Implementation of Magic-sets in a Relational Database System

Inderpal Singh Mumick*
AT&T Bell Laboratories
mumick@research.att.com

Hamid Pirahesh
IBM Almaden Research Center
pirahesh@ibm.com

Abstract

We describe the implementation of the magic-sets transformation in the Starburst extensible relational database system. To our knowledge this is the first implementation of the magic-sets transformation in a relational database system. The Starburst implementation has many novel features that make our implementation especially interesting to database practitioners (in addition to database researchers). (1) We use a cost-based heuristic for *determining join orders* (sips) before applying magic. (2) We push all equality and *non-equality* predicates using magic, replacing traditional predicate pushdown optimizations. (3) We apply magic to *full SQL* with duplicates, aggregation, null values, and subqueries. (4) We *integrate* magic with other relational optimization techniques. (5) The implementation is *extensible*.

Our implementation demonstrates the feasibility of the magic-sets transformation for commercial relational systems, and provides a mechanism to implement magic as an integral part of a new database system, or as an add-on to an existing database system.

1 Introduction

Magic-sets [BMSU86, BR91, Ram88, Mum91] is a query-rewrite optimization algorithm for recursive and nonrecursive queries written in Datalog or SQL. The magic-sets transformation has been implemented in deductive database systems for optimizing recursive queries: Coral [RSS92] implements magic templates [Ram88], Aditi [VRK+90], EDS [FF93], $\mathcal{LDL}$ [NT88], NAIL [MNS+87], and Glue-Nail [DMP93] implement the supplementary magic-sets transformation [BR91]. However, the deductive implementations have ignored several aspects critical to commercial relational systems — join ordering, predicates other than equality, integration with traditional optimizations, existential and universal quantification, subqueries, nonrecursive queries, (strict) adherence to SQL semantics, and, in one case, the need to maintain ground tuples while working with predicates other than equality.

*Part of the work of this author was done at IBM Almaden Research Center and Stanford University.

Motivation:

EXAMPLE 1.1 Let employee be a stored relation with attributes `empno`, `empname`, `workdept`, and `salary`, amongst others. Let department be a stored relation with attributes `deptno` and `deptname`, amongst others. The query

```
(D0): SELECT d.deptname, s.workdept, s.avgsalary
      FROM department d, avgMgrSal s
      WHERE d.deptno = s.workdept AND
            d.deptname = 'Planning'.
```

defines a query table containing the average salary of all the managers in the department with name 'Planning'. The complete query includes the definition of two views: avgMgrSal that computes the average salary of all managers in each department, and mgrSal that contains information on employees who are managers (We have dropped the `CREATE VIEW` keywords for brevity).

```
(D1): avgMgrSal(workdept, avgsalary) AS
        (SELECT workdept, AVG (salary)
         FROM mgrSal
         GROUPBY workdept).

(D2): mgrSal(empno, empname, workdept, salary) AS
        (SELECT e.empno, e.empname, e.workdept, e.salary
         FROM employee e, department d
         WHERE e.empno = d.mgrno).
```

The query graph for query D, (Figure 1, left side) shows the flow of information amongst the query blocks. department and employee are used to compute mgrSal, and the result is grouped upon to compute avgMgrSal. The `GROUPBY` view $D1$ is shown by a triplet of boxes –

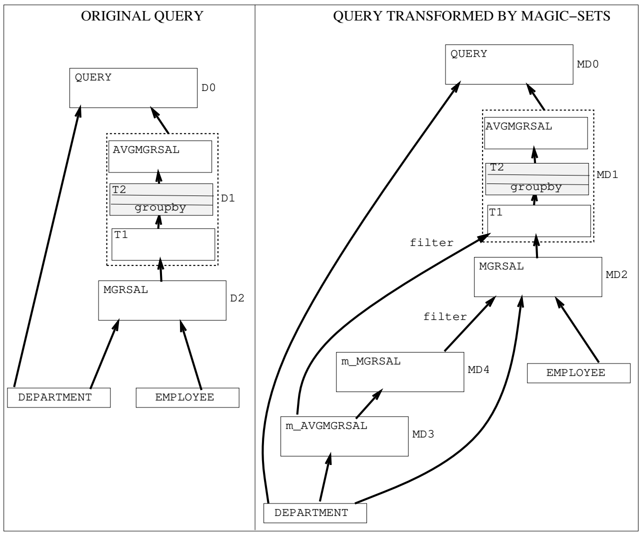

Figure 1: Magic Transformation introduces more joins, but leads to better performance.

the first box represents the **SELECT FROM WHERE** clause of the view $D1$, the second box represents the actual **GROUPBY** operation, and the third box accommodates a possible selection due to the **HAVING** clause (the reasons for having a triplet should not concern the reader here). department and avgMgrSal are then joined in the query block.

The right side of Figure 1 shows the flow of information if the magic-sets transformation was applied to this query graph. The department information can be used, through two other *magic* views, to limit the computation of the mgrSal view. Since only the 'Planning' department is of interest to the query, the magic views will cause mgrSal to be computed only for the managers in the 'Planning' dept. Clearly, this is a big reduction in the size of mgrSal, so it will be computed faster. Further, the grouping operation now groups a smaller relation, and is more efficient. The query block also works with smaller inputs, and can be evaluated faster than in the original query graph.

The transformed query graph is more complex - it has more query blocks, and more joins. However, the graph in Figure 1 does not represent the end of query-rewrite in Starburst. The graph is further optimized, using traditional relational optimizations, to generate a final equivalent query graph that has only one extra box, and only one extra join (Figure 4, discussed later). The additional join is very inexpensive, and our performance experiments (Table 1, Experiment G) show that the transformed query executes two and a half orders of magnitude faster. This example illustrates the savings that can be achieved using magic-sets, and the importance of integrating magic-sets with other query optimization techniques. □

We have done performance experiments on large benchmark data on IBM'S DB2 database, and seen that queries transformed by our extended magic-sets transformation can execute orders of magnitude faster. Some of the experiments were reported in [MFPR90a]. Results of those and other experiments are summarized in Table 1. The experiments compare the execution times

	Elapsed Time		
Query	Original	Correlated	EMST
Exp A	100.00	0.40	0.47
Exp B	100.00	2.12	0.28
Exp C	100.00	513.27	50.24
Exp D	100.00	5136.49	109.00
Exp E	100.00	52.56	7.62
Exp F	100.00	0.54	0.84
Exp G	100.00	2.41	0.49
Exp H	100.00	19.91	4.46

of nonrecursive queries rewritten using the extended magic-sets transformation (EMST), whose implementation we describe in this paper, and correlation (a leading optimization technique for complex SQL queries), and show that EMST is a far more stable optimization than correlation. The experiments thus show that the magic-sets transformation is invaluable for optimization of nonrecursive SQL queries, particularly for complex queries such as decision-support queries.

