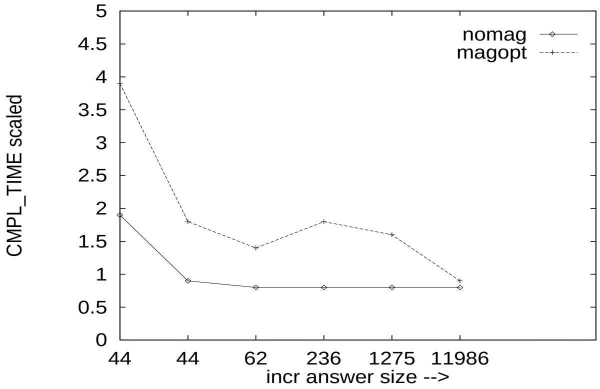

Figure 7: Total Compile Time

differs dramatically from query to query, though the only real difference between the queries is in the predicates on the underlying tables. Comparing the first and the second graphs, it is evident that **magopt** finds close to the best choice for each query, thereby forming the lower envelope of the available options. *This is the single most important performance result presented in this paper*, and demonstrates both the motivation for our work, and the success of our approach.

6.5 Compilation Time Results

The graph in Figure 7 shows the total compilation (rewriting + optimization) time due to the use of **magopt** and **nomag**. This time includes the second pass through the optimizer when magic sets rewriting is chosen (note that this is a linear scale, not a logarithmic scale). For the smaller queries, magic sets rewriting is chosen, and therefore the compilation time for **magopt** is approximately twice the time for **nomag**. We view this as an acceptable overhead, because in decision support queries, the compilation time is usually small compared to the execution time.

For the largest query, magic sets rewriting is not chosen, and so there is no second pass through the optimizer. As the graph shows, there is very little overhead for having considered the option of using magic. This is important in those cases where the optimization overhead may be significant. The graph in Figure 8 shows this more clearly; it plots the time used during the first pass of the optimizer, when Filter-joins are being considered by **magopt**. While **magopt** considers more plans, because it also considers plans involving a Filter-join, the time taken to do so is not very large. This performance result validates our claim that *cost-based magic optimization does not alter the order of complexity of query optimization*.

6.6 Experimental Variations

In separate experiments, we changed the complex view so that it was more expensive, and repeated the entire experimental variation of the outer query block. A couple of variations of the query execution environment (dropping some indexes) were also tried. The results are very similar to those shown, and changed only in the absolute numbers but not in their relative positions. Due to space constraints, we have not shown graphs of the results. In all the experiments, **magopt** made close to the best optimization choices. Based on this study, we observe that *the Filter-join based optimization*

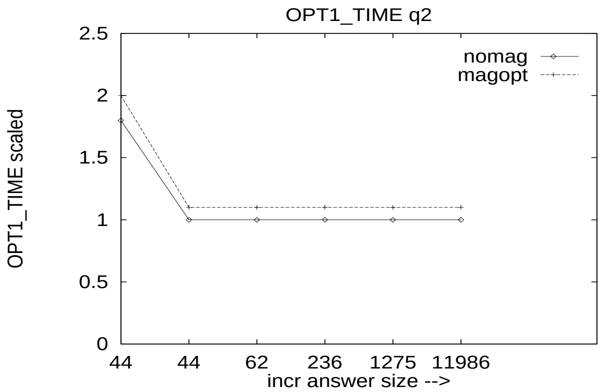

Figure 8: Initial Optimizer Overhead

technique is stable; its success is not dependent on the specific nature of the complex view, the nature of the query block in which the view is used, nor the existence of indexes.

7 The θ-Semijoin Algebra

In our implementation, we explored the options for magic sets rewriting by "piggybacking" them onto existing mechanisms in the optimizer. If the system were based on an algebraic rule-based optimizer, such as Volcano [GM93], another approach is to extend the query algebra to model magic sets as well. In this section, we present such an algebraic extension that helps to characterize the exact space of possible options. The algebraic equivalences lead to query transformations that can be applied in a cost-based manner, possibly using the cost estimation techniques already presented, or using a more elaborate exploration of the search space.

