Design and Implementation of a Semantic Query Optimizer

SREEKUMAR T. SHENOY AND ZEHRA MERAL OZSOYOGLU

Abstract—**In this paper we describe a scheme to utilize semantic knowledge in optimizing a user specified query. The semantics is represented as function-free clauses in predicate logic. The scheme uses a graph theoretic approach to identify redundant joins and restrictions present in a given query. An optimization algorithm is presented which eliminates redundant nonprofitable specifications from a query while adding additional profitable specifications to it. Dynamic and heuristic interaction of three entities—schema, semantics, and query—forms the basis of the algorithm. The implementation architecture of the algorithm and test results on a representative set of data are presented. Issues associated with updating of semantic constraints are addressed and an algorithm for semantic maintenance is introduced.**

Index Terms—**Algorithms, graph theory, heuristics, implication integrity constraints, query optimization, redundancy, relational databases, secondary index, semantic rules, subset integrity constraints.**

I. Introduction

QUERY optimization in relational databases continues to be an active issue in both academic and commercial fields for quite a long time now. The relevance for optimization stems from the flexibility provided by modern user-interfaces to databases. The interfaces and nonprocedural query languages facilitate the users to specify queries which may be computationally costly and inefficient to process. It then becomes not only meaningful but also important to reformulate the user specified query before executing it to an equivalent form that is computationally more efficient.

Query optimization can be formally defined as a process of transforming a query into an equivalent form (that produces the same result as the original one for all database states) which can be evaluated more efficiently.

Optimization in its conventional sense utilizes syntactic knowledge of the operations and storage details of the relations. The syntactic knowledge includes algebraic transformations and operator resequencing, whereas the storage details include indexes and clustering of storage. Several query processing algorithms are proposed in the literature [1]–[4], [8], [10], [11], [14], [17], [21], [23], [27], [31], [36], [39]–[41]. Most of the major commercial database management systems utilize these techniques to some extent to answer the ad hoc user queries.

Manuscript received July 14, 1989.

The authors are with the Department of Computer Engineering and Science and the Center for Automation and Intelligence, Case Western Reserve University, Cleveland, OH 44106.

IEEE Log Number 8930722.

Semantic processing adds a relatively new dimension to query optimization. Instead of just resequencing the operators or incorporating the indexed access of data files, it tries to exploit any available knowledge about the data. For instance, it utilizes knowledge about the domains of relations, nature of data, and constraints associated with database instances. Such relevant pieces of knowledge available to the optimizer, combined with its potential ability to intelligently process it, helps its generation of more optimal forms of the user specified query from an execution point of view.

Significance of semantic optimization can be made more apparent by certain inherent limitations of syntactic optimization techniques. Since syntactic optimizers lack the entire body of semantic knowledge assured to be satisfied by all the instances of a particular database, in many cases they produce suboptimal forms of the query for execution. Certain queries that can be answered without any relation scans cannot be detected by syntactic optimizers, thus resulting in redundant database access. Cases where queries contain dangling relations cannot be identified by syntactic techniques alone, thus forcing redundant joins to be performed. Also, syntactic optimizers cannot detect and eliminate semantically redundant restrictions or joins from user specified queries, and for the same reason they fail to introduce semantically redundant restrictions or joins which could, in turn, reduce the overall cost of the query.

Semantic query optimization is based on the semantic equivalence rather than the syntactic equivalence between different queries. Two queries are syntactically equivalent if their answers are the same for all the instances of the database. Two queries, possibly syntactically nonequivalent, are semantically equivalent if their answers are the same for all the instances of the database that satisfy the specified set of semantic rules. Semantic equivalence does not imply syntactic equivalence while syntactic equivalence trivially implies a semantic one. As an example, the two queries "retrieve (emp.all) where emp.Sal > 40K" and "retrieve (emp.all) where emp.Sal > 40K and emp.Job = 'Manager'" are not syntactically equivalent, but are semantically equivalent under the semantic rule 'emp.Sal > 40K → emp.Job = "Manager"'. Since the semantic equivalence between queries depends only on the database schema and the semantic rule set, the different queries can interchangeably be used to get the same results, provided the schema and the semantics are unaltered. Moreover, since these syntactically nonequivalent

queries can independently be optimized by a conventional syntactic optimizer, semantic processing does in fact expand the spectrum of equivalent forms of the specified query. Thus, the semantic query optimization is the process of finalizing, among all the possible syntactically and semantically equivalent forms of the query, the one which can be executed most efficiently.

There are various issues involved in semantic query processing. First, query and schema should be dynamically used to select the relevant semantics for optimization without an exhaustive search of semantic rule base. Second, there should be some mechanism to merge the selected semantics with the query. Third, there should be a cost analyzer to evaluate the costs of equivalent queries and rank them accordingly. Fourth, there should be a set of heuristics to guide the whole process in a meaningful way without a combinatorial explosion.

This paper is organized as follows. Section II presents a brief discussion on the related previous work. In Section III we discuss clausal representations of query as well as various types of constraints that constitute a semantic rule base for the optimizer. We introduce a simplified and generalized representation of implication constraints. Section IV addresses the issues related to the maintenance of semantic constraints. A maintenance algorithm is presented in this section with its associated data structures. Section V illustrates the role of heuristics as inference rules. Different graph schemes used to represent and transform the query are introduced in Section VI. A detailed discussion on various stages of semantic query transformation appears in Section VII. In Section VIII we present the transformation algorithm, its implementation architecture, and the implementation results. Section IX concludes the paper.

II. Previous Work

Semantic optimization has recently been the subject of detailed analysis from two different perspectives. Reference [18] formally introduced the issue in an artificial intelligence context and introduced a set of heuristics for query transformation. Reference [12] analyzed the problem in a database point of view.

Major heuristics discussed in [18] were *index introduction*, *join introduction*, *scan reduction*, and *join elimination*. Index introduction tries to obtain a constraint on an attribute of a relation which is restricted in the query and which has a clustered indexed attribute that is not restricted in the query. According to the strategy of join introduction, a relation should be a constraint target if it has a clustering link into a much larger relation that is constrained in the query, even if the relation itself is not in the original query. This heuristic contemplates addition of a join to the query, referred to as join introduction. In the case of scan reduction, the objective is to reduce the number of inner scans of the join by obtaining additional restrictions prior to the cross referencing part of the operation. Join elimination becomes possible if a relation is joined to just one other relation and none of its attributes contribute to the answer.

Later, a substantial amount of research followed, related to the theory and implementation of semantic rules for query processing [5]–[7], [15], [16], [22], [26], [34], [35].

Two of the above papers, [5] and [15], have maximum relevance to our current work. Reference [5] introduces the concept of semantic compilation, where all the relevant semantic rules are explicitly associated with each relation or view definitions. This allows any query on that relation or view to be semantically transformed with only a limited search of the rule base. The result of interaction of a query with compiled relations or views is a group of semantically equivalent queries, each of which can be potentially optimized using a syntactic optimizer. Reference [15] describes a graph theoretic approach integrated with tableau techniques and syntactic simplification algorithms to optimize queries containing inequality constraints. Referential integrity constraints like key dependencies, functional dependencies, and value bounds are used by the algorithm. 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.

