# Query Optimization in the Presence of Foreign Functions

Surajit Chaudhuri
Hewlett-Packard Laboratories
Palo Alto, CA 94304
chaudhuri@hpl.hp.com

Kyuseok Shim*
University of Maryland
College Park, MD 20742
shim@cs.umd.edu

## Abstract

The declarativeness of relational query languages is very attractive for developing applications. However, many applications also need to invoke external functions or to access data that is not stored in the database. It is not hard to express references to such foreign functions in the query language. However, the issue of cost-based optimization of relational queries in the presence of such foreign functions has not previously been addressed satisfactorily. In this paper, we describe a comprehensive approach to this problem. Our key observation is that the optimization must take into account semantic information about foreign functions. Therefore, we provide a simple declarative rule language to express such semantics. We present algorithms necessary for applying the rules and for generating the space of equivalent queries. The equivalent queries provide the optimizer with an enriched execution space. We show how we can modify the traditional join reordering algorithm based on dynamic programming to obtain an optimal plan from the execution space. We provide necessary extensions to the cost model that are needed in the presence of foreign functions.

*Work done while the author was visiting Hewlett-Packard Laboratories.

**Proceedings of the 19th VLDB Conference
Dublin, Ireland, 1993**

## 1 Introduction

Relational database systems provide the ability to conveniently query the data stored in their databases. However, in many applications, there is a need to integrate data and operations that are *external* to the database (let us refer to them as *foreign functions*). For example, it will be convenient to invoke UNIX library functions as part of a relational query. Moreover, for many problem domains, highly tuned applications exist. The ability to exploit such existing applications is important since redevelopment can be prohibitively expensive. Furthermore, for many applications, only part of the data that is needed may be stored in the database and much of the data may reside externally. Access to such external data is provided by a set of interface routines. For example, many specialized Geographic Information Systems (GIS) are available today that provide the ability to store and access geographic data. On the other hand, information on attributes (e.g., population of a city) is usually stored in a relational database. Thus, for GIS as well as for other applications, the ability to invoke foreign functions in a relational query is very useful.

The ability to answer relational queries efficiently relies on the ability of the optimizer to choose from the repertory of evaluation options. Therefore, when we add the ability to invoke foreign functions, we also must provide necessary extensions to the optimizer to ensure efficient execution of queries. In this paper, we will address the optimization and related issues for relational queries that invoke foreign functions. There are other dimensions to the problem of supporting foreign functions (e.g., format conversion, complex objects), that we do not address in this paper.

### 1.1 Motivating Application

To illustrate the key challenges to optimization introduced by foreign functions, we briefly describe an application that allows us to access information about businesses and their locations in the Bay Area. This application has been built in the Papyrus project [CHK$^+$91] at HP Laboratories. The details of this application appear in [KNP92].

The application is built on top of ETAK[1] MapEngine and a relational storage manager. The MapEngine is a geographic data manager[2] that provides the ability to store and query maps.

The relational store is used to maintain attribute information about businesses. In particular, the relation *Business* contains information about the type of businesses, their telephone numbers as well as addresses. The MapEngine is used to store the locations of the business establishments in the Bay Area. The function *Map* retrieves all points[3] in the map that correspond to business establishments. Similarly, a function *Map_Restaurant* is used to access points in the map that correspond to all restaurants in the Bay Area. The boolean function *Inside* is used to test whether a given point is within a given rectangular window. The MapEngine also provides an additional function *Mapclip* that, given a window, returns all points, corresponding to business establishments in the map that are in that window.

For our application, we need to support queries that span the relational system as well as the MapEngine. An example of such a query is to be able to retrieve names of restaurants that are located within a certain window in the map. In order to be able to answer such queries, each tuple of *Business* has an attribute which acts as an index for accessing information in the MapEngine. Similarly, each record in the MapEngine points to the tuple in the table *Business* where the attribute information about the corresponding business establishment is maintained.

## 1.2 Challenges for Optimization

The above application brings about two fundamental requirements on optimization, as discussed below.

First, the **semantic information** associated with foreign functions needs to be captured and exploited for optimization. This is illustrated by the following example. Additional examples of semantic information that is useful, in the context of our application, appear in Section 3. In that section, we will show how such semantic information can be represented in a declarative fashion.

**Example 1.1:** Let us consider the query to find, given a window, all points in the map that are in the window. The query can be answered by invoking *Map* and testing, with the function *Inside*, that each of the retrieved locations is inside the window. The query can also be answered by invoking *Mapclip*. Since the use of *Mapclip* can greatly reduce the cost of evaluation of the query, the above semantic information is significant. ▮

Next, we observe that the decision to modify a given query using semantic information may need to be **cost-based**. In other words, whether application of the semantic knowledge reduces cost of evaluation depends on the cost parameters and thus blind application of semantic information may sacrifice optimality, as the following example illustrates. Therefore, we must have an algorithm for cost-based optimization in the presence of semantic information.

**Example 1.2:** Let us consider the query to find all restaurants in the downtown Palo Alto. This query can be answered by selecting all restaurants from the table *Business* and then invoking the function *Mapclip*. Alternatively, we can invoke the function *Map_Restaurant* and then select the restaurants in downtown Palo Alto by invoking *Inside*. These two queries are equivalent, but the optimal plan for one of the queries may be better, even by an order of magnitude, compared to the optimal plan for the other query, depending on whether the indexing effect of restricting locations to downtown Palo Alto is more effective than indexing based on restricting the businesses to be restaurants. ▮

Thus, in the presence of foreign functions, there may be multiple ways to answer the same query and such semantic information is extremely valuable for query optimization and must be captured. However, the application of such semantic information for query optimization needs to be cost-based.

## 1.3 Overview of our Approach

In this paper, we present an approach to optimization in the presence of foreign functions that takes into account the observations made above. We allow the semantic information to be specified in a *declarative* way (using a simple extension to SQL) by using rewrite rules with clean semantics. The rewrite rules are used to generate alternative equivalent queries. An optimal plan is picked by our optimizer in a cost-based fashion that considers all such queries.

[1]ETAK Inc. designs vehicle navigation systems and produces digital map databases.

[2]For our application, we also integrated another spatial data manager. For brevity, we will not distinguish between the two spatial data managers.

[3]We will represent a point or a window as a single argument; although a variety of representations is possible.

In order to realize our approach to optimization, several technical challenges need to be addressed. The declarativeness of the rules shifts the responsibility of determining applicability of the rules to the optimizer. Thus, our algorithm must ensure that application of a rule results in a query equivalent to the given query (Section 3). Next, we must generate the set of equivalent queries using the rewrite rules (Section 4). Finally, we need to consider the problem of choosing among the optimal plans of these equivalent queries. The last step requires modifications to the well-known join reordering algorithm (as in System R [SAC+79]) that uses dynamic programming approach (Section 5). The presence of foreign functions introduces other extensions to query processing as well as to the cost model (Sections 6 and 7).

