Extensible/Rule Based Query Rewrite Optimization in Starburst

Hamid Pirahesh
Joseph M. Hellerstein*
Waqar Hasan†

IBM Almaden Research Center
San Jose, CA 95120, USA
pirahesh@almaden.ibm.com, joey@cs.berkeley.edu, hasan@hpl.hp.com

Abstract

This paper describes the Query Rewrite facility of the Starburst extensible database system, a novel phase of query optimization. We present a suite of rewrite rules used in Starburst to transform queries into equivalent queries for faster execution, and also describe the production rule engine which is used by Starburst to choose and execute these rules. Examples are provided demonstrating that these Query Rewrite transformations lead to query execution time improvements of orders of magnitude, suggesting that Query Rewrite in general — and these rewrite rules in particular — are an essential step in query optimization for modern database systems.

1 Introduction

In traditional database systems, query optimization typically consists of a single phase of processing in which access methods, join orders and join methods are chosen to provide an efficient plan for executing a user's declarative query. We refer to this phase as *plan optimization.* In this paper we present a distinct phase of query optimization, Query Rewrite, which has been implemented in the Starburst DBMS [HCL+90] to precede plan optimization in the processing of a query.

The goal of Query Rewrite is twofold:

1. *Make queries as declarative as possible:* In database languages such as SQL, it is often possible for a poorly expressed query, though ostensibly declarative, to force typical plan optimizers into choosing sub-optimal execution plans. A major goal of Query Rewrite is the transformation of such "procedural" queries into equivalent but more declarative queries.
2. *Perform natural heuristics:* Certain heuristics can be performed in Query Rewrite and are generally accepted in the literature as being valuable. A typical example is that of "predicate pushdown", in which predicates are applied as early as possible in the query (*i.e.* they are "pushed" from their original positions into table accesses, subqueries, views, etc.) Such rules can significantly improve query execution time, and while a few of these heuristics are done in typical plan optimizers, they often can be applied in a more general way during Query Rewrite.

*Current address: CS Division, Department of EECS, University of California, Berkeley, CA 94720

†Current address: Hewlett-Packard Laboratories, Palo Alto, CA 94305

Although it is accepted doctrine that query languages should be declarative, we shall see in our examples that alternative but equivalent expressions of a query can have widely varying performance, often differing by orders of magnitude. It is therefore our conviction that Query Rewrite is an essential step in query optimization, since it further ensures that the expression of the query will be insignificant with respect to its performance.

1.1 Queries with Path Expressions

The goals of Query Rewrite explained above are even more significant in Object Oriented applications, which typically generate complicated queries with "path expressions" connecting various collections of objects [BTA90, LLOW91, LLPS91]. In such applications, both the *complexity of the logic* and the *volume of the data* are far greater than in traditional DBMS applications [HSS88]. As a result, query optimization becomes increasingly important.

The following is an example of a query involving path expressions, using the Object SQL syntax defined in [BTA90].[1] This query is a small variation of an example presented in [BTA90]. The example database contains records of patients. Medical records are *set attributes* of patients. All accesses to data are via methods. Given a patient's record, the medical records are returned by the function *get_medical_records.* The example retrieves male patients who have been diagnosed with malaria or smallpox prior to '10/10/89'. The *FROM* clause enumerates the patients and the *WHERE* clause restricts the patients to males and checks for the existence of a malaria or smallpox diagnosis prior to a given date.

```
SELECT DISTINCT P
FROM Patient p  IN Patient_Set
WHERE p.sex == 'male' &&
  EXISTS ( SELECT r
           FROM Medical_record r  IN p.get_medical_record()
           WHERE r.get_date() < '10/10/89' &&
             ( r.get_diagnosis() == 'Malaria' ||
               r.get_diagnosis() == 'Smallpox' );
```

Queries such as this essentially involve path expressions in which, given a record, the related information is obtained through a path (*e.g.*, getting the medical records of a patient). Queries involving path expressions are very common in complex applications such as CAD/CAM. In general, many path expressions may be involved in a query, and each may have a length of more than one. Finding an efficient execution plan for such path expressions is a problem very similar to that of optimizing (nested) SQL subqueries. The above query is commonly executed by enumerating the male patients,

[1]This is a proposal for a standard OO query language, and is implemented by some commercial OODBMSs

and checking the medical records *for each* patient, testing for the predicates on date and diagnosis. This amounts to a nested-loop join, with the join order dictated by the user — the subquery is necessarily the inner of the join. However, if there are a large number of male patients, this could be very inefficient, since most of the records accessed are not qualified. Assuming both smallpox and malaria are rare diseases, the performance of such a query can be improved by orders of magnitude by first searching the medical records through an index on the *diagnosis* attribute of the medical records, and then finding the associated patients. This basically requires converting such queries to joins if possible, which can then be executed by a greater variety of plans.

Changing the join order in the above example improves the performance considerably, but has an undesirable side-effect. A given patient may have been diagnosed with both smallpox *and* malaria, and for each diagnosis, the associated patient record is sent to the output, resulting in duplicate patient records in the output. This is not correct; for example, this would cause the wrong result to be given by a query that counted the output records of the above query.

Duplicates may occur even without query transformations. Duplicate records frequently appear in base or intermediate results in applications, and are of great significance to queries involving aggregate functions, such as a query requesting an average of values of some column. Duplicates are part of SQL [ISO91] and OO models such as [LLOW91] (as "bags of objects"). The need and importance of having duplicates in a realistic implementation is widely recognized, particularly in relational DBMSs (RDBMSs). Hence, careful treatment of duplicates is essential, and is one focus of the work presented in this paper.

1.2 System Design

Naturally we do not propose to find an optimal expression of a query, just as traditional plan "optimizers" do not find an optimal execution plan for a query. Rather, we have built a set of Query Rewrite heuristics, expressed as production rules, which work together to address both of the rewrite goals stated above. These production rules are controlled by a rule engine written for the purpose and integrated into Starburst. Our rule system design has been instrumental in facilitating our experimentation with query transformations for two reasons. First, the rule system paradigm has made it easy for us to exploit the complicated triggering interactions between rewrite rules, saving us from the task of explicitly laying out the flow of control between rules. This is often cited as a dangerous complexity of rule systems ([ZH90], [Ras90], [HH91], etc.) but in our experience has been not only manageable but inherently useful. Second, a rule system is an excellent platform for extensibility, one of the key goals of Starburst's design. This extensibility has allowed us to write and test dozens of query transformations over the past two years, including those presented in this paper, magic sets transformations [MFPR90a, MPR90, MFPR90b], and numerous others.

As we shall see, the rules presented in this paper demonstrate that Query Rewrite can often speed up query execution by orders of magnitude, suggesting that query transformation schemes form a ripe area of research. Our extensible Query Rewrite system is designed with this in mind, and we expect to continue adding transformations to it.