Both of the above methods have certain limitations. Reference [5] fails to clearly categorize a given piece of semantic information as a rule or as a view. Also, no method is available to select or prioritize the rules associated with a relation or view in a query context. Moreover, no mechanism is available to quantify the profitability of a rule for a relation in a query context. In other words, integration of semantic rules with relations is considered in isolation with query context. In [15], explicit representation of arbitrary semantic rules is not supported. Prolog like view characterization is used to express a limited type of constraints on the variables appearing in view definitions. Since semantic details are integrated with view definitions, it becomes the responsibility of the end user to keep track of the semantics associated with each view. Since the constraints are hardwired to view definitions, they become unsharable by the similar attributes originating from the query. Also, any changes in the constraints at a later stage makes the maintenance of these views difficult.

In [32] we try to address the above difficulties by using explicit clausal representation [20] of integrity constraints as in [5], and by devising a mechanism for dynamic interaction between relations and constraints in a query context. Among the valid constraints selected for such interaction, only the profitable ones are finally used, profitability being decided by heuristic rules, global parameters, and some assumptions.

III. Clausal Forms of Query and Semantic Constraints

In this section we discuss the clausal representation [20], [37], [39] which provides a theoretical basis for specifying query as well as semantic constraints. After introducing the notations of clausal form, we describe the

specific representational details of query and semantic constraints.

A. Clausal Representation

A *clause* is an expression of the form "$A_1, \ldots A_n \rightarrow B_1, \ldots, B_m$" where $A_1, \ldots, A_n, B_1, \ldots, B_m$ are atoms (or atomic formulas), $n \geq 0$ and $m \geq 0$. An *atom* (or atomic formula) is an expression of the form "$p(t_1, \ldots, t_j)$" where p is a j-place predicate symbol, $t_1, \ldots, t_j$ are terms, and $j \geq 1$. A *term*, in its most general form, is a variable, a constant symbol, or a function $f(t_1, \ldots, t_k)$ where f is a k-place function symbol, $t_1, \ldots, t_k$ are terms, and $k > 0$.

In our discussion we consider only function-free terms. A function-free atomic formula, $p(t_1, \ldots, t_j)$, denotes either a *relation* or an *evaluable (built-in)* predicate. If it is a relation, it is the relation of its predicate restricted for equality by any constant appearing in a component and for equality between components that have the same variable. If it is an evaluable predicate, it is a binary comparison (arithmetic or set) predicate of the form $=, \neq, >, \geq$, "contains", etc. We follow the usual infix notation $X > Y$, instead of $>(X, Y)$, to represent the evaluable predicates.

A *literal* is either an atomic formula or a negated atomic formula. A nonnegated atomic formula is positive literal, and a negated one is a negative literal. A clause, thus, is a sum (logical OR) of literals. A clause with at most one positive literal is called a *Horn clause*, which can be of one of the following categories.

1) Integrity Constraint: No positive literal, one or more negative literals ($m = 0, n > 0$).

2) Unit Clause or Fact: A single positive literal, no negative literals ($m = 1, n = 0$).

3) Rule: A single positive literal, one or more negative literals ($m = 1, n > 0$).

The set of negative literals $A_1, \ldots, A_n$ of the above clause is sometimes referred to as the body of the clause, and the set of positive literals $B_1, \ldots, B_m$ as its head. The atoms appearing in the body are the joint (conjunctive) conditions of the clause, and the ones in its head are the alternative (disjunctive) conclusions. The conditions are sometimes referred to as *antecedent atoms* and the conclusions as *consequent atoms*.

B. Query

A *simple query* q, expressed in the context of a database scheme D, is syntactically similar to a unit clause (fact). The difference is that a unit clause asserts that a goal is true, whereas a query asks whether the goal is true. The variables appearing in a query are implicitly existentially quantified. Shared variables are used as a means of constraining a simple query by *restricting the range* of a variable.

A *substitution* is a finite set of pairs of the form $X_i = t_i$, where X_i is a variable and t_i is term, and $X_i \neq X_j$ for every $i \neq j$, and X_i does not occur in t_j, for any i, j. The result of applying a substitution @ to a term A, denoted by $A@$, is the term obtained by replacing every occurrence of X in A by t, for every pair $X = t$ in @. B is an *instance* of A if there is substitution @ such that $A@ = B$. *Answering* a query is the process of finding all the facts that are instances of the query. All such instances form the *solution* of the query.

Conjunctive queries are practically more relevant than the simple ones. A conjunctive query Q specifies a conjunction of goals posed as a query. Shared variables are used to specify *equality restrictions* as well as *equijoins* between terms in conjunctive queries. *Inequality restrictions* and *inequality joins* are specified by explicit terms. The explicit inequality operators used in our discussion are from $\{\neq, >, \geq\}$. The operators $<$ and $\leq$ are not explicitly considered because $a < b$ and $a \leq b$ can be represented by $b > a$ and $b \geq a$, respectively. Similarly, $a = b$ can be represented by the conjunction of $a \geq b$ and $b \geq a$.

C. Semantic Constraints

Constraints are laws or expressions associated with the database that represent certain required properties of the data. There are two broad classifications of constraints, i.e., state constraints and transition constraints [28]. In this paper, we restrict our discussion to state constraints. The state constraints can be further classified into conventional dependencies and semantic constraints. Conventional dependencies include functional (and key) dependencies, value bounds, referential constraints, etc. A detailed discussion can be found in [39]. Semantic constraints represent inter-relationships between chunks of data across the database relations.

In this work, we utilize two types of semantic constraints, viz. *subset constraints* S and *implication constraints* I, defined over the database scheme D. In other words our *complete semantic specification* has three components, D, S, and I. Clausal forms are used to represent both types of constraints. Both types of the semantic constraints of our interest can be represented by integrity constraints (conjunctions of negated predicates). In other words, we do not consider the other variants of horn clauses ("facts" or "rules") for constraint specification. Usually, semantic integrity constraints contain relational predicates as well as evaluable predicates.

D. Subset Constraints

Definition 3.1: The set of subset constraints S is a superset–subset relationship between the domains of two different attributes of possibly two different relations.

A subset constraint is represented by an integrity constraint (a conjunction of negated predicates) having two relational predicates and one evaluable predicate. The evaluable predicate specifies a set comparison between two attributes of the relations. Note that no restriction (constant substitution) is allowed on any attribute variables of the relational predicates.

An example of a subset constraint is "$r_1(X_1, Y_1, Z_1), r_2(X_2, Y_2, Z_2), X_1 \subseteq X_2 \rightarrow$", which is the same as "$r_1.X_1 \subseteq r_2.X_2 \rightarrow$" if we decide to prefix the attributes by the relation names. This constraint states that the condition "$r_1.X_1 \subseteq r_2.X_2$" is always false. In other words, it restricts the domain of $r_2.X_2$ to be a subset of the do-

main of $r_1.X_1$. If the evaluable predicate is complemented and moved to the head, the specification becomes "$\rightarrow r_2.X_2 \subseteq r_1.X_1$", equivalent to saying that domain of $r_1.X_1$ is a superset of that of $r_2.X_2$. Note that, as in a query, shared variables can be used in a subset constraint to specify equality implicitly.

A classic example of subset constraint is: "all managers are employees".

E. Implication Constraints

Formally, an implication constraint is represented by a clausal *integrity constraint* (a horn clause with no positive literal, and one or more negative literals), which is a conjunction of *negated predicates*. In the most basic form, these predicates could be *relational predicates* or *evaluable predicates*. The relational predicates represent the database relations (or views), whereas the evaluable predicates represent comparison between a variable and a constant, simple comparison between two variables, or comparison between two variables with an offset [13], [30].

Implication constraints restrict the relative domains of attributes. They specify valid ranges of values that certain attributes can have when some other attributes are restricted in the same or a different relation.

