Extensible Query Processing in Starburst

Laura M. Haas, J.C. Freytag[1], G.M. Lohman, and H. Pirahesh
IBM Almaden Research Center, San Jose, CA 95120

Abstract

Today's DBMSs are unable to support the increasing demands of the various applications that would like to use a DBMS. Each kind of application poses new requirements for the DBMS. The Starburst project at IBM's Almaden Research Center aims to extend relational DBMS technology to bridge this gap between applications and the DBMS. While providing a full function relational system to enable sharing across applications, Starburst will also allow (sophisticated) programmers to add many kinds of extensions to the base system's capabilities, including language extensions (e.g., new datatypes and operations), data management extensions (e.g., new access and storage methods) and internal processing extensions (e.g., new join methods and new query transformations). To support these features, the database query language processor must be very powerful and highly extensible. Starburst's language processor features a powerful query language, rule-based optimization and query rewrite, and an execution system based on an extended relational algebra. In this paper, we describe the design of Starburst's query language processor and discuss the ways in which the language processor can be extended to achieve Starburst's goals.

1. Introduction

As relational DBMSs have become more widely used, they have attracted applications beyond those for which they were originally designed. Engineering, office and geographic applications (among others) have attempted to use relational databases to store their data. However, each of these kinds of applications has requirements that standard relational systems are unable to satisfy. All have data that is structured in (sometimes) complex ways, and functions that manipulate that data. But the demands of each kind of application for new data types and functions are different.

For applications such as these, using a relational DBMS can be frustrating. While relational systems offer functionality that these applications need (e.g., the ability to relate arbitrary pieces of information, easy storage for basic facts, high level query language with automatic optimization, concurrency control, recovery), they do not provide sufficient support for the functions and data types needed by the applications. Thus, to derive any benefit from a relational system, the application itself must build the additional function it needs, on top of the DBMS. In particular, it must map its types to DBMS types, and it must evaluate all functions, even those in predicates, that are not built into the DBMS. This leads to complex and inefficient applications: complex, because the applications must each perform this extra work, and inefficient, because much data must be returned to the application for processing and predicate evaluation, and because there can be little or no internal DBMS support of the data for fast access (e.g., no specialized access paths or storage management).

The Starburst project at IBM's Almaden Research Center [SCHW86] aims to bridge this gap between relational DBMSs and the applications that would like to use them. Starburst will serve as a testbed for research in relational database technology and database applications, as well as extended database support for applications, hardware architectures and devices. *Extensibility*, or the ability to tailor the DBMS code to support particular applications or hardware, is the key to achieving these goals. Thus Starburst provides support for adding new storage methods for tables, new types of access methods and integrity constraints, new data types, functions, and new operations on tables. In most cases, these extensions will be made by knowledgeable database implementers, whom we will call database customizers (DBCs), and not by end-users or even skilled application programmers.

Starburst has two major components, the query language processor, Corona, and the data manager, Core, corresponding roughly to the RDS and RSS of System R [ASTR76]. Corona is responsible for compiling the user's query into an efficiently interpretable form. Thus its duties include parsing the query, semantic analysis, choosing an execution strategy, and producing the set of operators to implement the execution strategy (the Query Evaluation Plan). It also drives the execution, by interpreting the Query Evaluation Plan, and controls the data definition process (when tables, access methods, etc. are created and destroyed). During each of these activities Corona invokes the data manager's services as needed. Among the services provided by Core are record management (locating, retrieving, and storing records), buffer management, access path management, concurrency control and recovery.

One aspect of Core deserves further description, as it puts demands on the capabilities of Corona. This is Core's data management extension architecture [LIND87]. This architecture allows a DBC to add new kinds of attachments

---

1 Author's current address: ECRC, Arabellastrasse 17, D-8000 Munich, F.R. Germany

(access methods and integrity constraints) and new storage managers to Starburst. For example, a DBC could define a new storage manager which handles fixed-length records only -- but extremely efficiently. Corona must ensure that the correct storage manager is invoked when a table is accessed. Likewise, a DBC could define a new type of access method, e.g., an R-tree [GUTT84]. Corona must recognize when this access method is useful for a query and when to invoke it. Thus the language processor must accommodate these extensions to the data manager's capabilities.

The Starburst prototype runs under AIX on IBM PC/RT workstations, and is being implemented in C to increase portability. Starburst is still in the early prototype phase of implementation. The design is essentially complete and initial implementations exist for each of the components described below, with varying degrees of robustness. Some components, such as the parser, are complete and have been thoroughly tested; others have complete functionality but have not been integrated. We are in the process of integrating these components to make a coherent, usable system.

Many other systems provide facilities similar to those of Starburst. DASDBS [PAUL87], Exodus [CARE86], Genesis [BATO86], Postgres [STON86], Probe [DAYA86], and SABRE [ABIT86] are all extensible systems. Exodus and Genesis are distinguished by following what has been called the "toolkit" approach: both allow the database implementer to mix-and-match a set of components and then generate a DBMS from these components, so that the DBMS is fine-tuned for a particular application. By contrast, our approach in Starburst allows the DBMS to be extended in multiple directions, which promotes data integration across applications. DASDBS, Postgres and SABRE are based on the relational data model, and DASDBS allows nested relations. The other systems are based on functional data models. There are also several "object-oriented" systems with similar goals, including Orion [BANE87, KIM87] and Gemstone [MAIE86]. We chose to stay with the relational model because of its great power and simplicity, as well as its current popularity in the commercial realm. We hope to show that we can give users the support they need for their objects through extensions, while preserving the advantages of a relational DBMS.

Language processing has two components, one for data definition statements and one for data manipulation statements. In this paper we will focus on the handling of data manipulation statements in Starburst. An overview of data definition in Starburst can be found in [HAAS88]. In the next section we discuss Starburst's query language. Section 3 gives an overview of how a data manipulation statement is processed. Section 4 describes the internal representation of a query. This representation is critical to achieving extensibility. We then discuss the optimization of queries: first, a semantic, "rewrite" optimization in section 5, and then cost-based optimization in section 6. Section 7 describes the query execution component. Finally, we summarize our experiences with the implementation, describe its current status, and discuss some of the challenges that lie ahead for Starburst and for extensible systems in general.

2. Language

Starburst's language, Hydrogen, is based on SQL [IBM87]. As in SQL, queries are expressed in an English-like syntax (SELECT...FROM...WHERE...). Hydrogen includes and generalizes most SQL features, allowing aggregation, set operations and nested subqueries, in addition to the standard operators of the relational calculus. It expands on SQL in two important respects. First, it generalizes the SQL grammar to make the language more orthogonal [DATE84]. Second, it allows DBC's to add new functionality to the language, that is, Hydrogen itself is extensible.

The generalization of SQL takes two forms: many of the implementation-dependent constraints of SQL are relaxed, and a new construct is added to increase the orthogonality of the language. SQL has many constraints on where particular constructs may appear. Those constraints pertaining to views and the use of set operations (UNION, etc.) are particularly noticeable. For example, in SQL, if a view performs an aggregation, it cannot be used in a query in which it is joined to another table or view. This forces the user of the view to be aware of the view definition. In Hydrogen, by contrast, views can appear anywhere a base table can be used in a select statement, as well as in many update statements. Update through views will be allowed when the update is unambiguous; otherwise an error will be returned. Likewise, queries with set operations may appear wherever a select statement can be used -- in view definitions, subqueries, etc. The goal in Hydrogen is complete orthogonality: any operation on tables produces a table, and can be used wherever a table would normally be allowed.