Deductive implementations don't satisfy the needs of a commercial relational implementation: The magic-sets transformation has been implemented in deductive databases, but is not popular in relational database systems for nonrecursive queries. There are several reasons for this. First, deductive database theory and implementations have advocated that the magic-sets transformation is useful only for recursive queries, and that too only for non-linear recursive queries. Non-linear recursive queries do occur in applications, but are not common enough to demand extensive optimization priority; besides most relational systems do not support recursive queries any way. Second, the magic-sets transformation can rewrite a nonrecursive query into a recursive query, making it inapplicable in most relational systems that do not support recursion. Third, commercial RDBMSs usually have a complex query language (such as SQL), with features and semantics different from Datalog and other logic languages. For example, SQL has existential and universal quantifiers, nested subqueries, correlation, duplicates, aggregation, ordering, and nulls. The magic-sets transformation must strictly adhere to the semantics of SQL (or another complex query language). The deductive database implementations do not comply with the complex *non-logical* features. Fourth, it is not known how magic-sets transformation interacts with other optimization techniques. What is the relationship between predicate pushdown, decorrelation, and magic? How does one determine the join-order information needed for magic? In a commercial relational database, an understanding of the interaction with other traditional optimizations is critical for magic to be useful. Fifth, as Example 1.1 showed, the magic-sets transformation produces a complex query graph with many extra boxes and joins. Deductive database implementations of magic-sets do not optimize the graph any further. Lastly, optimizers in commercial RDBMSs have a complex implementation, and it is not easy to integrate a complex optimization like magic-sets transformation into an existing optimizer. No effort has been devoted to considering issues related to integrating the magic-sets transformation into a relational optimizer (EDS [FF93] being a recent and welcome exception).

In this paper, we give a detailed account of our implementation and explain our solutions to the above problems with implementing magic-sets in a real system: how to integrate magic-sets transformation with other optimization techniques, how to use magic as a general and uniform predicate pushdown technique in SQL, how to determine join orders, how to adorn queries, how to be extensible when new functionality is added to the database, how to reduce the complexity of queries produced by magic-sets, and how to handle complex language features such as aggregation. The implementation effort was modest – about 6 person-months of coding work.

This paper is organized as follows. Section 2 defines some terms and notation we will use in the rest of the paper. The Starburst query-rewrite architecture is presented in Section 3. We also discuss how the magic-sets transformation fits into the query-rewrite and the cost based join optimization phases. Section 4 describes the implementation of EMST as a rule in the Starburst query-rewrite system. The design decisions that make the implementation extensible are explained out in Section 5. Related work is discussed in Section 6, and Section 7 summarizes our contributions.

2 Preliminaries

In Starburst SQL [MPR90], a query defines one or more views, some of which may be recursive, and designates one of the tables as the query table. A view definition can be a single `SELECT` statement, of the form

(Q): `SELECT ... FROM ...`
`[WHERE ...] [GROUPBY ...] [HAVING ...]` .

Alternatively, a view may be defined as a `UNION`, `INTERSECTION`, and `EXCEPT` of multiple `SELECT` statements. A subquery q in an SQL statement also defines a view v. This view v is just like any other view in the query, except that its definition often depends upon the enclosing SQL statement.

A single select statement, of the form Q above will be called a **block**. The SQL code defining a view will be called a **blob**. A blob can consist of a single block, or a union, difference and intersection of blocks. Since a subquery is treated as a view, it has its own blob, which may contain a single block, or a union, difference and intersection of blocks.

EXAMPLE 2.1 Consider the query and views defined in Example 1.1. The query consisting of statements $D_0, D_1, \ldots$ is called query D. The SQL code defining view avgMgrSal in statement $D1$ is the *blob*. This blob consists of a single block, containing the code of the `SELECT FROM GROUPBY` statement. The blobs $D0$ and $D2$ each consist of a single block, containing the code of the corresponding `SELECT FROM WHERE` statement. □

Stratum Numbers: Given a query, assign stratum numbers to blobs as follows: Construct a dependency graph with blobs as nodes, and an edge from blob U to blob V if table U appears in the `FROM` clause of blob V. Construct an acyclic *reduced dependency graph* by collapsing every strongly connected component of the dependency graph to a single node (applicable only if the original program has recursion). A topological sort of the reduced dependency graph assigns a stratum number to each node. If a node represents a strongly connected component, all blobs in the strongly connected component are assigned the stratum number of the node. By convention, the base tables are assigned stratum number = 0.

The Query Graph Model

In Starburst, a query is internally represented by a query graph in the Query Graph Model (QGM) [PHH92]. The query graph is made of one or more QGM boxes. A QGM box is roughly equivalent to a single `SELECT` statement (block), or to a single Datalog rule. A QGM box represents a unit of evaluation, such as a join and select operation (select-box), a grouping and aggregation operation (groupby-box), a union operation (union-box), a difference operation (difference-box), an intersection operation (intersection-box), and so on.

EXAMPLE 2.2 Figure 1 shows the QGM graph for query D. Boxes labeled DEPARTMENT and EMPLOYEE represent the stored relations department and employee. The box labeled MGRSAL is a select-box defining the view mgrSal. The view avgMgrSal is split between three boxes – box $T1$ is a select-box representing the code "T1 `AS (SELECT` $\star$ `FROM` mgrSal)", T2 is a groupby-box representing the code "T2 `AS (GROUPBY workdept, AVG (salary) FROM` T1)", and AVGMGRSAL is the select-box for "avgMgrSal `AS (SELECT` $\star$ `FROM` T2)". □

We formalize the relationship between a QGM box and an SQL block. In general, a block is made up of one or more boxes. A block without a `GROUPBY` clause consists of a single *select-box*. A block with a `GROUPBY` clause is decomposed into three boxes, called a *groupby-triplet*, so that no selections get mixed up with the grouping operation. The groupby-triplet consists of a *select-box* implementing the `SELECT-FROM-WHERE` clause (box $T1$ in Figure 1), a *groupby-box* implementing the `GROUPBY` clause (box $T2$ in Figure 1), and another *select-box* implementing the selection given by the `HAVING` clause (box labeled AVGMGRSAL in Figure 1).

Inside each QGM box is a mini-graph representing the corresponding SQL statement. The vertices of the mini-graph are called *quantifiers*. There is one quantifier for each table referenced in the `FROM` clause or in the subqueries of the SQL statement. For instance, given view $D1$ for mgrSal, the box for MGRSAL has quantifiers e and d for the tables EMPLOYEE and DEPARTMENT. Arcs between boxes in Figure 1 represent tables referenced by quantifiers in the boxes. For example, the arcs from boxes AVGMGRSAL and DEPARTMENT to box QUERY mean that box QUERY contains quantifiers ranging over boxes AVGMGRSAL and DEPARTMENT, as should be expected since the SQL block $D0$ of the QUERY box computes a join of tables avgMgrSal and department. Thus, an out-edge from a box B represents a usage of the table computed by box B, and an in-edge into a box C represents a table referenced by box C. QGM represents predicates in the `WHERE` clause by edges between quantifiers. Correlation predicates are represented by edges between quantifiers in different boxes.