As an example, the constraint "Only managers make more than 40K" on the employee relation can be represented as

employee(Ssn,Name,Dept,Job,Grade,Sal,Bonus,Age),
Sal > 40K → Job = "Manager"

Here, "employee(. . .)" is the relational predicate and the other two are the evaluable ones. In our discussion, we eliminate the explicit representation of relational predicates and prefix the attributes with relation names for improving the readability without losing any generality. The above example with such a representation would be

employee.Sal > 40K → employee.Job = "Manager".

Here we introduce a simple generalization to the above representation. We complement and move the "consequent" predicate (employee.Job = "Manager") to the "antecedent" side, thus making it a "true" clausal integrity constraint (with no positive literal). The resultant representation in our example is

employee.Sal > 40K, employee.Job ≠ "Manager" →.

The constraint now can be read as 'there is no tuple in the employee relation with Sal > 40K and Job ≠ "Manager"'. There are two definite advantages to this modification. First, it simplifies the syntax of the constraint by associating it to a *simple conjunctive set of predicates*, thus eliminating the classification of "antecedent-consequent" atoms. Second, it generalizes the semantics of the constraint as any of the predicates in the conjunctive set qualifies to be the consequent one when complemented and moved to the "consequent" side. For example, from the above conjunctive set, we can also derive

employee.Job ≠ "Manager" → employee.Sal ≤ 40K

Semantically, an implication constraint represents an *impossible conjunctive combination*. From the database point of view, the conjunction represented by an implication constraint always evaluates to be false.

An implication constraint is said to be *local* if all its relational predicates refer to the same relation. Otherwise it is called a *cross* constraint because, in such cases, the implication relates more than one relation. The cross implication constraints involve at least one join specification between relations.

F. Example Database and Constraints

As a running example throughout the paper, we use the following schema and constraints.

Schema and Relation Sizes: (Indexed attributes are underlined)

employee (Ssn, Name, Dept, Job, Grade, Sal, Bonus, Age) [size: 37043 tuples]
storage (Dept, Material, Qty) [size: 1601 tuples]
material (Material, Risk, Storage_Limit) [size 1801 tuples]

Subset Constraints:

storage.Material is a subset of material.Material

Implication Constraints, Implicative and Conjunctive Forms:

IC_1: Only managers make more than 40K.

employee.Sal > 40K → employee.Job = "Manager".
employee.Sal > 40K, employee.Job ≠ "Manager" →.

IC_2: All managers are of grade 20 or higher.

employee.Job = "Manager" → employee.Grade ≥ 20.
employee.Job = "Manager", employee.Grade < 20 →.

IC_3: All materials stored in department d1 are of risk greater than 3.

storage.Dept = "d1", storage.Material = material.Material → material.Risk > 3.
storage.Dept = "d1", storage.Material = material.Material, material.Risk ≤ 3 →.

IC_4: Benzene is always stored in quantities more than 500.

storage.Material = "Benzene" → storage.Qty > 500.
storage.Material = "Benzene", storage.Qty ≤ 500 →.

IC_5: Employees of any department that stores anything in >600 are of age >35.

storage.Qty > 600, storage.Dept = employee.Dept → employee.Age > 35
storage.Qty > 600, storage.Dept = employee.Dept, employee.Age ≤ 35 →.

IV. Maintenance Algorithm for Integrity Constraints

Maintenance of semantic integrity constraints represents an isolated but important component of any seman-

tic query optimization system [29]. Even though the semantic constraints are relatively less frequently updated in comparison to the data itself, an efficient module to manage the semantic modifications is an important part of the optimizer.

A. Subset Constraints

Maintenance of subset constraints is relatively simple. Subset constraints contain exactly two relational attributes and one of the set-comparison operators $\{ \subset, \subseteq \}$.

Task: Accept a subset constraint if and only if it is not redundant and contradicting. The constraint is said to be redundant if it can be derived from the existing constraint set. It is said to be contradicting if, when combined with one or more existing constraints, it produces a null result.

As an example, if "$\rightarrow r.A \subset r.B$" and "$\rightarrow r.B \subset r.C$" are present in the existing set, a new constraint "$\rightarrow r.A \subset r.C$" is redundant. On the other hand, the constraint "$\rightarrow r.C \subset r.A$" is contradicting.

Assumption: It is assumed that the existing set of subset constraints is free from redundancy and contradiction. This assumption is trivially true for a null constraint set.

Data Structures: A directed and labeled graph, $G_s = (V_s, E_s)$, represents the subset constraints. V_s is the set of relational attributes involved in any of the existing subset constraints. A directed edge e exists in E_s from v_1 to v_2 (for v_1, v_2 in V_s) iff the subset constraint "$\rightarrow v_2 \subseteq v_1$" or "$\rightarrow v_2 \subset v_2$" is present in the existing subset constraint set. The label is used to denote the operator $\subset$ or $\subseteq$. If there are parallel directed edges of the same label between two vertices, they are replaced by a single edge with the same direction and the same label. If they are of different labels, they are replaced by a single edge with the same direction and "$\subset$" label. Label "$\subset$" is said to *predominate* the label "$\subseteq$". A directed path in the graph is a sequence of directed edges. A path is said to be of $\subseteq$-type if all the edges in that path are labeled "$\subseteq$". Otherwise, it is called a $\subset$-type path.

Algorithm:

Add the vertices v_1, v_2, corresponding the new edge (from v_1 to v_2) to the graph, unless they are already present.

If the new edge forms a directed loop with at least one "$\subset$" edge, it is a contradicting one.

If there already exists a directed path from v_1 to v_2 of the same or predominating type, then the new edge is redundant.

An edge is accepted to the graph (and the corresponding constraint to the semantic set) iff it is neither contradicting nor redundant.

B. Implication Constraints

From here onwards, we denote an integrity constraint by its conjunctive representation and use the words "constraint" and "conjunction" interchangeably, even though the constraint corresponding to the conjunction "C" is "$C \rightarrow$". Also, we just use "C" in place of "$C \rightarrow$" to designate a constraint whenever the meaning is unambiguous. In any case, "C" represents a conjunction of negated evaluable predicates (a clausal integrity constraint). If a set of predicates c is removed from a conjunction C, the resulting conjunction is denoted as $C - c$. Negation of a conjunction of predicates c is denoted as $\neg c$.

1) Deduction System for New Constraints: Using various rules of the first order predicate calculus, we can deduce new (redundant) constraints from the existing constraint set. Below we list a set of axioms and inference rules for a deduction system, adapted from Gentzens' work [9], [24], which is sometimes referred to as a natural deduction system for first order predicate calculus. The inference rules are divided into two parts: 1) the axioms and basic rules; and 2) rules for the connectives. Each rule is of the form $A \Rightarrow B$, where A and B are conjunctions of predicates, stating that the conjunction B is an integrity constraint if the conjunction A is an integrity constraint.

The rules listed below are from [24] with a slightly modified notation.

The Axioms and Basic Rules:

1) $(A \rightarrow) \Rightarrow (A, B \rightarrow)$
   (clause introduction)
2) $(A, B \rightarrow), (\neg A, B \rightarrow) \Rightarrow (B \rightarrow)$
   (clause elimination)

Rules for the Connectives:

3) $(A \rightarrow) \Rightarrow (A, B \rightarrow)$
   (& introduction)
4) $(A, C \rightarrow), (B, C \rightarrow), (\neg A \rightarrow \vee \neg B \rightarrow) \Rightarrow (C \rightarrow)$
   (& elimination)
5) $(A \rightarrow), (B \rightarrow) \Rightarrow (A \vee B \rightarrow)$
   ($\vee$ introduction)
6) $(A \vee B) \Rightarrow (A \rightarrow)$
   ($\vee$ elimination)
7) $(A, \neg B \rightarrow), (A, B \rightarrow) \Rightarrow (\rightarrow \neg A)$
   ($\neg$ introduction)
8) $(\neg A \rightarrow), (A \rightarrow) \Rightarrow (\rightarrow B)$
   ($\neg$ elimination)

It is important to note that the above set of rules represents a *complete deduction system* for propositional calculus [25], [24]. In other words, every valid implication integrity constraint C, with respect to a given constraint set S, can be deduced using the above inference rules. It is possible to derive other inference rules (e.g., transitivity of implication) from the above basic set. Refer to [24] for a detailed discussion.