To achieve this goal, *table expressions* [DATE84] have been introduced, and can appear anywhere a view or table can. Table expressions are expressions (usually queries) which produce a table as output. The output table can be named and referred to later in the same query. Table expressions add considerable power to the language. They allow users to modularize their queries by factoring out common subexpressions. While similar to views, they are more powerful, as they may contain references to host language variables, or be correlated with other parts of the query.

One important use of table expressions is for expressing recursion. Recursion can be expressed by forming cyclic references to named table expressions. Hydrogen can be used for logic programming by mapping rules to table expressions. Recursive queries may contain relational calculus operations, aggregation, and even externally defined functions (see below). As a result, one can also express path algebra computations [ROSE86, CARR79] in Hydrogen. Thus Hydrogen can be used as an integrated language for logic programming and database access. This allows the scope of optimization to include both the rules and the database queries, providing the opportunity to obtain a globally optimized execution plan.

DBCs can extend Hydrogen in several ways. They can define new functions on columns, new operations on tables, and new data types for columns. Several types of functions can be defined. Scalar functions (e.g., *Area(Width, Length)*) take one or more field values from a single (possibly composite) tuple, and return a single value. Scalar functions can be used anywhere a column can be referenced. Aggregate functions (e.g., *StandardDeviation(Salary)*) range over many tuples of a table and return a single (aggregated) value. Externally defined aggregate functions can be used in place of built-in aggregates. Both types of functions can be invoked by Starburst at low levels of the system to allow more efficient handling of the data. For example, by invoking functions in the predicate evaluator, Starburst can reduce the amount of irrelevant data that is returned to the user.

New set predicate functions may also be defined. Set predicate functions take as input a set (of tuples) and a predicate. In SQL the built-in set predicate functions include ALL and ANY, whose operands are a simple predicate (e.g., T1.c1 = T2.c1) and a subquery that defines the set of tuples over which that predicate should range (e.g., SELECT T2.c1 FROM T2 WHERE T2.c2 = 5). The function returns true if the predicate holds for all (any) elements of the set. In Hydrogen, a DBC could define a new set predicate function, e.g., MAJORITY, which would return true if the predicate is true for the majority of the elements of the set.

A fourth kind of function in Hydrogen is the table function. These functions take one or more tables (or table expressions) and possibly other parameters as their input, and produce a new table as output. For example, the function SAMPLE(*table, int*) might produce a new table consisting of *int* rows of *table*. Table functions can appear anywhere a table or table expression can. They may be used to define new operations on tables to supplement the relational calculus operations supported by the base system. For example, a DBC might add outer join or relational divide, or an operation for solving simultaneous equations. Although syntactically a new operation looks like a function call, table functions require sophisticated processing internally (see "4. The Query Graph Model").

Definition of new data types for columns has more impact on the DDL than the DML. However, they are extremely useful to the user writing DML queries, as they allow type-checking to better reflect the user's semantic intent, and enable better structuring of the user's data. Starburst will allow the definition of almost any type. Columns whose type is externally defined can appear anywhere a column with built-in type can appear, and functions can be defined on them. An overview of our approach to externally defined types appears in [WILM88].

With its increased orthogonality and extensibility, Hydrogen can express very complex queries. This is necessary for achieving Starburst's goal of supporting nontraditional applications, such as logic programming. As in SQL, Hydrogen has a relatively small number of built-in constructs. This keeps the grammar small and compact, and perhaps makes the language easier to learn. However, it creates two difficulties. First, while complex queries can be expressed, their representation in Hydrogen is usually quite complex as well. This could make Hydrogen very hard to read and write, making application development more difficult. Some "syntactic sugar" may be needed on top of Hydrogen to make Starburst sufficiently easy to use. In addition, even with a small number of constructs, there are frequently several ways of expressing the same query. In SQL, for example, the standard query asking for all employees who make more than their manager can be expressed either as a subquery or as a join [KIM82], with several variations on predicate order, etc. In Hydrogen, it could also be expressed several different ways using table expressions. Whenever feasible, the performance of a query should depend on its *meaning* rather than on its *expression*; this increases nonprocedurality. Identifying equivalent expressions for a query poses a significant challenge for optimization.

Overall, we are pleased with the expressiveness of Hydrogen, despite potential difficulties. We are experimenting with a powerful architecture for rewriting queries (see "5. Query Rewrite") to address the optimization issue. We will be better able to address the complexity issue when we have some experience developing applications for Starburst.

3. Overview of Language Processing

As in System R and R*, processing of the data manipulation language (DML) consists of two stages: first, compilation of the query, and then, execution [CHAM81]. These two stages may be separated in time, since the result of the compilation stage can be stored for future use. The different phases of query processing are shown in Figure 1. The query is first broken into tokens, then parsed into its internal representation, the Query Graph Model (QGM). Semantic analysis of the query is also done during parsing, so the QGM produced is guaranteed to be valid. The query rewrite phase transforms the QGM representation of the query into an equivalent QGM for better performance. This phase could be bypassed for faster query compilation at the expense of potentially lower runtime performance. The plan optimizer chooses an execution strategy (Query Evaluation Plan, or QEP) for the query based on estimated cost, and this strategy is then refined (Plan Refinement) for efficient interpretation by the Query Evaluation System at runtime.

4. The Query Graph Model

Starburst must be able to accommodate many types of extensions, including language extensions, data management extensions and language processing extensions. For many of these extensions, the DBC may wish to influence Corona's basic internal processing. For example, a DBC providing an outer join operation may wish to tell the system when predicates on the join can be "pushed down" to (applied to) the individual tables. The rules for predicate push-down for outer join are different than those for regular join [ROSE84]. At the same time, many base system functions (e.g., catalog

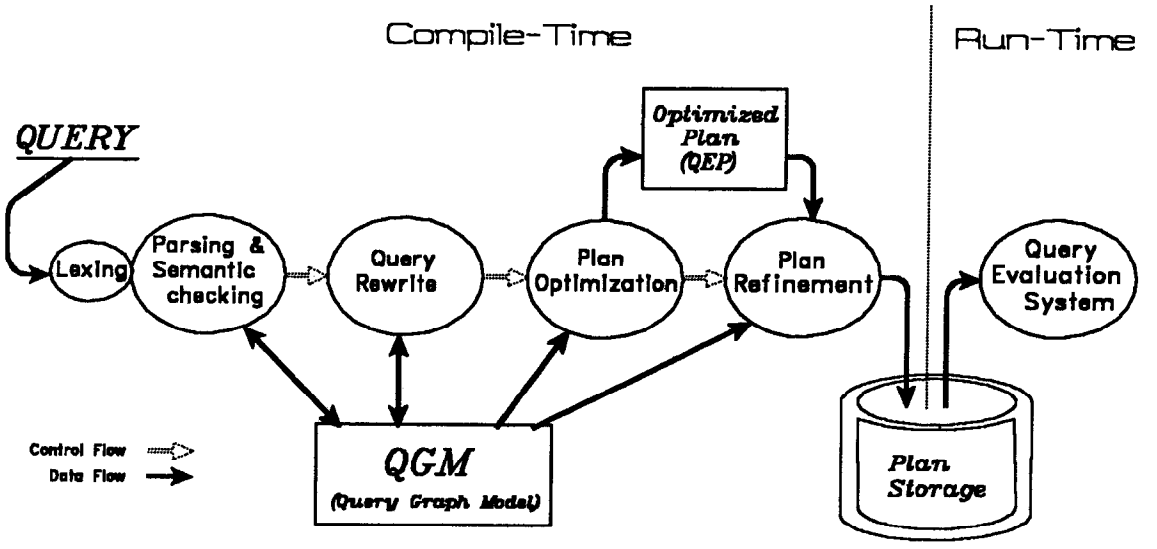

Figure 1: Phases of Query Processing

interface) can frequently be used by the extension. Ideally, the DBC's extensions should be able to invoke Corona's code for these common functions without altering them.

