SEPARABLE SEMANTIC OPTIMIZATION

by

Wei Hong and Eugene Wong

Memorandum No. UCB/ERL M89/69

31 May 1989

to estimate the benefit of keeping a redundant predicate. All unprofitable redundant predicates are removed before the query is passed to a conventional query optimizer. However, this approach does not guarantee overall benefit of the extra optimization because that depends on a correct prediction of what selection-join sequence will be chosen by the query optimizer and this prediction may well be wrong. Another problem with this scheme is that the idea of "redundant predicate" is not well formulated.

To overcome these difficulties, we first propose a new formulation of semantic query optimization that precisely defines optional (redundant) predicates in terms of closures and generators of the original query predicates. Using this formulation, we introduce a new query processing architecture, in which preprocessing and postprocessing modules are added before and after a conventional query optimizer. Preprocessing identifies all optional predicates and detects unsatisfiabilities. Postprocessing removes useless optional predicates. This postprocessing module is essential to guaranteeing that semantic optimization will always result in an improvement. Our approach has been implemented on POSTGRES[STON86a][ROWE87]. POSTGRES is a next-generation database management system that extends relational data model with abstract data types, procedure-typed attributes and attribute/procedure inheritance. It provides a novel way of supporting integrity control. Our method has been designed to make use of all these new features.

This paper is organized as follows. Section 2 proposes our new formulation to semantic query optimization. Section 3 describes the new query processing architecture and establishes some fundamental theorems about our approach. Section 4 gives detailed algorithms and data structures of our implementation on POSTGRES. Section 5 concludes the paper.

2 A New Formulation of Semantic Query Optimization

In this section, we present a new formulation of semantic query optimization. Examples from the following sample database will be used to illustrate ideas throughout the paper.

SEPARABLE SEMANTIC OPTIMIZATION

by

Wei Hong and Eugene Wong

Memorandum No. UCB/ERL M89/69

31 May 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

to estimate the benefit of keeping a redundant predicate. All unprofitable redundant predicates are removed before the query is passed to a conventional query optimizer. However, this approach does not guarantee overall benefit of the extra optimization because that depends on a correct prediction of what selection-join sequence will be chosen by the query optimizer and this prediction may well be wrong. Another problem with this scheme is that the idea of "redundant predicate" is not well formulated.

To overcome these difficulties, we first propose a new formulation of semantic query optimization that precisely defines optional (redundant) predicates in terms of closures and generators of the original query predicates. Using this formulation, we introduce a new query processing architecture, in which preprocessing and postprocessing modules are added before and after a conventional query optimizer. Preprocessing identifies all optional predicates and detects unsatisfiabilities. Postprocessing removes useless optional predicates. This postprocessing module is essential to guaranteeing that semantic optimization will always result in an improvement. Our approach has been implemented on POSTGRES[STON86a][ROWE87]. POSTGRES is a next-generation database management system that extends relational data model with abstract data types, procedure-typed attributes and attribute/procedure inheritance. It provides a novel way of supporting integrity control. Our method has been designed to make use of all these new features.

This paper is organized as follows. Section 2 proposes our new formulation to semantic query optimization. Section 3 describes the new query processing architecture and establishes some fundamental theorems about our approach. Section 4 gives detailed algorithms and data structures of our implementation on POSTGRES. Section 5 concludes the paper.

2 A New Formulation of Semantic Query Optimization

In this section, we present a new formulation of semantic query optimization. Examples from the following sample database will be used to illustrate ideas throughout the paper.

SEPARABLE SEMANTIC OPTIMIZATION

by

Wei Hong and Eugene Wong

Memorandum No. UCB/ERL M89/69

31 May 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

to estimate the benefit of keeping a redundant predicate. All unprofitable redundant predicates are removed before the query is passed to a conventional query optimizer. However, this approach does not guarantee overall benefit of the extra optimization because that depends on a correct prediction of what selection-join sequence will be chosen by the query optimizer and this prediction may well be wrong. Another problem with this scheme is that the idea of "redundant predicate" is not well formulated.

To overcome these difficulties, we first propose a new formulation of semantic query optimization that precisely defines optional (redundant) predicates in terms of closures and generators of the original query predicates. Using this formulation, we introduce a new query processing architecture, in which preprocessing and postprocessing modules are added before and after a conventional query optimizer. Preprocessing identifies all optional predicates and detects unsatisfiabilities. Postprocessing removes useless optional predicates. This postprocessing module is essential to guaranteeing that semantic optimization will always result in an improvement. Our approach has been implemented on POSTGRES[STON86a][ROWE87]. POSTGRES is a next-generation database management system that extends relational data model with abstract data types, procedure-typed attributes and attribute/procedure inheritance. It provides a novel way of supporting integrity control. Our method has been designed to make use of all these new features.

This paper is organized as follows. Section 2 proposes our new formulation to semantic query optimization. Section 3 describes the new query processing architecture and establishes some fundamental theorems about our approach. Section 4 gives detailed algorithms and data structures of our implementation on POSTGRES. Section 5 concludes the paper.

2 A New Formulation of Semantic Query Optimization

In this section, we present a new formulation of semantic query optimization. Examples from the following sample database will be used to illustrate ideas throughout the paper.

SEPARABLE SEMANTIC OPTIMIZATION

by

Wei Hong and Eugene Wong

Memorandum No. UCB/ERL M89/69

31 May 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Separable Semantic Optimization

Wei Hong Eugene Wong
EECS Department, University of California
Berkeley, CA 94720

May 31, 1989

Abstract

This paper describes an efficient new approach to semantic query optimization that is completely separable from conventional query optimizers and guarantees profitability. Moreover, it obtains a processing plan with minimum cost among all the equivalent query forms most of the time. This approach is based on our new formulation to semantic query optimization in terms of closures and generators. It consists of two major steps, preprocessing and postprocessing. Preprocessing explores all the optimization opportunities and detects unsatisfiabilities, and postprocessing makes the final decisions. Only one pass through a conventional optimizer is required. An implementation of our approach on POSTGRES is also described.

