Magic is Relevant

Presented at SIGMOD 1990

Inderpal Singh Mumick*
Stanford University

Sheldon J. Finkelstein†
IBM Almaden Research Center

Hamid Pirahesh
IBM Almaden Research Center

Raghu Ramakrishnan‡
University of Wisconsin at Madison

Any sufficiently advanced technology is indistinguishable from magic.

— Arthur C. Clarke, in *"Profiles of the Future"*

Abstract

We define the magic-sets transformation for traditional relational systems (with duplicates, aggregation and grouping), as well as for relational systems extended with recursion. We compare the magic-sets rewriting to traditional optimization techniques for nonrecursive queries, and use performance experiments to argue that the magic-sets transformation is often a better optimization technique.

1 Introduction

"Magic-sets" is the name of a query transformation algorithm ([BMSU86]) (and now a class of algorithms — Generalized Magic-sets of [BR87], Magic Templates of [Ram88], Magic Conditions of [MFPR90]) for processing recursive queries written in Datalog. Previously, these algorithms had not been deployed in standard relational database systems, and their value for such systems had not been assessed.

Relational database systems support a number of features beyond those in Datalog, including duplicates (which lead to multisets), aggregation and grouping. We extended the magic-sets approach (and the Datalog language) to handle these features ([MPR90]), and also showed that magic-sets can be extended to propagate conditions other than equality [MFPR90]. This paper synthesizes, extends, and applies those results; our goal is to demonstrate that magic-sets is a robust technique which can profitably be incorporated in practical relational systems not just for processing recursive queries (e.g., bill-of-materials) but also for nonrecursive queries. The technique is particularly valuable for complex queries such as decision-support queries.

The paper is organized as follows. We present a brief subsection describing SBSQL, the SQL language of Starburst that supports recursion, and give a realistic example of a non-linear query. Section 2 motivates the practicality of the magic-sets technique by describing its relationship to traditional transformations (such as predicate pushdown) for nonrecursive complex SQL queries. Section 3 defines the Magic-sets transformation and some related concepts, as published elsewhere. Section 4 describes our extension of the magic-sets transformation for relational database systems, resolving complications arising when our previous results [MFPR90, MPR90] are combined and applied to SBSQL. We show that recursions introduced by the magic-sets transformation of nonrecursive queries can be avoided, and that combining the adornment phase with the magic-sets transformation allows us to propagate arbitrary conditions using a simple adornment pattern. An overview of the implementation of magic-sets in the Starburst extensible relational database prototype at IBM Almaden Research Center ([HFLP89]) is presented in Section 5. Section 6 gives DB2 performance

*Part of this work was done at the IBM Almaden Research Center. Work at Stanford was supported by an NSF grant IRI-87-22886, an Air Force grant AFOSR-88-0266, and a grant of IBM Corporation.

†Author's current affiliation: Tandem Computers.

‡Part of this work was done while the author was visiting IBM Almaden Research Center. Work at Wisconsin was supported by an IBM Faculty Development Award and an NSF grant IRI-88-04319.

measurements demonstrating that magic-sets can improve the performance of complex nonrecursive SQL queries over traditional techniques such as correlation and decorrelation. Section 7 presents conclusions.

1.1 Starburst SQL (SBSQL)

The SQL language in Starburst (SBSQL) has been extended to include recursion, user defined functions, and abstract data types. SBSQL supports a number of operations on tables, including `SELECT` , `GROUPBY` , and `UNION` . The `SELECT` operation performs a join and selection on the input tables and outputs a set of (expressions on) columns of qualified tuples. A sophisticated user of Starburst (the *Database Customizer*) may define new operations (e.g., outer join) on tables, so the query-rewrite phase of Starburst needs to be adaptable to language extensions.

Definition 1.1 Table Expressions : A texp (table expression) in SBSQL is an expression defining a named derived table that can be used anywhere in the query in place of a base table. A texp includes a head and a body. The head of a texp specifies its output table (name, attribute names). The body of a texp is a SBSQL query specifying how the output table is computed. □

As an example, consider the following query that determines the employee number and salary of senior programmers along with the average salary and head count of their departments.

EXAMPLE 1.1 (Table Expression):

```
(Q): SELECT Eno, Sal, Avgsal, Empcount
     FROM emp, dinfo(Dno, Avgsal, Empcount) AS
          (SELECT Dno, AVG(Sal), COUNT(*)
           FROM emp GROUPBY Dno)
     WHERE Job = "Sr Programmer" AND
           emp.Dno = dinfo.Dno.
```

□

dinfo is a derived table defined by a table expression. `dinfo(Dno, Avgsal, Empcount)` is the head of the texp, and `(SELECT Dno, AVG(Sal), COUNT(*) FROM emp GROUPBY Dno)` is the body of the texp.

In this paper we use a variant of the standard SQL syntax for conciseness. We write texps separately as if defining a view, so that query (Q) of Example 1.1 is written as:

```
(T1): SELECT Eno, Sal, Avgsal, Empcount
      FROM emp(Eno, Sal, Dno, Job),
           dinfo(Dno, Avgsal, Empcount)
      WHERE Job = "Sr Programmer".

(T2): dinfo(Dno, Avgsal, Empcount) AS
          (SELECT Dno, AVG(Sal), COUNT(*)
           FROM emp GROUPBY Dno).
```

At times, as in $(T1)$, we will explicitly name the attributes of the tables in the `FROM` clause, with the same name in two positions being shorthand for an equality predicate on the two columns.

Table expressions allow us to write Datalog queries. The head and body of a Datalog rule map to the head and body of a texp. Multiple Datalog rules having the same head map to a single texp with a body that is the `UNION` of the queries associated with the bodies of the rules.

EXAMPLE 1.2 (Nonlinear Query): Consider a system with a large number of components. Components can be base units, or they can require support from another component. Let each component have a primary and secondary supporter. If the primary supporter fails, the secondary supporter takes on its functions. A component fails if both its primary <u>and</u> secondary supporters fail. Some components may breakdown on their own, and we are interested in computing the resulting failed set of components.

Let primary(C, P) and secondary(C, S) give the primary (P) and secondary (S) supporters of a component C. broken(B) is the set of components that have broken down by themselves. The set of failed components fail(C) is defined by the texp:

```
(F): fail(C) AS
        ((SELECT * FROM broken)
        UNION DISTINCT
         (SELECT p.C
          FROM primary p, secondary s, fail f1, fail f2
          WHERE f1.C = p.P AND f2.C = s.S AND
                p.C = s.C)).
```

fail is the name of the output table of the texp F. F is recursive since fail is referred to within F. Further, the recursion is non-linear (since fail appears twice in the same `FROM` clause), and an equivalent linear query cannot be written ([AC89]). □

In this paper, we will sometimes refer to an SBSQL query as a program.

2 Relationship of Magic-sets to Traditional Optimizations

This section gives an informal description of magic-sets and its relationship to more traditional transformation techniques, and compares our work with previous results. Section 3 defines the magic-sets transformation formally.