In Starburst we meet both goals, allowing the DBC to use as much of Corona's function as he can and to write the rest himself, by defining a generic internal representation for queries and giving the DBC access to this representation. We call this representation the Query Graph Model (QGM). QGM can be regarded as the schema for a main memory database storing information about a query. This database is the main interface between different phases of query compilation (Figure 1), and between Corona and any extensions.

We do not have space in this paper for a formal description of QGM semantics [PIRA89]. Instead, we will use the following query and its corresponding QGM (Figure 2(a)) to illustrate the most important QGM constructs:

```
SELECT partno, price, order_qty
 FROM quotations Q1
 WHERE Q1.partno IN
   (SELECT partno
     FROM inventory Q3
     WHERE Q3.onhand_qty < Q1.order_qty
           AND Q3.type = 'cpu')
```

This query returns the part number, price and order amount corresponding to each quotation for a cpu part that is in inventory, and for which the supply on hand is low.

In QGM, queries are represented as a series of high level operations on tables. These operations include: SELECT, which performs selection, projection and join; GROUP BY, which forms equivalence classes (groups) from tuples of the input table and applies aggregate functions to each group; INSERT; UPDATE; DELETE; UNION; INTERSECTION; and so on. Our query involves two SELECT operations, one for each SELECT...FROM...WHERE. These are shown as two separate boxes in Figure 2(a). Each operation has a head that describes its output table (the column names and datatypes[2]) and a body that represents the operation graphically. In this query, the outer SELECT consists of an access to the quotations table and a predicate over the result of a subquery. The inner SELECT contains an access to inventory, and a predicate with two conjuncts. Each access to either a stored or a derived table (one produced by another operation) is represented in the QGM by a vertex and a dotted line (range edge) connecting the vertex to the table or operation being accessed. Each conjunct of a predicate is represented by a rectangle on a solid line (qualifier edge) connecting one or more vertices (loops represent single table predicates). Thus in our query's QGM, Q1 represents the access to the quotations table (stored tables are shown as dotted boxes, to distinguish them from derived tables), Q2 the access to the result of the lower SELECT operation, and Q3 the access to inventory. There is a qualifier edge between Q1 and Q2 representing the predicate over the subquery result (the "IN" predicate), an edge between Q1 and Q3 representing the first conjunct of the lower SELECT's predicate (Q3.onhand_qty < Q1.order_qty), and a loop from Q3 to itself, representing "Q3.type = 'cpu'".

Vertices represent *iterators*, which may be either *setformers* or *quantifiers*. A setformer, such as vertex Q1 or Q3, creates a new table (set) from its input tables (essentially by projec-

2 For simplicity, we show only the column names in Figure 2

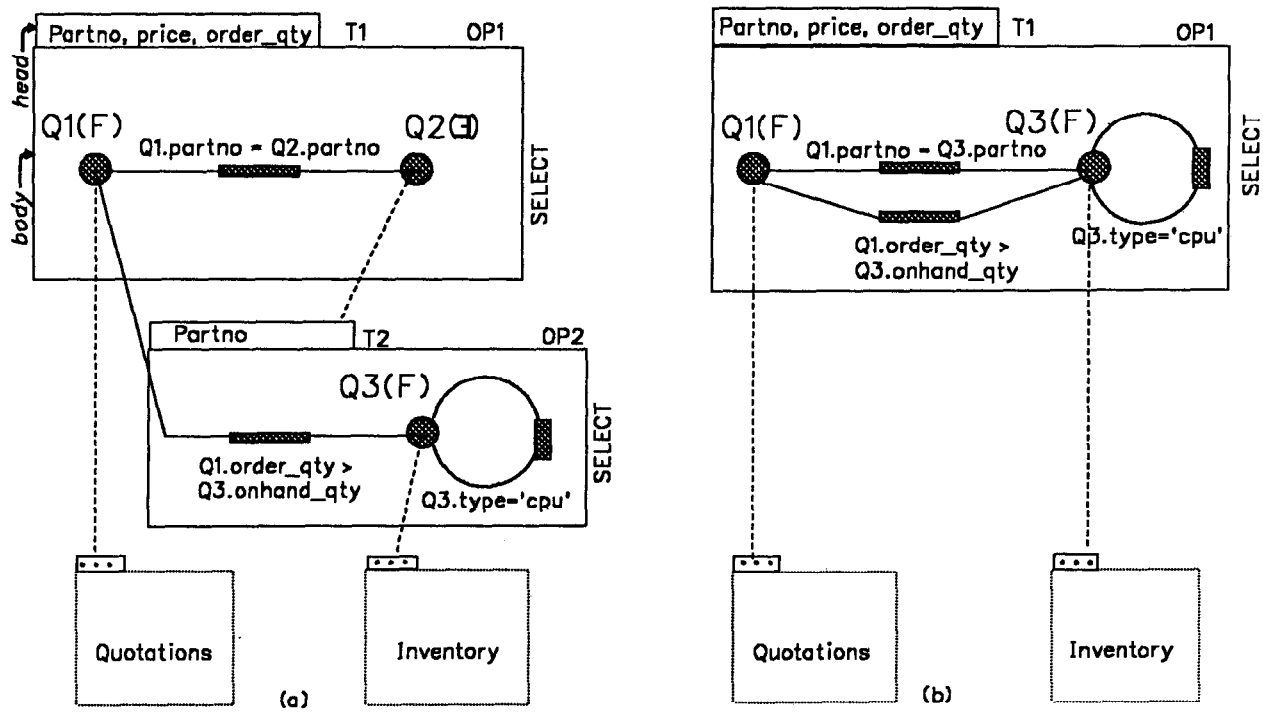

Figure 2: (a) QGM of a query (b) Same QGM after rewrite rules applied

tion). A quantifier (e.g., Q2) is a tuple variable ranging over its input tables. Setformers differ from quantifiers in that each element in the table over which a setformer ranges may be used to construct the output from this operation (the output table), whereas the elements ranged over by a quantifier will not be added to the output table, only used to restrict it. Thus Q3 is a setformer, producing a table of partno's. Q2 is a quantifier, ranging over this table; the elements it ranges over will be used to restrict those produced by Q1 in the upper box, but will not be added to the result of that upper box.