In the above list, the connectivity rules 3)–8) can be derived from the set of basic rules 1), 2), definition of implication operation (i.e., $A \rightarrow B$ can be defined as $\neg A \vee B$), and DeMorgan's laws. The rules of the basic set itself belong to two categories, augmentation and transitivity.

Augmentation (Rule 1) refers to the uninteresting process of appending arbitrary predicates to an existing conjunction. Since the existing conjunction—being an integrity constraint—always evaluates to false, all of its superset conjunctions also evaluate to false, thus technically qualifying to be integrity constraints. In rule 1, "A"

represents the existing conjunction and "B" represents a conjunction of arbitrary predicates. As an example, consider an integrity constraint 'employee.Sal > 40 K, employee.Job ≠ "Manager"' which states that everyone who makes more than 40K is a manager. It is trivial to generate the constraint by combining the predicate 'employee.Age > 40' to the above constraint to get "employee.Sal > 40K, employee.Age > 40, employee.Job ≠ "Manager"', which states that everyone who makes more than 40K and is over 40 years is a manager.

Transitivity (Rule 2) the other way of generating new constraints depends on the transitivity of the implication operation. Two constraints, C_1 and C_2, can be used to transitively generate a new one if there exists conjunctions c_1, c_2, where c_1 is a subset of C_1 and c_2 a subset of C_2, such that c_1 and c_2 are complements of each other. The new constraint C is defined as the conjunction of all the predicates from C_1 and C_2 excluding the ones in c_1 and c_2. In rule 2, "A, B" and "$\neg A$, B" represent C_1 and C_2, respectively, with "A" representing c_1 and "$\neg A$" representing c_2, to generate a new constraint "B" representing C. For example, consider the constraints IC_1 and IC_2 of the example database

IC_1: employee.Sal > 40K, employee.Job ≠ "Manager" →.

IC_2: employee.Job = "Manager", employee.Grade < 20 →.

IC_1 and IC_2 can be used (with c_1 = 'employee.Job ≠ "Manager"' and c_2 'employee.Job = "Manager"') to obtain

C: employee.Sal > 40K, employee.Grade < 20 →.

The proof that C is an integrity constraint is as follows:

Since C_1 is an integrity constraint, it is true that $(C_1 - c_1) \rightarrow \neg (c_1)$.

Since C_2 is an integrity constraint, it is true that $(c_2) \rightarrow \neg (C_2 - c_2)$.

Since $c_2 = \neg c_1$, and by transitivity, it is true that $(C_1 - c_1) \rightarrow \neg (C_2 - c_2)$.

In other words, "$(C_1 - c_1), (C_2 - c_2)$", which is C, is an integrity constraint.

Generating weaker constraints from the existing ones can be technically considered as a special case of transitivity involving *tautology conjunctions*. Tautology conjunctions are the ones that are always satisfiable as integrity constraints, independent of the database context. For example, the conjunction "employee.Grade > 20, employee.Grade < 18" is a tautology, as it represents the implication "employee.Grade > 20 → employee.Grade > 18", or equivalently, "employee.Grade < 18 → employee.Grade < 20". This tautology conjunction can be used with 'employee.Job = "Manager", employee.Grade < 20' to generate a weaker constraint 'employee.Job = "Manager", employee.Grade < 18'

It is also easy to observe from the above discussion that a conjunction C is implied by a set of conjunctions S iff a subset of C is implied by S.

Definition: A conjunction of predicates (integrity constraint) C is said to be *in contradiction* with respect to a set of conjunctions or predicates S if $\neg (C)$ is implied by S.

2) Maintenance of Implication Constraints: The task of constraint maintenance is to ensure that the constraint set is always free from redundancy and contradiction. This assumption is trivially true with empty constraint sets. If S is the existing set of constraints and C is the new constraint, C is tested for redundancy (i.e., $S \rightarrow C$) and contradiction (i.e., $S \rightarrow \neg C$) before acceptance. If accepted, existing constraints are tested for redundancy with C, and removed from the set if found redundant. As an example, assume the following two constraints exist in the semantic set

employee.Sal > 40K → employee.Job = "Manager"
employee.Sal > 40K → employee.Grade > 20

Consider a new constraint 'employee.Job = "Manager" → employee.Grade > 20', which is neither redundant nor contradicting. However, when added to the set, it makes the constraint 'employee.Sal > 40K → employee.Grade > 20' redundant.

Since the constraint set may contain constraints that do not contribute to making the new constraint redundant (or contradicting), we first identify the *relevant subset* of constraint set for this purpose. This identification restricts the constraints to be considered for redundancy/contradiction checking, thus, reducing its complexity. As a first step, the existing constraint set is partitioned into equivalence classes.

Equivalence Classes: The set $S = \{IC_1, IC_2, \ldots, IC_n\}$ of existing constraints is partitioned into equivalence classes such that any two constraints belong to the same equivalence class iff they have at least one variable (relational attribute) in common.

A variable (relational attribute) is said to *belong* to an equivalence class iff it belongs to a constraint that belongs to that equivalence class.

The example constraint set of the previous section can be partitioned into two equivalence classes, E_1, E_2, as

E_1 Containing Constraints:

IC_1: employee.Sal > 40K, employee.Job ≠ "Manager" →

IC_2: employee.Job = "Manager", employee.Grade < 20 →

E_2 Containing Constraints:

IC_3: storage.Dept = d1, storage.Material = material.Material, material.Risk ≤ 3 →

IC_4: storage.Material = "Benzene", storage.Qty ≤ 500 →

IC_5: storage.Qty > 600, storage.Dept = employee.Dept, employee.Age ≤ 35 →

Variables belonging to E_1 are

employee.Job, employee.Grade, employee.Sal

and those belonging to E_2 are

employee.Dept, employee.Age, storage.Dept, storage.Material,
storage.Qty, material.Material, material.Risk

Lemma 4.1: If C' is a minimal subset of a constraint C (or $\neg C$) implied by S, then it is implied by the constraints belonging to the one and only equivalence class that contains all the variables appearing in that subset.

Proof: Since C' is the minimal subset of C implied by S, it is not generated by augmentations. So C' is an existing constraint itself, or is generated by transitivity. If it is an existing constraint, the proof is trivial. Assume that it is a transitively generated one. No constraints from different equivalence classes can transitively interact because they do not have any variables in common. In other words, all the constraints contributing to the redundancy of C' must belong to a single equivalence class—the one that contains all the variables of C'.

Set of Relevant Constraints: We now identify a *set of relevant constraints* S_r (from S) for C such that $S \rightarrow C$ (or $\neg C$) iff $S_r \rightarrow C$ (or $\neg C$).

Case 1: All variables of C belong to the same equivalence class.

The set of relevant constraints for C is $E(C)$.

Case 2: Variables of C belong to more than one equivalence class, say $E_1, \ldots, E_n$.