7.1 Notation

We use the symbols R (with or without subscripts) to denote relations, θ (with or without subscripts) to denote predicates, E (with or without subscripts) to denote relational expressions, $attrs(E)$ to denote the attributes of the result of E, and $\bar{a}$ to denote a tuple of attributes. In keeping with SQL semantics, we treat relations as multisets of tuples, and use the multiset version of relational algebra [DGK82].

We assume familiarity with the algebra, and only describe the grouping/aggregation operator and the multiset θ-semijoin operator. We use a grouping/aggregation operator ${}_{\bar{g}}\mathcal{F}_{\bar{f}}$, where $\bar{g}$ denotes the groupby attributes and $\bar{f}$ denotes the aggregate operations performed on the groups defined by the groupby attributes. (This notation is borrowed from [EN94, Klu82], and is required to deal with SQL-style grouping and aggregation.)

Definition 7.1 (Multiset θ-Semijoin) The multiset version of the θ-semijoin operator, $\ltimes_\theta$, is defined as follows. Given relations R_1 and R_2,

$$(R_1 \ltimes_\theta R_2) \stackrel{\text{def}}{=} \sigma_{\exists \bar{t}_2 \in R_2, \theta(\bar{t}_2)}(R_1)$$

where $\theta(\bar{t}_2)$ denotes the predicate θ with attributes of R_2 replaced by their values from tuple $\bar{t}_2$. $\square$

The definition of θ-semijoin preserves the multiset semantics, i.e., for each tuple present in the result, the multiplicity is exactly the same as in R_1. For example, if the relation $R_1(A, B)$ is the multiset of tuples $\{(1,2),(1,2),(1,4)\}$, and $R_2(C, D)$ is $\{(3,5),(3,6),(3,7)\}$, then $R_1 \ltimes_{C \geq B} R_2 = \{(1,2),(1,2)\}$.

In the multiset relational algebra, θ-semijoin is a derived operator, and can be expressed using the θ-join, projection (π) and duplicate elimination (δ) operators as follows:[4]

$$(R_1 \ltimes_\theta R_2) \equiv (R_1 \bowtie_{Nat} (\delta(\pi_{attrs(R_1)}(R_1 \bowtie_\theta R_2))))$$

where $\bowtie_{Nat}$ denotes natural join. The intuition is that the expression $\pi_{attrs(R_1)}(R_1 \bowtie_\theta R_2)$ contains all the tuples in the θ-semijoin, but possibly with incorrect multiplicities. The duplicate elimination followed by the natural join is used to recover the desired multiplicity.

Some of the θ-semijoin equivalence rules we describe make use of functional dependencies present in relations; we denote the functional dependencies present in a relation R by $FD(R)$. In addition, the equivalence rules also make use of functional dependencies implied by predicates (such as θ-join or θ-semijoin predicates). For example, the predicate $x = y * y$ implies the functional dependency $\{y\} \rightarrow x$, and the predicate $x = y + z$ implies the functional dependencies $\{y, z\} \rightarrow x$, $\{x, y\} \rightarrow z$, and $\{x, z\} \rightarrow y$. We use the notation $FD(\theta)$ to denote the set of all functional dependencies implied by predicate θ.

7.2 Transformation Rules for θ-Semijoin

Optimizing SQL queries by making use of θ-semijoins involves specifying equivalence rules involving θ-semijoins, and other operators of the extended multiset relational algebra. Given a collection of equivalence rules, a transformational optimizer, like Volcano [GM93], can be used to enumerate and compactly represent the logical search space. Cost formulas for the operators in the algebra are used to compute cost estimates and choose the optimal way of evaluating the query.

We discuss some of the more interesting transformations involving the θ-semijoin and θ-join operators below; others are presented in [SSS95]. Algebraic transformations may require a renaming step, for example when changing the structure of the expressions. Like other work in the area, our equivalence rules ignore the renaming step, for simplicity of exposition; details of renaming can be worked out easily.