Many iterators can range over the same input table, as each represents a separate access to that table. Quantifiers have different types, which may be interpreted differently by different operations. For example, quantifiers of type *Exists* (∃) and *All* (∀) are interpreted by the SELECT operation as existential and universal quantifiers, respectively, Q2 is an existential quantifier corresponding to the IN keyword of the query. There is only one built-in type of setformer, *ForEach* (*F*), though others may be added (see below).

Vertices (setformers and quantifiers), edges (range and qualifier), and boxes (with heads) are the main constructs of QGM. They are extremely powerful, allowing us to express arbitrarily complicated queries. Further, because the interpretation of these constructs can vary depending on the type of the operation, QGM is extensible. We anticipate that the most common extension will be to add new operations on tables, which requires the DBC to give an interpretation to the qualifier edges and any new iterator types. The rest of QGM is "generic"; that is, the information is standard and has a standard interpretation, regardless of the operation. This is because most of QGM describes *tables*, both input and output, and not the operations themselves.

For example, assume we wished to add a left outer join operation. In this case, qualifier edges correspond to predicates, as with SELECT. However, the meaning of the predicate is different, because if matching inner values are not found, the outer tuple still must be included in the result. A simple way to flag this difference is to change the setformer of the outer table to a new type, say, *PF* (Preserve Foreach). Many of Corona's built-in functions (e.g., catalog lookup, name resolution, most of the semantic checking, etc.) can be used to generate QGM for queries containing outer join. The DBC must write code for those parts that are different. In this case, he might write code to check whether the outer join was associative or commutative (in general it would not be), and to change *F* setformers to *PF* where tuples must be preserved. Of course, other parts of the system must be changed before the user can actually write and execute an outer join query. Query rewrite rules, optimizer cost estimates, execution routines, and so on, are all needed. The important point here is that QGM is modifiable to allow the operation to be represented internally.

Similar query graphs have been described in the literature [RIES83, ZANI82]. QGM expands on these models in two respects. First, QGM has the full expressive power of Hydrogen. Thus it can represent intricacies of the language such as nested operations, different iterator types, aggrega-

tion, correlation, and recursion. Second, due to its generic interfaces, QGM can also be used to represent extensions to the system and to the language, such as externally defined table functions and iterator types.

In summary, QGM provides an understandable, detailed, and unambiguous semantic representation of a query that can be modified by the DBC as he adds operations to the system. Further, because of the generic nature of the *table* abstraction, the DBC can capitalize on the common services provided by Corona. Thus QGM simplifies the DBC's task, while giving him a great deal of flexibility and power.

5. Query Rewrite

There are several reasons why relational DBMSs need to rewrite queries. There are frequently several alternative phrasings of a query. Since relational query languages are ostensibly nonprocedural, these should perform equivalently. Further, complex applications often rely on abstractions provided by views to reduce the complexity of query specification. As view definitions are hidden from the query writer, only the DBMS can rewrite queries involving views.

There is an extensive set of transformations that could be performed during query rewrite. Views may be merged with the queries that use them, as in System R and INGRES [STON76]. Redundant joins may be eliminated [OTT82]. Predicates may be migrated from one operation to another to minimize the amount of data accessed (this includes "sideways information passing" [ULLM85]). With the introduction of recursion in DBMS queries, transformations such as magic sets [BANC86] should be incorporated. Semantic query optimization [KING81, SHEN87] uses integrity constraints to transform queries. All of these transformations may lead to better performance for the query, and new transformations are constantly being proposed (e.g., [DAYA87]).

It is unlikely that we will ever have a complete set of transformations, even for a standard relational language. And for an extensible language such as Hydrogen, as new language constructs and operations are added, new transformations will be needed. Thus this part of the system must be highly extensible. To achieve this, we have created a new phase in query processing for handling query rewrites at the QGM level, using a rule-based approach [HASA88]. Query rewrite is essentially a form of optimization. However, traditional DBMS optimization primarily deals with low level decisions, such as the methods for table accesses, and the methods and order of joins, creating a plan for query execution. Query rewrite occurs at a higher level of abstraction (in our case, QGM), but which is slightly lower (more procedural) than the query language level.

Although many rule-based systems exist, we have chosen to develop our own rule system for query rewrite. We require a rich set of primitives for manipulating query graphs, nested execution of rules, explicit control over the order of rule execution, and some sort of termination guarantee to avoid endless looping, yet preserve consistency. We estimated that the effort of integrating an existing system into Starburst and modifying it to meet these requirements would be greater than the cost of implementing our own.

In our rule system, the rule language is C. A rule consists of two parts, the *condition* and the *action*, each written as a C function. The rule writer is expected to ensure that every rule changes a consistent QGM representation into another consistent QGM representation, i.e., each rule completes a transformation. We provide a set of rules for rewriting the base system operations. A DBC may provide additional rules for these operations, or rules for new (externally defined) operations. Rules may be grouped into rule classes to limit the number of rules that have to be examined, to allow modularization of rules, and to give the DBC (and ourselves) more explicit control over the execution sequence.

As an example, consider the following rules:

Rule 1 (Subquery to Join):[3]

```
IF OP1.type=Select ∧ Q2.type='∃' ∧
   (at each evaluation of the existential predicate
    at most one tuple of T2 satisfies the predicate)
THEN
  Q2.type = 'F';          /*convert to join*/
```

Rule 2 (Operation Merging):

```
IF OP1.type=Select ∧ OP2.type=Select ∧ Q2.type='F'
  ∧ (NOT (T1.distinct=false
          ∧ OP2.eliminate_duplicate =true))
THEN
  merge OP2 into OP1;
  IF OP2.eliminate_duplicate = true
    THEN OP1.eliminate_duplicate = true;
```

The first rule says that an existential subquery can be converted to a join when there is at most one matching tuple of the subquery for each tuple of the main query. The second rule says that two SELECT operations may be merged as long as there is no conflict in the way they handle duplicates.[4] These rules can be applied to the QGM of figure 2(a), resulting in the transformed QGM of figure 2(b). By merging the operations, there is greater scope for optimization, which may result in an improved execution plan.

The rules we provide for base system operations fall mainly into three classes: predicate migration, projection push-down, and operation merging. Predicate migration allows predicates to be pushed down into lower level operations to minimize the amount of data retrieved. Predicates may also be

3 For purposes of illustration, we assume the subquery is a conjunct of the predicate

4 For a more complete explanation of these rules, and a more general version of Rule 1, see [HASA88]

replicated, and replicas migrated to multiple operations to reduce execution cost. Predicates may also be moved up, and then possibly down to other operations, which may be beneficial. Rules for projection push-down avoid the retrieval of unused columns of tables or views. These rules interact with those for predicate migration: for example, when a predicate is pushed to a "lower" operation, columns referenced only by that predicate are no longer needed by the higher operation. Operation merging rules merge QGM "boxes", creating the union of the predicates and iterators of the original operations to allow more scope for optimization. View merging rules fall into this category, as does the rule in our example. Not all operations are mergeable (for example, GROUP BY cannot merge with SELECT). Even two SELECT operations may not always merge, depending, for example, on how they handle duplicates. Other rules convert subqueries to joins [KIM82, GANS87], and apply miscellaneous transformations.