When the output of a QGM box is used multiple times (*e.g.*, a view may be used multiple times in the same query), common subexpressions are created. Recursive queries create cycles in QGM. The number of boxes in the query graph determines the complexity of the query; the greater the number of boxes, greater the complexity of the query, and greater the cost of optimization. However, a query is not necessarily more expensive to evaluate simply because it has more QGM boxes; indeed, queries produceed by the magic-sets transformation have more boxes, and can be evaluated more efficiently.

Magic-sets transformation

The magic-sets transformation [BR91, Mum91] optimizes database queries by defining a set of auxiliary *magic tables* that that are used as filters to restrict computation of the SQL query. For example, since the query $D0$ asks for only the avgMgrSal tuples that are in 'Planning' dept, the computation of the views avgMgrSal and mgrSal can be filtered so as to exclude all other departments. In Figure 1, magic tables m_avgMgrSal and m_mgrSal are defined to contain deptno of only the 'Planning' department. The filtering is then done by adding the magic tables to the `FROM` clause, and equijoin predicates to the `WHERE` clause, of each SQL statement. Figure 1 illustrates the filtering graphically. The supplementary magic-sets transformation [BR91] is a variant of the magic-sets transformation where *supplementary magic tables* are created as common subexpressions to avoid duplicating work. The ground magic-sets transformation [MFPR90b] is an extension of the magic sets idea to push down non-equality predicates in addition to equality predicates.

Each of the above magic transformations accepts an adorned query as input, thus requiring two phases for optimization – (1) an adornment phase, and (2) a transformation phase.

In the adornment phase, each table reference is annotated to indicate which attributes of the table are restricted by equality predicates, which are restricted by predicates other than equality (conditions), and which are free. The annotation is indicated by an adornment string using letters b for *bound* by an equality predicate, c for *conditioned*, and f for *free*.

EXAMPLE 2.3 (Magic-sets Transformation): The SQL queries corresponding to the boxes in Figure 1 for the query graph after magic transformation are:

($MD0$): SELECT d.deptname, s.workdept, s.avgsalary
FROM department d, avgMgrSal^{bf} s
WHERE d.deptno = s.workdept AND
d.deptname = 'Planning'.

($MD1$): avgMgrSal^{bf}(workdept, avgsalary) AS
(SELECT s.workdept, AVG (salary)
FROM m_avgMgrSal^{bf} m, mgrSal^{ffbf} s
WHERE m.workdept = s.workdept
GROUPBY workdept).

($MD2$): mgrSal^{ffbf}(empno, empname, workdept, salary) AS
(SELECT e.empno, e.empname, e.workdept, e.salary
FROM m_mgrSal^{ffbf} m, employee e, department d
WHERE m.workdept = e.workdept AND
e.empno = d.mgrno).

($MD3$): m_avgMgrSal^{bf}(workdept) AS
(SELECT DISTINCT deptno FROM department
WHERE deptname = 'Planning').

($MD4$): m_mgrSal^{ffbf}(workdept) AS
(SELECT DISTINCT workdept
FROM m_avgMgrSal^{bf}).

m_avgMgrSal^{bf} and m_mgrSal^{ffbf} are magic tables for the views avgMgrSal^{bf} and mgrSal^{ffbf} respectively, giving the relevant department numbers for which the view needs to be evaluated. The superscripts bf and $ffbf$ indicate that the views avgMgrSal^{bf} and mgrSal^{ffbf} will always be evaluated with the attribute `workdept` (first attribute of avgMgrSal and the third attribute of mgrSal) bound to a set of department numbers, and with the remaining attributes free. □

The magic-sets transformation requires a join-ordering of tables in the `FROM` clause so as to be able to choose a *strategy* needed to adorn and magic-transform the query. The choice of the join-order is very important for an efficient transformation, and is one of the weak points of all implementations of magic in deductive databases. Deductive database systems don't do any cost-based optimization to determine the join orders needed for magic-sets transformation.

3 System Architecture

Figure 2 shows the architecture of the Starburst database

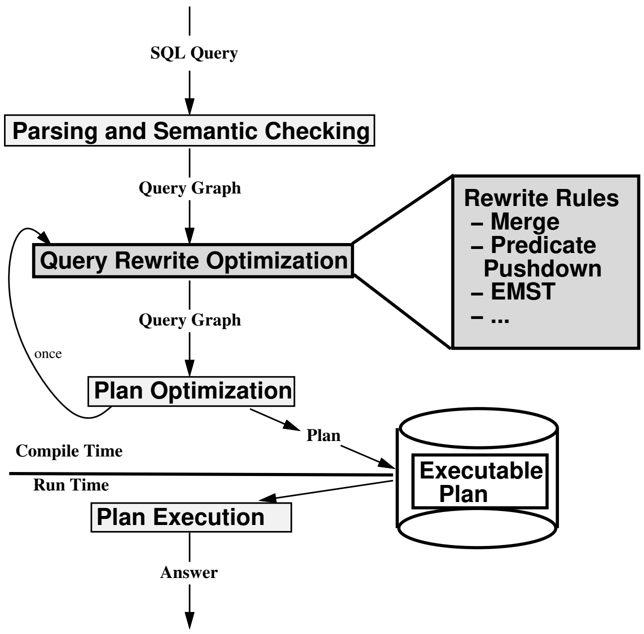

Figure 2: Starburst System Architecture.

system[HCL+90]. The system parses an SQL query into the QGM internal graphical form. The query then goes through two phases of optimization, a query rewrite phase (so called since it rewrites one query graph into another), and a plan optimization phase. Disregard the back arrow from plan optimization to query rewrite optimization for now. Plan optimization looks at a single QGM box and decides about optimal join orders, access methods, and so on [SAC+79, Loh88].

3.1 Query Rewrite Optimization

The query rewrite phase [PHH92] looks at the full query graph and does global optimization. The query rewrite optimizer performs transformations such as merging boxes (analog of unfolding in logic), pushing predicates and projections down into lower boxes so that they may be applied early during evaluation, redundant join elimination, redundant duplicate elimination, and the extended magic-sets transformation (*EMST*). The transformations are done using a production rule-based system that encodes each query transformation as a rewrite rule. The extended magic-sets transformation (EMST) is also implemented as a rewrite rule. A cursor facility traverses the query blocks depth first (other traversal methods are also possible), and a forward chaining engine applies the rules, including the EMST rule, at each query block. The EMST rule differs from a typical rewrite rules in that it needs join order information. Thus, the nature of the integration of the EMST rule into the query rewrite phase depends critically upon how cost-based join order optimization is done. This issue has been ignored in all previous implementations.

3.2 Cost Based Join Orders

The extended magic-sets transformation on a box is sensitive to the join order of the referenced tables. Optimal join orders are determined in Starburst by the plan optimizer, using extensive statistical information and cost estimates. To compute the optimal query plan, we should apply EMST for every possible join order (2^n options in a box with n quantifiers), and then run the plan optimizer on each transformed program to determine the program with the least cost. The plan optimizer would thus be executed 2^n times. However, even without applying EMST, the cost of looking at all join alternatives is usually unacceptably high when large number of tables are joined. Optimizers employ pruning techniques such as greedy search and dynamic programming to reduce the number of alternatives. Further, optimizers represent alternative plans in highly optimized data structures to make it feasible to store and search through large number of alternatives. If we apply EMST for each join order, the shape of generated alternative queries are very different compared with simple permutation of tables. As a result, the current technique of pruning and using compact data structures are not sufficient, resulting in unacceptable optimization time.