1 Introduction

Conventional query optimization has been a well studied area in the database field(see [JARK84a] for a survey). A conventional query optimizer, given a fixed query, searches through "all" possible plans and finds an optimal plan to process that query. The search space includes the following major dimensions: usage of indices, join methods, join ordering, etc. Decisions are made mainly based on syntactic knowledge and storage structures. Syntactic knowledge usually includes relational algebra and first-order logic properties. Storage structures includes availability of indices and database

statistics, e.g., tuple/attribute cardinalities. Semantic query optimization adds a new dimension to the search space, *semantically equivalent queries.* It tries to exploit the semantic knowledge (integrity constraints) about the current database and introduces more opportunities for query optimization.

Semantically equivalent queries are those that produce the same answer for all database instances that satisfy the integrity constraints. The basic idea of semantic query optimization is to transform the original query into semantically equivalent queries, which may yield a more efficient processing plan. Semantic optimization is becoming more important as better integrity control facilities are supported[STON87] and greater integration of Artificial intelligence and database technology are under way[BROD86].

The ultimate objective of semantic query optimization is to choose a semantically equivalent query that yields the best processing plan. A most straightforward way to do this is to feed all possible semantically equivalent queries through a conventional query optimizer and choose the plan with minimum cost. This approach finds the best possible processing plan, but it requires a number of passes through a conventional optimizer when a single pass is already fairly expensive. See [SHEK88] for more analysis on the significance of this optimization cost. To avoid calling a conventional optimizer many times, one can also apply certain heuristics to select a "better" semantically equivalent query than the original query before passing it through the optimizer. This requires only a single pass through the optimizer, but because the decisions are made before the actual query optimization, no estimates about actual cost of various parts of the evaluation of the query are available, the heuristics can only be based on some crude predictions or assumptions about evaluation costs. Therefore semantic optimization may end up choosing an equivalent query that is more expensive to process than the original query. The motivation of our research is to overcome this fundamental difficulty. In this paper, we describe a new approach to semantic query optimization that guarantees profitability while requires only a single pass through a conventional query optimizer.

There is a substantial amount of work that has been done on semantic query optimization. However, the results all suffer from either of the two kinds of problems mentioned above. They either require multiple passes through a conventional query optimizer, thus too expensive to incorporate into existing systems, or can not promise better processing plans after the additional optimization.

Semantic query optimization is first introduced in [KING81] and [HAMM80]. Both of them propose schemes that derive semantically equivalent queries by applying *semantic transformations* to the original query and decide which one is the best by passing all the equivalent queries through a conventional query optimizer. [KING81] also proposes a set of heuristics (e.g., index introduction, join elimination, etc) to reduced the number of promising semantic transformations. Some of these heuristics are used in our approach as described later in this paper.

[CHAK84] and [CHAK87] propose to use the resolution method in first order predicate logic to transform the original query into all possible equivalent queries. They have a semantic compilation scheme based on subsumption to associate the set of valid and useful integrity constraints with each relations. The purpose of this is to transform any queries on that relation by using the associated constraints without searching the entire rule base. They also extend this method to handle non-Horn-clause integrity constraints. However, they have no clear-cut way of deciding which equivalent query would lead to the best processing plan, Furthermore, using resolution to derive equivalent queries is likely to be slow.

[JARK84b] and [SHEN87] both use graph-theoretical approaches involving heuristics. [JARK84b] uses graphs to integrate tableau techniques and syntactic simplification algorithms to optimize queries containing inequality restrictions. Referential integrity constraints like key dependencies, functional dependencies, and value bounds are used to arrive at different forms of a given query. The graph is used to unify attribute values based on referential constraints, to detect cycles that imply equal values for different attributes, and to predict queries with null answers. The problem with [JARK84b] is that there is no scheme available for an explicit representation of arbitrary semantic constraints. The Prolog like view representation scheme allows to express a limited type of constraints on the variables appearing in view definitions. It becomes the responsibility of the user to keep track of semantic constraints contained in view definition. Any changes in the constraints at a later stage makes maintenance and usage of these views difficult.

[SHEN87] uses explicit clausal representation for integrity constraints and uses *query graphes* to identify redundant join predicates and redundant restriction predicates specified in a user query and introduce more redundant predicates from integrity constraints. Heuristics in [KING81] are used

to estimate the benefit of keeping a redundant predicate. All unprofitable redundant predicates are removed before the query is passed to a conventional query optimizer. However, this approach does not guarantee overall benefit of the extra optimization because that depends on a correct prediction of what selection-join sequence will be chosen by the query optimizer and this prediction may well be wrong. Another problem with this scheme is that the idea of "redundant predicate" is not well formulated.

To overcome these difficulties, we first propose a new formulation of semantic query optimization that precisely defines optional (redundant) predicates in terms of closures and generators of the original query predicates. Using this formulation, we introduce a new query processing architecture, in which preprocessing and postprocessing modules are added before and after a conventional query optimizer. Preprocessing identifies all optional predicates and detects unsatisfiabilities. Postprocessing removes useless optional predicates. This postprocessing module is essential to guaranteeing that semantic optimization will always result in an improvement. Our approach has been implemented on POSTGRES[STON86a][ROWE87]. POSTGRES is a next-generation database management system that extends relational data model with abstract data types, procedure-typed attributes and attribute/procedure inheritance. It provides a novel way of supporting integrity control. Our method has been designed to make use of all these new features.

This paper is organized as follows. Section 2 proposes our new formulation to semantic query optimization. Section 3 describes the new query processing architecture and establishes some fundamental theorems about our approach. Section 4 gives detailed algorithms and data structures of our implementation on POSTGRES. Section 5 concludes the paper.

2 A New Formulation of Semantic Query Optimization

In this section, we present a new formulation of semantic query optimization. Examples from the following sample database will be used to illustrate ideas throughout the paper.

```
department(name, floor)
project(name, mgr)
projpart(project, part, qty)
storage(dept, part, qty)
employee(name, age, sal, dept)
```

The queries we shall be dealing with are of the following general POSTQUEL [ROWE87] form:

retrieve (*Target_list*)
where *Predicate* and *Predicate* and ... and *Predicate*

Here, *predicates* (simple predicates) are of the form X *op* Y where X, Y can be either of the following three form: r.a (normal attribute), r.a.b.c (POSTGRES nested dot notation) or any constant; *op* can be any POSTGRES operations that roughly falls into the following categories: arithmetic ($=, \neq, >, \geq, <, \leq$), set-theoretic ($\subset, \in$), subclass ($IS$), user-defined (e.g., .OVERLAP. between rectangles). We have restricted query predicates to conjunctions of simple comparison predicates because those are the only predicates existing query optimizers can handle. Query 2.1 is an example of the queries that we consider.