There are two types of predicates in C.

Type 1: Predicates with all the variables from the same equivalent class.

Type 2: Predicates with variables from different equivalent classes.

Let P_0 be the set of all the type-2 predicates.

Partition $C - P_0$ to $P_1, \ldots, P_n$ such that all variables of each P_i belong to exactly one equivalence class.

Let the equivalence class containing the variables of P_i be denoted by $E(P_i)$, for $1 \leq i \leq n$.

The set of relevant constraints for P_i is $E(P_i)$, $1 \leq i \leq n$.

The following is a set of illustrative examples.

Example 1:

Let C be 'employee.Sal > 40K, employee.Grade < 20'.
All the variables of C belong to one equivalent class $E(C)$, i.e., E_1.

Note that E_1 is
IC_1: 'employee.Sal > 40K, employee.Job ≠ "Manager" →',
IC_2: 'employee.Job = "Manager", employee.Grade < 20 →'

The task is to verify whether $E_1 \rightarrow C$.
If $E_1 \rightarrow C$, the constraint C is redundant.
In this particular example, since $E_1 \rightarrow C$, the constraint C is redundant.

Example 2:

Let C be 'employee.Dept = storage.Dept, employee.Age > 40, employee.grade < 18'
Variables of C belong to different equivalent classes.
Type 1 predicates: employee.Age > 40, employee.Grade < 18
Type 2 predicates: employee.Dept = storage.Dept
P_0: employee.Dept = storage.Dept
P_1: employee.Age > 40
P_2: employee.Grade < 18

Note that $E(P_1)$ is E_1, i.e.,
IC_1: employee.Sal > 40K, employee.Job ≠ "Manager" →
IC_2: employee.Job = "Manager", employee.Grade < 20 →

Note that $E(P_2)$ is E_2, i.e.,
IC_3: storage.Dept=d1, storage.Material=material.Material, material.Risk≤3 →
IC_4: storage.Material = "Benzene", storage.Qty ≤ 500 →
IC_5: storage.Qty>600, storage.Dept=employee.Dept, employee.Age≤35 →

The task is to verify whether $E_1 \rightarrow P_1$ or $E_2 \rightarrow P_2$.
If $E_1 \rightarrow P_1$ or $E_2 \rightarrow P_2$, the constraint C is redundant.

In this example, since neither of the above implications are true, the constraint C is not redundant.

3) Constraint Derivation from the Relevant Set: Once the set of relevant constraints for implication checking is identified, the task is to verify whether the subset actually implies the new constraint (or its complement).

Let the set of relevant constraints contain '$C_1\rightarrow$', $\cdots$, '$C_n\rightarrow$'. If the new constraint '$C\rightarrow$' is implied by this set, $\neg(C_1), \cdots, \neg(C_n) \rightarrow \neg(C)$ must be a tautology or, in other words, $\neg(C_1), \ldots, \neg(C_n), C$ must be unsatisfiable. Since C_i's are conjunctions of predicates, "$\neg(C_1), \ldots, \neg(C_n)$" is a conjunction of disjunctions and can be rewritten as a disjunction of conjunctions of predicates from C_i's. Each of the conjunctions will contain n predicates, and there will be $t_1 * \ldots * t_n$ such conjunctions in the disjunction, where t_i is the number of predicates in C_i. When C is conjuncted with disjunction, the resulting disjunction will have the same number of conjunctions, i.e., $t_1 * \cdots * t_n$, each with $n + t$ predicates each, where t is the number of predicates in C.

Similarly, testing whether the set of sufficient constraints implies the complement of the new constraint is equivalent to testing the unsatisfiability of "$\neg(C_1), \ldots, \neg(C_n), \neg(C)$". This represents a disjunction of conjunctions from C_i's and C. Each conjunction will have $n + 1$ predicates, and there will be $t_1 * \cdots * t_n * t$ such conjunctions in the disjunction.

In both of these cases, the number of conjunctions increases *exponentially* with the number of constraints in the set, whereas the number of predicates in each conjunction has a *linear* growth.

For illustration, let C be

'employee.Sal > 40K, employee.Bonus > 25K, employee.grade < 18'
P_0: 'employee.Bonus > 25K'

P_1: 'employee.Sal > 40K, employee.Grade < 20'
$E(P_1)$, which is E_1:
'employee.Sal > 40K, employee.Job = ''Manager''',
'employee.Job ≠ ''Manager'', employee.Grade < 20'

The new constraint C is redundant iff the following is unsatisfiable: (Note: we drop the relation prefix ''employee'' for improved readability.)

¬(Sal > 40K, Job = ''Manager''),
¬(Job ≠ ''Manager'', Grade < 20),
(Sal > 40K, Grade < 18)

This can be simplified by eliminating the predicates appearing in the negated conjunctions which have equivalent ones in the nonnegated conjunction. The result is

¬(Job = ''Manager''),
¬(Job ≠ ''Manager'', Grade < 20),
(Sal < 40K, Grade < 18)

This is the same as

(Job ≠ ''Manager''),
(Job = ''Manager'' OR Grade > 20),
(Sal > 40K, Grade < 18)

which is

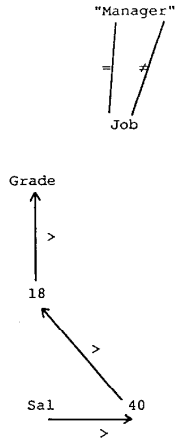
(Job ≠ ''Manager'', Job = ''Manager'', Sal > 40K, Grade < 18) OR
(Job ≠ ''Manager'', Grade ≥ 20, Sal > 40K, Grade < 18)

Here, both the conjunctions in the disjunction evaluate to be false, thus making the disjunction unsatisfiable. This indicates that the new constraint is redundant.

Checking Unsatisfiability: Unsatisfiability of the disjunction depends on unsatisfiability of its conjunctions. Here we sketch a simple algorithm for checking the unsatisfiability of a conjunction.

The conjunction is represented by a directed graph $G = (V, E)$. The *vertex set* V represents the variables, constants, and all the unique combinations of the variables and constants present in the conjunction. Note that we use distinct vertices for different constants belonging to the same domain as well as for the same variable in combination with different constants. The *edge set* E represents explicit as well as implicit comparisons. Explicit comparisons are the ones present in the conjunction. Implicit comparisons are between constants of the same domain and between the same variables in combination with the constants of the same domain.

The edges are labeled by their comparison operator. As discussed in the previous section, the operators are restricted to $>$, $\geq$, and $\neq$. The ''$>$'' and ''$\geq$'' edges are directed, their direction representing the direction of the comparison operator, whereas the ''$\neq$'' are undirected. Distinct vertices are used to represent different constants even though they belong to the same domain. This representation eliminates the need to assign weights to the edges as in [30], [15].



The graph representing the first conjunction of the above example is presented in Fig. 1. Note that the constants (18, 40) are represented by distinct vertices. Also, note the (implicit) edges between the vertices ''18'' and ''40''.

A *directed path* from vertex u to vertex v is the sequence of directed edges $e_1, \ldots, e_k$, $k \geq 1$, such that there exists a corresponding sequence of vertices $v_0, v_1, \ldots, v_p$ ($u = v_0$, $v = v_p$) satisfying $e_k = (v_{k-1}, v_k)$, for $0 < k \leq p$.

We claim that the graph (and the conjunction) is unsatisfiable if and only if any of the following conditions is true.

1) For any ''$>$'' edge, say from vertex a to vertex b, there is a directed path from vertex b to vertex a.