Introduction of θ-Semijoin: Relational algebra expressions generated directly from SQL queries typically do not contain the θ-semijoin operator.[5] We illustrate below how the θ-semijoin operator can be introduced into expressions with joins; similar equivalence rules can be derived for outerjoins, intersections and differences.

$$E_1 \bowtie_\theta E_2 \equiv E_1 \bowtie_\theta (E_2 \ltimes_\theta E_1)$$

The intuition here is that the result of the expression $E_1 \bowtie_\theta E_2$ only makes use of tuples of E_2 that join with tuples of E_1 on the predicate θ. Hence, first selecting the subset

[4] In the set version of relational algebra, θ-semijoin can be more simply expressed as $(R_1 \ltimes_\theta R_2) \equiv \pi_{attrs(R_1)}(R_1 \bowtie_\theta R_2)$. This equivalence does not hold in the multiset version of relational algebra.

[5] The translation from SQL to relational algebra presented in [CG85] uses θ-semijoins only to handle HAVING clauses.

of the tuples of E_2 that join with E_1 on θ, using the θ-semijoin operator, prior to performing the join with E_1, would preserve equivalence. Note that common subexpressions are introduced as a result of this step.

Pushing θ-Semijoin through Join: We present below a transformation rule that describes how to push θ-semijoins through joins.

$$(E_1 \bowtie_{\theta_1} E_2) \ltimes_{\theta_2} E_3 \equiv (E_1 \bowtie_{\theta_1} E_2') \ltimes_{\theta_2} E_3$$

where $E_2' = E_2 \ltimes_{\theta_1 \wedge \theta_2} (E_1 \bowtie_{True} E_3)$.

This transformation allows both E_3 and E_1 to be used to restrict the tuples computed for E_2'. Although as stated the transformation uses a cross-product, it is useful if some part of θ_2 uses only attributes from E_1 and E_3 — that part of θ_2 can be used in a succeeding step to convert the cross-product to a θ-join. The intuition behind the correctness of the transformation rule is that only those tuples $\bar{t}_2 \in E_2$ are required for which there exist tuples $\bar{t}_1 \in E_1$ and $\bar{t}_3 \in E_3$ for which $\theta_1(\bar{t}_1, \bar{t}_2)$ is true and $\theta_2(\bar{t}_1, \bar{t}_2, \bar{t}_3)$ is true.

Pushing θ-Semijoin through Aggregation: The following transformation rule describes how to push θ-semijoins through grouping and aggregation.

$${}_{\bar{g}}\mathcal{F}_{\bar{f}}(E_1) \ltimes_{\theta} E_2 \equiv {}_{\bar{g}}\mathcal{F}_{\bar{f}}(E_1 \ltimes_{\theta} E_2)$$

where θ involves only the attributes in $\bar{g}$ and $attrs(E_2)$.

The intuition here is that if the semijoin predicate θ involves only the attributes in $\bar{g}$ and $attrs(E_2)$, for each group of E_1, either all the tuples will be selected by $(E_1 \ltimes_{\theta} E_2)$, or none will. The tuple in the result of the $\mathcal{F}$ operator generated from each group will correspondingly be selected or not.

When θ involves results of the aggregation, the θ-semijoin operator cannot be pushed through aggregation in general. In some cases involving min and max, it *is* possible to push the θ-semijoin operator through ${}_{\bar{g}}\mathcal{F}_{\bar{f}}$; see [SSS95].

Simplification: Some of the θ-semijoin transformations can generate expressions where some predicates are checked more than once; for example, in the right hand side of the transformation above that introduces the θ-semijoin, the predicate θ is checked twice. The repeated checks are necessary in general, but in some special cases the repeated checks are redundant, and the expressions can be simplified by removing them. The following transformation can be used to eliminate repeated checks, when it is applicable.

$$E_1 \bowtie_{\theta_1 \wedge \theta_2 \wedge \theta_3} (E_2 \ltimes_{\theta_1 \wedge \theta_2 \wedge \theta_4} E_1) \equiv$$
$$E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \ltimes_{\theta_1 \wedge \theta_2 \wedge \theta_4} E_1)$$

where $attrs(E_2)$ functionally determine all the attributes in θ_2, under $FD(\theta_1) \cup FD(E_1)$.