Query 2.1
List the projects that are managed by someone who has salary over 30K and work in a department on the second floor that stores the parts that the projects need in quantities greater than 450. In POSTQUEL, this can be expressed as:

```
retrieve (project.name)
where
      projpart.part = storage.part
      and projpart.project = project.name
      and project.mgr = employee.name
      and storage.dept = employee.dept
      and employee.dept = department.name
      and projpart.qty > 450
      and emp.sal > 30K
      and dept.floor = 2
```

The semantic rules that we use are all Horn clauses consisting of predicates defined above. We believe that this is not a severe restriction for the same reason as before, viz., non-Horn clause rules with disjunctions and existential quantifiers(as in [CHAK84, CHAK87]) are not likely to be handled well by conventional query optimizers. We have considered three classes of semantic rules: *integrity* rules, *implicit* rules and user-defined *operator* rules. Integrity rules are normal integrity constraints. We shall use the following set of integrity rules as examples throughout this paper.

Rule1
The parts that a project needs have to be stored at the department where the project manager works.

```
projpart.part = storage.part
∧ projpart.project = project.name
∧ project.mgr = employee.name
→ storage.dept = employee.dept
```

Rule2
Storage quantity must be greater than project needs for the same part.

```
projpart.part = storage.part
→ projpart.qty ≤ storage.qty
```

Rule3
All project managers make more than 40K.

```
project.mgr = employee.name
→ employee.sal > 40K
```

Rule4
All employees older than 35 make more than 50K.

```
employee.age > 35
→ employee.sal > 50K
```

Rule5
Only those with age > 45 works on the first and second floor.

```
department.name = employee.dept
∧ department.floor ≤ 2
→ employee.age > 45
```

Rule6
Only department d1 can store parts in quantity > 450.

```
storage.qty > 450
→ storage.dept = d1
```

Rule7
Employees in departments in the basement (floor =0) that store part 'dynamite' make more than 40K.

```
employee.dept = department.name
∧ department.name = storage.dept
∧ department.floor = 0
∧ storage.part = 'dynamite'
→ employee.sal > 40K.
```

The implicit rules are those corresponding to well-known properties of the operations as listed below:
transitivity of $(>, \geq, <, \leq, \subset)$
equivalent relation properties of $=$
For any predicate P,
equality substitution: $(X = Y) \wedge P(X) \rightarrow P(Y)$
subclass substitution: $(X\ IS\ Y) \wedge P(Y) \rightarrow P(X)$
nested dot attribute properties: $(r.a \in r') \wedge P(r') \rightarrow P(r.a)$

User-defined operator rules are rules related to properties of user-defined operators. For example, we can have a user defined operator .COVER. between geometric objects. It is true if the region of one object completely covers the other. Obviously this operator also holds transitivity.

Now we introduce two fundamental concepts in our approach, *closure* and *generator*.

Definition.
Let $\mathcal{P}$ be a set of predicates and $\mathcal{H}$ be a set of Horn clauses. The *closure* of $\mathcal{P}$ under $\mathcal{H}$, $\mathcal{C}$ is a set of predicates defined recursively as the following:

1. $\mathcal{P} \subset \mathcal{C}$
2. if $(p_1 \wedge p_2 \wedge \ldots \wedge p_k \rightarrow p) \in \mathcal{H}$ and $p_1, p_2, \ldots, p_k \in \mathcal{C}$
   then $p \in \mathcal{C}$
3. $\mathcal{C}$ is the smallest set of predicates that satisfies 1 and 2.

Definition.
Let $\mathcal{C}$ be a set of predicates and $\mathcal{H}$ be a set of Horn clauses. The *generator* of $\mathcal{C}$ under $\mathcal{H}$, $\mathcal{G}$ is a subset of $\mathcal{C}$ such that $\mathcal{G} \rightarrow \mathcal{C}$ and no proper subset of $\mathcal{G}$ has this property.
Note: *When we use a set of predicates in a logic implication, we mean the conjunction of all the predicates in the set.*

These two concepts are exactly analogous to the concepts of closure and key in normalization theory[DATE83]. It is easy to see that here predicates correspond to attributes, Horn clauses correspond to functional dependencies, closure corresponds to X-closure and generator corresponds to a candidate key. Note that closures are unique, but there may be multiple generators.

Examples of closures and generators

The closure of **Query 2.1** is

```
{ projpart.part = storage.part, projpart.project = project.name,
  project.mgr = employee.name, employee.dept = department.name,
  storage.dept = employee.dept, projpart.qty > 450,
  employee.sal > 30, employee.sal > 40,
  employee.sal > 50, department.floor = 2,
  employee.age > 45, storage.dept = d1,
  projpart.qty ≤ storage.qty, storage.qty > 450 }
```

The only generator of **Query 2.1** is

```
{ projpart.part = storage.part, projpart.project = project.name,
  project.mgr = employee.name, employee.dept = department.name,
  projpart.qty > 450, department.floor = 2 }
```

Theorem 1 *Let $\mathcal{P}$ and $\mathcal{P}'$ be sets of predicates and $\mathcal{H}$ be a set of Horn clauses. Let $\mathcal{C}$ be the closure of $\mathcal{P}$ under $\mathcal{H}$.*
$\mathcal{P}' \leftrightarrow \mathcal{P}$ iff there exists a generator $\mathcal{G}$ of $\mathcal{P}$ such that $\mathcal{G} \subset \mathcal{P}' \subset \mathcal{C}$

Proof.
(if) $\mathcal{P} \rightarrow \mathcal{C} \rightarrow \mathcal{P}'$ and $\mathcal{P}' \rightarrow \mathcal{G} \rightarrow \mathcal{C} \rightarrow \mathcal{P}$

(only if) $\mathcal{P}' \rightarrow \mathcal{P}$ implies that $\mathcal{P}'$ contains a generator.
$\mathcal{P} \rightarrow \mathcal{P}'$ implies that $\mathcal{P}' \subset \mathcal{C}$.
Q.E.D.