1.3 Related Work

Designers of early RDBMSs such as System R [ABC+76] and INGRES [SWK76] recognized the importance of merging views, and achieved this under limited circumstances. In spite of the acknowledged importance of such transformations, few systems have expanded upon these early transformation designs.

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 eliminating nested subqueries. These papers recognize the importance of merging of subqueries. [Kim82, GW87] also deal with subqueries containing aggregation. We have reported our set of rules that deal with such subqueries in [MFPR90a, MPR90, MFPR90b].

Ganski's paper illustrates the complexity of query rewrite, since it has to emend some previous transformations which were incorrect. This complexity supports our approach of decomposing transformations into an extensible set of distinct rules, such that each rule can be shown to be correct separately.

We have paid particular attention to language orthogonality. Hence, operations such as UNION, INTERSECT, and EXCEPT (SQL's equivalent of set difference) may appear in subqueries, as is required by the SQL2 standard [ISO91]. The work mentioned above does not deal with these more complicated subqueries. Furthermore, our rules *guarantee* the merge of existential subquery conjuncts consisting of restriction, projection and join. This is partially due to our careful treatment of duplicates.

Anfindsen [Anf89] also has done a study of subquery transformations using IBM's DB2 RDBMS for performance measurements. Like us, Anfindsen reports orders of magnitude improvement in performance. However, [Anf89] restricts itself to those transformations which result in a SQL query that can be handled by DB2. In contrast, our approach is an internal and integrated part of an RDBMS, taking advantage of a richer internal language, and hence allowing for considerably more optimization. Anfindsen defines a concept similar to our **one-tuple-condition**, explained below, and gives sufficient conditions for which it is satisfied.

Many extensions have been added to Starburst [LLPS91], including XNF, a system supporting complex object queries which often generates extremely complicated SQL queries. Our Query Rewrite system has withstood the test of being used by these extensions, and in fact is key to making some of them work efficiently.

The work presented here should not be confused with the query rewrite facility of POSTGRES [SJGP90]. POSTGRES's query rewrite is part of an implementation for an active database. In POSTGRES, one can define a rule stating that certain incoming queries should perform additional or entirely different actions from what the user has specified. In some situations, POSTGRES implements this notion by rewriting the user's query. Note well that POSTGRES's query transformations are intended to change a query's *semantics*, not its performance. For example, POSTGRES's rewrite may be used to implement user-defined semantics for update of views. In contrast, our emphasis is on transformation for the purpose of optimizing query execution.

An earlier design of the Starburst Query Rewrite rule system is reported in [HP88].

1.4 Structure of the Paper

Section 2 presents the abstract representation of queries used by the rewrite rules. The rules themselves are presented in Section 3. Section 4 describes the rule engine designed for Query Rewrite. Summary and conclusions appear in Section 5.

2 Starburst's Query Graph Model

Queries are internally represented in a Query Graph Model (QGM). The goal of QGM is to provide a more powerful and conceptually more manageable representation of queries in order to reduce the complexity of query compilation and optimization. QGM supports arbitrary table operations, where the inputs are tables and outputs are tables. Examples of operations are SELECT, GROUP BY, UNION, INTERSECT and EXCEPT. The operation SELECT is that part of Starburst SQL which handles restriction, projection and join.

We present the QGM graph through an example. Suppose we have the following SQL query:

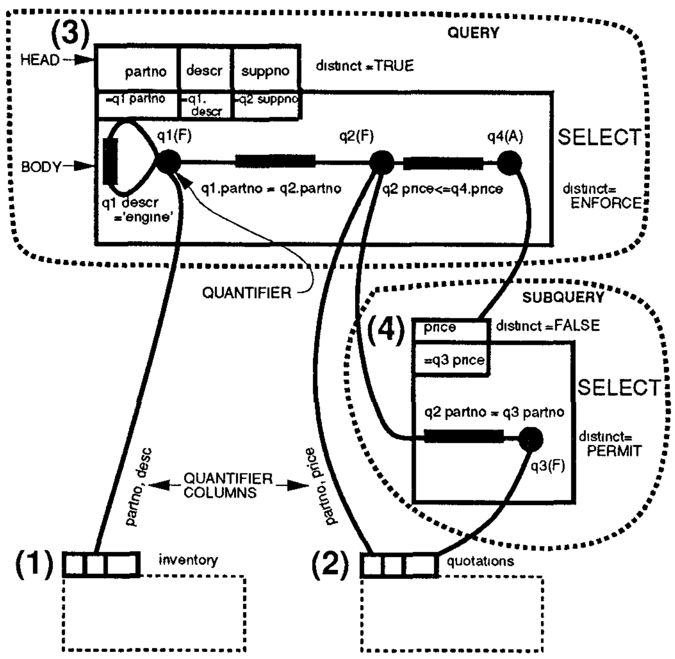

Figure 1: Example QGM graph

```
SELECT DISTINCT q1.partno, q1.descr, q2.suppno
FROM inventory q1, quotations q2
WHERE q1.partno = q2.partno AND q1.descr='engine'
  AND q2.price ≤ ALL
        ( SELECT q3.price FROM quotations q3
          WHERE q2.partno=q3.partno);
```

This query gives information about suppliers and parts for which the supplier price is less than that of *all* other suppliers. Figure 1 shows the QGM for this query. The graph contains four boxes. Boxes 1 and 2 are associated with base tables *inventory* and *quotations*. Box 3 is a SELECT box associated with the main part of the query, and Box 4 is a SELECT box associated with the subquery. Each box has a head and a body. The head describes the output table produced by the box, and the body specifies the operation required to compute the output table. Base tables can be considered to have empty or non-existent bodies.

Let's study Box 3. The head specifies output columns partno, descr and suppno, as specified in the select list of the query. The specification of these columns includes column names, types, and output ordering information. The head has a Boolean attribute called *distinct* which indicates whether the associated table contains only distinct tuples (head.distinct = TRUE), or whether it contains duplicates (head.distinct = FALSE).

The body of a box contains a graph. The vertices of this graph (dark circles in our diagrams) represent quantified tuple variables, called *quantifiers*. In Box 3, we have quantifiers $q1$, $q2$, and $q4$. Quantifiers $q1$ and $q2$ range over the base tables *inventory* and *quotations* respectively, and correspond to the table references in the *FROM* clause of the SQL query. Note that nodes $q1$ and $q2$ are connected via an inter-box edge to the head of the *inventory* and *quotations* boxes. The edge between $q1$ and $q2$ specifies the join predicate. The (loop) edge attached to $q1$ is the local predicate on $q1$. In fact, each inter-quantifier edge represents a conjunct of the *WHERE* clause in the query block — the conjuncts being represented in the diagram by the labelled rectangle along the edge. Such edges are also referred to as Boolean factors [SAC+79]. Quantifier 3 is a universal quantifier, associated with the *ALL* subquery in the *WHERE* clause. This represents that for *all* tuples associated with $q4$, the predicate represented by the edge between $q2$ and $q4$ is true.