The key element of our approach is the use of a declarative language for rewrite rules. We will comment on the pros and cons of such an approach in Section 8. Although our approach requires enhancements to the existing relational optimizers, it does not require an architectural redesign.

## 2 Queries, Foreign Functions

We will consider *conjunctive queries* for the purpose of this paper. Conjunctive queries correspond to the subset of SQL which has the following form. Observe that the the **WHERE** clause is a conjunction of conditions.

```
SELECT columnlist FROM Tablelist
WHERE  cond1 AND ... AND condk
```

We observe that every conjunctive query is a flattened select-project-join (SPJ) query. This subset of SQL is widely used.

A reference to a foreign function may occur as a condition, or as a table, or as a function in a SQL query.

**Example 2.1:** Let us consider a slightly modified version of the query that was informally stated in Example 1.1. Let us assume that we have a table **BUSINESS** that has five attributes: **NAME, TYPE, EARNING, SIZE** and **ETAKID**. The map on MapEngine is modeled as a foreign table **MAP** consisting of attributes **ETAKID** and **LOCATION**. The attribute **ETAKID** in both the tables refers to the key in the MapEngine. Recall from Example 1.1 that *Inside* acts as a condition that checks whether a point is within a window. Therefore, it can be represented as a condition in the **WHERE** clause of the query. Finally, we have a foreign function **EXPECTED-REVENUE** which takes the size of a restaurant as an input argument and estimates the average expected earning of a restaurant. The following query finds all restaurants that are in the map in the window **w** whose earnings are better than expected

```
SELECT BUSINESS.NAME, MAP.LOCATION
FROM BUSINESS, MAP
WHERE BUSINESS.TYPE =  'Restaurant'
AND BUSINESS.ETAKID = MAP.ETAKID
AND INSIDE(w, MAP.LOCATION)
AND BUSINESS.EARNING >
    EXPECTED-REVENUE(BUSINESS.SIZE)
```

▮

For notational convenience, we will represent conjunctive queries as *domain calculus* expressions as is done in nonrecursive Datalog [Ull88]. In domain-calculus, a conjunctive query is represented as a set of conjuncts (also called literals). Thus, references to either table, function or condition in the SQL statement will appear as a conjunct. The mapping to such a domain-calculus representation is straight-forward.

**Example 2.2:** The domain-calculus representation for the query in Example 2.1 is:

$$\begin{aligned}&Query(name, location) :-\\ &\quad Business(name, \texttt{Restaurant}, earn, size, eid),\\ &\quad Map(eid, location), Inside(\texttt{w}, location),\\ &\quad Exp\_Rev(size, exp), earn > exp\end{aligned}$$

The constants in the query are in typewriter fonts. ▮

In our notation, there are no explicit equality clauses. Instead, the equalities are implicitly represented as equality of variables in the expression. Like SQL, a query evaluates to a *bag of tuples* [4]. As illustrated in Example 2.2, a reference to foreign function in the domain-calculus representation appears as a conjunct. Therefore, we say that foreign functions are modeled as *foreign tables* (We will use the terms foreign functions and foreign tables interchangeably). Despite the fact that representations of a foreign table and a stored table appear syntactically similar, the distinction between the two will need to be drawn for query evaluation as well as for query optimization.

A foreign function may have *safety constraints*. Safety constraints are needed to ensure that during invocation, the foreign function is passed values to its "input" arguments. For example, before the conjunct *Inside* in Example 2.1 is evaluated, both its arguments need to be bound. Such safety constraints need to be specified when the foreign function is registered with the database.

[4]Note that the above is unlike the approach typically used in deductive databases, where a set semantics is associated with such a notation.

## 3 Rewrite Rules

The objective of the *rewrite rules* is to capture semantic information associated with foreign tables and their relationship to database tables. This information will be utilized to derive queries that are semantically *equivalent*[5] to the given query. Subsequently, a cost-based optimizer constructs an optimal plan for a query from the execution space of equivalent queries.

The language for expressing rewrite rules is declarative and requires simple syntactic extensions to SQL. Roughly speaking, a rewrite rule has the format:

```
REWRITE QUERY1  AS QUERY2
```

where QUERY1 and QUERY2 are relational queries such that the result relations have the same arity. Since in this paper, we are considering only conjunctive queries, we will adopt the following notation for rewrite rules.

$$L(\mathbf{x},\mathbf{y}) \rightarrow R(\mathbf{x},\mathbf{z})$$

The expressions $L(\mathbf{x},\mathbf{y})$ and $R(\mathbf{x},\mathbf{z})$ are conjunctive expressions and will be called the left-hand side and the right-hand side of the rule respectively. We note that $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ are ordered sets of variables. Any variable that occurs in either side of the rewrite rule is called an *universal variable* (e.g., any variable in $\mathbf{x}$).

### 3.1 Semantics

Our intent is to use rewrite rules to *derive* semantically *equivalent* queries. There are two aspects of semantics associated with a rewrite rule.

First, a rewrite rule $L(\mathbf{x},\mathbf{y}) \rightarrow R(\mathbf{x},\mathbf{z})$ asserts that over any database, the queries $Q_l$ and $Q_r$, as defined below, result in the same bag of tuples. In other words, $Q_l$ and $Q_r$ are equivalent (denoted $Q_l \equiv Q_r$).

$$\begin{aligned} Q_l(\mathbf{x}) &\ :-\ L(\mathbf{x},\mathbf{y}) \\ Q_r(\mathbf{x}) &\ :-\ R(\mathbf{x},\mathbf{z}) \end{aligned}$$

Observe that only the universal variables occur as projection variables of $Q_r$ and $Q_l$.

Next, a rewrite rule also specifies a rule for *derivation* of a new query. It says that an *occurrence* [6] of $L(\mathbf{x},\mathbf{y})$ in a query may be replaced by the subexpression $R(\mathbf{x},\mathbf{z})$ after appropriate renaming to derive a new query. The arrow in the rewrite rule is used to indicate that only an occurrence of the left-hand side of the rule should be substituted by the corresponding occurrence of the right-hand side (and *not* vice-versa) to generate equivalent queries. By the notation $Q \Rightarrow_r Q'$, we denote that $Q'$ is derived from $Q$ using the rewrite rule $r$.

The semantics of the rewrite rule, as discussed above, imposes directionality. Therefore, to express that either left-hand or right-hand side of the rule can be substituted by the other to derive a new query, we need two rewrite rules. For brevity, we will express that by a *bidirectional rule* using the notation $L(\mathbf{x},\mathbf{y}) \leftrightarrow R(\mathbf{x},\mathbf{z})$.