DBC's can take advantage of existing rules in any of these classes, or write their own rules in these or other classes. For example, predicate push-down rules are of two types: those that specify when a predicate may be pushed down *from* an operation, and those that specify when a predicate may be received by (pushed down *to*) an operation. A DBC adding a left outer join operation will want the *from* rules to apply automatically to the type F setformer (see "4. The Query Graph Model"), without his having to respecify them. However, they should not be applied to the type PF setformer, as they would then eliminate tuples which should be preserved. Left outer join (unlike the SELECT operation) does not keep predicates, but can receive them if they refer only to columns of the PF setformer, in which case they are pushed *through* the outer join operation to the operation ranged over by the PF setformer. Thus the DBC would have to write his own rule for receiving predicates.

Besides the rewrite rules, our rule system has two other major components: the rule engine and the search facility. The rule engine is independent of the individual rules, but is designed to meet the needs of query rewrite rules in general. It handles IF THEN rules, using a forward chaining strategy. Several control strategies are provided: sequential (rules are processed sequentially), priority (higher priority rules are given a chance first), and statistical (next rule is chosen randomly based on a user defined probability distribution). To keep the rule engine from spending too much time rewriting queries, it can be given a budget. When the budget is exhausted, the processing stops at a consistent state (of QGM). The search strategy is independent of both the rules and the rule engine, but is specific to QGM. Its role is to browse through QGM, providing the context for the rules to work on. Both depth first (top down) and breadth first search are supported.

There may be several alternative transformations for a query. For example, a join predicate may be pushed down on either of its iterators (but not both!). If a view is used in multiple places in a query, it may be merged into the query each time, or materialized once and used several times. To choose among these alternatives, cost analysis is needed. However, cost estimates are known only at the plan level, since we need to have decisions about join order, access methods, etc, in order to estimate costs. As a result, query rewrite must interact with plan optimization to estimate the cost of alternative QGM representations. We have initially chosen to generate the alternatives during query rewrite and select among them during optimization. We have therefore added a new operation, CHOOSE, to QGM to link together the alternatives. This operation can be eliminated when the optimizer chooses an alternative, or kept in the plan until runtime to allow a decision based on runtime parameters (e.g. the value of programming language variables). A similar operation for plans has been proposed by Graefe [GRAE89].

One drawback of deferring the choice of an alternative until optimization is that alternatives cannot be pruned during query rewrite. This is particularly troublesome for predicate migration, since the number of alternatives grows as a product of the number of iterators and predicates. We are currently investigating ways to integrate the query rewrite and optimization phases to avoid this problem.

We have built the rule system and defined rules for the base operations and for several extensions. Recently we have been adding rewrite rules for recursive queries, including rules to do magic set transformations [BANC86]. Much work remains to be done. In particular, we would like to do a comprehensive study of the rule system's performance. We would like to prove the correctness of the rewrite rules designed so far, and develop tools to help DBC's establish the correctness of their transformations. We are pursuing the design of a more restrictive rule language that would allow automatic analysis and optimal execution of rules, and are studying the applicability of efficient execution techniques such as RETE networks [FORG82] and rule indexing. Finally, we need to add rules for many more extensions, to determine whether the rule system is sufficiently flexible, and to study the interactions between rules for different extensions.

6. Plan Optimization

Given a QGM representation of the query, the optimizer estimates the cost of alternative query evaluation plans (QEPs) to execute that query, and chooses the cheapest. As with other query optimizers, the Starburst optimizer may be characterized by its approach to three major aspects of optimization: (1) plan generation, (2) plan costing, and (3) search strategy. Again, Starburst's goal of extensibility dictated a more flexible specification of each of these aspects of optimization than earlier optimizers for System R [SELI79] and R* [LOHM85], which embedded all execution strategies, cost formulas, and search strategies in the code. The challenge was to find an efficient yet easy-to-specify representation for each aspect, in order to facilitate the specification of a rich repertoire of alternative execution plans without sacrificing compiler performance. For greater extensibility, the

three aspects were designed to be largely orthogonal, so that each could be modified independently of the others.

The optimizer algorithm optimizes each QGM operation independently, bottom up, using a rule-driven *plan generator* and rules peculiar to that operation's type to construct QEPs and evaluate their cost. For example, the most common operation type, a SELECT operation, has rules for accessing stored tables in various ways, and rules for joining multiple iterators. A DBC adding a left outer join operation would add new rules for the outer join operation itself, but could benefit from the existing rules for single table accesses.

The Starburst plan generator generates alternative plans using a constructive, "building blocks" approach [LOHM88]. This approach is closer to that of the Genesis project [BATO87a] than the plan transformational approach of the Exodus project [GRAE87a, GRAE87b] or of Freytag [FREY87]. Executable plans are defined using a grammar-like set of parameterized production rules called *strategy alternative rules (STARs)*. These rules define higher-level "non-terminal" constructs from low-level "terminal" database operators strung together, in a way resembling a functional programming language [BACK78]. A "terminal" in this grammar is a *low-level plan operator (LOLEPOP)* that will be interpreted by the Query Evaluation System at run-time. LOLEPOPs are a variation of the relational algebra (e.g., **JOIN, UNION**, etc.), supplemented with physical operators such as **SCAN, SORT, SHIP**, etc. Each LOLEPOP is expressed as a function that operates on 0 or more streams of tuples, and produces 0 or more new streams (typically one). A Starburst *query evaluation plan* (QEP) is a nesting of invocations of LOLEPOPs. A STAR consists of a name (the nonterminals of our grammar), zero or more parameters, and one or more alternative definitions in terms of LOLEPOPs or other STAR names. IF conditions can be attached to any alternative, to determine the applicability of that alternative, and "∀" clauses can be used to generate a set of related alternatives.

Every table (either a base table or the result of a plan) has a set of properties that summarize its characteristics [BATO87b, ROSE87] and hence are important to the cost model. These properties are of three types: relational (e.g., tables joined, columns accessed, and predicates applied thus far), operational (e.g., order of tuples (if any), site of result) and estimated (e.g., (cumulative) cost, cardinality). Each LOLEPOP changes selected properties of its operands, in a way influenced by its parameters, usually adding cost. These changes, including the appropriate cost and cardinality estimates, are defined by a C function for each LOLEPOP. For example, **SORT** changes the order of tuples to the order specified in a parameter. **SHIP** changes the site to the specified site. **SCAN** changes a stored table to a memory-resident stream of tuples, but optionally can also subset columns and apply predicates that may be enumerated as parameters. The latter option will of course change the cardinality property as well.

A STAR may require certain properties of its operands, especially for certain join methods. For example, the merge join requires its input table streams to be ordered by the join columns. Required properties are achieved by additional *"glue" STARs* that find the cheapest plan satisfying the requirements. If necessary, glue STARs may add LOLEPOPs to make existing plans meet the requirements. For example, **SORT** can be added to change the tuple order, or **SHIP** to change the site [LOHM88].

When a QGM operation (especially SELECT) contains multiple iterators, a *join enumerator* then enumerates all valid join sequences by iteratively constructing progressively larger sets of iterators (*iterator sets*) from two[5] smaller iterator sets, starting initially from the plans generated earlier for sets of a single iterator. For each such pair of iterator sets, the join enumerator invokes the plan generator to generate and evaluate alternative QEPs for that join using the join STARs. The enumeration exploits join predicates referencing more than two iterators, implied predicates, and joins permitting composite inner tables, producing a potentially larger set of plans than did the R* and System R optimizers [ONO88].