The Starburst Solution: In Starburst, we use a cost-based heuristic to determine the join order to be used for applying EMST. Under our heuristic, optimization proceeds as follows:

1. Do query rewrite transformation without using the EMST rule (actually, a version of the EMST rule that does not depend on join orders and pushes only local predicates is used in Starburst, but this variant rule is not the subject of this paper). A join order is therefore not needed in this rewrite phase.
2. Do plan optimization to determine the best join order for each query block (traverse the edge from the query optimizer to the plan optimizer as shown in Figure 2).
3. Do query rewrite transformation using the EMST rule, using the join orders determined by the plan optimizer in Step 2. This new phase of query-rewrite is represented by the back edge from the plan optimizer to the query rewrite optimizer in Figure 2.
4. Do plan optimization to determine the new optimal join orders in each query block. As shown in Figure 2, the edge from from the query optimizer to the plan optimizer is thus traversed for the second time.
5. Compare the costs of the optimal plans before and after EMST transformation. Execute the cheaper plan.

Thus, under our cost based heuristic, we do plan optimization twice. The back edge from the plan optimizer to the query rewrite optimizer in Figure 2 is introduced into the Starburst architecture to support the cost-based join order determination for the magic-sets transformation. The heuristic has the following desirable properties: (1) The total cost of join order determination is $O(2^{n+1})$. (2) More important, the plan optimizer can continue to use pruning strategies during each invocation. (3) We can prove that under our cost-based heuristic, usage of the EMST rewrite rule cannot degrade a query plan produced without using the EMST rule: Do all the optimization you can without knowing join orders and without using EMST. Then, after you determine the optimal join orders, perhaps you can use the join orders to improve the plan you just selected.

We have experimented with our cost-based heuristic, and the preliminary results are extremely encouraging. The performance experiments of Table 1 [MFPR90a] used the cost-based heuristic (applied manually by iterating through the DB2 optimizer) to achieve impressive gains. The Starburst implementation provides a testbed for the above and other possible heuristics for investigating the interaction between cost-based optimization and magic-sets transformation.

3.3 The EMST Rule inside the Query-Rewrite

The join order heuristic requires that the query-rewrite occur twice, once without EMST, and once with EMST. In the Starburst implementation, we find it best to do the query rewrite in three phases, with tight control over execution of the EMST rule, as illustrated in Figure 3. Note that

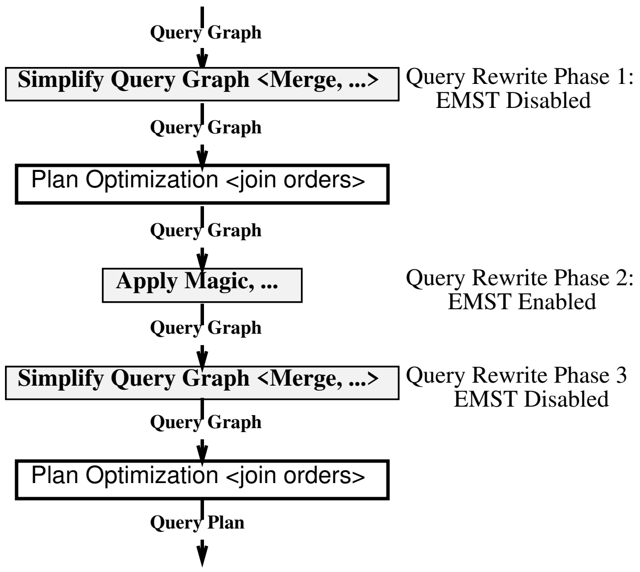

Figure 3: Query-Rewrite, EMST, and Plan Optimization.

plan optimization is still done only twice, as required by the join order heuristic. During the first phase of query-rewrite, rewrite rules other than EMST that do not depend on join orders, such as local predicate pushdown (implemented through a local magic rule not discussed in this paper), duplicate elimination pushdown, redundant join elimination, and the merge rule are applied. Plan optimization is done after the first phase to determine join orders. Then, in the second phase, EMST, as well as rewrite rules other than EMST are active, and the join orders determined by the plan optimizer are used. A third query-rewrite phase is used to simplify the query graph after the EMST application. The EMST rule is disabled in the third phase. Thus, the EMST rule is applied only during the second phase of query-rewrite.

EXAMPLE 3.1 Consider the query D of Example 1.1. Its query graph is reproduced in the upper left quadrant of Figure 4. In the phase 1 of query-rewrite, the merge rule merges the `AVGMGRSAL` select-box into the `QUERY` select-box, and the `MGRSAL` select-box into the `T1` select-box, deriving the QGM graph shown in the upper right quadrant of Figure 4. We now use the name AVGMGRSAL for the groupby box formerly called T2. The SQL representation of the query has not changed during phase 1 in this particular example. □

4 Implementing the Extended Magic-sets Transformation Rule

In this section, we give details of the EMST rule itself, the algorithm used, the data structures used, and the manner of application of the EMST rule. Like other rewrite rules, the EMST rule operates on one QGM box at a time. (QGM, or the query graph model, was described in Section 2.) A select-box in QGM is like an SQL blob with one `SELECT` block without a groupby clause. Thus the action of EMST on a select-box is very similar to the action of the magic-sets transformation on one Datalog rule. The action of EMST on other types of boxes, such as a difference-box or a union-box is different from the action of EMST on a select-box. Further, new types of boxes may be incorporated into the QGM. For example, an outer-join operation can be defined by defining an outer-join-box. The EMST rule is applied once on each QGM box as the query graph is traversed depth-first, culminating in a complete magic transformation of the query graph. We illustrate the EMST rule with a nonrecursive example for simplicity; however we stress that the EMST rule applies to nonrecursive and general recursive queries with stratified negation and aggregation. The major differences of EMST over the GMST algorithm presented in [Mum91] are (1) EMST combines adornment and magic-sets transformations into one step, (2) the modular nature of EMST, (3) the lack of control over the order in which various boxes (or blobs) are processed by EMST, and the (4) interaction permitted between EMST and other rewrite rules. These differences force adaptations of the algorithm that are discussed in this section. The techniques to adapt to these *real-world constraints* are very important for any system that hopes to implement the magic-sets transformation inside an integrated query-rewrite optimization system. The EMST rule uses other rewrite rules while transforming a box. For example, the EMST rule uses the predicate pushdown rule to push predicates into each referenced table to derive an adornment for the referenced table.

We first illustrate the EMST rule and its integration into the query-rewrite system (Figure 3) by applying EMST to the query D of Example 1.1. Other rewrite rules will be applied concurrently, as applicable. After the example, we will give details of the EMST algorithm.