**Example 3.1:** Consider the rewrite rule in Example 1.1. We will represent that rewrite rule as

$$\begin{aligned} &Map(eid, loc), Inside(window, loc) \\ &\quad \rightarrow Mapclip(eid, loc, window) \end{aligned}$$

Note that the safety constraint requires the last argument of *Mapclip* to be bound. In this example, the variables *eid*, *loc* and *window* are all universal variables. The semantics imply that, over any database, queries $Q_l$ and $Q_r$ must result in same bag of tuples.

$$\begin{aligned} &Q_l(eid, loc, window) :- \\ &\quad Map(eid, loc), Inside(window, loc) \end{aligned}$$

$$Q_r(eid, loc, window) :- Mapclip(eid, loc, window)$$

We also observe that the left-hand side of the rewrite rule has an occurrence in the query $Q$, given below. Thus, using the above rewrite rule (say $r$), we derive the query $Q'$, i.e, $Q \Rightarrow_r Q'$.

$$\begin{aligned} &Q(name, loc) :- \\ &\quad Business(name, \mathtt{Restaurant}, earn, eid), \\ &\quad Map(eid, loc), Inside(w, loc), \\ &\quad Intersect(\mathtt{w1}, \mathtt{w2}, w) \end{aligned}$$

$$\begin{aligned} &Q'(name, loc) :- \\ &\quad Business(name, \mathtt{Restaurant}, earn, eid), \\ &\quad Mapclip(eid, loc, w), Intersect(\mathtt{w1}, \mathtt{w2}, w) \end{aligned}$$

∎

**Example 3.2:** The following rule is used informally in Example 1.2. It says that in order to obtain locations of restaurants, we can either take a join between *Business* and *Map* or can invoke *Map_Restaurant*:

$$\begin{aligned} &Business(name, \mathtt{Restaurant}, earn, size, eid), \\ &\quad Map(eid, loc) \leftrightarrow Map\_Restaurant(eid, loc) \end{aligned}$$

In this example, *eid* and *loc* are universal variables. ∎

[5] We say that two queries are equivalent if they result in the same bag of tuples over any database.

[6] Identical to an expression upto renaming of variables or substituting a constant for a variable.

**Example 3.3:** The following rule for MapEngine says that instead of checking whether a point belongs to each of the two given windows, we can check whether the point belongs to the intersection of windows.

$$Inside(w1, point), Inside(w2, point) \rightarrow Inside(w, point), Intersect(w1, w2, w)$$

Using this rule, the problem of finding all businesses in multiple windows can be reduced to the problem of finding all businesses in the intersection of the windows. ▮

**Example 3.4:** Assume there is an index on *Map* for a given *eid*. We can refer to the access function for the indexed scan by *Mapwithid(eid, loc)*. In order to ensure that the optimizer takes the indexed scan as a possibility, we specify the following rewrite rule:

$$Map(eid, loc) \leftrightarrow Mapwithid(eid, loc)$$

The safety constraint on *Mapwithid* requires *eid* to be bound. ▮

## 3.2 Sound Application of a Rewrite Rule

Let us assume that $Q \Rightarrow_r Q'$. Since our objective is to use the rewrite rules for query optimization, we are interested in $Q'$ only if it is equivalent to $Q$. The following example shows that not all derived queries are necessarily equivalent to the given query.

**Example 3.5:** We observe that the query $Q$ has an occurrence of the left-hand side of the rewrite rule in Example 3.2.

$Q(loc)$ : –
  $Business(bizname, \textbf{Restaurant}, earn, size, eid)$,
  $Map(eid, loc), Owner(bizname, \texttt{bob})$

However, replacing the occurrence with the right-hand side of the rewrite rule results in query $Q'$, which is not semantically equivalent to $Q$.

$Q'(loc)$ : –
  $Map\_Restaurant(eid, loc), Owner(bizname, \texttt{bob})$

▮

In the above example, the crux of the problem is that the semantics of rewrite rules guarantees that the left-hand and right-hand sides of the rewrite rules are equivalent over universal variables ***only***. Thus, in order to ensure that the queries that we derive are semantically equivalent, we must ensure that only appropriate occurrences of the left-hand side of the rewrite rule in the query are replaced. It turns out, for a rewrite rule $L(\mathbf{x}, \mathbf{y}) \rightarrow R(\mathbf{x}, \mathbf{z})$ to have an appropriate occurrence in a query $Q$, the latter must have the form:

$$Q(\mathbf{u}) : -L(\mathbf{v}, \mathbf{w}), G(\mathbf{t})$$

where the set of variables $\mathbf{w}$ is disjoint from the set $\mathbf{u}$ as well as the set $\mathbf{t}$. Note that $G(\mathbf{t})$ represents the conjunction of the rest of the literals in the query and $\mathbf{t}$ represents the set of all variables that occur in $G$. If the query has the above form, then we can derive an equivalent query $Q'$.

$$Q'(\mathbf{u}) : -R(\mathbf{v}, \mathbf{s}), G(\mathbf{t})$$

The following definition of ***sound occurrence*** captures appropriate occurrences. The definition is simplified for presentation and assumes that there are no inequality conditions in the query. A complete description appears in [CS93].

**Definition 3.6:** Let $l \rightarrow r$ be a rewrite rule. An occurrence $l'$ of $l$ in $Q$ is a ***sound occurrence*** if the variable renaming is such that (a) only variables in $l$ that are mapped to constants in $l'$ are universal variables (b) The literals in $l'$ share with the rest of the literals in $Q$ only those variables that correspond to universal variables of $l$. ▮

**Example 3.7:** It can be seen that the occurrence in Example 3.5 is not a sound occurrence. We observe that the variable ***bizname*** is shared with the literal *Owner* which is not in the occurrence of the rule. However, ***bizname*** is ***not*** a renaming of a universal variable. Let us now consider a variant of the query in Example 3.5 where the literal *Owner* is replaced by *Historic(loc)* in $Q$. In this case, there is a sound occurrence. ▮

Using the definition of sound occurrence, we now present an algorithm for deriving a semantically equivalent query using rewrite rules. A ***sound application of a rewrite rule*** consists of two steps:

1. Identify a subexpression such that there is a sound occurrence of the left-hand side of the rule in the query.

2. Substitute the subexpression with the right-hand side of the rule (after renaming).

The following theorem, shown in [CS93], is a key property of sound application.

**Lemma 3.8:** ***Let $Q \Rightarrow_r Q'$. Then, $Q' \equiv Q$ iff $Q'$ is obtained by a sound application of $r$ to $Q$.***

We outline an algorithm *rewrite*$(r, Q)$ to generate all equivalent queries obtained by sound applications of the rewrite rule $r$ to a given query. First, we enumerate all possible occurrences of the left-hand side of the rewrite rule to the query. Next, for each occurrence, we test whether the occurrence is sound. If so, we generate the corresponding equivalent query (See Figure 1).