2.1 Predicate Pushdown

Practical relational database systems push selection predicates as far down as possible in the execution tree. Data is filtered so that irrelevant rows are not propagated; in some cases, predicates are applied implicitly via the access path chosen to retrieve data. Consider the following SQL query with `emp(Eno, Ename, Sal,` Bonus, Job, Dno, EkidsN) and dept(Dno, Mgrno, Location) as the base tables.

```
(Q): SELECT Ename, Mgrno FROM emp, dept
     WHERE Job = "Sr Programmer" AND
           Sal + Bonus > 50000 AND
           emp.Dno = dept.Dno AND
           Location = "San Jose" AND
           P(emp,dept).
```

P is some complex subquery. A system might use an index on Job to access only senior programmers, immediately apply the predicate on Sal + Bonus, access dept using an index on its Dno column, immediately apply the predicate on Location, and finally evaluate the subquery P. Using an index is often cost-effective, even though it can be thought of as introducing an extra join (with the index). Indexed access can eliminate retrieval of many irrelevant rows (and hence, perhaps, many irrelevant data pages). Once we have an emp row, emp.Dno can be passed down, so that the Dno predicate can be used for accessing (or filtering) dept.

The predicate on Sal + Bonus could have been applied after dept was retrieved, but instead "predicate pushdown" moved it next to the data access. Since evaluation of such predicates is inexpensive, predicate pushdown (eliminating irrelevant rows as soon as possible), is typically a very good strategy.

The semi-join operator [BC81, RBF+80] takes this idea a step further. If an employee's department is not in San Jose, that employee is just as irrelevant (for the above query) as if she were a Junior Programmer. By computing SJDno, the Dno's of the departments in San Jose, a system can follow a modification of the above execution plan in which employees are filtered (based on emp.Dno in SJDno) before the join with the dept table. As with indexes, an extra operation is introduced (the computation and join with SJDno). Applying this operation means that the dept table must be accessed twice, first to compute SJDno, then to access matching departments.

In the original plan above, emp was accessed first, then the value of Dno was passed to dept, so that relevant departments were accessed. The semi-join predicate restricting employees to those with Dno in SJDno was passed "sideways" in the opposite direction, from the department table. Using information passed sideways is the "magic-sets" approach — systematically introducing predicates based on information passed sideways, so that these predicates can be used to filter out irrelevant data as soon as possible.

Unlike the standard predicate pushdown transformation, magic may be applied in many different ways for a particular query; these correspond to the "sips" (sideways information passing strategies) chosen. Sips can be used flexibly; for each join order (sips order), we can pick any set of tables to generate bindings, and pick any subset of the bindings and push them down. This produces magic predicates, which are bindings on certain columns (similar to the semi-join predicate for SJDno above). The names used for magic tables show how they were created — these names have superscripts indicating restrictions on attributes of the original query's tables. The superscripts are called "adornments".

2.2 Correlation and Decorrelation

Several authors have studied SQL subqueries ([ISO89, ABC+76]) and described transformations for migrating predicates across them [Kim82, GW87, Day87]. Correlation, like magic, "pushes predicates down" into subqueries. Its inverse, decorrelation, "pulls predicates up" from subqueries. One major difference between magic and these other techniques is that magic applies uniformly to hierarchical (tree-structured) and recursive queries (as well as queries with common subexpressions); these other techniques have been applied only to hierarchical queries.[1] Performance comparisons of these techniques appear in Section 6.

EXAMPLE 2.1 (Correlation, Decorrelation and Magic):

```
(C): SELECT Ename FROM emp e1
     WHERE Job = "Sr Programmer" AND
           Sal > (SELECT AVG(e2.Sal) FROM emp e2
                  WHERE e2.Dno = e1.Dno).
```

Query (C) selects senior programmers who make more than the average salaries in their departments. As written, it involves correlation: for each employee who is a senior programmer, the average salary in her department is calculated, and the employee is selected if her salary is more. No irrelevant information is computed, but the average salary for a department might be calculated many times (if several employees were in the same department).[2] In addition, access to e1 and e2 must be done in a specific order (e1, then e2). Finally, processing for e2 is row-at-a-time rather than set-oriented, and set-orientation tends to be a major performance advantage of the relational model. Thus correlation diminishes the non-procedural and set-oriented advantages of the relational model, and may also perform redundant computations.

Query C can be transformed into the decorrelated query[3] D that uses a temporary table, dep-avgsal(Dno, Asal), defined within the query.

```
(D1): SELECT Ename FROM emp, dep_avgsal
      WHERE Job = "Sr Programmer" AND Sal > Asal
            AND emp.Dno = dep_avgsal.Dno.
```

[1] Correlation can also exist in recursive queries, but that is beyond the scope of this paper [PF89].

[2] Correlation might be implemented so that departmental salaries are stored in a temporary table. This has its own costs.

[3] In this paper, a program consisting of statements $X1$ to Xn is referred to as program X.

($D2$): dep_avgsal(Dno, Asal) AS
(SELECT Dno, AVG(Sal) FROM emp
GROUPBY Dno).

Unlike the correlated query, the decorrelated query is set-oriented. (Average salaries are computed for all the departments in one operation, rather than computing the average for one department at a time as an employee in the department is selected.) It is also non-procedural. (The two scans of employee can be switched around.) An execution plan might access employees by Dno, calculate the average salary for that Dno, forming a tuple of dep_avgsal, and then find all senior programmers in that Dno with a higher salary. But decorrelation also has a substantial disadvantage: average salary is determined for all departments, whether or not they have senior programmers. If there are many departments and only a few have senior programmers, the cost of the irrelevant computation will be substantial.

The magic-sets approach combines the advantages of correlation and decorrelation, though at a cost. After transformation, the magic query S is:

($S1$): SELECT Ename FROM s_mag, mag_avgsal
WHERE Sal > Asal AND
s_mag.Dno = mag_avgsal.Dno.

($S2$): mag_avgsal(Dno, Asal) AS
(SELECT Dno, AVG(Sal) FROM mag, emp
WHERE mag.Dno = emp.Dno GROUPBY Dno).

($S3$): mag(Dno) AS
(SELECT DISTINCT Dno FROM s_mag).

($S4$): s_mag(Ename, Dno, Sal) AS
(SELECT Ename, Dno, Sal FROM emp
WHERE Job = "Sr Programmer").

One possible execution plan for S selects employees who are senior programmers (s_mag), determines which departments have at least one of these employees in them (mag), computes the average salary only for those departments (mag_avgsal), and then selects each senior programmer (s_mag) who makes more than her departmental average. The accesses can be ordered in other ways, just as they could be for the decorrelated query, and the operations are set-oriented. Moreover, no irrelevant data is touched, since the average is computed only for departments that have senior programmers. However, magic comes at the cost of computing extra tables, s_mag and mag.[4] □

The magic query S is similar to the semi-join example given in Section 2.1. Information about relevant bindings (departments that have senior programmers) is passed "sideways" from emp to mag_avgsal.

[4] s_mag is used more than once, and potentially has to be stored in a temporary. mag is used once, and may be piped.

2.3 Previous Work

Kim [Kim82] originally studied the question of when quantified subqueries could be replaced by joins (or anti-joins). Ganski and Wong [GW87] and Dayal [Day87] did additional work on both eliminating nested subqueries[5] and making correlated subqueries more efficient. These papers recognize that correlated subqueries can be very inefficient because they are not set-oriented. They eliminate correlation by introducing additional relational operators, including outer join [GW87] and generalized aggregation [Day87]. Their transformations can be applied to SQL queries written in a specific form, where the user either pushed down join predicates from the query to the subquery or wrote a predicate referring to tables in both the subquery and the query. They use these predicates, which we think of as sideways predicates, to generate bindings in the query.