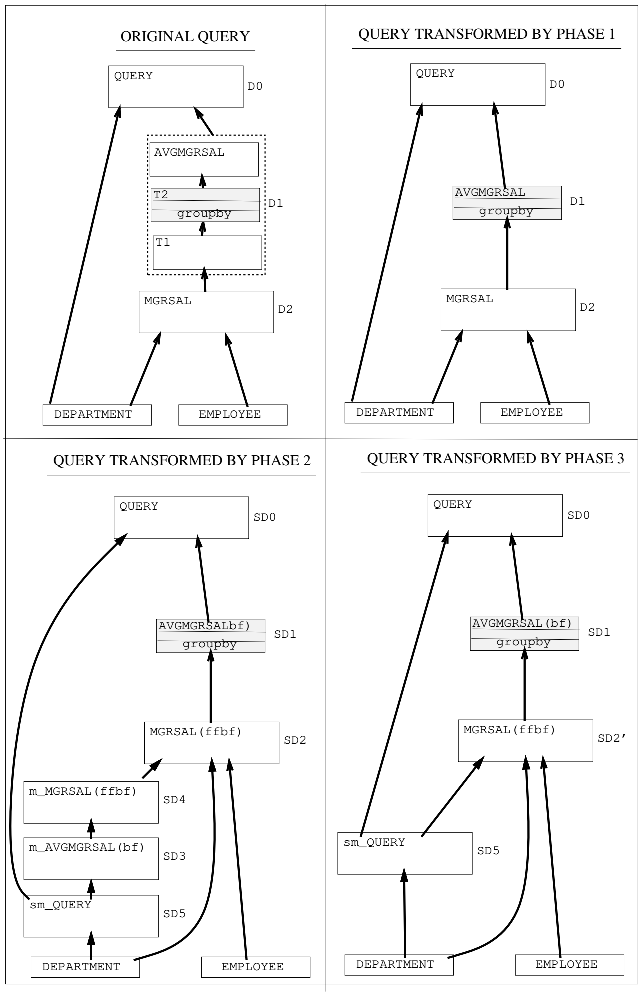

Figure 4: QGM query graph for query D, before, and after, phases 1, 2, and 3 of query-rewrite.

EXAMPLE 4.1 We start with the QGM graph shown in the upper left quadrant of Figure 4. The corresponding SQL query D is repeated in Figure 5.

Phase 1: In the first phase of query-rewrite (Example 3.1), the query graph was simplified to the QGM shown in the upper right quadrant of Figure 4. Following phase 1, let the plan optimizer produce the join order (department $\bowtie$ avgMgrSal).

Original Query before application of EMST:

($D0$): SELECT d.deptname, s.workdept, s.avgsalary
FROM department d, avgMgrSal s
WHERE d.deptno = s.workdept AND
d.deptname = 'Planning'.

($D1$): avgMgrSal(workdept, avgsalary) AS
(SELECT workdept, AVG (salary)
FROM mgrSal
GROUPBY workdept).

($D2$): mgrSal(empno, empname, workdept, salary) AS
(SELECT e.empno, e.empname, e.workdept, e.salary
FROM employee e, department d
WHERE e.empno = d.mgrno).

Query after EMST application in Phase 2:

($SD0$): SELECT q.deptname, s.workdept, s.avgsalary
FROM sm_query q, avgMgrSal^{bf} s
WHERE q.deptno = s.workdept.

($SD1$): avgMgrSal^{bf}(workdept, avgsalary) AS
(SELECT workdept, AVG (salary)
FROM mgrSal^{ffbf}
GROUPBY workdept).

($SD2$): mgrSal^{ffbf}(empno, empname, workdept, salary) AS
(SELECT e.empno, e.empname, e.workdept, e.salary
FROM m_mgrSal^{ffbf} m, employee e, department d
WHERE m.workdept = e.workdept AND
e.empno = d.mgrno).

($SD3$): m_avgMgrSal^{bf}(workdept) AS
(SELECT deptno FROM sm_query).

($SD4$): m_mgrSal^{ffbf}(workdept) AS
(SELECT workdept FROM m_avgMgrSal^{bf}).

($SD5$): sm_query(deptno, deptname) AS
(SELECT deptno, deptname FROM department
WHERE deptname = 'Planning').

Query after simplification in Phase 3: Same as after phase 2, except that $SD3$ and $SD4$ are eliminated, and $SD2$ is modified to $SD2'$ as below:

($SD2'$): mgrSal^{ffbf}(empno, empname, workdept, salary) AS
(SELECT e.empno, e.empname, e.workdept, e.salary
FROM sm_query sm, employee e, department d
WHERE sm.deptno = e.workdept AND
e.empno = d.mgrno).

Figure 5: SQL Queries before and after optimization by EMST.

Phase 2: In phase 2 of query-rewrite, the EMST rule transforms the select-box QUERY, and the groupby-box AVGMGRSAL.

select-box labeled QUERY: The predicate

"department.deptno=AVGMGRSAL.workdept"

can be pushed into the quantifier over groupby-box AVGMGRSAL, leading to an adorned box $\mathsf{AVGMGRSAL}^{bf}$. A supplementary-magic-box sm_QUERY is created to compute the selection (deptname='Planning') on the department table. The supplementary-magic-box is then used in QUERY and in defining a magic-box $\mathsf{m\_AVGMGRSAL}^{bf}$ for $\mathsf{AVGMGRSAL}^{bf}$. The magic-box is linked with the $\mathsf{AVGMGRSAL}^{bf}$ box, but it is not used to restrict computation in the $\mathsf{AVGMGRSAL}^{bf}$ box (QGM semantics for a groupby-box does not let us apply the restriction).

groupby-box labeled $\mathsf{AVGMGRSAL}^{bf}$: The magic-box linked with the groupby-box $\mathsf{AVGMGRSAL}^{bf}$ provides the predicate

"MgrSal.workdept = $\mathsf{m\_AVGMGRSAL}^{bf}$.workdept"

that can be pushed further into the box MGRSAL, leading to the adorned box MGRSAL^{ffbf}. The magic-box $\mathsf{m\_MGRSAL}^{ffbf}$ derives department numbers from $\mathsf{m\_AVGMGRSAL}^{bf}$, and passes them into the box MGRSAL^{bf}, completing the EMST application on box $\mathsf{AVGMGRSAL}^{bf}$.

select-box labeled MGRSAL^{ffbf}: No action is taken since all referenced tables are either magic tables or stored tables.

Final Result of phase 2: The magic transformation of the query graph is now complete, and the lower left quadrant of Figure 4 shows the resulting query graph. Besides the EMST rule, a distinct pullup rule is used twice in this phase to infer that there is no need to eliminate duplicates from the magic tables, since one can infer that duplicate magic tuples will not be generated. Thus, we arrive at the SQL query shown in Figure 5 as statements $SD0 - SD5$.