In the worst case, *rewrite*$(r, Q)$ is exponential in the size of the query. However, for queries with no repeated table names, there is a unique sound application of a rule and the algorithm for sound application takes time linear in the combined size of the query and the rewrite rule. Moreover, the size of the query is typically limited to a few literals. In practice, our algorithm for enumerating sound applications performs satisfactorily.

## 4 Optimization with Rewrite Rules

The traditional optimization problem is to choose an optimal plan for a query. However, sound applications of rewrite rules generate alternatives to a query that are semantically equivalent. Therefore, the optimization problem becomes that of picking the cheapest among the optimal plans of the set of equivalent queries in a cost-based fashion. Thus, our optimization algorithm consists of the following two steps.

1. Generate the set of equivalent queries.

2. Choose the cheapest among the optimal plans for each query obtained from Step 1.

The Step 2 of the algorithm is accomplished by an algorithm which extends the System R style of join enumeration using a dynamic programming approach. This algorithm will be described in Section 5. We now describe Step 1 below.

### 4.1 Generating Equivalent Queries

The set of equivalent queries that are obtained from a given query by sound applications of rewrite rules will be referred to as the *closure* of the query, defined below. We will present an algorithm to compute the closure.

**Definition 4.1:** The *closure* of a query $Q$ with respect to a set $R$ of rewrite rules is the set of queries:

$$closure(R, Q) = \{Q' | Q \overset{*}{\Rightarrow}_R Q'\} \blacksquare$$

The symbol $Q \overset{*}{\Rightarrow}_R Q'$ is used to denote the fact that $Q'$ has been obtained from $Q$ by a finite sequence of sound applications of the set $R$ of rewrite rules.

**Example 4.2:** Consider the following query.

$$Q(loc) : -Map(eid, loc), Inside(\mathtt{w1}, loc), Inside(\mathtt{w2}, loc)$$

We can apply the rule in Example 3.3 to generate the query:

$$Q'(loc) : -Map(eid, loc),$$
$$Inside(w, loc), Intersect(\mathtt{w1}, \mathtt{w2}, w)$$

Finally, an application of the rewrite rule in Example 3.1 results in a query $Q''$.

$$Q''(loc) : -Mapclip(eid, loc, w), Intersect(\mathtt{w1}, \mathtt{w2}, w)$$

In this example, we have observed a sequence of rule applications. ▮

The algorithm to compute the closure is given in Figure 1.

The algorithm *gen_closure* is iterative. During each iteration, derived queries that were not obtained before, act as the seeds to generate additional queries in the current iteration. To generate queries, the algorithm repeatedly invokes the function *rewrite*. The control-structure of this algorithm is similar to *semi-naive* algorithm that is used in deductive databases [Ban86].

**Lemma 4.3:** *The function gen_closure*$(R, Q)$ *computes closure*$(R, Q)$.

Assuming that each query has a bounded length, the complexity of *gen_closure* is polynomial in the size of $R$ and *closure*$(R, Q)$. In order to access the relevant rules efficiently, we maintain the rules in a rule-table which is indexed on the conjuncts that appear on the left-hand side of the rule. A detailed discussion of implementation is beyond the scope of this paper.

Observe that the termination of *gen_closure* depends on whether the closure of the query with respect to a set of rewrite rules is finite or not. For the applications that we have considered, we found that the closure of a query is typically limited to a few queries only. Therefore, neither termination nor the size of the closure posed any problems. Nonetheless, we have algorithms that test sufficient conditions for finiteness of closure [CS93]. These algorithms take time linear in the size of the set of rules and the query. In case the closure is not provably finite, or if we desire to restrict the set of equivalent queries that are generated, we can do partial enumeration of closure.

```
Function gen_closure(R, Q)
    begin
        S = Q;
        δ = Q;
        repeat
            new = ∅
            for each q in δ and r in R do
                new = new ∪ rewrite(r, q);
            endfor
            if new ⊆ S then return(S);
            δ = new − S;
            S = S ∪ δ;
        forever
    end
```

```
Function rewrite(r, Q)
    begin
        Q_r = ∅
        for every sound occurrence A
          of r in Q do
            Q_r = Q_r ∪ {A_Q}
            where A_Q is the derived query
            due to occurrence A
        endfor
        return(Q_r)
    end
```

Figure 1: Algorithm to Compute Closure

**Partial Enumeration of Closure:** For selective enumeration of closure, we can use a *budget* to specify an upper bound on the maximum time spent on enumeration. Another alternative is to bound the size of any query that is used as a seed to generate other queries. We can also modify the rewrite rules to achieve partial enumeration. This is illustrated by the following example.

**Example 4.4:** The following rewrite rule expresses the knowledge that all restaurants in the map are in a window **w**. Therefore, if we are asked to find restaurants that are in any window, we can intersect the given window with **w** before we search.

$$\begin{aligned}&Business(name, \texttt{Restaurant}, earn, size, eid),\\ &\quad Mapclip(eid, loc, window) \rightarrow\\ &\quad Business(name, \texttt{Restaurant}, earn, size, eid),\\ &\quad Mapclip(eid, loc, small\_win),\\ &\quad Intersect(window, \mathbf{w}, small\_win)\end{aligned}$$

Imagine a query which consists of the conjuncts in the left hand side of the rewrite rule. It is easy to see that the closure for this query is infinite.

However, we can represent the rule in Example 4.4 as follows.

$$\begin{aligned}&Business(name, \texttt{Restaurant}, earn, size, eid),\\ &\quad Mapclip(eid, loc, window) \rightarrow\\ &\quad Business(name, \texttt{Restaurant}, earn, size, eid),\\ &\quad SpecialMapclip(eid, loc, window)\end{aligned}$$

Our modified rewrite rule contains a new table name *SpecialMapclip*. Unlike the original rule, the new rule can not be used repeatedly since there is no rewrite rule where *SpecialMapclip* occurs in the left-hand side. Thus, the closure is finite. After the closure is generated, we substitute the expression for *SpecialMapclip(eid, loc, window)*. In effect, we have avoided generating the entire closure. ∎

In this example, we have shown how we can treat a subexpression as a single literal and thereby restrict the enumeration of closure. The choice of the subexpression determines the subset of closure that is selected for enumeration. It can be shown that by using this strategy, *any* set of rewrite rules can be rewritten to ensure that the effective closure is finite.