Note that it is safe to use a subset of the functional dependencies in performing this transformation, so a *complete* mechanism for deducing functional dependencies is not essential.

Eliminating θ-Semijoin: Intuitively, a θ-semijoin can be rewritten as a join followed by a projection if the join predicate along with the functional dependencies of the right operand of the θ-semijoin guarantee that each tuple of the left operand is selected by at most one tuple of the right operand. This intuition is formally captured by the transformation shown below:

$$E_1 \ltimes_{(E_2.\bar{y} = \bar{g}(attrs(E_1))) \wedge \theta_1} E_2 \equiv$$
$$\pi_{attrs(E_1)}(E_1 \bowtie_{(E_2.\bar{y} = \bar{g}(attrs(E_1))) \wedge \theta_1} E_2)$$

where $E_2.\bar{y}$ is a superkey of E_2, and $\bar{g}(attrs(E_1))$ is a function of attributes of E_1 that returns a tuple of values with the same arity as $E_2.\bar{y}$.

We have presented a representative sample of the equivalence rules involving the θ-semijoin operator. A larger collection of equivalence rules is presented in [SSS95].

7.3 Cost Model for θ-Semijoin

The θ-semijoin operation $R_1 \ltimes_{\theta} R_2$ can be efficiently implemented using minor changes to join techniques such as hash joins and index joins. One implementation treats the left operand R_1 of the θ-semijoin as the "outer" relation in the join technique. For each tuple in the outer relation R_1, instead of joining it with *each* matching tuple in the inner relation R_2, the tuple in R_1 can be returned as soon as a match is found. Sort-merge joins can similarly be adapted to implement θ-semijoins if the join predicate is an equijoin.

An alternative is to treat the right operand R_2 of the θ-semijoin as the "outer" relation in the join technique. For each tuple in the outer relation R_2, all matching tuples in the inner relation R_1 are returned. If a tuple in R_1 is already in the result as a consequence of matching a *different* R_2 tuple, it is not added to the result; an efficient implementation requires an index on the result of the θ-semijoin, in general. When the θ-semijoin predicate involves an equijoin with a superkey of R_2, it is guaranteed that a tuple in R_1 matches at most one tuple in R_2; no index on the result of the θ-semijoin is required in this case.

The cost formulas for the different join techniques are easily modified to derive cost formulas for the different ways of implementing the θ-semijoin operator. We omit details. The costing phase of a transformational optimizer, such as Volcano, uses cost formulas for the operators in the algebra to compute cost estimates for the different ways of evaluating the query.

Since the θ-semijoin operator is a derived operator (it can be expressed using θ-join, projection and duplicate elimination), it can also be implemented using algorithms for these operators. However, implementing it thus is inefficient.

7.4 Applications of θ-Semijoin Equivalence Rules

Rewriting using θ-semijoin equivalence rules is particularly useful for complex queries that use views that cannot be expanded out into a join. Examples include:

- Queries using views that are defined using aggregation. Such a query was presented in Section 2.
- Queries using views that use SELECT DISTINCT in the view definition. Such views can be expanded out in some, but not all, cases.

- Queries with outerjoins. Views used as arguments of outerjoins cannot be expanded out. Since outerjoins are not associative, they cannot be reordered.

A distinct advantage of rule-based optimizers is that it is easy to specify the transformations for a variety of operations. In fact, the set of transformation rules we present in [SSS95] are able to push θ-semijoins into the views in all the above mentioned examples.

If the entire set of θ-semijoin equivalence rules were added to an exhaustive optimizer, the search space could increase greatly. The above classes of queries represent the most useful classes of applications of the θ-semijoin equivalence rules, and suggest the following very useful heuristic:

> The θ-semijoin operator should be introduced only when dealing with aggregation, duplicate elimination and outerjoin operations.

An equivalence rule for aggregation was presented earlier, while others are described in [SSS95].