Phase 3: In the final phase of query-rewrite, the select-box $\mathsf{m\_MGRSAL}^{ffbf}$ is merged into the select-box MGRSAL^{ffbf}, and the select-box $\mathsf{m\_AVGMGRSAL}^{bf}$ is also merged into the select-box MGRSAL^{ffbf}, outputting the simpler query graph shown in the lower right quadrant of Figure 4. This merge was possible only because we inferred, in phase 2, that duplicates were guaranteed to be absent from the magic tables, and so we did not need to apply the `DISTINCT` operator in statements $SD3$ and $SD4$. The example thus highlights the need to apply other query rewrite rules to simplify the query graph produced by EMST. The resulting SQL query is identical to the one produced by phase 2, except that the view mgrSal^{ffbf} is now defined by statement $SD2'$, shown in Figure 5, and the views $SD3$ and $SD4$ are discarded. □

The optimized query in the lower right quadrant of Figure 4 has more blobs than the phase 1 query in the upper right quadrant. However, note that predicates have been pushed down in the optimized query, so they get applied earlier during evaluation, and computation is more efficient. Indeed, a query isomorphic to D was one of the queries tested in the DB2 benchmark (Experiment G), and the optimized query in the lower right quadrant showed two and a half orders of magnitude improvement in execution time over the query in the upper right quadrant.

4.1 Data Structures Used

The EMST rule introduces three special types of QGM boxes into the query graph model: the **magic-box** (e.g. $\mathrm{m\_MGRSAL}^{ffbf}$), **condition-magic-box**, and **supplementary-magic-box** (e.g. sm_QUERY). A box that is not of one of the above three types will be called a **regular** box (e.g. MGRSAL^{ffbf}). A quantifier that references any of the above special boxes will be called a *magic quantifier*.

The **magic-box** is a QGM box constructed during an EMST processing of a box whose adornment does not contain a c adornment. A magic-box contributes tuples to the magic table of the associated adorned box. The magic-box is either a select-box, or a union-box. Both the magic-boxes in Example 4.1 are select-boxes. A magic-box differs from regular select-boxes in that the EMST rule does not apply to a magic box. To other rewrite rules, the magic-box is indistinguishable from other select- or union-boxes.

The **condition-magic-box** is similar to a magic-box except that (1) it is constructed during the EMST processing of a box whose adornment contains a c adornment, and (2) The EMST rule processes a condition-magic-box. A condition-magic-box is ungrounded at time of construction, and may be grounded later (using the GMST algorithm [Mum91]).

The **supplementary-magic-box** is also constructed during the EMST processing of a box. A supplementary-magic-box contributes tuples to the supplementary table of the associated box being processed.

4.2 AMQ and NMQ Properties

QGM allows boxes of different operation types — `SELECT`, `UNION`, `GROUP-BY`, `INTERSECTION`, and `DIFFERENCE` being a few. New operation types can also be defined by the database customizer. The EMST processing of a box depends on the operation type of the box. It is difficult to write a separate EMST rule for every operation type, and unrealistic to expect future customizers of Starburst to provide an EMST rule for every operation they define. It is thus important to identify the properties of an operation type that determine the type of EMST processing on a box of that operation. The relevant property is whether a box B allows a quantifier (i.e., a table reference) to be added to the box with the semantics that the new table reference is to be joined with the table being output by box B earlier. If it does, box B is said to be an AMQ (accepts magic quantifier) box, and a magic quantifier over the magic-box can be inserted into the box B. Supplementary tables can also be created for an AMQ box. If a box B does not permit a quantifier to be added, or if it does not cause the inserted table reference to be joined with the other table references, then box B is said to be an NMQ (no magic quantifier) box, and box B cannot use the magic quantifier to restrict the computation done inside the box. However, an NMQ box may be able to pass the restriction represented by the magic table down into its quantifiers, so that one may compute restricted tables to start with.

A select-box is AMQ, while the union-, groupby-, and difference-boxes are NMQ.

4.3 Adorning a QGM-box

The adorn-box algorithm described below is used to adorn a QGM box from within the EMST rule. The given algorithm can be used to adorn an SQL block or a Datalog rule by appropriately mapping notation. For example, a quantifier corresponds to a table reference in an SQL block, and to a subgoal in a Datalog rule.

Consider a quantifier q in a box B, ranging over a box Bq. Adornment of the box Bq depends on the predicates restricting q's usage. To determine all the predicates on box Bq, we need to (1) Determine which predicates from box B, and elsewhere, can potentially restrict box Bq, and (2) What is the form of these predicates on box Bq.

When B is an AMQ box (e.g. the QUERY box), all and only the predicates in box B can restrict Bq. However, when B is an NMQ box (e.g. the AVGMGRSAL^{bf} box), it does not have a magic quantifier over its magic table, so the computation of box B is not restricted by predicates in its own magic table. However the predicates in the magic table of B can be used to restrict computation of a child box Bq. Further, predicates inside box B can also restrict computation of a child box Bq. As an example, a predicate on the outer table in a difference-box can restrict computation of the inner table.

The predicate pushdown rules [PHH92] to push predicates are query-rewrite rules defined separately, and independently, from EMST. A separate predicate pushdown rule is written for each type of box (select-, union-, ...) in the system. The predicate-push-down rules can map predicates from attributes in the `SELECT` clause onto attributes in the `FROM` clause, can push predicates through a groupby-box, including predicates on an aggregated column, and can recognize that a predicate on an inner table in a difference-box cannot be pushed into the outer table, while a predicate on the outer table can be pushed into the inner table.

Algorithm 4.1 adorn-box
Input: A QGM box B.
Output: Adornments for quantifiers in box B.
Method: For each quantifier q in box B:

1. Find the predicates that need to be pushed out of box B. If B is an AMQ box, all predicates in box B can be pushed down. If B is an NMQ box, all predicates in box B, as well as predicates implied by existing magic-boxes for B need to be pushed down.

   EXAMPLE 4.2 Let B be the QUERY box, and consider quantifier q ranging over the AVGMGRSAL box. The QUERY box is an AMQ box, so the predicate department.deptno = avgMgrSal.workdept can be pushed into the AVGMGRSAL box. □

   EXAMPLE 4.3 Now let B be the AVGMGRSAL^{bf} box, and consider quantifier q ranging over the MGRSAL box. The AVGMGRSAL^{bf} box is an NMQ box, so the predicate
   "MgrSal.workdept = m_AVGMGRSAL^{bf}.workdept"
   implied by the linked magic-box can be pushed into the MGRSAL box. □

2. Determine the quantifiers that are eligible to pass information into q. Eligibility depends on join-orders, and the correlation predicates the quantifier q participates in. To pass information into q, a quantifier in box B must precede q in the join-order, while a quantifier outside box B linked to q by a correlation must satisfy a condition based upon correlation levels.

   EXAMPLE 4.4 Continuing with Example 4.2: The join order (department $\bowtie$ avgMgrSal) was determined in a previous plan optimization phase. Since department precedes avgMgrSal in the join order, the quantifier over department is eligible to pass information into q = avgMgrSal. □

3. Use predicate-push-down rules to push down all of the above determined predicates subject to the eligibility of quantifiers. The set of pushed predicates on the referenced tables is thus determined.