**Pruning the set of Equivalent Queries:** The set of equivalent queries that are generated by *gen_closure* are considered by the cost-based optimizer to pick the optimal plan. Since optimization of queries is expensive, it is appropriate that we eliminate queries that are not promising, i.e., not likely to yield an optimal plan. It is possible to designate certain rewrite rules as *always-improving*. Thus, if $r$ is always-improving and if $Q \Rightarrow_r Q'$, then we do not optimize $Q$ since it is assumed that $Q'$ will always result in a better optimal plan. For example, the rewrite rule in Example 3.3 may be marked as always-improving. An interesting approach will be to use crude cost measures to approximate the cost of query evaluation to weed out queries that are not promising.

## 5 Choosing an Optimal Plan

The presence of rewrite rules and foreign tables introduces new dimensions to the traditional optimization problem. First, the presence of foreign tables requires introduction of new join methods as well as cost models that are appropriate for foreign tables. Second, the traditional join enumeration phase must ensure that only those reordering of the joins are considered which satisfy the safety constraints. In other words, we need to ensure that the bindings that are passed to the foreign functions satisfy the safety constraints. Finally, our task is to choose an *optimal* plan when there are multiple queries which are equivalent.

In this section, we address the last of the above three twists to the traditional optimization problem. We will address the issue of extensions to the cost model in Section 7. For this section, we will assume that the cost model can assign a real number to any given plan in the execution space (defined below) and satisfies the *principle of optimality* [GHK92, CLR90], which is implicit in relational optimizers that use dynamic programming. We will omit any discussion on the problem of ensuring safety since this is a rather well-studied problem (See [Ull88]).

The optimization problem is to choose a plan of least cost from the execution space. The execution of a query can be represented syntactically as *annotated join trees* [GHK92] where the the internal node is a join operation and each leaf node is a database table[7]. Thus, the *execution space* consists of the space of all join trees[8] for each equivalent query obtained from Step 1 of optimization (Section 4).

Since the execution space is the union of the execution spaces of the equivalent queries, we can obtain the following simple extension to the optimization algorithm:

1. Optimize each query using the traditional algorithm and obtain the best plan for the query.

2. Choose the cheapest among the best plans obtained in Step 1.

For Step 1, we can use any traditional relational optimizer [SAC+79]. The space requirement for this algorithm is the maximum space required for optimization of any of the equivalent queries. However, the algorithm has a poor time complexity since it fails to take advantage of the *common subexpressions* among equivalent queries to reduce the optimization time.

## 5.1 Algorithm that Reuses Optimal Plans

Our algorithm uses dynamic programming and extends the well-known join enumeration algorithm in System R [SAC+79]. In that algorithm, the commonality among the subqueries [9] of a *single* query to reduce the time complexity of the optimization. In our case, we exploit the commonality among subqueries across *multiple* equivalent queries during optimization. Thus, the key idea is to save and share the optimal plan for the common subqueries.

The problem of identifying common subqueries between two *arbitrary* conjunctive queries is computationally hard. However, we detect commonality that arises due to application of rules. Thus, if we apply the rule $l \rightarrow r$ to obtain a new query $Q'$ from $Q$, then we know that $Q'$ and $Q$ must share the subquery consisting of the literals $(Q - l)$. Such shared subqueries can be detected without any overhead during application of the rules. The following example illustrates this technique. The optimal plan for such common subqueries are shared.

**Example 5.1:** Consider an application of the rewrite rule in Example 3.2 to query $Q$ that results in the query $Q'$. The MapEngine file *Historic* contains the location of all historic sites and the foreign table *Price(loc, amount)* provides the value of the real estate.

$$\begin{aligned} Q(amount) :- & Historic(loc), \\ & Business(bizname, \textbf{Restaurant}, earn, eid), \\ & Map(eid, loc), Price(loc, amount) \end{aligned}$$

$$\begin{aligned} Q'(amount) :- & Historic(loc), \\ & Map\_Restaurant(eid, loc), Price(loc, amount) \end{aligned}$$

Observe that both the queries have the same first and last literals in the body. This commonality is detected at the time the rewrite rule is applied by observing which literals are left unchanged by the application of the rule. Thus, the optimal plan for the common subquery *Historic(loc)*, *Price(loc, amount)*, is used while optimizing the query $Q$ as well as $Q'$. ∎

It is well-known that dynamic programming based algorithms can be presented either as top-down or bottom-up (see [CLR90]). A top-down dynamic programming based algorithm is presented in Figure 2.

We optimize each query one at a time. The optimal plans for all shared subqueries are stored in a *plan table* and are *never* rederived. Hashing is used to look up the plan table. Thus, whenever, a plan needs to be constructed, we consult the plan table to check whether the plan already exists. We have omitted the base cases in

[7] In our case, a leaf node can be a foreign table as well.

[8] The implementation of our optimizer considers only the *left-deep* join trees, i.e., join trees where the right child of every join node is a base table (leaf).

[9] We use the term *subquery* interchangeably with the term *subexpression of a query*.

```
Procedure OptPlan(Q) :
begin
    if $existsoptimal(Q)$ then return;
    Let $Q = (q_1, \ldots, q_n)$;
    Let $S_i = Q - \{q_i\}$;
    for each $i$ do
        OptPlan($S_i$);
        $P_i$ = Plan for $Q$ from $S_i$ and $q_i$
    endfor;
    Choose best among $P_i$
    and add to plan table.
end
```

Figure 2: Join Enumeration Algorithm

*Optplan* where the query has at most two literals (i.e., a single join). The above cases as well as the generation of $P_i$ from $S_i$ and $q_i$ (See Figure), are handled by a *local optimizer* which is invoked by this join-enumeration algorithm. The local optimizer uses information about the cost-model. As in traditional optimizers, our optimizer treats the built-in boolean conditions (*sargable predicates*) specially.

**Example 5.2:** Consider Example 5.1. Let us assume that the query $Q$ is represented by the string (1234). However, once the rewrite rule is applied, a new literal *Map_Restaurant(eid, loc)* is created and the representation for $Q'$ will be (145). The optimization of the query $Q$ will create the optimal plan for (14) which is then stored in the plan table. During the optimization of the query $Q'$, first the plan table is consulted to see whether a plan for (145) already exists. Since it does not exist, we must construct the optimal plans for each subquery. In particular, before constructing the optimal plan for (14), the plan table is consulted and the existing optimal plan for (14) is reused for optimization. ∎

The algorithm *Optplan* has the desirable feature that for *no* shared subquery, the optimal plan is rederived. Moreover, only plans for *shared* subqueries are retained in the plan table.

## 5.2 Discussion

It is possible that for a given foreign table, there may be different implementations which differ on the safety constraints. The join enumeration method invokes the implementation whose safety constraints are satisfied.

We note that when there is a budget on optimization time, our strategy of optimizing one query at a time is convenient because it is possible to terminate the optimization once the optimal plan for one of the equivalent queries has been constructed.