7.5 θ-Semijoin and Constraint Magic Rewriting

Magic rewritings [BR91, MP94, SS94] optimize database queries by defining a set of auxiliary *magic* (or *filter*) relations, that are used as filters to restrict computation of the query. Constraint Magic rewriting [SS94] is the most general of these rewritings, and we present a derived transformation rule that captures the key intuition of Constraint Magic rewriting for a single join. Applying this transformation rule repeatedly on a sequence of joins has an effect similar to Constraint Magic rewriting, for a single block SQL query. Finally we indicate a heuristic for applying the method to SQL queries that use view relations in addition to database relations; the heuristic simulates the behavior of Constraint Magic rewriting (with full left-to-right SIPS) on such queries.

CM Transformation Step Using θ-Semijoin: The following transformation rule captures the basic step of Constraint Magic rewriting [SS94]:

$$(E_1 \bowtie_{\theta_1} E_2) \ltimes_{\theta_2 \wedge \theta_3} F \equiv (E_1' \bowtie_{\theta_1} E_2') \ltimes_{\theta_2 \wedge \theta_3} F$$

where θ_2 involves only the attributes in $attrs(F) \cup attrs(E_1)$,[6] and E_1' and E_2' are defined below:

$$E_1' = E_1 \ltimes_{\theta_2} F$$
$$E_2' = E_2 \ltimes_{\theta_1 \wedge \theta_3} (E_1' \bowtie_{\theta_2} F)$$

We refer to the above transformation as the *Constraint Magic Transformation (CMT) Step*. The CMT step can be derived from simpler equivalence rules; details are presented in [SSS95].

CMT Step and Constraint Magic Rewriting: The expressions defining E_1' and E_2' in the CMT step capture the essence of Constraint Magic rewriting. The intuition is as follows. Suppose we have a set of bindings F on the result of a join of relations E_1 and E_2. In Constraint Magic rewriting of this join, the "filter" relation (also called the "magic" relation) F is first used to restrict computation of E_1 to tuples relevant to F. Then, the set of E_1 tuples thus restricted are used along with the filter relation F to restrict computation of E_2. This strategy is exactly captured in the CMT step.

More formally, the connection can be established as follows. Consider a view defined as:

$$V \stackrel{\text{Vdef}}{=} (E_1 \bowtie_{\theta_1} E_2)$$

with a filter relation F, and a parametrized predicate $\theta_2 \wedge \theta_3$ where θ_2 involves only the attributes in $attrs(F) \cup attrs(E_1)$. (This is the same as the LHS of the CMT step.) The Supplementary Constraint Magic rewriting first defines supplementary (or PartialResult) relation S_1 below:[7]

$$S_1 \stackrel{\text{Vdef}}{=} F \bowtie_{\theta_2} E_1''$$

where E_1'' is the result of supplementary Constraint Magic rewriting of E_1 with the filter relation F and the parametrized predicate θ_2.[8] View V is then replaced by view V' defined below:

$$V' \stackrel{\text{Vdef}}{=} S_1 \bowtie_{\theta_1 \wedge \theta_3} E_2''$$

where E_2'' is the result of the supplementary Constraint Magic rewriting on E_2, with the filter relation S_1, and the parametrized predicate $\theta_1 \wedge \theta_3$.

Note the close correspondence between E_1' (in CMT) and E_1'' (in Constraint Magic), between the right operand of the θ-semijoin in the definition of E_2' (in CMT) and S_1 (in Constraint Magic), and between E_2' and E_2''. The main difference between the Constraint Magic rewriting and the CMT step on a single join is that Constraint Magic rewriting uses θ-joins rather than θ-semijoins. Although the final expression using θ-semijoin is more complex than the definition of V' generated by Constraint Magic rewriting, the added complexity is required to preserve the multiset semantics.