In contrast, our Extended Magic Sets (EMS) technique takes queries without user-specified correlation and determines which sideways predicates should be pushed down. This is an advance, since users might miss some opportunities for predicate pushdown, and some cannot even be specified syntactically.[6] However, if users specify correlated queries, it is desirable to make them set-oriented. For this, we rely on techniques similar to the ones presented in [GW87] and [Day87]. Hence we view Ganski's and Dayal's work as complementary to our work. Ganski's paper illustrates the complexity of query-rewrite, since it emends some previous transformations. This complexity supports our unified structured approach, in which we systematically transform an algebraically general class (that includes recursion) of queries.

3 Definitions

Magic-sets Transformation: The Magic Sets algorithm rewrites a query so that the fixpoint evaluation of the transformed query generates no irrelevant tuples. The idea is to compute a set of auxiliary tables that contain the bindings used to restrict a table. The table expressions in the query are then modified by joining the auxiliary tables that act as filters and prevent the generation of irrelevant tuples. As a first step, however, we produce an *adorned* query in which tables are adorned with an annotation that indicates which arguments are bound to constants, which are restricted by conditions, and which are free, in the table expression using the table. For each table, we have an adorned version that corresponds to all uses of that table with a binding pattern that is described by the adornment; different adorned versions are essentially treated as dif-

[5] We use such subquery elimination rules (called merge rules) in Starburst to eliminate nesting before applying the magic-sets transformation.

[6] E.g., users cannot push predicates into views.

ferent tables (and possibly solved differently). For example, p^{bf} and p^{fb} are treated as (names of) distinct tables. An *adornment* for an n-ary table is defined to be a string of b's, c's and f's. Argument positions that are treated as free (have no predicate on them) are designated as f, and positions that are bound to a finite set of given values (by equality predicates) are designated as b. Argument positions that are restricted in the goal by some non-equality predicate (condition), are designated as c.

The magic-sets transformations of [BMSU86, BR87] propagate bindings (equality predicates) in Datalog, using b and f adornments. Conditions are ignored. [MPR90] extends the magic-sets transformation to propagate bindings in programs with duplicates and aggregation. The extension to conditions ([MFPR90]) needs to be adapted to work in presence of duplicates, and we present the idea in Section 4.1. We ignore c adornments and conditions in the following definition:

The Magic-sets algorithm can be understood as a two-step transformation in which we first obtain an adorned query P^{ad} and then apply the following transformation:[7]

We construct a new query P^{mg}. Initially, P^{mg} is empty.

1. Create a new `DISTINCT` table m_p for each table p in P^{ad}. The arity is the number of bound arguments of p.

2. For each table expression in P^{ad}, add a *modified version* to P^{mg}. If table expression r has head, say, $p(\overline{t})$ ($\overline{t}$ is shorthand for all the attributes of p), the modified version is obtained by joining the table $m\_p(\overline{t^b})$ into the body (m_p denotes the magic table of p, and $\overline{t^b}$ denotes the arguments of p that are bound.).

3. For each table expression r in P with head, say, $p(\overline{t})$, and for each table $q_i(\overline{t_i})$ referenced in its body, add a *magic table expression* to P^{mg}. The head is $m\_q_i(\overline{t_i^b})$. The body contains all tables that precede q_i in the sips (defined below) associated with r, and the magic table $m\_p(\overline{t^b})$.

4. Create a *seed* tuple $m\_q(\overline{c})$ from the equality predicates in the outermost query block, where $\overline{c}$ is the set of constants equated to the bound arguments of q.

Note that there is a magic table associated with each table in P^{ad}. If several table expressions with the same head are generated, they are replaced with a single table expression in which the body is the union of the bodies.

[7] In Section 4.3, we show how to combine these two steps.

Intuitively, magic-sets transformation involves adding *magic* tables to the `FROM` clause and equijoin predicates to the `WHERE` clause of each SQL statement.

EXAMPLE 3.1 (Magic-sets Transformation): Consider the query D of Example 2.1. We need to evaluate the average salary of a department in the view dep_avgsal if, and only if, the department has a senior programmer, as otherwise the average salary is not relevant. Magic-sets achieves this optimization by defining a magic table ($M3$), and rewriting $D1$ and $D2$ as $M1$ and $M2$:

($M1$): SELECT Ename FROM emp, dep_avgsal^{bf}
WHERE Job = "Sr Programmer" AND Sal > Asal
AND emp.Dno = dep_avgsal^{bf}.Dno.

($M2$): dep_avgsal^{bf}(Dno, Asal) AS
(SELECT Dno, AVG(Sal)
FROM m_dep_avgsal^{bf}, emp
WHERE m_dep_avgsal^{bf}.Dno = emp.Dno
GROUPBY Dno).

($M3$): m_dep_avgsal^{bf}(Dno) AS
(SELECT DISTINCT Dno FROM emp
WHERE Job = "Sr Programmer").

m_dep_avgsal^{bf} is a magic table for the view dep_avgsal^{bf}, giving the relevant departments for which the view needs to be evaluated. The superscript bf indicates that the view dep_avgsal^{bf} will always be evaluated with the first arguments bound to a set of tuples, and with the second argument free. □

Supplementary Magic-sets: In the magic query M of Example 3.1, the predicate Job = "Sr Programmer" is repeated in statements $M1$ and $M3$. The program S in Example 2.1 stores the result of the selection in s_mag, and uses it as a common subexpression when evaluating $S1$ and $S3$. s_mag is called the supplementary magic-set. Program D is transformed into program S using the supplementary magic-sets transformation ([BR87]). We use supplementary magic-sets in Section 6 because the performance advantage of using common subexpressions is important. For ease of exposition, we use magic-sets in other sections.

SIPS: A Sideways Information Passing Strategy is a decision as to how to pass information sideways in the body of a table expression while evaluating the table expression. The information passed comes from the predicates in the table expression. [BR87, MFPR90] define SIPS formally.

A sips can be *full*, meaning that all eligible predicates are used as soon as possible, or *partial*. A full sips can be defined by an ordering on the tables in the `FROM` clause. We refer to this order as the sips order.

The magic-sets transformation passes information sideways between tables being joined, according to a given sips order. In this paper, we assume that tables are listed in the `FROM` clause in the sips order.

Dependency Graphs: Dependency graphs are commonly used to detect recursions. In a table expression, the tables in the body (the `From` clause) are used to define the table in the head. If table q defines table r in some table expression, we denote this by $q \rightarrow r$, which is called a dependency edge. We define $\twoheadrightarrow$ to be the transitive closure of $\rightarrow$. A query is recursive iff its dependency graph has cycles, that is, if there exists a table q such that , $(q \twoheadrightarrow q)$. All tables in a strongly connected component (scc) of the dependency graph are said to be mutually recursive.

4 The Extended Magic-sets Transformation

The magic-sets transformation defined in Section 3 is applicable to relational systems with duplicates and aggregation. The definition borrows results from [MPR90], where semantics of duplicates and aggregation in presence of recursion is defined, and the use of aggregation is limited to the classes of *monotonic* and *magical stratified* programs, which are closed under the magic-sets transformation.

The magic-sets transformation was long believed to be useful only for propagating bindings (equality predicates). Our recent paper, [MFPR90], addresses the extension of the technique to propagating conditions (non-equality predicates) in Datalog programs, using a ground magic-sets transformation (GMT). In Section 4.1 we extend GMT to work in the presence of duplicates.