Since potentially we are optimizing many queries, we use a ***branch and bound*** strategy along with the top-down algorithm. Thus, if a partial plan is found to have exceeded the cost of the optimal plan that has been found so far, then that partial plan need not be completed since it is guaranteed to be suboptimal.

We are exploring the opportunities for ***heuristics*** in guiding the search. For example, heuristics may be used to determine the order in which queries are optimized.

**Bottom-up Algorithms:** It is also possible to use bottom-up variants of our algorithms. There can be at least two possible variants in a bottom-up approach. One possibility is to optimize all the equivalent queries together. Thus, optimal plans for all subqueries of size $n$ are constructed before any optimal plan for *any* subquery of size $(n+1)$ is constructed. This approach has the advantage that it requires less space than the top-down approach (by reusing the space). On the other hand, since the subqueries for all equivalent queries are constructed together, the time for the completion of the optimal plan for the first query is longer than that for the top-down approach. Another variant of the bottom-up algorithm is where optimization is done one query at a time but the optimal plans of shared subqueries are saved. While this rectifies the shortcoming of the previous approach, it suffers from the problem of not being able to share the plan for the maximal shared subquery.

**Inexpensive Tables:** In a traditional relational optimizer, the selection conditions are not reordered during join-enumeration. Rather, the selection conditions are evaluated as early as possible. Since the cost of reordering joins is exponential in the number of literals being reordered, this helped save optimization time.

The invocations of some foreign tables may also be inexpensive. For example, the foreign table *Inside(w, loc)* checks whether a point *loc* is inside the window $w$. An invocation of *Inside* is inexpensive and *Inside* may be considered like a selection condition in a relational query. Thus, we allow foreign tables to be designated as ***inexpensive tables***. In a query, the literals that correspond to inexpensive tables are not reordered but are evaluated as early as possible in the join-order without violating the safety constraints.

We call the rest of the literals as *reorderable*, which are then considered for join enumeration. Thus, given

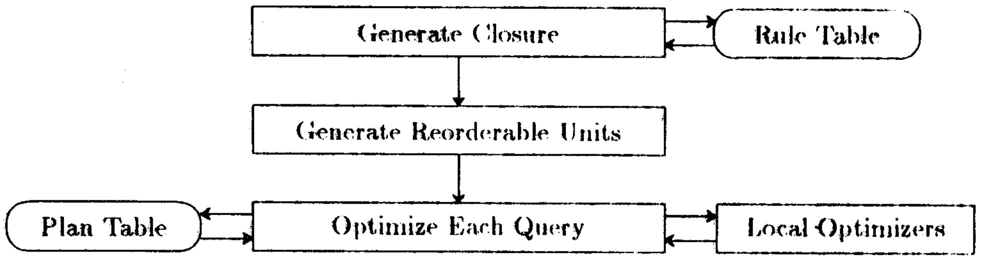

Figure 3: Overview Of Optimization

a query, the optimizer needs to identify reorderable literals and place the inexpensive tables for evaluation as early as possible in the join order. Thus, the presence of inexpensive tables in a query introduces the step to generate *reorderable units* [CS93].

**Affiliated Tables:** In addition to being inexpensive to evaluate, the selection conditions in relational queries play an additional role. Their presence influences the cost of scan. Thus, in System R [SAC+79] architecture, selection conditions are pushed down to the RSS layer. In general, the cost of evaluation of a foreign table may be influenced by the presence of a set of other foreign tables. While rewrite rules may be used to capture such dependencies among foreign tables, we provide another alternative in our optimizer.

At the time of registering a foreign table one could specify a set of *affiliated tables*. Each affiliated table must be a condition, i.e., all its arguments need to be bound (e.g, *Inside*). The cost of invoking a foreign table is influenced by the presence of affiliated tables in the query. During the join enumeration, the optimizer considers the foreign table and its affiliated tables together. The cost model provides the join enumerator the cost of invoking the foreign table in the presence of affiliated tables. The details of affiliated tables appear in [CS93].

**Example 5.3:** Let us consider a hidden *implementation* for *Mapclip* that takes an arbitrarily set of windows and returns all points that are in each of the windows. We designate such an implementation as $MapclipService(eid, loc, W)$ where $W$ is a set of windows. Thus, the cost of evaluating *Mapclip* depends on the presence of *Inside* in the query. Therefore, we designate *Inside* as an affiliated table for *Mapclip*. Then, given the query $Q$, the query optimizer considers it as a single invocation of *Mapclip* with two affiliated tables.

$$\begin{aligned} Q(eid) \quad :- \quad & Mapclip(eid, loc, \mathtt{w1}), Inside(loc, \mathtt{w2}), \\ & Inside(loc, \mathtt{w3}) \end{aligned}$$

The cost model is responsible to return the cost of invocation in the presence of affiliated tables. In effect, this query will be compiled as the invocation of $MapclipService(eid, loc, \{\mathtt{w1}, \mathtt{w2}, \mathtt{w3}\})$. ∎

The summary of the steps in the optimization process is shown in Figure 3.

## 6 Query Processing for Foreign Functions

The objective of this section is to introduce the query processing techniques for foreign functions that our optimizer considers. This discussion is relevant for the cost model, presented in the next section. We consider the join operation and make the simplifying assumption that the foreign table occurs as the right child of a join node in a left-deep tree. Therefore, in a left-deep join tree, the table with which the foreign table joins is referred to as the *left table*.

Any access to a foreign table must respect its safety constraints. Therefore, before a tuple can be obtained from a foreign table, it must be passed the bindings that are required due to safety constraints and for each such binding, an invocation is made for the foreign table. Such a technique is inefficient for invoking a foreign table which has a high cost of invocation and for each invocation returns many tuples as output.

In our approach, query processing for foreign functions consists of viewing the "join" with a foreign table to have two phases: *invocation* and *residual join*. The *invocation* phase consists of passing the values for the bound arguments of the foreign table from the left table. There are several ways in which this invocation may occur:

- *Simple Invocation:* For each tuple in the left table, an invocation is made.
- *Group Invocation:* In this scheme, for each *distinct values* of the bound arguments from the left table, a single invocation is made.

The group invocation technique adds the overhead of identifying the set of distinct values for the bound arguments. However, it has the advantage of fewer invocations, which is important for foreign tables for which each invocation is expensive Moreover, if the left table is already sorted on the bound arguments prior to join, then group invocation is superior. The section on cost model would capture the tradeoff in the two approaches.

Since an invocation generates a set of tuples, the step of *residual join* is similar to a traditional join and any join method may be used. The selection conditions that apply to one or more *free* (output) arguments of the foreign table, are evaluated during this phase. The simplest choice for the residual join is nested loop where the tuples generated for each invocation are treated as the matching tuples of the inner table. This residual join method can be combined with the two techniques for invocation.