In Box 3, $q1$ and $q2$ participate in joins, and their columns are used in the output tuples. These quantifiers have type *F*, since they come from the query's *FROM* clause. Quantifier 4 has type *A*, representing a universal (*ALL*) quantifier. SQL's predicates EXISTS, IN, ANY and SOME are true if at least one tuple of the subquery satisfies the predicate. Hence, all of these predicates are existential, and the quantifiers associated with such subqueries have type *E*. Each quantifier is labeled with the columns that it *needs* from the table it ranges over.

Box 4 represents the subquery. It contains an F quantifier $q3$ over the *quotations* table, and has a predicate that refers to $q2$ and $q3$.

The body of every box has an attribute called *distinct* which has a value of *ENFORCE, PRESERVE* or *PERMIT*. ENFORCE means that the operation must eliminate duplicates in order to enforce head.distinct = TRUE. PRESERVE means that the operation can preserve the number of duplicates it generates. This could be because head.distinct = FALSE, or because head.distinct = TRUE and no duplicates could exist in the output of the operation even without duplicate elimination. PERMIT means that the operation is permitted to eliminate (or generate) duplicates arbitrarily. For example, the *distinct* attribute of Box 4 can have the value PERMIT because its output is used in a universal quantifier ($q4$ in Box 3), and universal quantifiers are insensitive to duplicate tuples. This will be covered in more detail in Section 3.

Like each box body, each quantifier also has an attribute called *distinct* which has a value of *ENFORCE, PRESERVE* or *PERMIT.* ENFORCE means that the quantifier requires the table over which it ranges to enforce duplicate elimination. PRESERVE means that the quantifier requires that the exact number of duplicates in the lower table be preserved. PERMIT means that the table below may have an arbitrary number of duplicates. Existential and universal quantifiers can always have distinct = PERMIT, since they are insensitive to duplicates.

In the body, each output column may have an associated expression corresponding to expressions allowed in the select list of the query. In Figure 1, all of these expressions are simple identity functions over the referenced quantifier columns.

SQL2 has table expressions, which are similar to view definitions, and can be defined anywhere a table can be used. In Starburst, table expressions and views, just like queries and subqueries, have a QGM, with one or many boxes, and become part of the QGM graph of queries referring to them.

The output of a box can be used multiple times (*e.g.*, a view may be used multiple times in the same query), creating common subexpressions. Recursive queries create cycles in QGM. As the size of the graph grows, the cost of optimization also grows. The number of QGM boxes in a query typically ranges from 2 to 10. For much more complex queries, such as those produced by XNF, this number often ranges from 10 to 100.

2.1 Environment for Performance Measurements

It is not uncommon for queries to take hours or even days to complete. Query Rewrite can improve performance by several orders of magnitude — in many cases converting an over-night query to an interactive one. We will be demonstrating this fact during the course of the discussion by measuring the performance effect of our rewrite rules on various queries. In this section we present the environment used for these measurements.

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 (involving subqueries, views, etc), rather than a transaction workload, where queries are relatively simple. There is no accepted standard complex query workload, although several have been proposed ([TOB89, O'N89]). To measure the performance effect of the rewrite rules, we employ a

Table	Tuple Size	#Tuples	#4K Pgs	#Indices
itm	34	170 000	1 850	1
itl	78	2 550 000	57 980	2
itp	43	339 440	4 250	3
pur	398	128 000	11 640	1
wor	119	120 000	4 000	1

Table 1: Benchmark Database

version of the IBM DB2 benchmark database described in [Loo86], scaled up by a factor of 10.

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

Since Starburst's Query Rewrite system can produce SQL representations of its output, we can easily measure its effects on a widely-used commercial DBMS. This allows us to demonstrate the general applicability of Query Rewrite to typical DBMSs, not just Starburst. Our performance measurements were done on the DB2 relational DBMS. We measured the elapsed time (total time taken by system to evaluate the query), and CPU time (the time for which CPU is busy) of each query both before and after applying the rewrite rules. Both representations of the query went through the usual DB2 query compilation process, including plan optimization. Performance figures for several of the queries we measured are given in Section 3.

3 A Suite of Rewrite Rules for Guaranteeing SELECT Merge

In designing our rewrite engine and rules, we attempted to abide by the following:

Rewrite Philosophy

Whenever possible, a query should be converted to a single SELECT operator.

A single SELECT operator ranging over base tables represents the prototypical relational query, involving straightforward restriction, projection, and join. There are a variety of high-performance algorithms for executing such queries, and methods for choosing among these are well understood. Also, as noted above, more complex queries often force the plan optimizer into choosing a particular plan — for example subqueries force particular join methods and orders, and views (as we shall see) unnecessarily restrict the possible join orders for the query. Finally, plan optimizers typically can only make decisions based on the environment of a single query block (*i.e.* a single QGM box). As a result, multi-operator queries usually do not result in optimal plans, and should be converted to single SELECT operators whenever possible. There are, of course, many optimizations that can be applied to SELECT operators themselves. But we consider conversion to a single SELECT, when possible, to be among the most important goals of query transformation.

As a result, we focus in this paper on those rewrite rules in Starburst which are used to guarantee that all views (table expressions) and existential subqueries are merged whenever possible. The only queries in Starburst which do not get rewritten as single SELECT queries are those which contain non-existential or non-Boolean factor subqueries, set operators,[2] aggregates, or user-defined extension operators (such as OUTER JOIN). The system allows rewrite rules

[2]Even some of these get converted to a single SELECT, as we shall see!

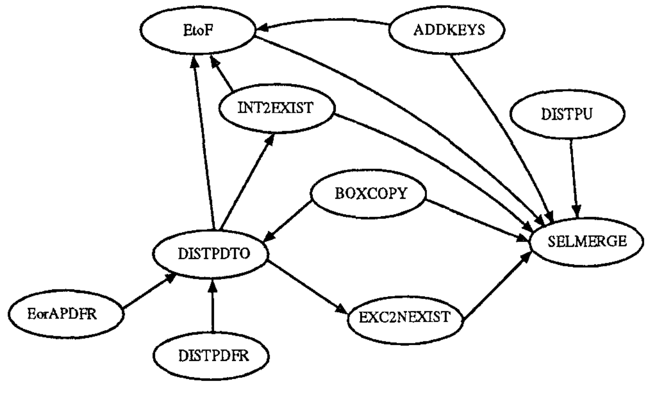

Figure 2: Triggering Interactions Between Rules

```
if (in a SELECT box (upper box)
    a quantifier has type F
  AND ranges over a SELECT box (lower box)
  AND no other quantifier ranges over lower box
  AND
    ( upper.head.distinct = TRUE
    OR
      upper.body.distinct = PERMIT
    OR
      lower.body.distinct != ENFORCE))
  {merge the lower box into the upper box;
    if ( lower.body.distinct = ENFORCE
         AND upper.body.distinct != PERMIT)
       {upper.body.distinct = ENFORCE;}}
```

Table 2: Rule 1 — SELMERGE

to be defined for such queries as well, so even though these are not converted to a single SELECT, they are often subject to some useful transformation. For example, Rosenthal [RGL90] defines a set of such transformations for outer join.