Further, we discuss how the magic-sets technique may be useful in purely nonrecursive systems (Section 4.2), and we present a one-phase algorithm for adorning and magic transforming a query that lets us push arbitrary conditions using just the b, c, and f adornments (Section 4.3).

4.1 Pushing Conditions using Magic

The ground magic-sets transformation for propagation of conditions, as presented in [MFPR90], does not preserve duplicate semantics. We consider a simple example:

EXAMPLE 4.1 Consider the following program P. p_1 and p_2 are arbitrary built-in predicates (conditions), and u, v, s, and w are EDB relations.

$(P1)$: $r(X, Z)$ AS
  ((SELECT X, Z
   FROM $t^{cf}(X, Y)$, $u(Y, Z)$ WHERE $p_1(X)$)
  UNION ALL
  (SELECT X, Z
   FROM $t^{cf}(X, Y)$, $v(Y, Z)$ WHERE $p_2(X)$)).

$(P2)$: $t^{cf}(X, Z)$ AS
  (SELECT X, Z FROM $s(X, Y)$, $w(Y, Z)$).

Let $s = [(1, 2), (1, 2)]$, $w = [(2, 3)]$, $u = [(3, 4)]$, and $v = [(3, 4)]$. Let $p_1(1)$ and $p_2(1)$ be true. The duplicate semantics of P defines r to be the multiset $[(1, 4), (1, 4), (1, 4), (1, 4)]$.

GMT transforms the definition $P2$ into:

$(M2)$: $t^{cf}(X, Z)$ AS
  (SELECT X, Z FROM $m(X, Y)$, $w(Y, Z)$).

$(M3)$: $m(X, Y)$ AS
  ((SELECT X, Y FROM $s(X, Y)$ WHERE $p_1(X)$)
  UNION DISTINCT
  (SELECT X, Y FROM $s(X, Y)$ WHERE $p_2(X)$)).

$P1$ is copied into $M1$ to complete the magic program. The view m has one copy of the tuple $(1, 2)$, consequently the view r has two copies of $(1, 4)$. As a result programs P and M are not duplicate equivalent. Simply defining m by `UNION ALL` does not help us, for then m will have four copies of $(1, 2)$, and r will have eight copies of $(1, 4)$.

As an aside, if either r or t was a `DISTINCT` table, GMT would preserve the query semantics. □

GMT constructs customized magic-sets (m), known as supplementary magic-sets, for each `SELECT` clause by combining the magic set ($p_1(X)$) with a table in the `FROM` clause. Preservation of duplicate semantics requires us to eliminate overlapping magic tuples (X values common to p_1 and p_2), while retaining duplicates in tables copied from `FROM` clause (s). Such an operation is not possible if the magic tables are never constructed, as is the case in GMT.

A straightforward solution is to construct the magic-sets explicitly, writing $M2$ and $M3$ as:

$(E2)$: $t^{cf}(X, Z)$ AS
  (SELECT X, Z
   FROM $m(X)$, $s(X, Y)$, $w(Y, Z)$).

$(E3)$: $m(X)$ AS
  ((SELECT X FROM $s(X, Y)$ WHERE $p_1(X)$)
  UNION DISTINCT
  (SELECT X FROM $s(X, Y)$ WHERE $p_2(X)$)).

m is the magic-set. The `UNION DISTINCT` operator eliminates duplicates after the union. Some joins are repeated in the above construction, such as the join with s. In the Starburst implementation, we have a solution that lets us use the supplementary transformation; we omit the description due to lack of space.

4.2 Magic-sets Transformation for Nonrecursive Programs

It is well-known that the magic-sets transformation has the undesirable property of merging scc's. Consequently a nonrecursive program can become recursive.

EXAMPLE 4.2 (Recursion due to Magic): In the program P,

$(P1)$: SELECT A, B FROM $r(A, C)$, $q(C, B)$
WHERE $A = 10$.

$(P2)$: $r(A, C)$ AS (SELECT A, C FROM $q(A, D)$, $t(D, C)$).
$(P3)$: $q(E, F)$ AS (SELECT E, F FROM $s(E, F)$).

q is used twice, once in $(P1)$, and once in $(P2)$, with a bf adornment at both places. q^{bf} gets bindings from r^{bf} $(P1)$, and from m_r^{bf} $(P2)$. Its magic-sets is thus a Union. The magic-query is

$(M1)$: SELECT A, B FROM $r^{bf}(A, C), q^{bf}(C, B)$
WHERE $A = 10$.

$(M2)$: $r^{bf}(A, C)$ AS (SELECT A, C
FROM $m\_r^{bf}(A), q^{bf}(A, D), t(D, C)$).

$(M3)$: $q^{bf}(E, F)$ AS
(SELECT E, F FROM $m\_q^{bf}(E), s(E, F)$).

$(M4)$: $m\_r^{bf}(10)$.

$(M5)$: $m\_q^{bf}(A)$ AS
(5a) ((SELECT C FROM $r^{bf}(A, C)$ WHERE $A = 10$)
UNION DISTINCT
(5b) (SELECT A FROM $m\_r^{bf}(A)$)).

Query (M) is recursive, as its dependency graph has the cycle $q^{bf} \rightarrow_{(M2)} r^{bf} \rightarrow_{(5a)} m\_q^{bf} \rightarrow_{(M3)} q^{bf}$ □

Many existing DBMS's do not support recursion. Usability of the EMS in such systems will be severely limited if recursive queries are produced as a result of the magic-sets transformation.

Consider Example 4.2. Table q^{bf} is recursive, but the newly introduced recursion is through the magic table, m_q^{bf} (as it must be for any recursion introduced by the magic transformation). $m\_q^{bf}(10)$ is computed from (5b), and leads to tuples in q^{bf} by (M3). These generate tuples for r^{bf} through $(M2)$. Tuples in r^{bf} generate new tuples in m_q^{bf} (5a) and thence in q^{bf}. But now, the new q^{bf} tuples cannot fire the body of (M2) to generate new r^{bf} tuples Thus the recursion does not "feed into itself", and terminates after one loop. The program can therefore be written nonrecursively.

We can avoid the introduction of recursion in the magic program by not recognizing common subexpression. If we treat the two uses of q in program P as two different tables, $q1$ and $q2$, the magic-sets transformation will not introduce recursion, as the reader may verify by performing the magic-sets transformation on a program P' derived from P with $q1$ and $q2$ defined according to $P3$.

We now make precise the intuition underlying the above example.

Proposition 4.1 *Given a query P, let M be the query obtained by magic transformation of P according to a set of full sips. Then, (A) If P is a tree structured*[8] *query, M is a dag*[9]*, and (B) if P is a dag, M will have bounded recursion that can be avoided altogether by not forming the common subexpressions.* □

4.3 Simple-bcf Adornments

The magic-sets transformations of [BMSU86, BR87, MFPR90] assume that an adorned program is available as input. The transformation thus requires two phases. The program is adorned in the first phase, and magic transformed in the second phase. In this subsection, we present a one-phase algorithm that does adornments and magic transformation together, and show how it can help in reducing the complexity of adornments.

We view adornments as providing three functions:

Function 1: The adornment α on a table t is an abstraction for the restriction on the table t at the point where it is used. This abstraction, α, and not the actual restriction, is used to decide how the table expressions for t^{α} will be evaluated.