The combination of simple invocation and the choice of nested loop join technique results in a join algorithm which is similar to the traditional nested loop join. We call this join technique *foreign nested loop join* (FNL). The combination of group invocation and the nested loop join results in an algorithm very similar to the sort merge join and we refer to it as *foreign sort-merge join* (FSM). An outline of the FSM algorithm is presented in Figure 4. The FSM algorithm is the algorithm of choice when the invocation of foreign tables is expensive.

In order to reduce the number of invocations, caching the results of invocation was suggested in Postgres and such an alternative can be used with our approach as well. The correctness of caching (or group invocation) depends on the assumption that foreign tables are invariant during query processing. Such an assumption is not always true (e.g., if the foreign table is a random number generator).

## 7 Extensions to the Cost Model

The cost model must be able to compute the cost of any given plan. For traditional relational optimizers, a descriptor for a table includes statistical information about the table such as the number of unique values in each argument position (i.e., in each column) and the expected number of tuples in the table. The cost model uses the descriptors to compute the cost of an operation (e.g, a join). The cost model also produces a new descriptor which contains the statistical information of the intermediate table which is obtained after the join.

Our approach to the cost model is to preserve the relational descriptor for the database tables and intermediate tables. However, two extensions are needed. First, we need to provide a descriptor for foreign tables. Next, we have to explain, how such a descriptor can be combined with a relational descriptor.

```
Function $FSM(Left, FTable)$
(Left is left table, FTable is a Foreign Table)
  begin
    $Join = \emptyset$
    $Temp\_Left = GROUPBY(Left, Bound)$
    where sorting and Grouping is by
    the bound arguments of FTable
    for every group $L_i$ of $Temp\_Left$ do
        $FT_i = Invoke(FTable, Bval_i)$
        where $Bval_i$ are the values in group $L_i$
        for bound arguments of $Bound$
        $Join = Bag\_Union(Join, Merge(L_i, FT_i))$
    endfor
    return($Join$)
  end
```

Figure 4: Foreign Sort Merge Join Algorithm

### 7.1 Descriptor for Foreign Tables

For each foreign table, the following information can be registered. A full description of the registration language appears in [CS93]. This cost model is an extension of the model proposed in [CGK89].

- *Safety Constraints:* This information is not directly used by the cost model, but is used by the optimizer to determine permissible join-orders.
- *Cost:* The cost of invoking the foreign table once.
- *Fanout:* The number of "output tuples" expected for each invocation.
- For each attribute:
    - *Domain Size:* We need to provide the *size* of the representation of each domain element. We also need to specify the cardinality of the domain. A permissible assignment to cardinality is `infinite`.
    - *Unique Value Factor:* The expected number of unique values the attribute has for each invocation. If this parameter is not explicitly provided, the fanout is used to approximate this factor. If all the domains are finite, uniform distribution assumption is used to compute this factor.

Observe that the parameters in the descriptor need not necessarily be constants, but can depend as well on any constants that appear in the query during compilation [CS93].

**Example 7.1:** A possible descriptor for the foreign table $Intersect(window1, window2, window3)$ could be characterized by a cost of .012ms, a fanout of 1, unique value factor of 1. The size of each domain element is that corresponding to a real and the domain has cardinality `infinite`. The fanout is 1 since intersection of two windows result in one window. The safety constraint on the function is that the first two argument positions must be bound before it is invoked. ▮

## 7.2 Computing the Descriptor

In this section, we address the extensions that are needed to compute a descriptor. For simplicity, we only consider the scenario where the foreign table occurs as a right leaf node of left deep join trees. We can assume the existence of a descriptor for the *left* table with which the foreign table joins. In our optimizer, one can register a *customized function* to compute the descriptor for the table resulting after the join. Such a function can take as its argument the descriptor for the left table. In the rest of this section, we provide a default way to compute the descriptor for the intermediate table.

We introduce the *left uniqueness factor* as a cost parameter. The left uniqueness factor estimates the expected number of *distinct* invocations of the foreign table for a given descriptor for the left table. We have considered several ways to approximate the left uniqueness factor. In this paper, we present the simplest approximation.

For the foreign table, some argument positions may be required to be bound. Therefore, there exists a corresponding set of attributes $A$ in the left table which provide the values for the bound arguments of the foreign table. Let $P$ be the product of the expected number of unique values for the set of attributes $A$ in the left table. We use the descriptor of the left table to compute $P$. Let $N$ be the number of tuples in the left table. We observe that the number of distinct invocations can exceed neither $P$ nor $N$. Therefore, we can use $min(P, N)$ to estimate the left uniqueness factor. Our formula provides an upper bound of the left uniqueness factor.

**Example 7.2:** Consider the following query which provides the location of the terminals for the bus routes. Assume that the descriptor for *Terminal* has 100 tuples and the number of expected unique values in the second argument is 10.

$$Query(route, loc) :- \\ Terminal(route, cid), Map(cid, loc)$$

Therefore, $P = 10$ and $N = 100$. Hence, the left uniqueness factor is 10. ▮

In the following discussion, we assume that there are no selection conditions other than equality between the left table and the bound arguments of the foreign table. The effect of selection conditions on free arguments as well as the effect of projection on foreign tables on the descriptor are taken into account by treating the result of the join of left table with the foreign table as an intermediate table (like any interior node of the join tree). Therefore, we provide the cost formulas for the invocation phase only.

- *Number of Tuples:* The estimated number of tuples after the join is $N' = F * N$, where $F$ is the fanout of the foreign table and $N$ is the number of tuples in the left table.
- *Number of Unique Values:* The estimated number of unique values corresponding to the $i$th argument of the foreign table is given by: $UVF_i * UI$ where $UVF_i$ is the unique value factor for the $i$th attribute. The parameter $UI$ is the left uniqueness factor.
- *Cost:* We will provide the cost of foreign nested loop and foreign sort-merge join. We assume that $N$ is the number of tuples in the left table, $C$ is the cost of invoking the foreign table and $UI$ is the uniqueness factor. The following costs are for the invocation phase only.
  - *Foreign Nested Loop:* $C * N$.
  - *Foreign Sort-Merge:* $Cost_{sort}(N) + UI * C$

## 8 A Critique of Declarative Rewrite Rules

The declarative nature of our rewrite rules provides ease of specification of semantic knowledge. For example, in the MapEngine application, the semantic knowledge was captured using a few rewrite rules and the optimization algorithm ensured that the rules were exploited to produce an optimal plan. Nonetheless, the trade-off between such a declarative language and a procedural language is that between the ease of specification with the need for expressivity and possibly efficiency concerns. For example, as discussed in Section 5, we sometimes find it convenient to use the notion of *affiliated tables* (instead of rewrite rules) in order to express the knowledge that a set of conditions need to be "pushed-down". In contrast to our approach, Starburst [PHH92] uses a procedural language to express the semantic knowledge. While making specification of semantic knowledge harder, such an approach enables rewrite rules to express any desirable transformation and leaves the design of search algorithms open-ended. We should note that the ability to invoke and optimize foreign functions is a limited form of extensibility. In contrast, the Starburst architecture has far more ambitious goal of providing extensibility.