Starburst's wider range of plans necessitates more powerful mechanisms for limiting the search. Our optimizer has both query-specific parameters to limit the search space as well as an extensible search strategy. Each alternative for a STAR will have a rank associated with it, so that alternatives exceeding a given rank can be pruned by the plan generator. In addition, a prioritized queue mechanism parameterizes the order in which STARs are evaluated. Merely by changing the priorities, this general mechanism can implement breadth-first, depth-first, or many other strategies. Two other parameters allow the join enumerator to prune join sequences having composite inners ("bushy trees") or no join predicate (Cartesian products), as System R and R* always did.

We have implemented the complete join enumerator [ONO88] and an initial STAR-based plan generator [LEE88]. The plan generator contains (1) a general-purpose STAR evaluator, (2) a search strategy that chooses the next STAR to evaluate, and (3) an array of STARs. This design permits the optimizer designer to add, change, or delete rules in the STAR array without affecting the code for the search strategy or the rule evaluator. Similarly, the search strategy can be changed without affecting the rule evaluator or the STARs. Each time the plan generator is invoked, it constructs a tree of alternative QEPs as STARs are evaluated and replaced by their alternative definitions, much as is done by a macro processor, until all STARs are fully refined to LOLEPOPs. Then the property function for each LOLEPOP is called to evaluate its effect on the properties of its operands, starting with statistics on stored tables. Currently, the STAR array is initialized manually in the code, but ultimately we hope

5 Joins are typically implemented as dyadic operators, but this limitation can be removed in Starburst.

to build a compiler that will permit the DBC to specify STARs in a very high level functional programming language.

This design is very powerful, and highly extensible. Using STARs, we can readily express all the strategies of the R* optimizer, plus new strategies for composite inners (e.g., (A*B)*(C*D)), new join methods, new access methods, index ANDing, filtration methods such as semi-joins and Bloom-joins [MACK86], dynamic creation of indexes on intermediate results, conversion of subqueries to joins, and materialization of tables at any point to force projection, all in under 20 rules. We can add new properties to existing operators, add new operators, form new plans by combining existing operators in new ways, restrict or broaden the search space, and change the search method. Many of these changes can be accomplished by simple modifications to existing rules or by giving a different value to a parameter. Others require somewhat more sophistication by the DBC (for example, adding a new property). An easily extensible optimizer is critical to achieving Starburst's goals. We plan to experiment with a variety of extensions to demonstrate the power and extensibility of our optimizer.

7. The Query Evaluation System

At runtime, Starburst's query evaluation system (QES) takes a query evaluation plan (QEP), and evaluates it against the database. The QES calls Core to retrieve tuples from tables and attachments into its own set of buffers, where it performs complex operations such as join or aggregation. For extensibility, we need a QES that can be easily changed and extended to accommodate, for example, new types of access methods or new join methods, without changing its overall structure or implementation. We achieve this goal by developing an algebraic interface between the optimizer and the QES. A QEP is an operator tree similar to a query specification in the relational algebra. Each operator takes one or more streams of tuples as input and produces one or more streams of tuples (usually one) as output. We implement the concept of streams by *lazy evaluation* to keep "intermediate" results between operators as small as one tuple.

There are several advantages to this approach. First, using the stream abstraction allows us to add new operators easily. The interaction between operators can be standardized, so that the details of obtaining a tuple from and handing a tuple to another operator can be hidden by predefined macros. The implementers of new operators then can reuse those macros without reimplementing them for each extension. Second, the algebraic interface guarantees the independence of different parts of the QES. As long as the operators use the model of streams as the interface between them, we do not anticipate changes in existing operators due to an extension or change in another part of the QES. Finally, the algebraic approach not only provides an interface to a component that *interprets* QEPs, but it can also serve as the input specification to a component that *compiles* QEPs into iterative programs [FREY86].

Starburst will include about a dozen built-in LOLEPOPS. By far the most difficult of these to implement are the JOIN operators. One reason for this is that in Starburst, we treat subqueries as special types of join. This simplifies QGM, query rewrite and optimization considerably, but causes several problems for the execution system. First, the join operators must be able to handle different *kinds* of joins; that is, joins between different iterator types in QGM. The different kinds of joins include "regular" join, "exists" join (or semi-join), "scalar-subquery" join (one in which the subquery is introduced by a scalar comparison operator), and "op-ALL" join (corresponding to the universal quantifier). This set of join kinds can also be extended as new iterator types are added to QGM. Each join operator takes as one of its parameters a function name, representing the join kind. In this way a single operator can handle many different join kinds.[6]. By clearly separating the "control structure" of the join, i.e., the join method, from the function performed during the join, i.e., the join kind, we provide an additional degree of flexibility for the implementation of joins. For example, "left outer" join could be added as a join kind, allowing the left outer join operator to take advantage of existing methods of join evaluation. Alternatively, or in addition, a new join method specific to left outer join could be added simply by adding a new LOLEPOP.

Secondly, Hydrogen, like SQL, allows both correlated and uncorrelated subqueries, which must then be handled by the join operators. In Corona, we replace the mechanisms of "evaluate-at-open" and "evaluate-at-application" [LOHM84] for uncorrelated and correlated subqueries, respectively, by a single uniform mechanism called "evaluate-on-demand". With this approach, we evaluate subqueries only as they are needed. We also include logic to avoid re-evaluating the subquery when the correlation values have not changed, thus improving the performance during execution.

The third problem arises when evaluating queries containing OR predicates with subqueries. Consider the Hydrogen query

```
SELECT * FROM T1
WHERE T1.A1 = 5 OR T1.A2 =
      (SELECT B2 FROM T2 WHERE T2.B1 = 16).
```

The two predicates must be executed by separate operators. The first could be evaluated using the **FILTER** operator, the other by a **JOIN** operator using any join method combined with the "scalar-subquery" join kind to join T1 and the subquery table(s). However, the **FILTER** operator, if applied first, cannot just discard a tuple which does not satisfy the predicate. Instead it must be handed over to the **JOIN** operator for further consideration. For this, we have designed an additional **OR** operator that is based on the "stream philosophy" and does not require any change to the operators used to evaluate the predicate terms.

---

6 This does not imply that every join method can be combined with every join kind.

During the initial implementation of the QES, we realized that it has a very regular structure (due to the algebraic approach) and simple interfaces. To support the DBC who extends the QES, we would like to have an additional tool that relieves him from knowing which parts of the system to change in order to add or remove operators from the component. We have considered providing a *QES generator*. In some well-defined way, the DBC could specify each operator and its corresponding execution routine (and possibly a printing and building routine). The generator would then add the necessary data structures and procedures that "glue" these operators together to form the QES.