Function 2: t^{α} is evaluated in an identical fashion for all restrictions that are abstracted by the adornment α (same sips, sips orders, join orders and adornments for tables referenced in t's table expressions). Thus if the abstraction is not a good one, t^{α} will be solved less than optimally for some of the restrictions. An adornment should be faithful ([MFPR90]) in that it should allow an optimal evaluation to be chosen for all restrictions (within the class of restrictions the adornment pattern is trying to capture) generating that adornment.

Function 3: Adornments specify when two uses of t can share the same copy of t as a common subexpression. The motivation behind the requirement that the uses have the same adornment is that the magic-sets generated from the uses be over the same arguments, which permits union.

The bcf adornment pattern introduced in [MFPR90] uses the c adornment for independent conditions only. A condition on an attribute X is said to be independent if it can be expressed without reference to any free (f) attribute. Thus $X > 10$ is independent. $X > Y$ is independent if Y is bound, otherwise it is dependent. The adornment algorithm and the following GMT of [MFPR90] work on the assumption that only independent conditions are pushed down, and that no conditions are deduced from the given ones. [MFPR90]

[8] A tree structured query does not have common subexpressions.

[9] A dag can have common subexpressions, but it does not have recursion.

also suggests that with the two-phase algorithm, it is not possible to capture and push down dependent and more general types of conditions using the bcf adornment pattern. Stronger adornment patterns are needed to push down such conditions.

In our one-phase algorithm, we generate magic-sets for a table t as we generate its adornments, *before* adorning the bodies of the table expression defining t. Later, when we determine the sips in the table expressions of t^α, and adorn the tables referenced, we know the actual conditions on t^α (since we can look up the magic-set), rather than just the adornment α. We then use these actual conditions in making a better choice on how to evaluate the table expression.

Define the simple-bcf adornment pattern to be similar to the bcf pattern of [MFPR90] (discussed in Section 4.1), except that a c adornment on an attribute now represents any type of condition on that attribute, not just an independent condition.

We now explain how having the magic-sets available while adorning a table expression for t enables the simple-bcf pattern to fulfil the three functions of adornments given earlier, even though the c adornment represents an arbitrary condition.

In the following lemma we borrow the definition of grounding tables from [MFPR90]. Given a condition p on a derived table t, a set of tables in the `FROM` clause of t containing all the attributes referenced in p, is called a grounding set. In statement $P2$ of Example 4.1, s is the grounding table for the restriction $X > 10$.

Lemma 4.1 *Let p_1 and p_2 be two restrictions that condition the same attributes of a derived table t. Then, if a set G is a grounding set for p_1, G is also a grounding set for p_2.*

Using Lemma 4.1, we show that the one-phase algorithm with simple-bcf adornments performs all the functions we want adornments to perform.

Function 1: With the actual restrictions on a table available at the time its body is adorned, adornments are no longer needed for Function 1. As a result, the abstraction they represent is not important.

Function 2: Function 2 can be done by the simple-bcf pattern for the class of arbitrary conditions; this follows from Lemma 4.1 and the Ground Magic-sets Transformation [MFPR90]. We illustrate with an example.

EXAMPLE 4.3 (Simple-bcf Adornment):

($P1$): `SELECT` X, Y, Z `FROM` $t(X, Y, Z)$
`WHERE` $X > 10$ `AND` $Y > 10$.

($P2$): `SELECT` X, Y, Z `FROM` $t(X, Y, Z)$
`WHERE` $X > Y$.

($P3$): $t(X, Y, Z)$ `AS` (`SELECT DISTINCT` X, Y, Z
`FROM` $q_1(X), q_2(Y), u(X, Z)$).

Both queries $P1$ and $P2$ generate the adornment t^{ccf}. By Lemma 4.1, $\{q_1, q_2\}$ is a grounding set for every restriction that conditions the first two arguments of t. q_1 and q_2 should be adorned differently for the two uses of t^{ccf} while u should be adorned u^{bf} for both uses. If we were adorning the program without constructing magic-sets, and without using any information besides the ccf adornment on t, we could not adorn as desired. However, using our one-phase algorithm, we get

($M1$): `SELECT` X, Y, Z `FROM` $t^{ccf}(X, Y, Z)$
`WHERE` $X > 10$ `AND` $Y > 10$.

($M2$): `SELECT` X, Y, Z `FROM` $t^{ccf}(X, Y, Z)$
`WHERE` $X > Y$.

($M3$): $t^{ccf}(X, Y, Z)$ `AS`
(`SELECT DISTINCT` X, Y, Z
`FROM` $m\_t^{ccf}(X, Y), u^{bf}(X, Z)$).

($M4$): $m\_t^{ccf}(X, Y)$ `AS`
((`SELECT` X, Y `FROM` $q_1^c(X), q_2^c(Y)$
`WHERE` $X > 10$ `AND` $Y > 10$)
`UNION DISTINCT`
(`SELECT` X, Y `FROM` $q_1^f(X), q_2^c(Y)$
`WHERE` $X > Y$)).

Note the different adornments for q_1 in the two `SELECT` statements of ($M4$), and the b adornment on X in ($M3$) after the magic-set is available. □

In general, for a c adorned table t, GMT moves the grounding tables of t^α into a supplementary table.[10] For any two restrictions $r1$ and $r2$ that generate the same simple-bcf adornment α on t, the grounding tables $\overline{q}$ are the same (Lemma 4.1). $\overline{q}$ will be copied into two tables, $M1$ along with $r1$, and $M2$ along with $r2$. If these restrictions are of an entirely different nature, different adornments and different sip orders for the tables $\overline{q}$ might be selected in $M1$ and $M2$. However, in the original texp for t^α, the supplementary magic-sets is seen as generating *bindings* for all b or c argument, whatever the nature of the restrictions. Since we do not distinguish between types of bindings while deciding on evaluation of a table expression, t^α's evaluation will be optimal for both restrictions.

Function 3: The simple-bcf pattern performs Function 3. If two uses of a table have the same arguments conditioned, their ground magic-sets are also over the same arguments.

[10] Subsection 4.1 pointed out that in some cases, this must be considered as a magic table. That is not a problem, but for simplicity let us assume we always have the supplementary table.

5 An Outline of the Extended Magic-sets (EMS) Implementation

We are implementing EMS in Starburst. We have written the pseudo-code, and have C code that executes the transformation in simple cases. In this section we give a sketch of our implementation.

EMS is a part of the query-rewrite phase of the Starburst optimizer. Rewrites are done by a (production) rule-based system that encodes each query transformation as a rewrite rule ([HP88]). A forward chaining engine traverses the query graph depth first (normally), applying rewrite rules. EMS is applied to graph elements representing table expressions, and it is applied to one table expression at a time. Multiple firings of the EMS rule, as the graph is traversed, cumulatively produce a transformed query.

Starburst includes a number of rewrite rules besides the magic-sets rule. The predicate pushdown rules determine what predicates get pushed from table expressions into referenced tables, and in what form. The EMS rule then places the predicates in the right place (as a magic-set).

The way in which magic-sets is applied to a table expression can depend upon the operation in the table expression. For example, magic cannot be applied to operations such as `GROUPBY` (the bad operations) exactly as it is applied to operations such as `SELECT` (the good operations).

When magic processing acts on a table expression for table t, previous processing ensures that the head t is adorned, a magic-table mt for t is available, and (for good operations) that mt is grounded and joined into the body of the table expression. Also, the sips order within the table expression must be known. [11]

During magic processing of t, all predicates in the table expression are pushed into each table r referenced in the table expression (using the predicate pushdown rules). An adornment α for each r is determined, and a table expression for r^α, with a body identical to that of the table expression for r, is created. The magic-sets m_r^α for r^α from its use in t is formed, and if r is good, m_r^α is grounded and added to the table expression for r^α.