CM Transformation of an SQL Block Using θ-Semijoin: The algebraic expression V generated by transforming a single block SQL query is of the form:

$$V : \pi_{\bar{pa}}({}_{\bar{ga}}\mathcal{F}_{\bar{af}}(\ldots(R_1 \bowtie_{\theta_1} R_2)\ldots \bowtie_{\theta_{n-1}} R_n))$$

Given a filter relation F on V, denoted by $V \ltimes_{\psi} F$, the following sequence of transformations can be applied to $V \ltimes_{\psi} F$. First, identify the strongest subset of ψ, denoted by ψ_n, that can be pushed through the groupby/aggregation operator. If the original query did not use GROUPBY, ψ_n is the same as ψ. Then, $\ltimes_{\psi_n} F$ can be pushed inside the projection operator, and the groupby/aggregation operator, to obtain:

$$\pi_{\bar{pa}}({}_{\bar{ga}}\mathcal{F}_{\bar{af}}((\ldots(R_1 \bowtie_{\theta_1} R_2)\ldots \bowtie_{\theta_{n-1}} R_n) \ltimes_{\psi_n} F))$$

Finally, the CMT step can be repeatedly applied on the expression

[6] Note that this applicability condition can always be satisfied by choosing θ_2 to be True.

[7] Assuming that E_1 is chosen as the first relation in the sideways information passing (SIP) order.

[8] Actually, the projection of F on the relevant attributes is used in Constraint Magic rewriting, but we use F itself for simplicity of exposition. The projection can be introduced after carrying out the CMT step.

$$((\ldots(R_1 \bowtie_{\theta_1} R_2)\ldots \bowtie_{\theta_{n-1}} R_n) \ltimes_{\psi_n} F)$$

as described below. First define $S_i, i \geq 1$ as follows:

$$S_1 \stackrel{\text{def}}{=} R_1$$
$$S_{i+1} \stackrel{\text{def}}{=} (S_i \bowtie_{\theta_i} R_{i+1}), i \geq 1$$

Also, let $\psi_i, i < n$ denote the strongest subset of ψ_{i+1} that uses only attributes of F and S_i, and $\gamma_i, i < n$ denote the rest of ψ_{i+1}. The first application of the CMT step transforms $(S_{n-1} \bowtie_{\theta_{n-1}} R_n) \ltimes_{\psi_n} F$ to $(S'_{n-1} \bowtie_{\theta_{n-1}} R'_n) \ltimes_{\psi_n} F$, where $S'_{n-1} = (S_{n-1} \ltimes_{\psi_{n-1}} F)$ and $R'_n = R_n \ltimes_{\theta_{n-1} \wedge \gamma_{n-1}} (F \bowtie_{\psi_{n-1}} S'_{n-1})$.

Now, consider S'_{n-1}; the θ-semijoin can be pushed into the definition of S_{n-1} in exactly the same manner as above. Thus the CMT step is applied on each $S_i, n \geq i \geq 2$. Note that there are two occurrences of S'_{n-1}, i.e., it is a common subexpression of two expressions. By using labeled expressions we can avoid the cost of optimizing and evaluating the expression twice. Using labeled expressions is very important to avoid an exponential blow up as we go down from S_n to S_1.

CM Transformation of SQL Queries With Views: We start from an SQL query block, and perform the θ-semijoin transformation of the block as described above. This block may contain uses of view relations, and after the transformation the use of a relation R_i may have a semijoin of the form $R_i \ltimes_{\beta_i} F_i$, or $R_i \ltimes_{\beta_i} (F \bowtie_{\psi_{n-1}} S'_{n-1})$. Let E_i denote the entire semijoin expression involving R_i. If R_i is a view relation, a specialized version R'_i of the view definition of R_i, with the semijoin pushed into it, can be created recursively using the θ-semijoin transformation of the SQL block defining R_i. Finally, if all of β_i can be pushed into the view definition of R_i, then E_i is replaced by R'_i, else only R_i in E_i is replaced by R'_i.

The relationship between the CMT step and Constraint Magic rewriting discussed earlier for a single join also carries over to the case of views, and to queries defined using multiple views.

We have thus shown that, for SQL queries, the effect of Constraint Magic rewriting is obtained as a special case of the θ-semijoin transformations, in particular by using the CMT step. If we explore the full space of equivalent expressions, Constraint Magic rewriting will be examined as an option, and the cheapest expression in the search space will be chosen.