In this section we will describe the set of rules in Starburst which lead up to the merger of SELECT boxes. In Figure 2 we show the dependencies among the rules, particularly when the execution of one rule (at the tail of an arrow) can cause another (at the arrow's head) to have its condition satisfied. We make no claim to be exhaustive in presenting these dependencies. The goal is rather to present the most important dependencies as an illustration of the utility of each rule in causing the merger of SELECT boxes. Since the goal of this suite of rules is the merger of SELECT boxes, the "SELECT merge" (SELMERGE) rule presented first will be transitively dependent on each of the other rules here, and will form the focus of our measurements.

Before proceeding it should be noted that the number of rules we require is kept tractably low by enforcing locality of reference: each rule is written with reference to a specific context (*e.g.* a box, or a quantifier), and as a result rules which involve more than one box can be written in a box-by-box manner, rather than a paired-box manner. This keeps the number of rules on the order of the number of box types, rather than the *square* of that number or worse. We shall see many examples of rule locality in the suite of rules presented here, and in the next section we will see how our rule engine supports it.

3.1 Rules to Guarantee View Merge

Rule 1. SELECT Merge

The SELMERGE rule (Table 2) takes two SELECT boxes connected by an F quantifier (*e.g.* a query over a view) and merges them into one box. The benefit of this transformation is that it makes more join orders possible; in the resulting single SELECT box the plan optimizer can choose as a join order any permutation of the tables under F quantifiers, whereas in the original query the tables referenced in the table expression (lower box) could not be interleaved with those of the box above. Note that if we can apply this rule to all boxes in a query, we end up with a single SELECT box, so this rule leads directly to the realization of our Rewrite Philosophy. In order to exploit the utility of this rule, the rest of the rules in this section will attempt to make the condition of this rule satisfied in as many situations as possible.[3]

The issue of duplicates forces us to introduce some complexity to ensure the rule's correctness. Ignoring duplicates for a moment, it should be clear why this rule works: it follows directly from the commutativity of joins and predicate applications. Since joins and predicate applications are commutative, we can interleave those of the lower and upper boxes, which is equivalent to saying that we can merge the two. This argument applies directly to the case in which neither the upper nor the lower box removes duplicates.

Some analysis is required to see that this rule handles duplicates correctly. We break down the cases for duplicates and argue the correctness of each case:

- *upper.head.distinct = TRUE*: This can happen in one of two ways:
  - If *upper.body.distinct = ENFORCE*, then any duplicates produced by the lower box in the original query will be removed by the upper box, and thus no duplicates are lost or introduced by the merge.
  - If *upper.body.distinct = PRESERVE*, then all the F quantifiers in the upper box produce sets without duplicates. If *lower.body.distinct = PRESERVE* then we can simply merge the lower box into the upper box, without any effect on duplicates. If *lower.body.distinct = ENFORCE* then we must set *upper.body.distinct* to *ENFORCE* to ensure that no duplicates will be produced after merge occurs. Note that we cannot have *lower.body.distinct = PERMIT* — if we did, then the lower box could produce as many duplicates as it found convenient, meaning that we could not have *upper.head.distinct = TRUE* and *upper.body.distinct = PRESERVE*, a contradiction.
- *upper.body.distinct = PERMIT*: In this case we may ignore the issue of duplicates by definition.
- *lower.body.distinct != ENFORCE*: In this remaining case, the previous two conditions must be false, *i.e.* we know that *upper.head.distinct = FALSE* and *upper.body.distinct = PRESERVE*. As a result we cannot merge the boxes if *lower.body.distinct = ENFORCE*, since we would be unable in a single box to remove the duplicates from the quantifiers of the lower box, and preserve those of the remaining quantifiers of the upper box. However, if *lower.body.distinct != ENFORCE* we need not worry about this issue, and thus can merge.

Note that the only cases in which we *cannot* apply SELMERGE to two SELECT boxes connected by F quantifiers are when the lower box has multiple quantifiers ranging over it, or when *upper.head.distinct = FALSE*, *upper.body.distinct = PRESERVE* and *lower.body.distinct = ENFORCE*. We shall see that these cases are handled by the BOXCOPY and ADDKEYS rules respectively, guaranteeing that SELMERGE will eventually get to be executed.

We have chosen a relatively simple query to measure the effect of the above rule in the performance environment explained in Section 2. In practice, queries are typically more complicated, and the merge rule only becomes applicable after many of the rules enumerated below are applied. Although this example is simple, many commercial DBMSs miss this optimization.

[3]As noted in [HP88], the importance of triggering this rule is emphasized when we remember that early relational systems such as System R supported *only* mergable views.

Query	CPU Time	Elapsed time
Before Rewrite	20 min 34.51 sec	24 min 19.80 sec
After Rewrite	0 min 1.10 sec	0 min 7.20 sec

Table 3: Example 1, Before and After Rewrite

```
if (  in a SELECT box
        either **quantifier-nodup-condition**
        or **one-tuple-condition**
        holds for all F quantifiers)
   {  head.distinct = TRUE;
      body.distinct = PRESERVE;}
```

Table 4: Rule 2 — DISTPU

Consider a view which gives the item number and vendors for an item which vendors have supplied since the year 85. This view is used in a query which gives information about certain items and their vendors.

Example 1.

```
 CREATE  VIEW itpv AS
   ( SELECT DISTINCT itp.itemn, pur.vendn
     FROM itp, pur
     WHERE itp.ponum = pur.ponum AND pur.odate > '85');

SELECT itm.itmn, itpv.vendn FROM itm, itpv
WHERE itm.itemn = itpv.itemn
  AND itm.itemn ≥ '01' AND itm.itemn < '20';
```

The rewrite logic first recognizes that the result of the query is DISTINCT by applying the DISTPU rule explained below. Then the merge rule is applied. The resulting query is:

```
SELECT DISTINCT itm.itmn, pur.vendn
FROM itm, itp, pur
WHERE itp.ponum = pur.ponum AND itm.itemn = itp.itemn
  AND pur.odate > '85'
  AND itm.itemn ≥ '01' AND itm.itemn < '20';
```

As a result of merging the view with the query, the plan optimizer can use an index to access the tables within the view, and therefore it chooses a plan which exploits this fact while doing a join on behalf of the query. The results of executing this query with and without rewrite are shown in Table 3. After applying the rewrite rule, we get an 1100 × improvement in CPU time (and hence in pathlength) and a 200 × improvement in the elapsed time.