   EXAMPLE 4.5 Continuing with Example 4.4: The predicate
   "department.deptno = avgMgrSal.workdept"
   is pushed into the AVGMGRSAL box. □

   EXAMPLE 4.6 Continuing with Example 4.3: The predicate
   "MgrSal.workdept = m_AVGMGRSAL^{bf}.workdept"
   is pushed into the MGRSAL box. □

4. Select a bcf adornment appropriate for the set of predicates on the referenced table.

   EXAMPLE 4.7 Continuing with Example 4.5: The predicate
   "department.deptno = avgMgrSal.workdept"
   is used to derive the adornment bf for the AVGMGRSAL box. □

EXAMPLE 4.8 Continuing with Example 4.6: The predicate
"MgrSal.workdept = m_AVGMGRSAL^{bf}.workdept"
is used to derive the adornment *ffbf* for the MGRSAL box. □

The Starburst implementation uses the c adornment to represent dependent and independent conditions. However, there are complex NMQ operations (such as outer-joins) where the refined *bcf* adornments of [Mum91] are needed to represent dependent conditions.

4.4 EMST application on one QGM box

We define a procedure magic_process(B) to apply EMST on one QGM box B. magic_process(B) assumes that the table defined by B has been adorned, and that magic-boxes or condition-magic-boxes for B are available. However, when magic_process is first invoked on the query graph, only a null (n) adornment is supplied. magic_process creates an $ff\ldots f$ adornment for the top query box, and proceeds normally.

Algorithm 4.2 magic-process
Input: A QGM box B in a query graph G.
Output: Query graph G' after applying EMST rule to box B.
Method: For each quantifier q in box B, in the given join order:

1. Determine the quantifiers eligible to pass information into q.
2. Apply the adorn-box Algorithm 4.1 to compute the pushed predicates and the adornment α for quantifier q.
3. Make q range over a box Bq with adornment α. The box referenced by q may already have adornment α, or a copy with adornment α may have been made earlier, or such a copy may be created at this step.

   EXAMPLE 4.9 Continuing with Example 4.7: A copy of the box AVGMGRSAL for the adornment *bf* is created, and the quantifier ranging over AVGMGRSAL is made to range over the box AVGMGRSAL^{bf}. □

   EXAMPLE 4.10 Continuing with Example 4.8: A copy of the box MGRSAL for the adornment *ffbf* is created, and the quantifier ranging over MGRSAL is made to range over the box MGRSAL^{ffbf}. □

4. If $\alpha \neq ff\ldots f$ do:

   (a) If box B is an AMQ box, and if it is desirable to construct a supplementary-magic-box before position i, do so by moving all the eligible quantifiers from box B into a supplementary-magic-box, and placing in B a quantifier over the supplementary-magic-box. It is not desirable to construct a supplementary-magic-box at a position just before the magic quantifier or before the first non magic quantifier in the join order, or if the supplementary-magic-box would include one quantifier and no predicates.

   EXAMPLE 4.11 Continuing with Example 4.9: Since B, the QUERY box, is an AMQ box, we construct a supplementary-magic-box at the join order position just before the quantifier over AVGMGRSAL^{bf}. The supplementary-magic-box, sm_query, contains the preceding quantifier over department, along with the selection predicate "department.deptname = 'Planning' (statement $SD5$ in Figure 5). The QUERY box now refers to the supplementary-magic-box sm_query rather than to the box department □

   EXAMPLE 4.12 Continuing with Example 4.10: Since B, the AVGMGRSAL^{bf} box, is not an AMQ box, a supplementary box is not constructed. □

   (b) Construct a magic-box or a condition-magic-box from q for the magic table of box Bq referenced by q. The eligible quantifiers, including the magic quantifier, and the pushed predicates will go into the magic box.

   However if B is an NMQ box and has c adornments, the condition-magic-box CMB for box B is ungrounded. Descendant boxes of box B will ground the predicates in the condition-magic-box CMB, and each descendant will ground CMB differently. Therefore, for this case, contents of the box CMB must be copied into the magic-box (or condition-magic-box) constructed for box Bq.

   EXAMPLE 4.13 Continuing with Example 4.11: The magic box m_avgMgrSal^{bf} is constructed, with a single quantifier over sm_query. □

   EXAMPLE 4.14 Continuing with Example 4.12: The magic box m_mgrSal^{ffbf} is constructed, with a single quantifier over m_avgMgrSal^{bf}. □

   (c) Add the magic- or the condition-magic-box m constructed above to the magic table for box Bq.

   If Bq is an NMQ box, the box m is linked to box Bq so that it may be retrieved from box Bq, but no magic quantifier is introduced. If Bq is an AMQ box, and m is a magic-box, a magic quantifier referencing magic-box m is inserted into box Bq. If Bq is an AMQ box, and m is a condition-magic-box, a supplementary-magic-table is created, and the condition-magic-box m is ground to construct a supplementary-magic-box (as required by GMST [Mum91]).

   EXAMPLE 4.15 Continuing with Example 4.13: Since AVGMGRSAL^{bf} is an NMQ box, its magic box m_avgMgrSal^{bf} is simply linked to it; no magic quantifiers over the magic box get created. □

   EXAMPLE 4.16 Continuing with Example 4.14: Since MGRSAL^{ffbf} is an AMQ box, a quantifier over its magic box m_mgrSal^{ffbf} is added to the definition of mgrSal^{ffbf} (statement $SD2$ in Figure 5). □

5 Extensibility

Extensibility is a major design goal in Starburst. A database customizer can add (1) features to the SQL language, leading to new operations in QGM boxes, (2) new optimization rules, (3) new traversal orders on the query graph, and (4) new access methods and cost-estimating functions, and so on. The query-rewrite optimizer, including the EMST rule, are required to be extensible with all customizations. The implementation described in Section 4 meets the extensibility goals.

When a database customizer defines a new operation, such as an outer-join, it is expected that the customizer will write predicate pushdown rules for outer-join. Specification of the predicate pushdown rules was purposefully kept independent of the EMST rule to facilitate definitions of predicate pushdown rules by the customizer. Further, the customizer is required to state whether a quantifier can be inserted into the box with a join semantics (AMQ or NMQ) – a simple property to state.

When new rewrite rules are added, a decision must be taken on the firing priority and sequence with respect to all other rules in the system. The EMST rule does not introduce additional complexity. The introduction of magic-, condition-magic-, and supplementary-magic-boxes helps extensibility by allowing EMST to distinguish between different types of select-boxes, while protecting other rules from noticing the difference. The EMST rule can be applied to the QGM boxes in any order of traversal, achieving the same final transformation, though the time taken to arrive at the result will differ. Join orders are determined outside the EMST rule, so a change in cost-estimating functions does not affect EMST.

6 Related Work

Our previous work on magic-sets transformation in relational systems has concentrated on performance evaluation [MFPR90a], and the adaptation of the magic-sets algorithm to duplicates and aggregates [MPR90]. We have now completed an implementation, and this paper describes the results of our implementation effort.