7.6 Discussion

Several commercial database systems use unique tuple-ids as implicit attributes in order to distinguish between multiple occurrences of a tuple. They carry out transformations of SQL queries that could affect the multiset semantics of the query (such as replacing nested subqueries by joins), and use unique tuple-ids to get the correct multiset semantics, if required, as follows. The tuple-ids of relations in the FROM clause (i.e., those that contribute to the multiset semantics) are selected along with other attributes in the SELECT clause, and duplicate elimination is performed on the resultant relation. Finally the tuple-id attributes are projected away to get the desired multiset result.

This approach has the benefit of using joins in place of θ-semijoins, allowing more join reorderings to be explored. However, it has several drawbacks. First, it cannot be used in association with grouping/aggregation. Second, the optimizer has to keep track of the tuple-ids across operations and perform duplicate elimination. Our approach of using the θ-semijoin operator is cleaner since it can uniformly deal with SQL queries that have multiset semantics, as well as queries that have set semantics. Our approach also avoids the costs of explicit duplicate elimination and of maintaining and dealing with unique tuple-ids.

8 Related Work

Magic sets rewriting was originally used in the area of recursive query processing in deductive databases [BMSU86, RLK86]. The impact of different choices of SIPS has been discussed in [BR91], and the idea of using approximations of the magic set has been explored in [Sag90, SS88]. We should note that this paper deals with *non-recursive* SQL queries that are supported by all commercial relational database systems. Magic sets has been shown to be applicable to non-recursive SQL queries [MFPR90], and has been implemented in the Starburst database system [MP94].

Cost-based optimization techniques similar to those for magic sets may also be applied to complex SQL queries involving correlation (using the magic decorrelation transformation [SPL96]), and expensive functions [Hel95].

The research on semijoins in distributed databases (e.g., [BGW+81, LMH+85]) assumed that relations were simple stored relations, and therefore the costs of performing the semijoins could be easily computed. Further, issues like the choice of SIPS were not considered, usually because communication costs were assumed to outweigh local processing costs (consequently, the chosen semijoin was always as restrictive as possible). Instead, optimization focused on the correct order in which to execute the semijoins [BGW+81]. System R* [LMH+85], on the other hand, assumed that local processing costs outweigh communication costs; consequently, semijoins were not considered during optimization.

The literature on heterogeneous databases has not yet dealt with issues like remote views in a complex query. However, such issues are becoming increasingly important, and our work should be applicable to this domain as well.

9 Conclusions

This paper makes two major contributions. First, it proposes a practical solution for integrating the magic sets rewriting algorithm into a cost-based optimization framework. The solution ensures that the rewriting is applied in an effective manner, and only when it is beneficial. The implementation of the optimization technique in DB2 C/S V2 demonstrates the feasibility of the proposed solution. Further, the performance results validate the claim that magic rewriting can be effectively optimized in a cost-based manner for a wide range of queries without significant optimization overhead. The second contribution is the formalization of these ideas by introducing the multiset θ-semijoin algebraic operator. A representative collection of algebraic equivalences involving this operator have been presented, which can be used to model magic rewriting. The algebraic characterization cleanly defines the entire space of evaluation options. Further, the transformation rules can be used by a rule-based optimizer to optimize complex queries in a cost-based fashion. While these contributions were originally the result of two independent research efforts, they are complementary in nature. Together, they address the theory and practice of applying the magic sets rewriting optimization in a cost-based manner.

Acknowledgements

Praveen, Joey, Raghu, Hamid and Cliff acknowledge the contributions of John McPherson, Guy Lohman, Beau Shekita, Dave Simmen, Lori Strain, Monica Urata, and Surendra Verma at IBM Almaden, Navin Kabra and Jignesh Patel at University of Wisconsin, and Inderpal Mumick for the original magic rewriting code. Divesh, Peter and Sudarshan would like to thank Brian Hart for discussions that lead them to start thinking of how Constraint Magic rewriting might be implemented using generalizations of semijoins, Bill McKenna for discussions regarding cost-based algebraic optimizers and feedback on the paper, and Tim Griffin for discussions about bag operators.