8. Conclusions

Implementation of the Starburst prototype is well underway. Initial implementations of all pieces of Corona described in this paper have been built on top of the basic Core functions, and are being integrated. Hydrogen statements of arbitrary complexity can be parsed and translated into QGM, and the QGM rewritten by an initial set of rules for view merging and predicate push-down. Much of the infrastructure of the optimizer is in place, including the inference engine for STARs. The interpreter routines for most of the QEP operators have been written and tested in an artificial environment, but few of the routines for transforming the optimizer's output QEP into an executable QEP exist. We have executed our first single-table SELECT. We have some initial proof of the extensibility of the language processing. We have been able to extend the early parts of the system to add a left outer join operation, so that queries with outer join can now be parsed, represented in QGM and manipulated correctly by the rewrite rules. Adding rules for new operations to the optimizer seems to be easy, though defining them may be hard, and adding new operators to the QES has been trivial (once the code for the operators has been written).

Although work on Corona is far from complete, we have already learned several important lessons about designing and building an extensible system. The success we have had in building and extending Corona to date is based on two principles: orthogonality and the table abstraction. Understanding which parts of the system are logically orthogonal allows us to separate them, leading to a very modular design. Thus, complex components are broken into independent pieces, which can be replaced or extended individually. Keeping the extensions orthogonal from the base system also creates a basis for sharing across application areas. This reliance on orthogonality exists throughout Corona. In addition, a powerful internal model of processing is critical. Basing the design of language processing on the table abstraction (and the QGM representation) led to simpler designs and a clean interface for extensions. Since both the extensions and the base system are written in terms of the same high level model, sharing between different extensions and the system should be enhanced.

On the whole, we are pleased with the design of Corona. All of the pieces are quite powerful and extensible. However, some concerns remain. In Corona, extensions at different levels of the system are done in different ways, well suited to their separate problems. For example, query rewrite uses production rules, and cost optimization employs grammar-like rules, while the QES takes an algebraic approach. We need to see in practice how well these approaches fit together, whether they can be combined in a single unified framework and whether there is any benefit to combining them. Performance of the system is another open question. These issues can only be resolved through completion of the prototype and experimentation with actual applications.

There are many other open problems in extensible systems. Currently, customization of Corona (and Starburst in general) requires a skilled programmer, and, for many extensions, experience with database technology and implementation. While even this level of extensibility is a great improvement over standard DBMSs, we feel that it is important to investigate tools for making customization easier. What are the appropriate tools? How easy can this process be made? How can independent extensions be combined, or more importantly, how can we ensure that independent extensions do not conflict? For example, how do we ensure that transformation rules for one extension do not interfere with the rules of another extension? In general, how do we ensure the "sanity" of the DBMS after several extensions have been made? Many other challenges will arise as we look at specific applications and further extensions to Starburst's functionality. For example, in a distributed environment where different Starburst nodes may be extended in different directions, query planning and coordination of metadata may be difficult.

In the near term, we are finishing the implementation of our prototype. In the future, we plan to study the uses of this extensible technology. We will test the extensibility of our system by building (or getting others to build) application systems for Starburst. We are currently exploring knowledge-based systems as a first application area, examining issues such as how to represent and support frames and rules in the database, what function should reside in the DBMS, and what function should be left in the application. We will also continue to pursue advances in database technology, including new access methods, join algorithms, parallelism and other execution strategies, using Starburst as a testbed for our research.

In summary, this paper described the overall design of the Starburst language processing component. The main challenge for language processing is to handle extensions to the data manager, the language, and the language processing itself. We feel that Corona has gone a long way towards meeting this goal. Among the novel features that have helped us achieve this extensibility are an internal model of the query based on tables, the use of rule-based processing for query rewrite, the separation of the different components of optimization and using grammar-like rules to generate plans, and an execution system based on the relational algebra. Still, there are many exciting open questions in extensible database research.

Acknowledgements

Several others have contributed greatly to the design of Corona. Mavis Lee and Waqar Hasan designed and implemented the rule-based optimizer prototype and the rule engine for query rewrite, respectively. Kiyoshi Ono created the optimizer infrastructure and the join enumerator. Bill Cody designed and built the lexical analysis and the parser. Bruce Lindsay and George Lapis designed and implemented much of the infrastructure for Corona, as well as the extensible data definition, authorization and catalog management components. Shel Finkelstein, in investigating how Starburst could support knowledge-based systems applications, has suggested many ways to increase Corona's extensibility, and has frequently forced us to re-examine our designs. Our colleagues and collaborators on Starburst -- Walter Chang, John McPherson, C. Mohan, Peter Schwarz, Paul Wilms and Bob Yost -- have improved our designs and clarified our ideas, while causing us to stretch to accommodate their innovations in Core. We are grateful to Bruce Lindsay, Shel Finkelstein, John McPherson, Pat Selinger, Irv Traiger, Paul Wilms and several anonymous referees who carefully and constructively read a draft of this paper.

9. Bibliography

[ABIT86] Abiteboul, S., M. Scholl, G. Gardarin and E. Simon, Towards DBMSs for Supporting new Applications, *Procs. of the Twelfth International Conference on Very Large Databases* (Kyoto, Aug. 1986).

[ASTR76] Astrahan, M., 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** (June 1976).

[BACK78] Backus, J., Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *Comm. ACM* **21:8** (Aug. 1978).

[BANC86] Bancilhon, F., D. Maier, Y. Sagiv and J. Ullman, Magic sets and other strange ways to implement logic programs, *5th ACM Symposium on Principles of Database Systems, Cambridge* (1985).

[BANE87] Banerjee, J., W. Kim, H.J. Kim and H. Korth, Semantics and Implementation of Schema Evolution in Object-Oriented Databases, *Procs. ACM SIGMOD* (San Francisco, May 1987).

[BATO86] Batory, D., GENESIS: A Project to Develop an Extensible Database Management System, *Procs. 1986 Int. Workshop on Object-oriented Database Systems* (Asilomar, Sept. 1986).

[BATO87a] Batory, D., A Molecular Database Systems Technology, *Tech. Report TR-87-23* (Dept. of Comp. Sci., Univ. of Texas at Austin, June 1987).

[BATO87b] Batory, D., Extensible Cost Models and Query Optimization in GENESIS, *IEEE Database Engineering* **10:4** (Dec. 1986).

[CARE86] Carey, M., D. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J. Richardson and E. Shekita, The Architecture of the EXODUS Extensible Database System, *Procs. 1986 Int. Workshop on Object-oriented Database Systems* (Asilomar, Sept. 1986).

[CARR79] Carr, B., Graphs and Networks, *Oxford University Press: New York, NY, 1979, ch.3* (1979).

[CHAM81] Chamberlin, D., M. Astrahan, W. King, R. Lorie, J. Mehl, T. Price, M. Schkolnick, P. Selinger, D. Slutz, B. Wade and R. Yost, Support for Repetitive Transactions and Ad-Hoc Queries in System R, *ACM Trans. on Database Systems* **6:1** (Mar. 1981).

[DATE84] Date, C., A Critique of the SQL Database, *ACM SIGMOD Record* (1984).

[DAYA86] Dayal,U. and J. Smith, PROBE: A Knowledge-Oriented Database Management System, *On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies, Springer Verlag* (Brodie & Mylopoulos (eds.), 1986).

[DAYA87] Dayal, U., Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers, *Procs. of the Thirteenth International Conference on Very Large Databases* (Brighton, Sept. 1987).

[FORG82] Forgy, C., RETE: A fast algorithm for the many pattern/ many object pattern match problem, *Artificial Intelligence 19:17-38* (1982).