This theorem is actually a characterization of all the semantically equivalent queries of a given query. They are the queries with the same Target_list but qualifications composed of predicates from subsets of the closure of the original query predicates that contain a generator, i.e.,

Corollary 1 *Query* $Q' = \{T' \mid \mathcal{P}'\}$ *is semantically equivalent to query* $Q = \{T \mid \mathcal{P}\}$ *under semantic constraints* $\mathcal{H}$ *iff* $T = T'$ *and* $\mathcal{G} \subset \mathcal{P}' \subset \mathcal{C}$, *where* $\mathcal{G}$ *and* $\mathcal{C}$ *are a generator and the closure of* $\mathcal{P}$ *under* $\mathcal{H}$.

This corollary precisely defines the search space for semantic optimization.

For a given generator $\mathcal{G}$, all the predicates in $\mathcal{C}$ - $\mathcal{G}$ are *optional(redundant)*. We are free to include or exclude any of them into query predicates provided that $\mathcal{G}$ is included, without changing the semantics of the query. Obviously, if there are multiple generators of a query, then the concept *optional* is not well defined. It can only be defined with respect to a particular generator.

The problem of semantic query optimization now becomes how to find and choose a generator and select a subset of all the *optional* predicates under the chosen generator that will help reducing the cost of query processing best. In general, finding all the generators is hard (as finding all keys given functional dependencies is hard). For different generators the set of optional predicates is different. This very likely requires multiple passes through a query optimizer.

We believe that in practice, for most queries and integrity rules there is only a single generator. This will become clear after we introduce the single-generator characterization theorem in the next section. From now on, we will emphasize on *single-generators queries*. Single-generator queries have the nice property that the set of optional predicates is well defined and as we will show later, there is an efficient algorithm to identify the generator and the semantic optimization only requires a single pass through a conventional query optimizer to find the best plan. For *multiple-generator queries*, we may just choose a generator within the original query predicates

with which we can at least do better than processing the original query without semantic optimization.

Now we can assume to have a single generator for any given query and therefore we can compute the set of optional predicates. The problem of semantic optimization becomes to choose a subset of optional predicates to make the query processing least costly. Here are some straightforward observations about how to choose useful optional predicates. Because of the behavior of conventional query optimizers, the following optional predicates are bound to be useless and therefore can be eliminated from consideration:

- *restrictions while stronger restrictions already exist.* For example, in the closure of **Query 2.1** `employee.sal > 30K` and `employee.sal > 40K` are apparently useless because `employee.sal > 50K` is also in the closure.

- *non-equijoins*, in general, joins with operations that are neither merge-joinable nor hashjoinable[STON86b]. For example, `projpart.qty < storage.qty` in the closure of **Query 2.1**.

For other optional predicates, more elaborate criteria based on whether they can introduce more efficient access pathes and whether they can reduce the cost of joins will be introduced in the next section. We will also show in the next section how we can detect unsatisfiabilities in computing the closure.

3 A New Architecture for Query Processing

In the previous section, we have shown that the problem of semantic query optimization is reduced to the problem of choosing optional predicates to help with query processing. Traditionally this can be done by introducing a preprocessing phase to modify the original query by adding profitable optional predicates and delete unprofitable optional predicates before calling the query optimizer. However, before calling the query optimizer very little is known about how the query can be processed. The best we can do at this point is to make the decisions based on some guesses on the choice and cost of access pathes, which may likely lead to even more expensive plans. To overcome this problem, we decide to delay the choice of optional predicates

until after calling the query optimizer at a *postprocessing* phase. This is a major architectural modification to query processing. The new query processing architecture is shown in Figure 3.1. The motivation behind this is that at the postprocessing phase we know for sure what optional predicates the query optimizer likes and made use of and what optional predicates the query optimizer considers as extra burdens, so that we can make the best selection of optional predicates. In order to get the optimizer's "opinion" on each of the optional predicates, we need to let each of them be examined by the optimizer. We can achieve this by modifying the original query into an equivalent query with the original predicates replaced by their closure and then call the optimizer.

The responsibility of the *preprocessor* in the architecture is to compute the closure of query predicates and find the(a) generator thus identify the set of optional predicates. In computing the closure, it also tries to detect the unsatisfiability of the query predicate and returns a null result if the query predicates are unsatisfiable, thus prevents wasting time on any further processing. If the query predicates are satisfiable, it then makes up a query with the original target list and the closure of predicates and passes that to the optimizer. The responsibility of the *postprocessor* is to examine the use of each optional predicates in the optimal plan the optimizer found for the closure query and deletes those optional predicates that does not reduce the cost of the query processing. We now discuss the algorithms in the two new modules.

In preprocessing we need algorithms for computing the closure, finding the(a) generator and detecting unsatisfiability. The algorithm to find closure is very straight forward as follows.

Algorithm for finding closure

Input: $Q = \{T \mid \mathcal{P}\}$ a query, $\mathcal{R}$ a set of integrity rules.
Output: $\mathcal{C}$ the closure of $\mathcal{P}$ under $\mathcal{R}$.
Notation: for any rule $r \in \mathcal{R}$, let $lhs(r)$ be the left hand side, i.e., the antecedent part of r and $rhs(r)$ be the right hand side, i.e., the consequence part of r.