Rule 2. Distinct Pullup

In the DISTPU rule (Table 4) a SELECT box *upper* infers that no duplicate elimination is needed to guarantee that its output tuples are distinct. It does this by isolating the following properties:

- **one-tuple-condition**: given a quantifier and a set of predicates, this condition is TRUE iff at most one tuple of the quantifier satisfies the set of predicates.
- **quantifier-nodup-condition**: given an F quantifier in a SELECT box, this condition is TRUE iff at least the primary key or a candidate key of the F quantifier appears in the output.

Upper must find that either **quantifier-nodup-condition** or **one-tuple-condition** holds for each of its F quantifiers and their associated predicates. If this is not true of some F quantifier, then the projection of the Cartesian product of the boxes below will have

```
/* DISTPDFR */
if (  in a box with type SELECT, UNION,
        INTERSECT or EXCEPT,
        body.distinct = PERMIT or ENFORCE)
   {  for (each F quantifier in the body)
        quantifier.distinct = PERMIT;}

/* DISTPDTO */
if (  in a box with type SELECT, UNION,
        INTERSECT or EXCEPT,
        all quantifiers ranging over the box
        have quantifier.distinct = PERMIT)
   {body.distinct = PERMIT;}
```

Table 5: Rule 3 — DISTPDFR/DISTPDTO

duplicates, and *upper* must remove them; if one of the two conditions is true for each F quantifier in *upper*, then the projection of the Cartesian product will have no duplicates.

Note the "locality" of this rule – in writing the rule we need not worry about the type of the boxes below us, rather we focus on the F quantifiers over those boxes.

Rule 3. Distinct Pushdown From/To

In this pair of rules (Table 5), a box informs the boxes under it that it does not require them to eliminate duplicates. It does so by "pushing" the distinct attribute *from* itself *to* the boxes below it. This may save the lower boxes below from needing a sort or hash for duplicate elimination, and may also allow the lower boxes to be subject to rules which can introduce duplicates (such as EtoF below.)

For the DISTINCT set operators (UNION, INTERSECT, and EXCEPT) the DISTPDFR rule is correct because of the semantics of duplicate elimination in those operators. The DISTINCT set operators are defined as removing duplicates from all their *inputs* before any further processing [ISO91]. Thus these boxes will disregard any duplicates produced by boxes below them, and can safely signal this by pushing DISTINCT down along their quantifiers.

In the case of a SELECT box with body.distinct = PERMIT, we do not worry about the issue of duplicates. To see the correctness of the DISTPDFR rule for a SELECT box with body.distinct = ENFORCE, it suffices to notice that any tuple resulting from such a box is a projection of the concatenation of tuples $t_1, \ldots, t_n$ from the n inputs (under F quantifiers) to the box. Regardless of how many copies of each t_i there are in the corresponding input table i, no more than one tuple projected from $t_1 \cdot \ldots \cdot t_n$ will be in the output of the SELECT DISTINCT box. Thus each input can safely remove or introduce duplicates without affecting the output of the SELECT box above.

The DISTPDTO rule is quite simple — if all boxes ranging over a given box indicate their indifference to the number of duplicates produced by their inputs, then that box may introduce or remove duplicates at will, and hence can set its body's distinct flag to PERMIT, and its head's distinct flag to FALSE.

Note the use of rule locality here: the task involves two boxes, and thus is broken into two separate rules, with the information passed via the quantifier between the boxes. Each operator need only concern itself with its own behavior in the activity of pushing down the DISTINCT attribute, and need not know anything about the other operators involved in the activity. Note further that if processing halts after DISTPDFR but before DISTPDTO (as can happen in our rule engine, see below), the QGM is still consistent and valid.

Rule 4. E or A Distinct Pushdown From

This rule (Table 6) is a special case of distinct pushdown "from", which exploits the fact that existential and universal quantifications are blind to duplicates. That is, the *number* of tuples in a subquery

```
if (  in a SELECT box
        a quantifier has type = E or A)
   {quantifier.distinct = PERMIT;}
```

Table 6: Rule 4 — EorAPDFR

```
if (  in a SELECT box
        more than one quantifier
        ranges over the box)
   {  Make a copy of the box;
      Take one of the quantifiers
        ranging over the original box
        and change it to range over the new copy;}
```

Table 7: Rule 5 — BOXCOPY

which satisfy the existential predicate is insignificant; existential predicates merely require that one of the tuples of the subquery match. Similarly the number of duplicates in a subquery has no bearing on a universal predicate; either *all* tuples in the subquery match the universal predicate, or not all do.

Example 2.

```
CREATE   VIEW richemps AS
  ( SELECT DISTINCT empno, salary, workdept
    FROM employee
    WHERE salary > 50000);

SELECT mgrno FROM department dept
WHERE NOT (EXISTS (
        SELECT * FROM richemps rich, project proj
        WHERE proj.deptno = rich.workdept
          AND rich.workdept = dept.deptno));
```

This example returns those managers who have no rich employees in their department. By applying EorAPDFR and DISTPDTO, we make the subquery have body.distinct = PERMIT, which results in the view richemps being merged into the subquery. After rewrite, the query is:

```
SELECT mgrno FROM department dept
WHERE NOT (EXISTS (
        SELECT * FROM employee emp, project proj
        WHERE proj.deptno = emp.workdept
          AND emp.workdept = dept.deptno
          AND emp.salary > 50000));
```

Rule 5. Common Subexpression Replication

This rule (Table 7) breaks common subexpressions in a QGM by replicating them. Doing so can allow one or both of the resulting boxes to merge.[4]

Rule 6. Add Keys

Given two SELECT boxes *upper* and *lower*, such that *lower* is ranged over only by an F quantifier in *upper*, ADDKEYS (Table 8) guarantees that *upper* and *lower* will be merged. It does so by modifying any SELECT box which preserves duplicates to be able to safely eliminate duplicates. We achieve this by adding "key" columns (or unique tuple ID's) to the inputs, which are passed up into the SELECT box. Once this is done, we can eliminate duplicates from the SELECT box without any effect, since each tuple in the

[4]If queries are correlated, the copy logic is more complicated. This issue is beyond the scope of this paper, but is treated correctly in Starburst's version of this rule.

```
if (  in a SELECT box
        head.distinct = FALSE)
   {  for(each F quantifier)
        if (  the key of the F quantifier
                  does not appear in the output)
           {  Add the key to the head;
              head.distinct = TRUE;}}
```

Table 8: Rule 6 — ADDKEYS

box has a unique key formed by the concatenation of the keys of the inputs.

Again, note the "locality" of this rule – boxes below are referenced by the F quantifiers which range over them, and thus the types of the boxes below become insignificant.

In the following example, a view is declared, giving distinct negotiated prices of ordered items. The query uses the view to calculate the negotiated price for each item type.