[FREY86] Freytag, J.C. and N. Goodman, Rule-Based Translation of Relational Queries into Iterative Programs, *Procs. ACM SIGMOD* (Washington, D.C., May 1986).

[FREY87] Freytag, J.C., A Rule-Based View of Query Optimization, *Procs. ACM SIGMOD* (San Francisco, May 1987).

[GANS87] Ganski, R. and H. Wong, Optimization of Nested SQL Queries Revisited, *Procs. ACM SIGMOD* (San Francisco, May 1987).

[GRAE87a] Graefe, G. and D. DeWitt, The EXODUS Optimizer Generator, *Procs. ACM SIGMOD* (San Francisco, May 1987).

[GRAE87b] Graefe, G., Software Modularization with the EXODUS Optimizer Generator, *IEEE Database Engineering* **10:4** (Dec. 1986).

[GRAE89] Graefe, G., Dynamic Query Evaluation Plans, *Oregon Graduate Center TR No. CS/E 88-003* (1988).

[GRIF76] Griffiths, P. and B. Wade, An Authorization Mechanism for a Relational Database System, *ACM Trans. on Database Systems* **1:3** (Sept. 1976).

[GUTT84] Guttman, A., R-Trees: A Dynamic Index Structure for Spatial Searching, *Procs. ACM SIGMOD* (Boston, June 1984).

[HAAS88] Haas, L, W. Cody, J. Freytag, G. Lapis, B. Lindsay, G. Lohman and H. Pirahesh, An Extensible Processor for an Extended Relational Query Language, IBM Research Report RJ 6182 (April 1988).

[HASA88] Hasan, W. and H. Pirahesh, A Rule System for Query Rewrite Optimization in Starburst, IBM Research Report RJ6367 (Aug. 1988).

[IBM87] IBM Systems Application Architecture, Common Programming Interface: Database Reference, SC 26-4348-0 (Sept. 1987).

[KIM82] Kim, W., On Optimizing an SQL-like Nested Query, *ACM Trans. on Database Systems* **7:3** (Sept. 1982).

[KIM87] Kim, W., J. Banerjee, H.-T. Chou, J. Garza and D. Woelk, Composite Object Support in an Object-Oriented Database System, *Procs. ACM Conf. on Object Oriented Programming Systems, Languages and Applications* (Orlando, Oct. 1987).

[KING81] King, J., QUIST: A system for semantic query optimization in relational database, *Procs. of the Seventh International Conference on Very Large Databases* (1981).

[LEE88] Lee, M., J.C. Freytag and G. Lohman, Implementing an Interpreter for Functional Rules in a Query Optimizer, *Procs. of the Fourteenth International Conference on Very Large Databases* (Los Angeles, Aug. 1988).

[LIND87] Lindsay, B., J. McPherson and H. Pirahesh, A Data Management Extension Architecture, *Procs. ACM SIGMOD* (San Francisco, May 1987).

[LOHM84] Lohman, G., L. Haas, R. Kistler, P. Selinger and D. Daniels, Optimization of Nested Queries in a Distributed Relational Database, *Proc. of the Tenth International Conference on Very Large Database* (Singapore, Aug. 1984).

[LOHM85] Lohman, G., C. Mohan, L. Haas, B. Lindsay, P. Selinger, P. Wilms and D. Daniels, Query Processing in R*, *Query Processing in Database Systems, Springer-Verlag* (Kim, Batory, & Reiner (eds.), 1985).

[LOHM88] Lohman, G., Grammar-like Functional Rules for Representing Query Optimization Alternatives, *Procs. of ACM-SIGMOD* (Chicago, IL, June 1988).

[MACK86] Mackert, L. and G. Lohman, R* Optimizer Validation and Performance Evaluation for Distributed Queries, *Procs. of the Twelfth International Conference on Very Large Databases* (Kyoto, August 1986).

[MAIE86] Maier, D., J. Stein, A. Otis and A. Purdy, Development of an Object-Oriented DBMS, *Procs. ACM Conf. on Object Oriented Programming Systems, Languages and Applications* (Portland, Sept. 1986).

[ONO88] Ono, K. and G. Lohman, Extensible Enumeration of Feasible Joins for Relational Query Optimization, IBM Research Report RJ6625 (Dec. 1988).

[OTT82] Ott, N. and K. Horlander, Removing Redundant Join Operations in Queries Involving Views, *Heidelberg Scientific Center, TR 82.03.003* (March 1982).

[PAUL87] Paul, H., H. Schek, M. Scholl, G. Weikum and U. Deppisch, Architecture and Implementation of the Darmstadt Database Kernel System, *Procs. ACM SIGMOD* (San Francisco, CA, May 1987).

[PIRA89] Pirahesh, H. and S. Finkelstein, An Extensible Query Graph Model, internal working paper.

[RIES83] Ries, D., A. Chan, U. Dayal, S. Fox, K. Wen-Te and L. Yedwab, Decompilation and Optimization for ADAPLEX: A procedural Database Language, *Technical Report, CCA-82-04* (Sept. 1983).

[ROSE84] Rosenthal, A. and D. Reiner, Extending the Algebraic Framework of Query Processing to handle Outerjoins, *Procs. of the Tenth International Conference on Very Large Databases* (Singapore, Aug. 1984).

[ROSE86] Rosenthal, A., S. Heiler, U. Dayal and F. Manola, Traversal Recursion: A Practical Approach to Supporting Recursive Applications, *Procs. ACM Sigmod* (Boston, June 1984).

[ROSE87] Rosenthal, A. and P. Helman, Understanding and Extending Transformation-Based Optimizers, *IEEE Database Engineering* **10:4** (Dec. 1986).

[SCHW86] Schwarz, P., W. Chang, J.C. Freytag, G. Lohman, J. McPherson, C. Mohan, H. Pirahesh, Extensibility in the Starburst Database System, *Procs. 1986 Int. Workshop on Object-oriented Database Systems* (Asilomar, Sept. 1986).

[SELI79] Selinger, P., M. Astrahan, D. Chamberlin, R. Lorie, and T. Price, Access Path Selection in a Relational Database Management System, *Procs. ACM SIGMOD* (June 1979).

[SHEN87] Shenoy, S. and Z. Ozsoyoglu, A System for Semantic Query Optimization, *Procs. ACM SIGMOD* (May 1987).

[STON76] Stonebraker, M., E. Wong, P. Kreps and G. Held, The design and implementation of INGRES, *ACM Trans. on Database Systems* **1:3** (Sept. 1976).

[STON86] Stonebraker, M. and L. Rowe, The Design of POSTGRES, *Procs. ACM SIGMOD* (Washington, D.C., May 1986).

[ULLM85] Ullman, J., Implementation of Logic Query Languages for Databases, *ACM Trans. on Database Systems* **10:3** (Sept. 1985).

[WILM88] Wilms, P., P. Schwarz, H. Schek and L. Haas, Incorporating Data Types in an Extensible Database Architecture, *3rd Int'l Conference on Data and Knowledge Bases* (Jerusalem, June 1988).

[ZANI82] Zaniolo, C. and M. Melkanoff, A Formal Approach to the Definition and the Design of Conceptual Schemata for Database Systems, *ACM Trans. on Database Systems* **7:1** (March 1982).