Magic processing is performed for every table expression visited in a traversal of the query graph. We avoid repeatedly processing a table expression except for bad table expressions under special conditions. The following theorem holds for our EMS algorithm (assuming we first get rid of all cycles in query Q consisting entirely of bad tables.[12])

[11] The appropriate interplay of cost optimization and transformational rewrite optimization is an open problem mentioned in the conclusion of this paper.

[12] By inserting a `SELECT` operation in the cycle.

Theorem 5.1 *For any query graph Q, EMS terminates, and $EMS(Q)$ is equivalent to Q under the evaluation strategy of Starburst.*

The adornment and the magic-set transformation are combined in a one-phase algorithm (Section 4.3). Mostly, the simple-bcf adornment is used, although bad operations require special refined adornments. EMS is extensible with respect to (1) new operations in table expressions, and (2) the traversal strategy (depth-first, breadth-first, bottom-up, etc.).

6 Performance

In this section we present performance measurements that illustrate how EMS accelerates complex queries (such as decision-support queries) consisting of several query blocks. It is not uncommon for such queries to take hours (or even days) to complete. Query transformation can improve performance by several orders of magnitude.

A comprehensive performance evaluation requires a definition of a benchmark database and a set of queries for a particular workload. We focus on a complex query workload (with multiple predicates, joins, aggregations and subqueries), rather than a transaction workload, where queries are relatively simple. Although transaction benchmarks have been proposed, [A$^+$85, TPC89], complex query workloads are still at a preliminary stage ([TOB89, O'N89]). To measure the performance effect of the magic-sets transformation, we employ a scaled up (by a factor of 10) version of the DB2 benchmark database described in [Loo86].

Magic-sets transformations have been studied in the context of recursive queries, and the usefulness of magic-sets for recursive queries is explained in [BR86, BR87]. In this section we study nonrecursive queries.

Our performance measurements were done on the IBM DB2 V2R2 relational DBMS using the DB2PM performance monitoring tool [DB88] to determine elapsed time (total time taken by system to evaluate the query) and I/O time (the time for which I/O devices are busy). We measured the performance of each query both before and after applying the magic-sets transformation. Both representations of the query went through the query compilation process, including cost optimization. Performance figures for several of the queries we measured are described below.

The DB2 benchmark database is based on an inventory tracking and stock control application. Workcenters, represented by wkc table have locations (locatn). Items (itm) are worked on at locations within workcenters, and the table itl captures this relationship. Each item may have orders (itp). Some physical characteristics of the database are shown in Table 1.

Predicate pushdown and set-oriented computation are the two key factors in query optimization and ex-

Table	Tuple Size	#Tuples	#Cols	#4K Pgs
itm	34	170 000	4	1 850
wkc	28	500	10	5
itl	78	2 550 000	13	57 980
itp	43	339 440	14	4 250

Table 1: Benchmark Database

ecution. The magic-sets transformation enables us to take advantage of both. Advantages of pushing down local predicates, such as (Job = "Sr Programmer") in query $D1$ of Example 2.1, are well-known. We concentrate on pushdown of join predicates that pass information sideways (SIPS predicates), such as (emp.Dno = dep_avgsal.Dno) in query $D1$ of Example 2.1.

Set-oriented computation is desirable as it usually leads to improved performance over an equivalent fragmented or tuple-at-a-time computation. Pushing join (or sips) predicates by correlation fragments the computation, causing the subquery to be evaluated once for each value passed down. As a result, we may lose the efficiency of sequential prefetch ([TG84]) because each computation fragment does not access enough pages to take full advantage of sequential prefetch in terms of amortizing the cost of an I/O call across a large number of pages. Inefficiency can also arise in accessing data through nonclustered indices. If computation is not fragmented, we extract the TID (tuple ID) of qualified tuples from the index, sort the results by page IDs, and then do the I/Os ([MHWC90]). Hence, each relevant data page is retrieved only once. In a fragmented computation the same page may be retrieved many times, once by each computation fragment that is interested in a tuple on the page. Further, each fragment has a certain fixed cost associated with operations such as opening and closing scans, and sort initializations (e.g., initialization of the tournament trees when tournament sorts are used). In a set-oriented computation, the fixed costs are incurred only once. With correlation the fixed costs are incurred once for each evaluation of the subquery. Query transformations that result in non set-oriented computation can therefore degrade performance significantly, as we see in Section 6.3.

Evaluation of performance of magic-sets is based on the two key factors discussed above: predicate pushdown (or sideways information passing) and set-oriented computation. The effect of predicate pushdown depends on how bindings affect the query plan of (a piece of) a query. For example, the magic-sets transformation may provide bindings for a column, so an index on that column becomes an efficient access path. The effect of set-oriented computation depends on the cardinality of the binding set (with and without duplicates). The higher the cardinality, the greater is the benefit of using set-oriented information passing. There are numerous queries where the above two factors are important. We now present some of the many queries we used in our experiments.

6.1 Experiment 1

In this experiment, selective bindings are passed to a subquery. The collection of bindings does not contain duplicates. The experiment uses the view $V1$ which, for each item and workcenter, computes the average time spent[13] on that item in locations within the workcenter.

```
(V1): vitemtime(itemn, wkcen, avgtime) AS
        (SELECT itemn, wkcen, AVG(loctime) FROM itl
         GROUPBY itemn, wkcen).
```

Consider the query $Q1$: For items ordered with a quantity (qcomp) of 450, find the average time spent on that item in locations in each workcenter that work on the item.

```
(Q1): SELECT DISTINCT itm.*, wkcen, avgtime
      FROM itp, itm, vitemtime
      WHERE itp.qcomp = 450 AND itp.itemn = itm.itemn
            AND itp.itemn = vitemtime.itemn.
```

The following plan to solve $Q1$: "Compute the view vitemtime, store it in a temporary table, and use is to compute $Q1$", took about three hours to execute. The plan is inefficient since the view is computed for all items even though the query needs the view on a small subset of the items (the predicate on quantity is very selective). We can avoid the redundant computation by passing into the view (through query rewrite) a set of bindings on items for which the view needs to be computed.

The bindings can be passed by either correlation or magic-sets. With correlation, the predicate (itp.itemn = vitemtime.itemn) is pushed into the table expression corr_itemtime, filtering out computation of many groups. The correlated query[14] is:

```
(C1): SELECT DISTINCT itm.*, wkcen, avgtime
      FROM itp, item,
           corr_itemtime(itemn, wkcen, avgtime) AS
           (SELECT itemn, wkcen, AVG(loctime) FROM itl
            WHERE itp.itemn = itl.itemn
            GROUPBY itemn, wkcen)
      WHERE qcomp = 450 AND itp.itemn = itm.itemn.
```

The plan for $C1$ evaluates the view corr_itemtime multiple times. During each evaluation, the index

[13] A location works on an item for loctime = finishtime − starttime

[14] The view becomes a correlated table expression. Standard SQL does not allow correlated table expressions. We did the experiment using a variant of $C1$ whose execution cost is close, but definitely less, than the execution cost of $C1$.

on itemn column of the itl table is used, and only the relevant tuples are retrieved. The predicate on itp.qcomp is such that there are no duplicates in the bindings (itp.itemn) passed into the view.