The magic-sets implementation in Starburst has goals and addresses issues very different from the goals and issues addressed by previous and concurrent deductive database implementations (Aditi [VRK+90], *LDL* [NT88], *NAIL!* [MNS+87], Glue-Nail [DMP93], Coral [RSS92], and EDS [FF93]).

The Starburst implementation is done in the context of an extensible relational system, with extended SQL as the query language. With the exception of EDS [FF93] all other implementations of magic-sets have been done for extended versions of Datalog. EDS works on extended SQL, but applies magic only to the Datalog style recursive subcomponent of extended SQL. SQL has many non-logical features, such as duplicates, aggregates, grouping, nested subqueries, existential and universal quantification, null values, and outer joins, all of which puts demands on the magic-sets design. We handle all complex features of SQL, and are fully compatible with SQL semantics. Further, our implementation is *extensible* to new features.

The Starburst implementation of magic-sets pushes conditions as well as equality predicates, and it does so while staying within the SQL evaluation semantics that requires all tuples to be ground terms. All other implementation either do not push conditions [VRK+90, NT88, MNS+87, DMP93, FF93], or introduce non-ground tuples [RSS92]. By pushing conditions along with equality predicates, the Starburst implementation integrates traditional predicate pushdown techniques that work on *local* conditions into the magic-sets transformation. The Starburst implementation is the first implementation to incorporate the magic conditions algorithm [MFPR90b], and the first implementation to use *bcf* and the more complex refined adornment classes [Mum91, MPR94].

Starburst is also unique in doing the magic-sets transformation on nonrecursive queries. The other systems have concentrated on recursive queries, where the importance of magic-sets transformation is questionable since the more specialized techniques for linear recursive queries, when applicable, do better than magic-sets.

Previous implementations of magic-sets transform the query in two steps: the query is adorned in the first step, and magic transformed in the second step. The EMST algorithm in Starburst combines the two steps: it creates magic tables concurrently while adorning the original query. The one-step processing in the EMST algorithm reduces the complexity of adornments and provide more precise constraint information while adorning the query [Mum91].

To summarize, compatibility with SQL, extensibility, uniform treatment of conditions and equality predicates, use of cost-based join order estimation, integration with traditional relational optimizations, and the focus on nonrecursive queries, are the major strengths of our implementation, differentiate us from other implementations, and strongly demonstrate the feasibility of using magic-sets transformation in a commercial relational database system.

7 Conclusion

We have implemented an extended magic-sets transformation (EMST) for nonrecursive and stratified recursive SQL queries in Starburst. The implementation required about 6 person-months of coding effort. The EMST transformation is integrated into a rule-based query rewrite system, and is implemented as a rule that transforms the query in a modular fashion, one unit at a time. The integration of EMST into the complete query-rewrite rule system enables us to eliminate the unnecessary complexity introduced by EMST in the query graph. EMST uses *bcf* adornments, can push equality and condition predicates, can push local and join predicates, adorns and transforms queries in one phase, can handle correlations, and is extensible. We have developed a cost-based heuristic to determine the join orders to be used for the magic-sets transformation, with the desirable property that the magic-sets transformation cannot degrade a query plan generated without doing the magic-sets transformation.

Commercial database vendors are now realising the extreme importance of optimization for complex decision support queries. In some cases, much effort has been spent to optimize TPCD benchmark queries [TPCD94] by hand in order to achieve better performance. The magic-sets transformation provides an opportunity to optimize decision support queries in a stable manner [MFPR90a]. However, the relational vendors have found the magic-sets technique difficult to understand, and impractical to implement. We have explained magic-sets at an intuitive level, and have given sufficient details of an actual implementation in a relational system to convince the vendors that (1) The perceived problems with implementing magic-sets have been solved by us, and (2) The magic-sets transformation can be implemented in relational database systems with modest effort (about 6 person-months).

Our implementation shows that it is feasible to build a magic-sets transformation module as a layer above existing relational databases, with feedback from the database systems about a join order to use for the magic-sets transformation, and with the guarantee that the resulting system will be as or more efficient than the existing system. Such an architecture brings the magic-sets transformation to the open world, and we hope it will encourage products from *SQL conditioning* companies. We also believe that our work will lead to many more implementations of the magic-sets transformation in commercial relational systems, especially for nonrecursive queries.

References

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS 1986*.

[BR91] C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10:255–300, 1991.

[Day87] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *VLDB 1987*.

[DMP93] M. Derr, S. Morishita, and G. Phipps. Design and implementation of the Glue-Nail database system. In *SIGMOD 1993*.

[FF93] J. Fessy and B. Finance. Early experience with recursion optimization in an extensible rewriter. In Proceedings of the Fourth Australian Database Conference (ADC), 1993.

[GM92] A. Gupta and I. Mumick. Magic-sets transformation in non-recursive systems. In *PODS 1992*.

[GW87] R. Ganski and H. Wong. Optimization of nested SQL queries revisited. In *SIGMOD 1987*.

[HCL+90] L. Haas, W. Chang, G. Lohman, J. McPherson, P. 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.

[Kim82] W. Kim. On optimizing an SQL-like nested query. *ACM TODS* , 7(3), September 1982.

[Loh88] G. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *SIGMOD 1988*.

[MFPR90a] I. Mumick, S. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is relevant. In *SIGMOD 1990*.

[MFPR90b] I. Mumick, S. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic conditions. In *PODS 1990*.

[MNS+87] K. Morris, J. Naughton, Y. Saraiya, J. Ullman, and A. Van Gelder. YAWN! (Yet Another Window on Nail!). In *Data Engineering 10:4*, 1987.

[MPR90] I. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *VLDB 1990*.

[MPR94] I. Mumick, H. Pirahesh, and R. Ramakrishnan. Adornments in database programs. To appear in Computers and AI, 1994. Preliminary version appeared in *Proceedings of the International Workshop on Deductive Databases, International Conference on Logic Programming*, 1993.

[Mum91] I. Mumick. *Query Optimization in Deductive and Relational Databases*. PhD thesis, Stanford University, 1991.

[Mur92] M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. In *VLDB 1992*.

[NT88] S. Naqvi and S. Tsur. *A Logic Language for Data and Knowledge Bases*. Computer Science Press, 1988.

[PHH92] H. Pirahesh, J. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in Starburst. In *SIGMOD 1992*.

[Ram88] R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. In Proceedings of the Fifth International Conference and Symposium on Logic Programming, 1988.

[RSS92] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: Control, relations and logic. In *VLDB 1992*.

[SAC+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *SIGMOD 1979*.

[TPCD94] F. Raab. TPC-D Working Draft 6.5. In Preparation. Transaction Processing Performance Council. February 1994.

[Ull89] J. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes 1 and 2*. Computer Science Press, 1989.

[VRK+90] J. Vaghani, K. Ramamohanarao, D. Kemp, Z. Somogyi, and P. Stuckey. The Aditi deductive database system. In *Workshop on Deductive Databases, North American Conference on Logic Programming (NACLP)*, 1990.