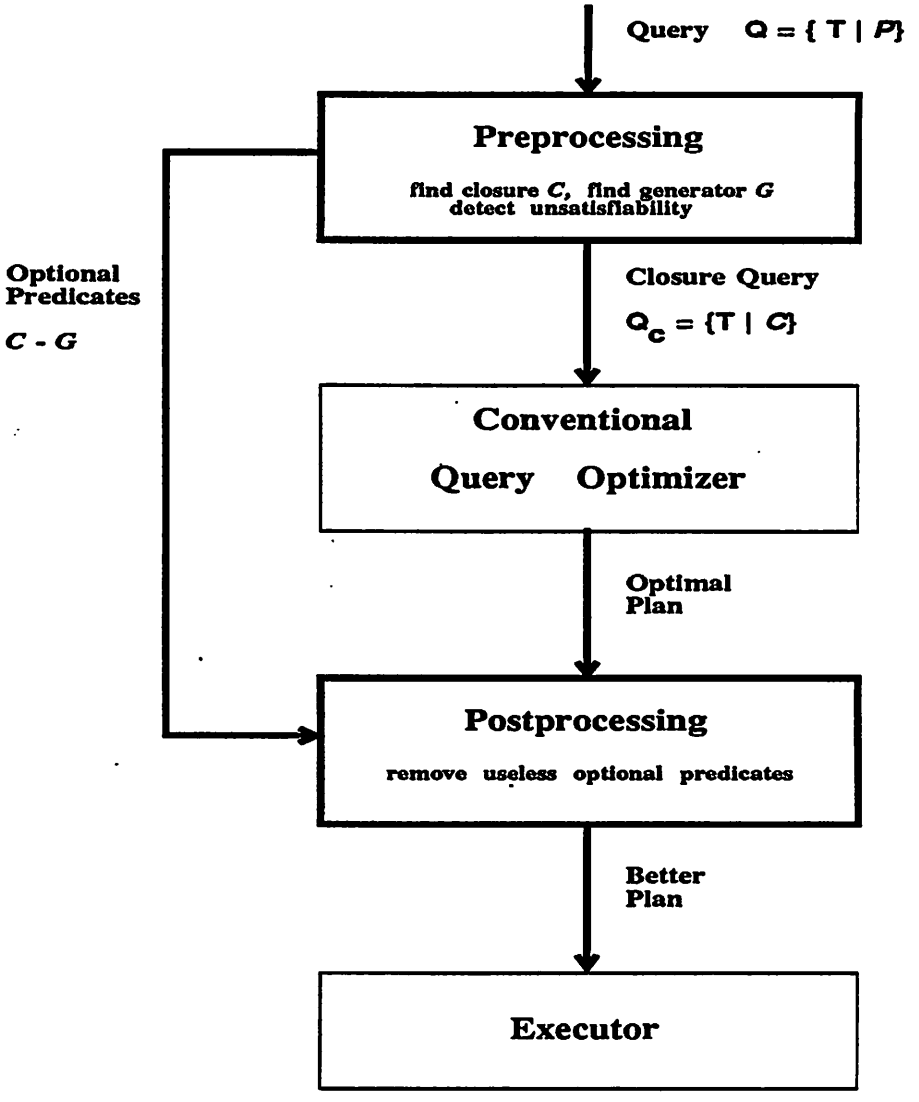

Figure 3.1 A New Query Processing Architecture

Repeat
    for all $r \in \mathcal{R}$
    if $lhs(r) \subset \mathcal{C}$
    then $\mathcal{C} = \mathcal{C} \cup \{rhs(r)\}$
until $\mathcal{C}$ stops growing.

It is essentially the same as the X-closure algorithm in normalization theory except that a few minor changes may be incorporated. For example, we may discard obviously useless optional predicates right away and we may "hardwire" implicit rules into the program to increase efficiency. The major issue here is to design an efficient storage structure for the integrity rules so that possibly relevant rules can be quickly retrieved and need only be retrieved once. This issue will be discussed in the next section in the context of POSTGRES. All the rules retrieved and actually used in deriving predicates in the closure are those that are relevant to the given query and therefore kept separately for later use in preprocessing.

The algorithm for finding generator is based on the following fundamental theorem.

Theorem 2 (A necessary and sufficient condition for there to be a single generator)

Let $\mathcal{P}$ be the set of predicates of a query Q and $\mathcal{H}$ the set of integrity rules. Let $\mathcal{R} \subset \mathcal{H}$ be the set of integrity rules that are relevant to Q, i.e., $\mathcal{R} = \{(p_1 \wedge p_2 \wedge \ldots \wedge p_k \rightarrow p) \in H | \mathcal{P} \rightarrow p_i, i = 1, 2, \ldots, k\}$. Let $\mathcal{N}$ be the set of predicates that appear in $\mathcal{R}$ but never appear on the righthand side of any rule, i.e., the set of predicates that can not be implied.

Query Q has a single generator iff $\mathcal{N} \rightarrow \mathcal{P}$

Proof.

(if) Since predicates in $\mathcal{N}$ can never be implied by other predicates, they have to be included in every generator. Because $\mathcal{N} \rightarrow \mathcal{P}$, $\mathcal{N}$ is a generator itself. Therefore, $\mathcal{N}$ is the only generator.

(only if) Suppose $\mathcal{N}$ does not imply $\mathcal{P}$ and $\mathcal{P}$ has only a single generator. Let $\mathcal{P}^*$ and $\mathcal{N}^*$ be the closure of $\mathcal{P}$ and $\mathcal{N}$ respectively. Let $\mathcal{S} = \mathcal{P}^* - \mathcal{N}^*$. Obviously, $\mathcal{S} \neq \phi$. (Otherwise, $\mathcal{P}^* = \mathcal{N}^*, i.e., \mathcal{N} \rightarrow \mathcal{P}$.) Because $\mathcal{P}^*$ has only a single generator, $\mathcal{S}$ must also have a single generator. Otherwise, suppose $\mathcal{K}_1, \mathcal{K}_2$ are different generators of $\mathcal{S}$, $\mathcal{N} \cup \mathcal{K}_1$ and $\mathcal{N} \cup \mathcal{K}_2$ would

be different generators of $\mathcal{P}$. Suppose $\mathcal{K}$ is the only generator of $\mathcal{S}$. Let $p \in \mathcal{K}$. We know that p appears on the right hand side of some Horn clause. Suppose the Horn clause is $p_1 \wedge p_2 \wedge \ldots \wedge p_n \rightarrow p$, $p_i \neq p$, $i = 1, \ldots, n$. We have $\{p_1, p_2, \ldots, p_n\} \cup (\mathcal{K} - \{p\}) \rightarrow \mathcal{K}$. Thus there must exist a generator in $\{p_1, p_2, \ldots, p_n\} \cup (\mathcal{K} - \{p\})$, but the generator does not contain p, therefor is different from $\mathcal{K}$. Thus, there are more than one generators in $\mathcal{S}$. Contradiction.