Example 3.

```
 CREATE   VIEW  itemprice AS
  ( SELECT DISTINCT  itp.itemno, itp.NegotiatedPrice
    FROM  itp
    WHERE  NegotiatedPrice > 1000);

SELECT  itemprice.NegotiatedPrice, itm.type
FROM  itemprice, itm
WHERE  itemprice.itemno = itm.itemno;
```

The ADDKEYS rule is applied to the (upper) SELECT box of the query, allowing SELMERGE to merge the view itemprice into the query. Note that SELMERGE changes the query's *body.distinct* attribute to be ENFORCE, thus removing the duplicates originally removed in the view. The resulting query is:

```
SELECT DISTINCT  itp.NegotiatedPrice, itm.type, itm.itemno
FROM  itp, itm
WHERE  itp.NegotiatedPrice > 1000 AND  itp.itemno = itm.itemno;
```

This SQL representation of the rewritten query does not exactly capture the semantics of the transformed QGM. In the actual rewritten query, the output column itm.itemno is used during duplicate elimination, but its values are not delivered to the output of the query.

3.2 Guaranteeing Existential Subquery Merge

The rules in the previous section guarantee that SELECT boxes get merged whenever the only quantifiers over the lower box are F quantifiers. The following rule attempts to facilitate merging by creating this situation as often as possible. In particular, we shall see that the next rule guarantees the merger of existential subquery conjuncts and the SELECT boxes above them.

Rule 7. E to F Quantifier Conversion

In this rule (Table 9) we convert Boolean factor existential subqueries to table expressions, by changing the type of quantifier over the subquery from E to F. Note that the ADDKEYS rule guarantees that the condition of this rule will eventually be satisfied for all such subqueries. As noted above, converting a subquery to a table expression (and hence a member of a join) increases possible orders of join execution. It may also allow for additional merging, if the subquery is another SELECT box.

This rule is the QGM equivalent of a rule proven correct in [Day87].[5] We do not prove its correctness here, but an intuition of

```
if (  in a SELECT box
        there is a quantifier of type E
        forming a Boolean factor
     AND
     (    head.distinct = TRUE
        OR
          body.distinct = PERMIT
        OR
          one-tuple-condition))
   {  set quantifier type to F;
      if (  one-tuple-condition is FALSE
             AND head.distinct = TRUE)
         {body.distinct = ENFORCE;}}
```

Table 9: Rule 7 — EtoF

Query	CPU Time	Elapsed time
Before Rewrite	88 min 01.25 sec	91 min 49.20 sec
After Rewrite	2 min 42.97 sec	6 min 24.60 sec

Table 10: Example 4, Before and After Rewrite

the rule's correctness can be seen by considering the two-quantifier case. As an example, consider the following query, which gives the order information for items built at certain locations and worked on at certain workcenters. Note that the itp table has a key, and hence contains no duplicates.

Example 4.

```
SELECT  * FROM  itp
WHERE  itp.itemn  IN
  ( SELECT  itl.itemn FROM  itl
    WHERE  itl.wkcen = 'WK468' AND  itl.locan = 'LOCA000IN');
```

In order to execute this query we must output one copy of a tuple from itp iff there is at least one tuple in itl.itemn which satisfies the appropriate predicates. If we apply DISTPU to convert this subquery to a table expression, and then apply SELMERGE, we get a single SELECT query, *i.e.*

```
SELECT DISTINCT  itp.* FROM  itp, itl
WHERE  itp.itemn = itl.itemn
  AND  itl.wkcen = 'WK468' AND  itl.locan = 'LOCA000IN';
```

In the transformed query, we output one copy of a tuple from itp $\times$ itl such that the appropriate predicates are satisfied (including "itp.itemn = itl.itemn", which was implied previously by the "IN" construct). Since all columns from itl are projected away and duplicates are removed, this produces the same results as the original query.

The above example is from the performance environment explained in Section 2. The results of the performance measurements are shown in Table 10. After rewrite we get a $32\times$ improvement in CPU time and a $14\times$ improvement in elapsed time.[6] During transformation, the DISTPU rule recognizes that the result of the query is distinct. Then the EtoF rule converts the subquery to a table expression, without adding any extra keys to the output, since it has already recognized that the output is distinct. Then the SELMERGE rule merges the table expression, greatly enhancing the performance of the query.

We can now observe why Boolean factor existential subqueries over SELECT boxes are guaranteed to merge. Consider any SE-

[5]The exact rule is *Semijoin*$(R, S; J)$ = *Delta-Project*(*Join* $(R, S; J); R.*)$. We actually generalize slightly here by isolating the case where **one-tuple-condition** is satisfied.

[6]Again, many RDBMS, including the commercial ones, are unable to perform this optimization.

```
if (  in an INTERSECT box
        body.distinct != PRESERVE)
   {  set the box to be of type SELECT;
      choose an arbitrary quantifier Q1;
      /* Q1 will keep type F */
      for ( each quantifier Q != Q1 in the box)
         {  Q.type = E;
            add the predicate
            EXISTS(SELECT * FROM Q
                      WHERE
                                Q1.c1 = Q.c1
                      AND    Q1.c2 = Q.c2
                      AND ...
                      AND    Q1.cn = Q.cn);}}
```

Table 11: Rule 8 — INT2EXIST

LECT box *upper* with a Boolean factor SELECT subquery (*i.e.* a SELECT box *lower* over which it ranges with an E quantifier). Because of the EorAPDFR and DISTPDFR/TO rules, we can assume that the subquery has *body.distinct = PERMIT*. Now, we want to be able to fire the EtoF rule, but we cannot do so if *upper.head.distinct = FALSE*, *upper.body.distinct = PRESERVE* and the **one-tuple-condition** does not hold for the quantifier between the two boxes. In this case, we can apply the ADDKEYS rule to force *upper.head.distinct = TRUE*, and at that point we can apply EtoF. After EtoF is applied, we have *upper* ranging over *lower* with an F quantifier, and *lower.body.distinct != ENFORCE*, so the conditions for SELMERGE are satisfied, and *lower* can be merged into *upper*.

Rule 8. INTERSECT to Exists

This rule (Table 11) converts a set INTERSECT operator (which may be n-ary) into an existential subquery, which can subsequently be converted (via EtoF and SELMERGE) into a single SELECT box. Typically, RDBMSs execute the INTERSECT operation by sorting the operands and then merging them. This method of execution is a variant of the sort merge join. We rewrite the INTERSECT operation as a join, and therefore benefit from other join methods besides sort-merge. This can improve the performance by many orders of magnitude, and hence is an essential query transformation.[7]

Recall that the semantics of SQL's INTERSECT operator are to first remove duplicates from the inputs and then send to the output one copy of every tuple that appears in all of the inputs. This is equivalent to choosing any one input and sending to the output one copy of each of its tuples which appears in all other inputs. This rule simply captures that equivalence — it produces a SELECT DISTINCT box with one F quantifier (the arbitrarily chosen input) and E quantifiers over all other inputs, which filter out tuples of the F-quantified input that do not have a match in all of the other inputs. Note that matching tuples require the large conjunction in the predicate of the subquery — tuples must match on all columns to be equal.[8]

As an example, consider the following query which finds the intersection of the items that employee 1279 works on and the items that are scheduled to be worked on in workcenter WK195 on date 9773.