## 9 Related Work

Many extensible systems have been proposed [CH90] with varying degrees of support for extensibility in the optimizer [BG92, GD87, Loh88, PHH92, SJGP90]. The query rewrite optimization [PHH92] in Starburst is most directly related to our approach. As discussed in the preceding section, the query rewrite optimization in Starburst relies on a procedural language. The rule programmer is responsible for ensuring termination and search algorithms. A key reason for such a design decision is to reserve the ability to express rules of arbitrary complexity. For example, the rewrite rule language in Starburst is used not only to express semantic knowledge, but also to express the rules for query transformation used in optimization (e.g., flattening a query). In contrast, we focused on only extending the optimizer to handle queries with foreign functions. Our narrow focus enabled us to use a rewrite language that is declarative. The termination and the search algorithm to generate equivalent queries are part of the optimizer and need not be specified by the rule programmer. Furthermore, in Starburst, an application of a rewrite rule is used as a heuristic. In contrast, we use rewrite rules to generate alternatives for the optimizer, from which the latter chooses an *optimal* plan in a cost-based fashion.

The idea of using semantic knowledge to transform a query into one which yields a cheaper optimal plan has been examined in the context of semantic query optimization (See [CGM90]). Unlike our approach, they use a conventional query optimizer to optimize equivalent queries and thus do not share optimization of common subexpressions. Furthermore, our algorithm for generating equivalent queries is based on conjunctive query equivalence, instead of resolution based techniques [CGM90] and preserves the duplicate semantics of SQL.

Query optimization in the presence of foreign function was examined in [CGK89, HS93]. Neither of these approaches provides any opportunity to use semantic knowledge. The contribution of [CGK89] is to present a cost model for optimization in the presence of foreign functions such that a traditional dynamic programming algorithm can be used. We have further extended their cost model. Recently, [HS93] presents an optimization algorithm for a restricted class of foreign functions. In his work, foreign functions are restricted to be conditions (boolean predicates). Thus, he does not consider the foreign functions that generate data tuples (e.g., *Mapclip*).

Aref and Samet [AS91] present a variety of strategies for choosing a plan in scenarios where retrieval requires accessing a relational database as well as a spatial data repository. Using rewrite rules, we can express the different alternatives that they consider.

## 10 Conclusion

The ability to invoke foreign functions in a relational query is important for many applications since it provides them the opportunity to exploit existing code and data that is external to the database. Such integration raises several issues. In particular, it provides us with new challenges in optimization.

In this paper, we have described a comprehensive approach for optimization in the presence of foreign functions. An optimizer, based on our approach, has been implemented at HP Laboratories. We provide a declarative rewrite rule system which can be used to express semantics of foreign functions. The rewrite rules are specified using simple extensions to SQL. The rewrite rules are used to present the optimizer with a set of equivalent queries. We have provided an algorithm to enumerate the equivalent queries. Our optimization algorithm is able to guarantee optimality of the plan over the enriched space of optimization. We have developed an extension to the traditional dynamic programming algorithm that exploits commonality among the equivalent queries. Our framework includes extensions to the cost model and query processing techniques that are necessary for foreign functions. For efficiency in optimization, we provide the ability to specify that certain tables are *inexpensive* or *affiliated*. Finally, we can use our framework to optimize relational queries where the database stores materialized views [CS93]. Intuitively, materialized views provide the optimizer with semantically equivalent queries to choose from.

**Acknowledgement**

We are indebted to Umesh Dayal, Waqar Hasan and Marie-Anne Neimat for their insightful comments and for many useful discussions with them. Thanks are also due to Curt Kolovson, Ravi Krishnamurthy, Spyros Potamianos and Timos Sellis. We are grateful to Hamid Pirahesh and anonymous reviewers for comments that helped us improve the draft.

## References

[AS91] W. G. Aref and Hanan Samet. Optimization strategies for spatial query processing. In *Proceedings of the 17th International VLDB Conference*, pages 81–90, Barcelona, Spain, September 1991.

[Ban86] F. Bancilhon. Naive evaluation of recursively defined functions. In M.L. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems*, pages 165–178. Springer-Verlag, New York, 1986.

[BG92] L. Becker and R. H. Güting. Rule-based optimization and query processing in an extensible geom tric database system. *ACM Transactions on Database Systems*, pages 247–303, June 1992.

[CGK89] D. Chimenti, R. Gamboa, and R. Krishnamurthy. Towards an open architecture for LDL. In *Proceedings of the 15th International VLDB Conference*, pages 195–203, Amsterdam, The Netherlands, August 1989.

[CGM90] U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, pages 162–207, June 1990.

[CH90] M. Carey and L. Haas. Extensible database management systems. *ACM SIGMOD Record*, Dec 1990.

[CHK+91] T. Connors, W. Hasan, C. Kolovson, M. A. Neimat, D. Schneider, and K. Wilkinson. The papyrus integrated data server. In *Proceedings of the First International Conference on Parallel and Distributed Systems*, Miami Beach, Florida, Dec 1991.

[CLR90] T. Cormen, C. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[CS93] S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. Technical report, Hewlett Packard Laboratories, 1993.

[GD87] G. Greafe and D. J. DeWitt. The exodus optimizer generator. In *Proceedings of the 1987 ACM-SIGMOD Conference on the Management of Data*, pages 160–172, San Francisco, CA, May 1987.

[GHK92] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *Proceedings of the 1992 ACM-SIGMOD Conference on the Management of Data*, pages 9–18, San Diego, CA, May 1992.

[HS93] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the ACM SIGMOD International Symposium on Management of Data*, Washington D. C., May 1993.

[KNP92] C. Kolovson, M. A. Neimat, and S. Potamianos. Interoperability of spatial and attribute data managers: A case study. In *The 3rd International Symposium on Large Spatial Databases*. Miami Beach, Florida, Dec 1992.

[Loh88] Guy M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proceedings of the 1988 ACM-SIGMOD Conference on the Management of Data*, pages 18–27, Chicago, IL, June 1988.

[PHH92] H. Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/rule based query optimization in starburst. In *Proceedings of the 1992 ACM-SIGMOD Conference on the Management of Data*, pages 39–48, San Diego, CA, May 1992.

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

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in database systems. In *Proceedings of the 1990 ACM-SIGMOD Conference on the Management of Data*, pages 281–290, Atlantic City, NJ, May 1990.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol 1*. Computer Science Press, 1988.