Q.E.D.

The algorithm we use for finding a generator is based on the above theorem. First we find the set of predicates $\mathcal{N}$ that is included in all possible generators. Then we test whether the query has a single generator by the theorem. If there is only a single generator, then $\mathcal{N}$ is the one, otherwise there are multiple generators, in which case, we do not try to find all the generators, we simply add predicates from $\mathcal{P}$ to $\mathcal{N}$ until it implies $\mathcal{P}$, $\mathcal{N}$ together with those added predicates from $\mathcal{P}$ consists of a generator within $\mathcal{P}$. By using a generator in $\mathcal{P}$, we can guarantee that the semantic optimization never makes things worse.

We do not really need a separate algorithm for detecting unsatisfiability. Based on the following theorem, unsatisfiability can be detected along the way of computing the closure.

Definition.
Let $\mathcal{P}$ be a set of predicates and $\mathcal{H}$ be a set of Horn clauses. $\mathcal{P}$ is *satisfiable* under $\mathcal{H}$ if there exits certain tuples in a database state satisfying $\mathcal{H}$ that satisfies $\mathcal{P}$. Otherwise, $\mathcal{P}$ is *unsatisfiable.*

Theorem 3 (A necessary and sufficient condition for satisfiability)
Let $\mathcal{P}$ be a set of predicates, $\mathcal{H}$ be a set of Horn clauses and $\mathcal{C}$ be the closure of $\mathcal{P}$ under $\mathcal{H}$.

$\mathcal{P}$ is satisfiable under $\mathcal{H}$ iff each individual predicate in $\mathcal{C}$ is satisfiable.

Proof.
(if) If $\mathcal{P}$ is unsatisfiable, then $\mathcal{P} \rightarrow f$, where f is an unsatisfiable predicate. Therefore $f \in \mathcal{C}$, i.e., there exists an unsatisfiable predicate in $\mathcal{C}$.

(only if) If there exists an unsatisfiable predicate, $f \in \mathcal{C}$, i.e., $\mathcal{P} \rightarrow f$, then $\mathcal{P}$ is unsatisfiable.

Q.E.D.

Based on this theorem, we can simply modify the algorithm for finding closures slightly to detect unsatisfiability. Each time a new predicate is derived into the closure, we check its satisfiability. As long as there is no unsatisfiable predicates such as `r.a > r.a` in the closure, the query predicates are satisfiable.

In postprocessing we only need some criteria for deciding whether a certain optional predicate helps to reduce the cost of query processing. Before we introduce the criteria, we need to classify the optional predicates. Optional predicates must be implied by some other predicates, thus they must appear at the righthand side of some rules. They can be classified into the following three classes according to the form of rules that derive them:

- *single-relation-derived restrictions* if it can be implied by predicates on the same relation, e.g., `department.name = d1, employee.sal > 50K` in **Query 2.1**.
- *multiple-relation-derived restrictions* if it can only be implied by predicates on more than one relations, e.g., `employee.sal > 40K, employee.age > 45` in **Query 2.1**.
- *derived joins* if it involves two relations, e.g., `storage.dept = employee.dept, projpart.qty ≤ storage.qty` in **Query 2.1**

Given the behavior of the current optimizers, we observe that there are only two possible ways an optional predicate may be used in a query processing plan. First, they may be used to introduce an efficient access path, for example a restriction predicate, `employee.sal > 40K` may be useful if there is a B-tree index on employee.sal, a join predicate, `employee.dept = department.dname`, may be useful if it can be evaluated by efficient merge-join or hash-join. Second, they may just act as additional "tuple filters"(additional restrictions on the tuples fetched). The first case certainly reduces the cost of query processing. It corresponds to the *index introduction* and *join introduction* heuristics in [KING81]. The second case reduces the cost of query processing only when the additional restriction reduces the size of input to some join operation. For example, if

```
employee.dept = department.dname
∧ department.floor = 1
→ employee.age > 50
```

Then we can use `employee.age > 50` to reduce the size of employee before the join operation `employee.dept = department.dname`. Note that only multiple-relation-derived restrictions are relevant in this case because single-relation-derived restrictions can not reduce the size of a join input relation (they automatically hold) and for multiple-relation-derived restrictions, their antecedents do not hold until after the join operation and thus they reduce the size of join inputs. This case corresponds to the *scan reduction* heuristic in [KING81]. There is another heuristic in [KING81], *join elimination* that we do not consider although it can be easily included in our approach. The reason for this is because this heuristic requires subset integrity constraints which is very hard to preserve and in POSTGRES with the nested dots notation in queries, very few queries might be subject to this optimization.

Algorithm for Choosing Optional Predicates

Let $\mathcal{C}$ and $\mathcal{G}$ be the closure and generator of a query $Q = \{T \mid \mathcal{P}\}$. Let $\mathcal{O} = \mathcal{C} - \mathcal{G}$, the set of optional predicates. Let PLAN be the optimal plan that the optimizer finds for the closure query $\{T \mid \mathcal{C}\}$.

```
for all p ∈ O do
    if p introduces a new index in PLAN or
       p is a multiple-relation-derived restriction
         and is used before join
    then keep p in PLAN
    else delete p from PLAN
```

Example of choosing optional predicates.

The set of optional predicates (after eliminating obviously useless ones) of **Query 2.1** (the set difference of closure and generator, see the previous section) is

p_1: `storage.dept = employee.dept`, p_2: `employee.sal > 50`,
p_3: `employee.age > 45`, p_4: `storage.dept = d1`,
p_5: `storage.qty > 450`

Among these optional predicates, p_2 and p_4 are single-relation-derived restrictions, p_3 and p_5 are multiple-relation-derived restrictions and p_1 is a

derived join. Suppose in the plan found by the query optimizer, p_1 is used to perform a merge-sort join between `storage` and `employee`, p_2 is used to take advantage of the index on `employee.age` but p_4 is just an extra tuple filter in the plan, p_3 is used before the join between `storage` and `employee` but p_5 is used after the join between `storage` and `employee` because the join uses the index on `storage.dept`. As a result, we keep p_1, p_2, p_3 in the plan, but delete p_4 and p_5 from it.