[7]A similar rule (EXC2NEXIST) exists in Starburst for converting an EXCEPT operator into a negated existential subquery, which can subsequently be involved in SELECT merge.

[8]This can be simplified by including in the conjunction only the key columns of the tables.

Query	CPU Time	Elapsed time
Before Rewrite	9.65 sec	13.92 sec
After Rewrite	.42 sec	1.77 sec

Table 12: Example 5, Before and After Rewrite

Example 5.

```
 SELECT items FROM wor
 WHERE empno = 'EMPN1279'
INTERSECT
 SELECT itemn FROM itl
 WHERE entry_time = '9773' AND wkctr = 'WK195';
```

The intersect rewrite rule converts the query to an existential subquery, which in turn is converted to join by the EtoF rule, and merged by the SELMERGE rule. The query after rewrite is:

```
SELECT DISTINCT itemn FROM itl, wor
WHERE empno = 'EMPN1279' AND entry_time = '9773'
  AND wkctr = 'WK195' AND itl.itemn = wor.itemn;
```

The results of executing this query with and without rewrite are shown in Table 12.[9] After conversion to join, the plan optimizer considers both merge join and nested loop join methods. Due to the presence of an index on the join column, itemn, nested loop is chosen, resulting in much better performance.[10]

4 The Rule Engine

In keeping with the extensibility goals of the Starburst project, it was decided that a rule system was the appropriate platform for allowing query transformations to be easily added to the system, and subsequently reordered or modified. Existing rule engines did not appear to be appropriate for our needs, and thus we designed our own. As will become apparent in the following discussion, we required numerous capabilities not available in typical rule systems (such as OPS5 [BFKM85]). Starburst's Query Rewrite rule engine incorporates the following features:

1. *Rules of Arbitrary Complexity:* Rules in our engine are pairs of functions in a procedural language such as C: a condition function, which does an arbitrary check and sets a flag TRUE or FALSE, and an action function, which, if the condition function sets the flag TRUE, is invoked to take arbitrary action. The fact that our rules are C functions is essential — we require our rules to be able to manipulate QGM, which is represented in Starburst as a network of C structures. Although rules are written in a language such as C, they are compiled into native machine language for efficient execution.
2. *Context Facility:* The data structure passed into any invocation of the rule system includes a pointer for user information, which is in turn passed into the rule functions themselves, where it can be read and/or modified. For our purposes in Query Rewrite, we use this pointer to store a current "context" in the QGM (a box, quantifier, or predicate), and our rewrite rules are written with reference to this context. This allows for the rule locality discussed earlier. Special rules are added to the rule set to

[9]For this experiment we used the original benchmark database, not the one scaled up by 10.

[10]DB2 does not support INTERSECT. In the experiment, we chose UNION instead, which can be executed by a nearly identical strategy. Obviously, the number of output tuples for UNION is different, however, since this number is small in our experiment, the error in the cost difference is negligible. In fact, DB2 chose a better execution strategy for UNION than the one sketched above, and therefore the performance numbers for the original query are conservative.

"advance" the context once we exhaust possibilities of modification to the current context. Thus these rules "traverse" a QGM graph.[11]

3. *Rule Classes and Extensible Conflict Resolution:* We partition our set of rules into rule classes, each of which can be thought of as a separate rule set. This gives us a number of advantages. First, it allows us to group rules into sensible units, allowing for better comprehension of rule activity. Second, since rules are arbitrary C procedures, a rule can invoke another rule class as a subroutine. This results in better modularity and increased comprehensibility. Finally, different rule sets can have different *conflict resolution* schemes [MF78] for choosing the next rule to fire. Starburst currently supports two schemes, a *sequential* scheme which cycles through a set of ordered rules, and a *priority* scheme, which always fires the highest order rule that has its condition satisfied. New conflict resolution schemes can be easily added to the system.

   The rule class mechanism provides some optional structure for the rules, giving us a measure of control without resorting to a fully procedural system. A glance at Figure 2 should illustrate that our rule interactions make it very difficult to explicitly express the order in which rules should be applied. The concept of production rules frees us of this burden. However, we do wish to exercise *some* control over the order in which rules are fired. For instance, we want to apply INT2EXIST before SELMERGE, in order to ensure that an INTERSECT box above a SELECT box will be converted and merged into one SELECT box. By carefully organizing our rule classes and their conflict resolution schemes, we ensure this ordering.

   Note that the full spectrum of control is supported by our engine, from totally data-driven to totally procedural: a totally data-driven scheme could be modelled with a single rule class in which the control scheme was randomized, and a totally procedural scheme would be a rule set made up of one rule which was triggered once and performed an arbitrary procedure or program. In Starburst's Query Rewrite, we use the rule engine to strike a balance between these extremes — we have multiple rule classes, some of which use the sequential control scheme, others of which use the priority scheme. The resulting system retains much of the organization of a procedural program, while taking advantage of the easy extensibility and flexible interactions inherent in a rule system.

4. *Guaranteed Termination:* After a specified number of rules have been considered, our rule engine will terminate execution, whether or not there remain rules eligible for execution. This number is under the control of the rule programmer. Note that since execution can end after any rule is checked or executed, we are forced to make each rule be an atomic change mapping a valid QGM to an equivalent valid QGM. There can be no invalid or "transitory" states of the QGM between rules. This restriction turns out to be quite positive, as it enforces a conceptual cleanliness on each rule.

5. *Rule Engine Controls:* Starburst users can locally enable or disable rules "on the fly" without affecting other users. This allows rule developers to enjoy the convenience of the rule system paradigm without affecting concurrent users of the database. The controls also allow us to trace and explain rule activity, a useful method for debugging the potentially complex interactions between rules.

Our experience with this rule system has been quite positive. Once all the rule engine controls were added to the system, it became fairly easy to add new rules to the existing set without causing unusual rule interactions. The time required to add a rule to the system was determined largely by the complexity of the rule's transformation to QGM; typically we did not need to spend much time debugging our rule set as a whole. The success we have had with our rule system is unusual, and can be attributed both to the unique features of the system design listed above, and also to our application of the rule system. Since all of our rewrite rules produce valid QGMs, and since each individual rule does not *degrade* query performance, the worst behavior our rule system can display is to leave a query untouched. In practice this happens only to simple queries which require no optimization.

[11]The choice of traversal is itself extensible. We currently support both depth-first and breadth-first traversal of QGM.

5 Conclusions: Engine and Rules

We have built an extensible Query Rewrite system for Starburst, and shown that it can provide query execution improvements of orders of magnitude. Others have proposed query transformations before ([Kim82, GW87, Day87, Anf89]) but our work subsumes many of these transformations, and is the first system implementation (to our knowledge) to organically incorporate query transformation schemes into a full RDBMS. We have put considerable effort into designing the system to address the problems faced by comprehensive research prototypes and industrial grade RDBMSs, particularly in handling of complex queries, and queries associated with complex objects [LLPS91]. The capabilities of our Query Rewrite system surpass by a significant margin those of current RDBMSs, commercial systems included.