Praveen Seshadri was supported by an IBM Graduate Student Fellowship. Raghu Ramakrishnan was supported by a David and Lucile Packard Foundation Fellowship in Science and Engineering, a Presidential Young Investigator Award, with matching grants from Digital Equipment Corporation, Tandem and Xerox, and NSF grant IRI-9011563. Joseph Hellerstein was supported by NSF grant IRI-9157357.

References

[BGW+81] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve and J. B. Rothnie. Query processing in a system for distributed databases (SDD-1) *ACM Transactions on Database Systems*, 6(4):602–625, 1981.

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv and J. D. Ullman. Magic sets and other strange ways to execute logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1–15, 1986.

[BR91] C. Beeri and R. Ramakrishnan. On the power of Magic. *Journal of Logic Programming*, 10(3&4):255–300, 1991.

[Blo70] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[CG85] S. Ceri and G. Gottlob. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE Transactions on Software Engineering*, 11(4):324–345, 1985.

[DGK82] U. Dayal, N. Goodman, and R. H. Katz. An extended relational algebra with control over duplicate elimination. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1982.

[EN94] R. Elmasri and S. B. Navathe. *Fundamentals of database systems*. Benjamin/Cummings Publishers, 2nd edition, 1994.

[GHK92] S. Ganguly, W. Hasan and R. Krishnamurthy. Query optimization for parallel execution. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1992.

[GM93] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the IEEE International Conference on Data Engineering*, 1993.

[HCL+90] L. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[Hel95] J. M. Hellerstein. Optimization and execution techniques for queries with expensive methods Ph.D. Thesis, University of Wisconsin, August 1995.

[IK84] T. Ibaraki and T. Kameda. Optimal nesting for computing N-relational joins. In *ACM Transactions on Database Systems*, 9(3):482–502, 1984.

[INSS92] Y. Ioannidis, R. Ng, K. Shim and T. K. Sellis. Parametric query optimization. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 103–114, 1992.

[KBZ86] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 128–137, 1986.

[Klu82] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM*, 29(3):699–717, 1982.

[LMH+85] G. M. Lohman, C. Mohan, L. M. Haas, D. Daniels, B. G. Lindsay, P. G. Selinger and P. F. Wilms. Query processing in R*. In *Query Processing in Database Systems*, (W. Kim, D. S. Reiner, and D. S. Batory, eds.), Springer-Verlag, 30–47, 1985.

[LNSS93] R. J. Lipton, J. F. Naughton, D. A. Schneider and S. Seshadri. Efficient sampling strategies for relational database operations. *Theoretical Computer Science*, 116:195–226, 1993.

[MFPR90] I. S. Mumick, S. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is relevant. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1990.

[MP94] I. S. Mumick and H. Pirahesh. Implementation of magic-sets in a relational database system. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1994.

[RLK86] J. Rohmer, R. Lescoeur, and J. M. Kerisit. The Alexander method: A technique for the processing of recursive axioms in deductive databases. In *New Generation Computing*, 4(3):273–285, 1986.

[RSSS94] R. Ramakrishnan, D. Srivastava, S. Sudarshan and P. Seshadri. The CORAL deductive system. *The VLDB Journal, Special Issue on Prototypes of Deductive Database Systems*, 1994.

[SAC+79] P. G. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 23–34, 1979.

[Sag90] Y. Sagiv. Is there anything better than magic? In *Proceedings of the North American Conference on Logic Programming*, 235–254, 1990.

[SPL96] P. Seshadri, H. Pirahesh, and T. Y. C. Leung. Decorrelating complex queries. In *Proceedings of the Twelfth International Conference on Data Engineering*, 1996.

[SS88] S. Sippu and E. Soisalon-Soinen. An optimization strategy for recursive queries in logic databases. In *Proceedings of the Fourth International Conference on Data Engineering*, 1988.

[SS94] P. J. Stuckey and S. Sudarshan. Compiling query constraints. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1994.

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

[TPCD94] TPC benchmark group. TPC-D Draft, December 1994. Information Paradigm. Suite 7, 115 North Wahsatch Avenue, Colorado Springs, CO 80903.

[Yao77] S. B. Yao. Approximating the number of accesses in database organizations. *Communications of the ACM*, 20(4):260–261, 1977.