The following Theorem guarantees the benefit of our approach.

Definition.
We define the cost of a query Q, $Cost(Q)$ be the cost of an optimal plan of Q and we define the cost of a processing plan is measured by the standard formula

$$C_{i/o} + W \times C_{cpu}$$

where $C_{i/o}$ stands for the I/O cost of the plan and equals to the number of disk blocks read or written, C_{cpu} stands for the CPU cost of the plan and equals to the number of tuples processed and W is a fudge factor.

Theorem 4 *Let $\mathcal{C}$ and $\mathcal{G}$ be the closure and generator of a query $Q = \{T \mid \mathcal{P}\}$. Let $Q_f = \{T \mid \mathcal{P}_f\}$ where $\mathcal{P}_f = \mathcal{C} - \mathcal{F}$, and $\mathcal{F}$ is the set of optional predicates that are deleted after our postprocessing. For any query $Q_x = \{T|P_x\}$ that is semantically equivalent to Q,*

$$Cost(Q_f) \leq Cost(Q_x)$$

Proof.
According to Corollary 1, $\mathcal{G} \subset \mathcal{P}_x \subset \mathcal{C}$. Let $Q_c = \{T \mid \mathcal{C}\}$, obviously Q_c has less I/O cost than Q_x because it has more opportunities for using indices and Q_c has less CPU cost than Q_x because it has more restrictive qualification therefore more tuples may be eliminated earlier. Therefore $Cost(Q_c) \leq Cost(Q_x)$. According to the criteria we use in postprocessing, $Cost(Q_f) = Cost(Q_c)$. Therefore, $Cost(Q_f) \leq Cost(Q_x)$.

Q.E.D.

From the proof of the above theorem, we can see that with the standard cost function of query processing, it turns out that the closure query has the minimum cost among all equivalent queries, i.e., the postprocessing phase is not even necessary. This surprising result is due the simplism of the standard cost function. Apparently, the more predicates the query has to satisfy the more CPU cycles have to be consumed to process them. This factor is not taken into account in the cost function. However, we believe that the number of disk I/O's and the number of tuples to process are the major cost factors while the number of predicates in a query condition only have minor effect of the query processing cost. The optimality of our approach is supported by the following theorem.

Theorem 5 *Suppose $Q_x = \{T|\mathcal{P}_x\}$ is a semantically equivalent query of Q and Q_f is the same as in Theorem 4. If $Cost(Q_x) = Cost(Q_f)$, then $\mathcal{P}_f \subset \mathcal{P}_x$. In other words, Q_f has the least number of predicates to process among those with minimum Cost.*

Proof.
Because $Cost(Q_x) = Cost(Q_f)$, i.e., the evaluation of Q_x and Q_f fetch the same number of disk pages and process the same number of tuples. Let $\mathcal{P}_f = \mathcal{G} \cup \mathcal{O}_f$ where $\mathcal{G}$ is the generator and $\mathcal{O}_f$ is the set of optional predicates chosen at the postprocessing phase. Obviously $\mathcal{G} \subset \mathcal{P}_x$. According to the criteria of choosing $\mathcal{O}_f$, if any predicate in $\mathcal{O}_f$ does not belong to $\mathcal{P}_x$, query Q_x either loses a more efficient access path or has to fetch more tuples in joins, and thus $Cost(Q_f) < Cost(Q_x)$. Therefore, $\mathcal{O}_f \subset \mathcal{P}_x$ and $\mathcal{P}_f = (\mathcal{G} \cup \mathcal{O}_f) \subset \mathcal{P}_x$.

Q.E.D.

4 Implementation on POSTGRES

We discuss the implementation of preprocessing and postprocessing separately. The current version POSTGRES are mostly written in Franz LISP Opus 43. The preprocessing and postprocessing are also implemented in this language.

Preprocessing

The most important part of preprocessing is to find the closure. There are several implementation details that need to be filled out in the algorithm for finding closures in the previous section:

1. how to store integrity rules in the database so that the rules that are relevant to a query can be fetched efficiently,

2. how to organize the relevant rules so that they are not repeatedly examined,

3. how to identify the membership of a predicate to a set of predicates quickly.

Given a set of integrity rules $\mathcal{R}$ and a set of query predicates $\mathcal{P}$, the set of relevant rules are defined as the following set:

$$\{r \in \mathcal{R} \mid \mathcal{P} \rightarrow lhs(r)\}$$

The left-hand-side(lhs) of a rule and $\mathcal{P}$ are both sets of simple predicates. If we only consider arithmetic comparison operations, then sets of predicates corresponds to convex polyhedra in a geometric space. The problem of identifying relevant rules can be reduced to the problem of identifying convex polyhedra that are contained in a given convex polyhedra. Although indices like Cell Tree[GUNT87] might help with this search, considering the high dimension(number of attributes) of our search space, they would not be very useful. We have to relax our constraint and fetch a superset of relevant rules from the database. A straightforward candidate for this superset is *the set of rules that only involves relations in the original query.* This is based on our assumption of integrities rules that the set of relations that appear in the right-hand-side of a rule must also appear in the left-hand-side of the rule. This looks like a reasonable assumption. With this assumption we can conclude that the set of relevant rules can never involve relations that are not relations of the original query. This is not true for attributes because rules may derive new attributes that are not in the original query. Unfortunately, there is no indexing schemes that can help with this set containment search either. We finally resort to an even larger superset, *the set of rules that contain at least one relation in the original query.* We have an efficient indexing scheme to support searching

of this set. Actually, this search is analogous to the keyterm search in a bibliographic database, in which one searches for the set of books containing a certain keyword. [LYNC88] proposes a simple and efficient indexing scheme for supporting this kind of search problems and shows that it can be easily implemented in POSTGRES by using user-defined indices, user-defined operators and abstract data types. We decide to use this indexing scheme to support integrity rule searching.