With the magic-sets transformation, the supplementary magic-set, s_mag, is computed as a temporary ($M1a$), s_mag is used in computing a reduced view mag_itemtime ($M1b$), and the original query is rewritten using the reduced view ($M1c$).

```
(M1a): s_mag AS
       (SELECT DISTINCT itm.* FROM itp, itm
        WHERE qcomp = 450 AND itp.itemn = itm.itemn).

(M1b): mag_itemtime(itemn, wkcen, avgtime) AS
       (SELECT itl.itemn, itl.wkcen, AVG(loctime)
        FROM s_mag, itl WHERE s_mag.itemn = itl.itemn
        GROUPBY itl.itemn, itl.wkcen).

(M1c): SELECT DISTINCT s_mag.*, wkcen, avgtime
       FROM s_mag, mag_itemtime
       WHERE s_mag.itemn = mag_itemtime.itemn.
```

The plan to solve $M1$ computes the view mag_itemtime by a nested-loop join, with s_mag (a small table) as the outer and itl (a large table) as the inner, using the index on itemn column of itl to access only the relevant itl tuples. The join is followed by grouping and aggregation.

Performance results are summarized in Table 2. For each query, we give the elapsed and I/O times. The figures are normalized with respect to a value 100 for the elapsed time of the original query. Thus, an elapsed time figure of 0.40 for correlation in Experiment 1 means that the correlated query took 0.40% of the time taken by the original query, while the I/O figure of 0.280 means that the I/O system was busy for 0.280% of the *elapsed time* for the *original* query.

Both correlation and magic-sets improved performance by *two orders* of magnitude, reducing the elapsed time from over three hours to under one minute. Neither technique was significantly better than the other, since both led to very similar plans for computing the vitemtime view, which was the expensive part of query $Q1$. With correlation, the bindings on itemn were directly used to access itl through an index. With magic-sets, a nested loop join retrieved the set of bindings from s_mag and used them to access itl in exactly the same way. Correlation was marginally faster because the magic query $M1$ needed to compute the supplementary magic-set.

6.2 Experiment 2

This experiment examines the effect of duplicate values in the set of bindings on performance. Experiment 1 is modified by changing the predicate on itp so that it gives us 95 items, each with 100 orders. As a result there are 100 copies of each distinct binding value (itemn). Performance results are summarized in Table 2.

Query	Experiment 1		Experiment 2	
	Time	I/O	Time	I/O
Original	100.00	6.855	100.00	6.386
Correlated	0.40	0.280	2.12	0.013
Magic	0.47	0.390	0.28	0.014

Table 2: Relative Elapsed and I/O Times for Queries of Experiments 1 and 2.

Correlation computes the view corr_itemtime for every copy of every binding value coming from the outer query. Magic-sets does significantly better because it eliminates duplicate bindings before storing them in s_mag. Correlation can be improved so as to eliminate duplicate view evaluations. The result of each evaluation, along with the binding value used in the evaluation, can be saved in a temporary table, and duplicate evaluations replaced by a table lookup. We believe that such a modification will make correlation competitive with magic-sets on Experiment 2.

6.3 Experiment 3

This experiment shows the advantage of set-oriented information passing using magic-sets. There are no duplicates in the binding set.

Consider the view $V3$: For each workcenter, find the average times spent on items by locations of a certain type in this workcenter. $V3$ is similar to $V1$, except that we filter out some locations, and project out the itemn column from the output.

```
(V3): itlvagg(wkcen, avgtime) AS
      (SELECT DISTINCT wkcen, AVG(loctime) FROM itl
       WHERE locan ≥ "loca50" AND locan ≤ "loca55"
       GROUPBY itemn, wkcen).

(Q3): SELECT wkc.deptn, wkc.wkcen, avgtime
      FROM wkc, itlvagg
      WHERE nmach ≤ 3 AND wkc.wkcen = itlvagg.wkcen.
```

Query $Q3$ asks for the information from view $V3$ for workcenters having 3 or fewer machines. It is executed by computing the full view itlvagg and joining it with the wkc table.

Correlation and Magic-sets rewrite $Q3$ so as to use the predicate (nmach ≤ 3) to avoid computing the full view. Only 10 workcenters satisfy the predicate; 10 binding values are thus passed into the view — individually by correlation, as a set by magic-sets. We also use a variant of $Q3$ with the predicate (nmach ≤ 6) that is satisfied by 100 workcenters.

In the correlated rewrite, the predicate (wkc.wkcen = itlvagg.wkcen) is pushed into the view. The magic query is obtained in a manner similar to that in Experiment 1. The supplementary magic-set, s_mag,

Query	10 bindings		100 bindings	
	Time	I/O	Time	I/O
Original	100.00	25.74	100.00	25.67
Correlated	513.27	232.47	5136.49	2322.61
Magic	55.45	23.22	109.00	23.62

Table 3: Relative Elapsed and I/O Times for Queries of Experiment 3.

Query	Time	I/O
Original	100.00	7.93
Correlated	52.56	25.62
Magic	8.68	3.64

Table 4: Relative Elapsed and I/O Times for a Variation of Experiment 3.

computes the relevant workcenters, and is used to restrict the view ($M3b$). The reduced view is used to answer the query ($M3c$).

```
(M3a): s_mag(deptn, wkcen) AS
         (SELECT deptn, wkcen
          FROM wkc WHERE nmach ≤ 3).

(M3b): mag_itlvagg(wkcen, avgtime) AS
        (SELECT DISTINCT itl.wkcen, AVG(loctime)
         FROM s_mag, itl
         WHERE locan ≥ "loca50" AND locan ≤ "loca55"
               AND s_mag.wkcen = itl.wkcen
         GROUPBY itl.wkcen, itl.itemn).

(M3c): SELECT deptn, s_mag.wkcen, avgtime
       FROM s_mag, mag_itlvagg
       WHERE s_mag.wkcen = mag_itlvagg.wkcen.
```

Performance results are summarized in Table 3. Correlation fares very poorly. For 10 binding values it performs five times worse than the original query, and degrades by a factor of ten when the binding set is increased by a factor of ten. The magic query performs reasonably well; it is clearly preferable to the original query with 10 bindings, and is competitive with 100 bindings. As the cardinality of the binding set increases, performance of the magic query is stable.

In the correlated query, the view itlvagg is computed by accessing the itl table via an index on locations. However, the binding set contains workcenters, not locations, so that each computation of the view repeats the indexed access to all tuples of itl that satisfy the location predicate. itl is a large table, even when limited to a few locations, and the access cost is substantial. As most of the access costs are repeated for each binding value, the cost of the query is almost linear in the cardinality of the binding set. In the corresponding magic query, itl is accessed only once and joined with the full set of bindings. The modification to correlation suggested in Subsection 6.2 cannot improve performance of correlation on this experiment.

Experiment 3 shows the stability of the magic-sets transformation — even when it turns out not to be the optimal choice (because predicate selectivity estimates are wrong), it tends not to be much worse than the winning alternative. Since the primary goal of optimization is to avoid bad plans (and the secondary goal is to find a pretty good one), the magic-sets transformation often meets optimization goals better than correlation and decorrelation, which are considerably less stable. Unstable query transformations require the optimizer to estimate the cost of queries carefully. Due to the extremely high cost of the optimization process, the role of stable heuristics is becoming increasingly important ([Pir89]). For this reason, the stability of magic-sets is very valuable.