Among the query transformations presented in this paper, we generalize previous work to handle duplicates correctly, and thus are able to guarantee the merge of existential subquery conjuncts. Furthermore, we convert set operators (INTERSECT, EXCEPT) to subqueries, allowing a rich set of join methods to be used for executing set operators. Although this is quite simple, it has not been dealt with in the past.

Our decision to design a rule engine for Query Rewrite seems to have been sound. We have encountered few of the oft-cited problems of rule systems (nondeterminism of outcome, difficulty of comprehending system execution, slow performance, etc.), and made significant use of the inherent advantages of a rule system (ease of extensibility, abstraction of rule programmer from control flow, etc.) The extensibility of a Query Rewrite system is key — it allows new functionality to be added easily both to the query language and the underlying technology (*e.g.* faster and cheaper memory), and should also allow plan optimizers to be "taught" to avoid their shortcomings, which may only be discovered when the system is used in production [Pir89, HCL+90]. Our extensible Query Rewrite system addresses both of these issues.

6 Acknowledgments

As usual, George Lapis was a great help in numerous implementation details for many of the rules. Don Haderle, Yun Wang of the DB2 organization, and Josephine Cheng of DBTI were of great help in our understanding of query transformations in the context of industrial grade systems. This lead to significant improvement in the completeness of our rewrite rules. Many discussions with Sheldon Finkelstein lead to better conceptualization of QGM and the rules. We benefited considerably from the plan optimizer, query refinement, and runtime environment of Starburst. Thanks are due to Bruce Lindsay, Guy Lohman, John McPherson, Mavis Lee, Hanh Nguyen, and Kiyoshi Ono. We also thank John McPherson and Adam Glass for comments on an earlier version of this paper.

References

[ABC+76] M. Astrahan, M. Blasgen, D. Chamberlin, K. Eswaran, J. Gray, P. Griffiths, W. King, R. Lorie, P. McJones,

J. Mehl, G. Putzolu, I. Traiger, B. Wade, and V. Watson. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[Anf89] Ole Jirgen Anfindsen. A Study of Access Path Selection in DB2. Technical report, Norwegian Telecommunications Administration and University of Oslo, Norway, October 1989.

[BFKM85] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5*. Addison-Wesley Publishing Co., 1985.

[BTA90] Jose Blakeley, Craig Thompson, and Abdallah Alashqur. Strawman reference model for object query language. In *First OODB Standardization Workshop, X3/SPARC/DBSSG/OODBTG*, Atlantic City, New Jersey, May 1990.

[Day87] Umeshwar Dayal. Of Nests and Trees: A Unified Approach to Processing Queries that Contain Nested Subqueries, Aggregates, and Quantifiers. In *Proc. 13th International Conference on Very Large Data Bases*, pages 197–208, Brighton, September 1987.

[GW87] Richard A. Ganski and Harry K. T. Wong. Optimization of Nested SQL Queries Revisited. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 23–33, San Francisco, May 1987.

[HCL+90] L.M. Haas, W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starburst Mid-Flight: As the Dust Clears. *IEEE Transactions on Knowledge and Data Engineering*, pages 143–160, March 1990.

[HH91] Joseph M. Hellerstein and Meichun Hsu. Determinism in Partially Ordered Production Systems. Research Report RJ 8089, IBM Almaden Research Center, March 1991.

[HP88] Waqar Hasan and Hamid Pirahesh. Query Rewrite Optimization in Starburst. Research Report RJ 6367 , IBM Almaden Research Center, August 1988.

[HSS88] T. Haerder, H. Schoning, and A. Sikeler. Parallelism in Processing Queries on Complex Object. In *Proc. International Symposium on Databases in Parallel and Distributed Systems*, Austin, December 1988.

[ISO91] ISO_ANSI. Database Language SQL ISO/IEC 9075:1992, 1991.

[Kim82] W. Kim. On Optimizing an SQL-like Nested Query. *ACM Transactions on Database Systems*, 7(3), September 1982.

[LLOW91] Chares Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The Objectstore Database System. *Communications of the ACM*, October 1991.

[LLPS91] Guy Lohman, Bruce Lindsay, Hamid Pirahesh, and Bernhard Schiefer. Extensions to Starburst: Objects, Types, Functions, and Rules. *Communications of the ACM*, October 1991.

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

[MF78] J. McDermott and C. Forgy. Production System Conflict Resolution Strategies. In D.A. Waterman and Fredrick Hayes-Roth, editors, *Pattern Directed Inference Systems*, pages 177–199. Academic Press, 1978.

[MFPR90a] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic is Relevant. In Proc. SIGMOD 90 [Pro90], pages 247–258.

[MFPR90b] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic Conditions. In *Proc. 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 314–330, Nashville, March 1990.

[MPR90] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The Magic of Duplicates and Aggregates. In *Proc. 16th International Conference on Very Large Data Bases*, Brisbane, August 1990.

[O'N89] P. O'Neil. Revisiting DBMS Benchmarks. *Datamation*, pages 47–54, September 15, 1989.

[Pir89] Hamid Pirahesh. Early Experience with Rule-Based Query Rewrite Optimization. In G. Graefe, editor, *Workshop on Database Query Optimization*, CSE Technical Report 89-005. Oregon Graduate Center, May 1989.

[Pro90] *Proc. ACM-SIGMOD International Conference on Management of Data*, Atlantic City, May 1990.

[Ras90] Louiqa Raschid. Maintaining Consistency in a Stratified Production System Program. In *Proc. AAAI National Conference on Artificial Intelligence*, 1990.

[RGL90] Arnon Rosenthal and Cesar Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. In Proc. SIGMOD 90 [Pro90].

[SAC+79] Patricia G. Selinger, M. Astrahan, D. Chamberlin, Raymond Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Boston, June 1979.

[SJGP90] M. Stonebraker, A. Jhingran, Jeffrey Goh, and Spyros Potamianos. On rules, procedures, caching and views in data base systems. In Proc. SIGMOD 90 [Pro90].

[SWK76] M.R. Stonebraker, E. Wong, and P. Kreps. The design and implementation of ingres. *ACM Transactions on Database Systems*, 1(3):189–222, September 1976.

[TOB89] C. Turbyfill, C. Orji, and Dina Bitton. AS3AP - A Comparative Relational Database Benchmark. In *Proc. IEEE Compcon Spring '89*, February 1989.

[ZH90] Yuli Zhou and Meichun Hsu. A Theory for Rule Triggering Systems. In Francois Bancilhon, Costantino Thanos, and Dennis Tsichritzis, editors, *Proc. International Conference on Extending Data Base Technology*, Advances in Database Technology - EDBT '90. Lecture Notes in Computer Science, Volume 416, Venice, March 1990. Springer-Verlag.