2) For any ''$\neq$'' edge, say between the vertices a and b, equality between a and b is implied by the conjunction, by a directed cycle of ''$\geq$'' edges involving a and b.

It is obvious that any of these conditions is sufficient to imply unsatisfiability. To see the necessity of these conditions for unsatisfiability, consider a graph where none of these conditions is true. Existence of no ''$>$'' edge is in contradiction because the only way for such a contradiction is a reverse directed path. For ''$\neq$'' edges, it is easy to see that no two edges interact to generate new information. (In other words, ''$a \neq b$'' and ''$b \neq c$'' do *not* imply anything between ''a'' and ''c''.) The only situation where a ''$\neq$'' edge can contribute to contradiction is when it is shunted by an (implied) equality between its vertices. Assuming that the size of the domain of the variables is much larger than the number of variables itself, we can always assign different values to the variables to simultaneously satisfy all the inequalities.

Verification of both of the above conditions basically requires verifying reachability of a specific vertex from another one using selected types of edges. This can be achieved in various ways, by finding transitive closure, shortest paths, or transitive reduction [42], all of which take time proportional to $O(n^3)$ where n is the number of vertices in the graph.

A related method is discussed in [30]. In that paper [30, theorems 21, 22, p. 70], a proof is sketched to show

that satisfiability of conjunction is NP complete when (and only when) "$\neq$" comparison is allowed between the variables. In that proof, satisfiability of the subgraph containing only "$\neq$" edges is reduced to k-colorability, thus concluding the NP completeness. However, reduction to k-colorability holds only if the cardinality of the domain of the variables is *less than* the number of variables in the subgraph—an assumption which is rarely true in any practical situation. If the cardinality of the domain is greater than or equal to the number of variables in the subgraph, colorability (and hence satisfiability) of the subgraph becomes polynomial—by assigning all different colors to different vertices, since we have sufficiently different colors. Satisfiability of general conjunctive predicates (with inequality comparisons) is shown to be polynomial in [19], when the size of the domain of attributes is greater than the number of variables used in the query for that domain. Reference [38] discusses various cases of the implication problem by converting them into a satisfiability problem, based on the above-mentioned results from [30].

V. Heuristics and Inference Rules

Before formally describing the utilization of constraints in semantic query transformation, we present a brief overview of various heuristic and inference rules used in semantic optimization. The illustrations are based on the example database presented in Section III-F.

We use four heuristic rules as suggested in [18], namely, *restriction elimination*, *index introduction*, *scan reduction*, and *join elimination*. The heuristic strategy of join introduction as in [18] is not used in our approach. In the following illustration we use a quel-like language [36] for expressing queries.

Restriction Elimination: Remove a restriction from the query, if found redundant.

Query Q_1: List all the departments that store benzene in more qty more than 400.
Quel Form: retrieve (storage.Dept) where storage.Material = "Benzene" and storage.Qty > 400.
Rule(s): storage.Material = "Benzene" → storage.Qty > 500.
Query Q_1': retrieve (storage.Dept) where storage.Material = "Benzene".
Result: The unnecessary restriction on the attribute "qty" of the relation "storage" is eliminated.

Index Introduction: Introduce a restriction on an indexed attribute, if implied by the query.

Query Q_2: Find all the employees who make more than 42K.
Quel Form: retrieve (employee.Ssn, employee.Name) where employee.Salary > 42K.
Rule(s): employee.Salary > 42K → employee.job = "Manager".
Query Q_2': retrieve (employee.Ssn, Name) where employee.Salary > 42K and employee.Job = "Manager".
Result: A new constraint is obtained on the indexed attribute "Job" of the relation "employee".

Scan Reduction: Reduce the number of inner scans of the join by obtaining additional restrictions prior to the cross referencing operation.

Query Q_3: List all employees working in departments storing anything in > 625.
Quel Form: retrieve (employee.Ssn, employee.Name) where employee.Dept = storage.Dept and storage.Qty > 625.
Rule(s): storage.Qty > 600, storage.Dept = employee.Dept → employee.Age > 35.
Query Q_3': retrieve (employee.Ssn, employee.Name) where employee.Dept = storage.Dept and storage.Qty > 625 and employee.Age > 35.
Result: The new constraint on attribute "Age" of relation "employee" can be applied to the relation prior to the cross matching step of its join to the relation "storage", thus reducing the qualifying tuples from the relation "employee" and hence the number of scans of the relation "storage."

Join Elimination: Eliminate a relation if it is joined to just another relation and none of its attributes contribute to the output.

Query Q_4: Get all the materials stored in "d1", in qty > 400, of risk > 2.
Quel Form: retrieve (storage.material) where storage.dept = "d1" and storage.qty > 400 and storage.Material = material.Material and material.Risk > 2.
Rule(s): storage.Dept = "d1", storage.Material = material.Material → material.Risk > 3
material.Material is a superset of storage.Material
Query Q_4': retrieve (storage.Material) where storage.Dept = "d1" and storage.Qty > 625.
Result: Join with the relation "material" is eliminated.

VI. Graph Representations of a Query

A *query* Q is a conjunction of join specifications of the form "$r_1.A_1$ *op* $r_2.A_2$" and the restriction specifications of the form "$r_1.A_1$ *op* k" where r_1, r_2 are relations, A_1, A_2 are attributes, k is a constant, and *op* is one of the comparison operators $\{\neq, \geq, >\}$. The operators "$<$"

and "$\leq$" are not explicitly considered because $a \leq b$ and $a < b$ are the same as $b \geq a$ and $b > a$, respectively. Similarly, the equality operator "$=$" is not explicitly represented because $a = b$ can be replaced by the conjunction of $a \geq b$ and $b \geq a$. This limits the set of operators to have only "$\neq$", "$\geq$", and "$>$". The *answer* of a query Q is the set of all tuples of the relations referenced in Q that satisfy Q, projected on the specified target attributes of Q.

Query Graph G_q: A query Q is represented by a *query graph* G_q which is a directed graph whose vertices are the attributes of the relations (attribute vertices) as well as the constants (constant vertices) involved in Q. The edges of G_q are the join and restriction specifications in Q. A join specification "$r_1.A_1$ *op* $r_2.A_2$" is represented by an edge from $r_1.A_1$ to $r_2.A_2$ with a label *op*. Similarly, a restriction specification "$r_1.A_1$ *op* k" is represented by an edge from $r_1.A_1$ to the constant k with a label *op*. The direction of an edge identifies the left and right operands of the label associated with it. The edges representing a join specification are referred to as "*join edges*" whereas the ones denoting the restrictions are called "*restriction (constant) edges*".

Consider the query Q_4 of Section V for Fig. 2.

Query Q_4: Get all the materials stored in "d1", in qty $>$ 400, of risk $>$ 2.

Quel Form: retrieve (storage.Material) where storage.Dept = "d1" and storage.Qty $>$ 400 and storage.Material = material.Material and material.Risk $>$ 2.

Graph G_q: (Indexed attributes are underlined, target attributes are identified by "?". For the sake of clarity, equalities are represented by undirected single edges rather than pairs of "$\geq$" edges of opposite directions.)

Canonical Condensed Graph G_c: Since a given query can have syntactically different, but semantically equivalent, forms and hence different equivalent query graphs, it becomes necessary to arrive at a canonical form of representation before processing the query. We adopt the notion of a *condensed graph* G_c as the canonical representation of a query. In its condensed form, a query graph is represented by a minimal set of join and restriction edges.