After we fetch the set(superset) of relevant rules from the database, they can be simply represented as LISP lists. The algorithm to find the closure may require multiple passes through this list of rules. The number of passes depends on the order of examining the rules. For example, suppose we have two rules, $r_1 : p_1 \rightarrow p_2$, $r_2 : p_2 \rightarrow p_3$, and we want to find the closure of $\{p_1\}$. If we examine the rules in the order r_1, r_2, then we only need one pass, but if we examine the rules in the order r_2, r_1, we need two passes. We want to examine the rules in the order that yields minimum number of re-examination.

Definition.
We define a relationship (>>) between rules. For two rules r_1 and r_2, $r_1 >> r_2$ if $rhs(r_1) \in lhs(r_2)$.

If >> is a partial order, then obviously if we examine the rules observing this partial order, i.e., if $r_1 >> r_2$, then examine r_1 before r_2, we only need to examine each rule once. Otherwise, there exists a cycle, $r_1 >> r_2 >> \ldots >> r_1$, which we believe is very rare with database integrity rules. Even in this case, we only need to re-examine r_1 and its descendends.

The way we implement this on POSTGRES is that we organize the list of rules into a directed graph based on >>, and represent it as an adjacency list. We examine the rules by traversing the graph in the topological order. We revisit a node only if it is in a cycle. The algorithm is a straightforward modification from the well-known topological sort algorithm.

The last detail to fill out is how to quickly decide the membership of a predicate in a given set of predicates. This can be efficiently achieved by building a LISP hash table or assoc list on (attribute predicates) so that given an attribute, we can quickly find the set of predicate that involve this attribute. Given a predicate, we first use the hash table to find the set of predicates containing the same attributes, and then compare with each one of them.

Postprocessing

The detail we need to give about postprocessing is how to apply the criteria of choosing optional predicates to a given query processing plan found by the POSTGRES query optimizer. A POSTGRES query plan is a tree composed of nodes like **NESTLOOP, MERGESORT, HASHJOIN, SEQSCAN, INDEXSCAN, SORT,** etc[POST88]. Each node is a LISP structure with a field called *qpqual* that contains the list of predicates that the output tuples of this node have to satisfy. These predicates do not contribute to the use of any efficient access paths. They only act as tuple filters. Now the algorithm of choosing optional predicates can be written in the POSTGRES context as the following:

Let $\mathcal{C}$ and $\mathcal{G}$ be the closure and generator of a query $Q = \{T \mid \mathcal{P}\}$. Let $\mathcal{O} = \mathcal{C} - \mathcal{G}$, the set of optional predicates. Let PLAN be the optimal plan the optimizer finds for the closure query $\{T \mid \mathcal{C}\}$.

```
for all p ∈ O do
    if p is in qpqual field of a node
       if p is not a multiple-relation-derived restriction
          or is not in a SEQSCAN or INDEXSCAN node
       then delete p from PLAN
```

5 Conclusions

In this paper we have proposed and described a new approach to semantic query optimization. We emphasize on solving the fundamental problem of semantic query optimization, i.e., complexity v.s. benefit. Our approach has both low complexity and warranty of benefit. Our approach also has good modularity and can be easily incorporated into any existing systems by adding preprocessing and postprocessing before and after the query optimizer. We view the closure-generator formulation of semantic optimization and the introduction of postprocessing as our major contributions to this field. The new formulation clearly characterizes the space of semantic query optimization and naturally leads to our preprocessing-postprocessing architecture. Deferring decisions on selecting optional predicates to postprocessing is the key to guarantee benefit of our approach.

Most of the things that have been described in this paper have been implemented on POSTGRES. Experiments for measuring profitabilities are under way.

References.

[BROD86] Brodie, M. L. and Mylopoulos, J., "On Knowledge Base Management Systems," Springer-Verlag, 1986.

[CHAK84] Chakravarthy, U. S., et al, "Semantic Query Optimization in Expert Systems and Database Systems," EDS 1984

[CHAK87] Chakravarthy, U. S., et al, "Semantic Query Optimization: Additional constraints and control strategies," EDS 1987

[DATE83] Date, C. J., "An Introduction to Database Systems," (3rd edition), Addison-Wesley, Reading, Mass., 1983.

[GUNT87] Gunther, O., "Efficient structures for geometric data management," Mem. No. UCB/ERL M87/77, Nov. 1987.

[HAMM80] Hammer M., and Zdonik, S B Jr., "Knowledge based query processing," VLDB 1980, p137-147.

[JARK84a] Jarke, M., "Query optimization in database systems," ACM Computing Surveys, Vol. 16, No. 2, June 1984, p111-152.

[JARK84b] Jarke, M., "External semantic query simplification: A graph-theoretic approach and its implementation in Prolog, EDS 1984, p467-482.

[KING81] King, J. J., "QUIST: A system for semantic query optimization in relational databases," VLDB 1981, p510-517.

[LOBO87] Lobo, J. and Minker, J., "A metaprogramming approach to semantically optimize queries in deductive databases," EDS 1987, p387-420.

[LYNC88] Lynch, C. and Stonebraker, M., "Extended user-defined indexing with application to textual databases," VLDB 1988.

[POST88] "Query Tree/Query Plan Specification," POSTGRES group document, UCB, 1988.

[ROWE87] Rowe, L. and Stonebraker, M., "The POSTGRES data model," VLDB 1987.

[SHEK88] Sheckhar, S., et al, "A formal Model of Trade-off between Optimization and Execution Costs in Semantic Query Optimization" VLDB 1988

[SHEN87] Shenoy, S. and Ozsoyoglu, Z M., "A system for semantic query optimization," ACM-SIGMOD 1987

[SIEG87] Siegel, M., "Automatic rule derivation for semantic query optimization," EDS 1987

[STON86a] Stonebraker, M. and Rowe, L., "The Design of POSTGRES," SIGMOD 1988.

[STON86b] Stonebraker, M., "Inclusion of New Types in Relational Database Systems," Data Base Engineering, 1986

[STON87] Stonebraker, M., et al, "The rule manager for relational database systems,", Mem No. UCB/ERL M86/85.