Table 4 summarizes the performance results of Experiment 4, a variation on Experiment 3 with 10 bindings. The view is similar to $V3$, but a join of itl with itp and another table is needed before grouping. As a result, the grouped relation is large, and the grouping cost is significant. Magic-sets performs better than both correlation (due to set-oriented computation) and the original query (due to reduction in cost of grouping), and is a clear winner.

7 Conclusion

In this paper, we showed that the magic-sets transformation can be extended to handle general SQL constructs. We sketched the implementation of Extended Magic-sets as part of the rewrite component of a relational database system prototype, and presented a performance study contrasting magic-sets with correlation and decorrelation. Many significant results were abbreviated or omitted, including aspects of refined adornments, simple adornments and implementation details.

We believe that this paper demonstrates that the magic-sets technique (which formerly was a tool only for Datalog and logic programming) should be considered a practical extension of existing rewrite optimization techniques. Magic is indeed "relevant" for relational database systems; it is a general technique (applicable to nonrecursive as well as recursive queries) for introducing predicates that filter out accesses to irrelevant rows of tables as soon as possible. Database systems have been using limited variants of this for many years.

We do not suggest that the magic-sets transformations should be employed whenever they are applicable. Rather, magic is a valuable alternative that appears to be more stable than both correlation and decorrelation, subject to trade-offs that must be evaluated by a cost optimizer [SAC+79, Loh88]. Magic may be especially valuable for queries (such as decision-support queries) involving large numbers of joins, complex nesting of query blocks, or recursion. Such queries may be infeasible unless magic-sets is applied.

A number of special optimization techniques have been proposed in the literature. Some of these can be viewed as alternatives to magic-sets that try to exploit special properties of certain queries, such as linear queries on acyclic data (for e.g., Henschen-Naqvi [HN84], Counting [BMSU86]), or special operators to express a restricted (and important) class of queries such as transitive closure (e.g., The alpha operator [Agr87]). When applicable, the above techniques are sometimes better than the magic-sets transformation. However, Example 1.2 illustrates that there are useful queries that cannot be expressed using linear recursion. The importance of magic-sets is that it is applicable to all (extended) SQL queries and provides a general optimization framework with good, stable performance. There are also techniques that further refine the magic-sets approach for certain classes of queries (e.g., factorization [NRSU89]). These complement the magic-sets approach by recognizing special properties of the program and optimizing the transformed program suitably.

Although we are implementing magic as an extension of the rewrite optimization component in the Starburst extensible relational database prototype, many practical questions remain. One difficult open problem is the integration of rewrite optimization and cost optimization. Cost optimization may take time and space exponential in the number of tables joined. Transformations such as magic-sets may introduce exponentially many alternative queries, each of which requires cost optimization of a query more complex than the original. Clearly there is a structural relationship among the many query transformations, but we do not understand this problem well enough yet to reduce it to a manageable level by either algebraic techniques or by engineering heuristics.

8 Acknowledgements

The Starburst project at IBM Almaden Research Center provided a stimulating environment for this work. We particularly thank Ted Messinger for helping us in setting up the DB2 benchmark database. Bruce Lindsay, Guy Lohman, Ulf Schreier, Jeff Ullman and the referees made helpful comments on drafts of this paper. We thank Jeff Ullman and members of the *NAIL!* group at Stanford for many insightful discussions on this subject.

References

[A+85] Anon et al. A Measure of Transaction Processing Power. Datamation, April 1985.

[ABC+76] M. Astrahan et al. System R: Relational approach to database management. ACM TODS, 1(2), June 1976.

[AC89] F. Afrati and S. Cosmadakis. Expressiveness of Restricted Recursive Queries. In STOC 1989.

[Agr87] R. Agrawal. Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries. In Data Engineering 1987.

[BC81] P. Bernstein and D. Chiu. Using Semijoins to Solve Relational Queries. JACM, 28(1):25–40, 1981.

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic Sets and other Strange Ways to Implement Logic Programs. In PODS 1986.

[BR86] F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In SIGMOD 1986.

[BR87] C. Beeri and R. Ramakrishnan. On the Power of Magic. In PODS 1987.

[Day87] U. Dayal. Of Nests and Trees: A Unified Approach to Processing Queries that Contain Nested Subqueries, Aggregates, and Quantifiers. In VLDB 1987.

[DB88] IBM. IBM Database 2 Performance Monitor Report Reference, December 1988. Number: IBM SH20-6858.

[GW87] R. Ganski and H. Wong. Optimization of Nested SQL Queries Revisited. In SIGMOD 1987.

[HFLP89] L. Haas, J. Freytag, G. Lohman, and H. Pirahesh. Extensible Query Processing in Starburst. In SIGMOD 1989.

[HN84] L. Henschen and S. Naqvi. On Compiling Queries in Recursive First-Order Databases. JACM, 31(1):47–85, January 1984.

[HP88] W. Hasan and H. Pirahesh. Query Rewrite Optimization in Starburst. IBM Research report, RJ 6367 (62349), August 1988.

[ISO89] ISO_ANSI. ISO-ANSI Working Draft: Database Language SQL2 and SQL3; X3H2; ISO/IEC JTC1/SC21/WG3, 1989.

[Kim82] W. Kim. On Optimizing an SQL-like Nested Query. TODS, 7(3), Sep 1982.

[Loh88] G. Lohman. Grammar-like Functional Rules for Representing Query Optimization Alternatives. In SIGMOD 1988.

[Loo86] C. Loosley. Measuring IBM Database 2 Release 2 - The Benchmark Game. InfoDB, 1(2), 1986.

[MFPR90] I. Mumick, S. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic Conditions. In PODS 1990.

[MHWC90] C. Mohan, D. Haderle, Y. Wang, and J. Cheng. Single Table Access Using Multiple Indexes: Optimization, Execution, and Concurrency Control Techniques. In EDBT 1990.

[MPR90] I. Mumick, H. Pirahesh, and R. Ramakrishnan. The Magic of Duplicates and Aggregates. In VLDB 1990.

[NRSU89] J. Naughton, R. Ramakrishnan, Y. Sagiv, and J. Ullman. Arity Reduction Through Factoring. In VLDB 1989.

[O'N89] P. O'Neil. Revisiting DBMS Benchmarks. Datamation, Sep 1989.

[PF89] H. Pirahesh and S. Finkelstein. Semantics of the Query Graph Model. IBM Internal Report, 1989

[Pir89] H. Pirahesh. Early experience with rule-based query rewrite optimization. In Optimization workshop, SIGMOD 1989.

[Ram88] R. Ramakrishnan. Magic Templates: A Spellbinding Approach to Logic Programs. In ICLP 1988.

[RBF+80] J. Rothnie Jr. et al. Introduction to a System for Distributed Databases (SDD-1). TODS, 5(1):1–17, March 1980.

[SAC+79] P. Selinger et al. Access Path Selection in a Relational Database Management System. In SIGMOD 1979.

[TG84] J. Teng and R. Gumaer. Managing IBM Database 2 Buffers to Maximize Performance. IBM System Journal, 23(2), 1984.

[TOB89] C. Turbyfill, C. Orji, and D. Bitton. AS3AP - A Comparative Relational Database Benchmark. In IEEE Compcon 1989.

[TPC89] TPC benchmark group. TPC Benchmark, A Draft 6-PR Proposed Standard, 1989. Available from ITOM International Co., PO Box 1450, Los Altos, CA 94023.