The condensed graph G_c is derived from the query graph G_q by first grouping the attribute vertices of G_q into *equivalence classes*. Any two vertices of G_q belong to the same equivalence class if they are connected by explicit equality edges or if there is a directed cycle of join edges incident with both the vertices. A vertex forms an equivalence class by itself if it is not a part of any equijoin.

The equivalence classes (as well as the constant vertices) of G_q are mapped as vertices in G_c as follows.

The nonequijoin ($\geq$, $>$) edges of G_q that are not part of an equality specification are represented in G_c by their *transitive reduction*. The transitive reduction [42] of a query graph with only join edges is a graph with the fewest number of join edges among all such query graphs having the same transitive closure. The transitive reduction is obtained by first mapping all the nonequijoin edges from G_q to G_c for the corresponding equivalence class vertices, removing all the redundant edges from G_c, and then replacing any multiple edges between two vertices by an equivalent single edge.

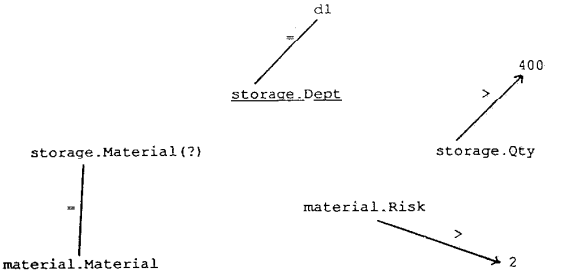


Fig. 2. Query graph G_q.

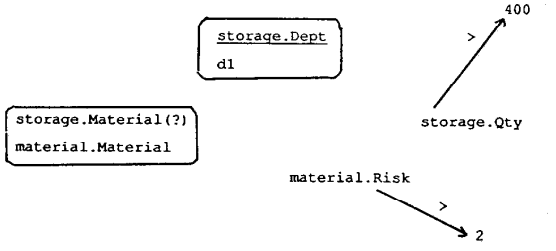


Fig. 3. Canonical condensed graph G_c.

The restriction (constant) edges of G_q are represented in G_c by their transitive reduction over the join edges. For this, first each restriction edge of G_q is mapped into G_c to restrict the vertex corresponding to the equivalence class. It is possible that addition of restriction edges could lead to the merger of equivalence classes. With each such addition, all the restriction and join edges in that connected component are tested for syntactic redundancy, and the redundant ones are removed from the graph. For example, with an existing join edge of "$A > B$" and a restriction edge of "$A < 3$", the addition of the restriction edge "$B > 3$" makes the existing restriction edge "$A < 3$" syntactically redundant and removable.

The condensed graph G_c for the query graph G_q illustrated above is shown in Fig. 3. The result of this condensation is that the vertices (storage.material, material.material) and (storage.dept, d1) that are connected with equality edges in G_q form single multimember nodes in G_c.

VII. Sequential Phases of Semantic Transformation

Semantic query transformation is the process of obtaining alternative query forms that are semantically equivalent to the original one. The motivation of semantic optimization is to arrive at a more profitable query yielding

the same answer, which could be syntactically different from the original query.

In our approach, semantic optimization of a query consists of two major phases, namely, semantic expansion, and semantic reduction. Semantic reduction is composed of two stages, namely, relation elimination and edge elimination. These as well as other auxiliary steps are described below.

A. Derivation of Canonical Condensed Form

This first step, as described in the previous section, obtains a canonical representation G_c of the query through transitive reduction of join and restriction edges present in G_q. The transitive reduction property of the graph is then retained by the query transformation algorithm in all its following stages by removing syntactically redundant edges and/or merging the equivalent classes. This first stage is independent of any semantic details and depends only on the query and the operator syntax. Besides arriving at a canonical form of the query, this stage facilitates any early detection of contradictions in join or restriction specifications that could lead to a null answer.

B. Semantic Expansion

Semantic expansion iteratively adds any new restriction or join edges implied by the combination of (condensed) query graph and semantic implication constraints. This is achieved by identifying the implication constraints whose antecedent atom(s) are satisfied by the graph and adding the restriction or join edges corresponding to their consequent atom to the query. Each time, the transitive reduction property of the graph is restored if the added edge happens to violate it. From the original form, addition of each such edge takes the query graph through various semantically equivalent forms until it reaches a stage G_m where no more new restrictions or joins could be implied.

The purpose of semantic expansion is to incorporate any useful restrictions (possibly on indexed attributes) or joins that are not present in the original query. This assures that the query contains a semantically maximal (and syntactically minimal) set of edges that satisfy both the query and implication constraints.

Semantic expansion of the condensed graph is illustrated in Fig. 4. Both the antecedent atoms of the second implication constraint (i.e., "storage.Dept = d1" and "storage.Material = material.Material") are satisfied by the condensed graph, thus making it possible to add the corresponding consequent atom (i.e., "material.Risk > 3") to the graph. Due to the transitive reduction property of the graph, the added edge "material.Risk > 3" supersedes, and hence eliminates the existing one "material.Risk > 2".

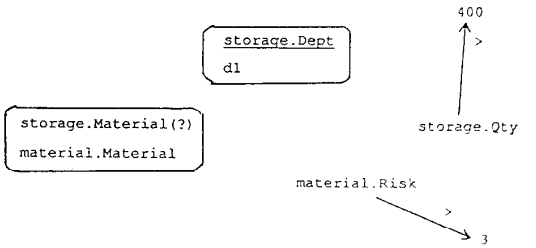


Fig. 4. Semantic expansion G_m.

C. Relation Elimination

Relation elimination stage identifies all semantically redundant relations from G_m. Relations identified to be redundant, if any, are removed from the query graph. A relation is considered to be redundant if it becomes dangling so that none of its attributes or restrictions contribute to the answer. Since the query graph is connected, elimination of a relation leads to the removal of its join to the rest of the query graph. A relation elimination is hence considered to be profitable because it eliminates the need of performing a join. The graph, when all the redundant relations are removed from G_m, is denoted by G_{r1}.

There are various conditions that a relation should satisfy in order for it to be classified as redundant. First of all, it should be free from any target attributes of the query because a relation containing target attributes cannot be removed from a query. The second condition is that all the restrictions on nonjoin attributes should be redundant. By this, all such restrictions could be removed without altering the query semantics. In this stage, the relation will have restrictions, if any, only on join attributes. The third condition is that the relation should have at most one join vertex, and the fourth condition is that the relation does not have any nonequijoins. The third and fourth conditions allow the transfer of all the restrictions on the only join attribute to the other side of the respective joins. This makes the relation free from all restrictions. Finally, there should be at least one other relation with a join attribute, say "$S.B$", in the equivalence class containing the join attribute of this relation, say "$R.A$", such that the subset constraint "$R.A$ is a super set of $S.B$" holds.

More formally, a relation R is redundant if it satisfies all of the following conditions.

a) R is target-free.

b) All the restriction edges on nonjoin vertices of R are redundant.

c) R has at most one join attribute.

d) R does not have any nonequijoins.

e) There is at least one other relation with a join attribute, say "$S.B$", in the join class containing the join attribute of R, say "$R.A$", such that the subset constraint "$R.A$ is a super set of $S.B$" holds.

Relation elimination of the graph G_m illustrated above is as shown in Fig. 5. The relation "material" gets qualified as redundant due to the conditions described above. It is free from any target attributes [condition a], and the restriction "material.risk > 3" is semantically redundant [condition b]. The only join attribute of the relation is "material.material" [condition c] and it is an equijoin [condition d]. The subset constraint "material.material is a superset of storage.material" [condition e] completes

Fig. 5. Relation elimination G_{r1}.

the requirements for making the relation "material" semantically redundant, and hence removable from the query.

D. Edge Elimination

A restriction or join edge is redundant if it is satisfied by the consequent atom of an implication constraint of which all the antecedent atoms are satisfied by the query. Semantically redundant join edges can always be removed from the graph, since their basic purpose is to aid semantic expansion by providing additional paths for information flow. The strategy of removing a redundant restriction edge from a relation largely depends on whether selections will be performed in that relation before joins. This, in turn, depends on whether the join attribute in that relation is indexed or not. If the relation has at least one indexed join attribute, it is assumed that the restrictions in that relation will be performed along with the join, and not before it. This is because, with a given set of matching values for the join attribute, location of tuples becomes easy through the (indexed) join attribute and the restriction(s) could be checked during the same time. On the other hand, if no join attribute of the relation is indexed, we assume that the selections in that relation will be performed before joins.

In the cases where selections are performed before joins, locally redundant restriction edges on indexed attributes become profitable provided none of the antecedent atoms of the corresponding local implication constraint are on indexed attributes. This is because the redundant restriction introduces an indexed scan to replace the sequential scan of the relation. Similarly, all the cross redundant restrictions become profitable if selections are performed before joins. The reason is that such a restriction additionally limits the effective size of the relation before the join operation, thus resulting in a scan reduction. A restriction, even though redundant, is considered to be profitable if it is on a join attribute since it may provide a better join strategy.

The graph resulting from deleting all nonprofitable edges from G_{r1} is denoted as G_{r2}. For the query graph G_{r1} shown in the above example, no edge is qualified for elimination. That is, G_{r2} is the same as G_{r1} in this case.

Unlike in the case of relation elimination where all the redundant joins are removed, redundant edges are retained if they are found profitable. But identifying a restriction to be profitable, as mentioned above, depends mainly on the estimation of the sequence of selections and joins. This sequence, in reality, is determined by various factors outside the scope of this work, like relation sizes, optimizer statistics, optimizer intelligence, and sequence of specification of equality joins. This might result in erroneous classification of profitable restriction at times. But by and large, this strategy provides a simple and reliable method to achieve the identification.

E. Conversion from the Condensed Query Graph

When the semantic expansion and reduction are completed, the query graph is converted back from its condensed form to the original one. This is achieved by replacing each multimember node of G_{r2} by any spanning tree on its attribute vertices connected by equijoin edges. Any one attribute vertex, an indexed one if available, of the multimember node is chosen for joining the spanning tree to other spanning trees or single attribute vertices. Also, the restriction(s) on the multimember node are mapped as restriction(s) on all the attribute vertices. This form of the graph, being the final one, is denoted by G_f.

Note that, as in the case of restriction elimination, this strategy of graph conversion could also produce suboptimal results. The cost of equijoins involving three or more attribute vertices depends on the edges in the corresponding spanning tree as well as the order in which they are considered for join. Similarly, selecting an attribute vertex in the spanning tree to join with other vertices could also make a difference in cost. For example, while selecting the edges of the spanning tree, priorities are given to the attributes of these relations which have one or more other joins between them. In general, issues like relation sizes and selectivities should also be considered in selecting the spanning tree for multimember equivalence classes.

The converted form G_f of the graph G_{r1} is as illustrated in Fig. 6. The result is the replacement of the multimember node (storage.dept, d1) by its spanning tree. This final graph can be translated to a quel statement:

```
retrieve (storage.material) where
storage.dept = d1 and
storage.qty > 400
```

The end result of the query transformation process, in this example, is elimination of the relation "material" from the original user-specified query.

VIII. Transformation Algorithm and Implementation Details

In this section we formalize the algorithm for semantic transformation, and discuss its correctness and cost saving. We also present its implementation architecture and examine the test results.

A. Algorithm for Query Transformation

/* Stage 1: Obtain canonical condensed form—Construct G_c from G_q: */

a) (Map the Vertex Set): For each connected component c in G_q, partition the vertices into equivalence classes so that any two vertices are in the same equivalent class if they are connected by an explicit equijoin edge or if there is a directed join cycle with only "$\geq$" edges in-

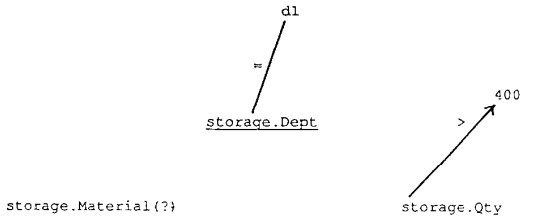


Fig. 6. Converted form G_f.

cident at both of them. A vertex forms an equivalence class by itself if it is not a part of any equijoins. These equivalence classes (as well as constants) of G_q are mapped as vertices of G_c.

b) (Map the Edge Set): For any two equivalence classes E_i, E_j, $i \neq j$, in G_q, let S be the set of edges between the vertices in E_i and the ones in E_j. Let v_i, v_j be the vertices in G_c corresponding to E_i, E_j. If S_{ij} is empty, then there is no edge between v_i and v_j in G_c. If all the edges in S_{ij} have the same label "$\geq$", or "$>$", then there is a single edge with the same label from v_i to v_j in G_c. If S_{ij} contains edges with different labels, then there is a single edge with the "$>$" label between v_i and v_j (unless there is a contradiction).

c) (Remove the Redundant Edges): The edges in each connected component of G_c are successively examined in any order, and those implied by transitivity are removed.

/* Stage 2: Obtain semantic expansion, G_m: */

a) Mark any unmarked implication constraints whose all predicates but one are currently implied by G_c. (We denote this predicate as an unimplied predicate.) Terminate the stage if no new implication constraint gets qualified for marking.

b) If there is at least one marked but unused implication constraint, add an edge corresponding to its unimplied predicate to G_c.

c) If any edges are added in step b), restore G_c to its transitive reduction. This may include removal of syntactically redundant edges and/or merger of vertices. Repeat step a).

/* Stage 3: Eliminate redundant relations to obtain G_{r1}: */

While there is a relation R that satisfies the following:

a) R is target-free.

b) All the restriction edges on nonjoin vertices of R are redundant.

c) R has at most one join attribute.

d) R does not have any nonequijoins.

e) There is at least one other relation with a join attribute, say "$S.B$", in the join class containing the join attribute of R, say "$R.A$", such that the subset dependency "$R.A \supseteq S.B$" holds.

eliminate R from the query graph.

/* Stage 4: Eliminate redundant edges, to obtain G_{r2}: */

remove all the redundant (those implied by rest of the query graph) join edges.

remove all the redundant (those implied by rest of the query graph) restrictions if they are not profitable.

a) A redundant restriction on a join attribute is profitable.

b) If no join attribute of the relation is indexed, then all the cross redundant restrictions in that relation are profitable.

c) If no join attribute of the relation is indexed, then all locally redundant restrictions in that relation are profitable only if they are on indexed attributes and the corresponding antecedent restrictions of the implication constraints are on nonindexed attributes.

/* Stage 5: Expand multimember nodes of G_{r2} to obtain G_f: */

a) Replace each multimember node of G_c by a spanning tree on its attribute vertices connected by equijoin edges. While selecting vertices of the spanning tree, assign priorities for attributes of those relations which have one or more other joins between them.

b) Select any attribute vertex, an indexed one if available, of the multimember node for joining the spanning tree to other spanning trees or single attribute vertices.

c) Map the restriction(s) of the multimember node as the restriction(s) on all the attribute vertices.

B. Correctness of the Algorithm

Let the original form of the query be denoted by Q_0, the expanded form on completion of the expansion stage of the algorithm by Q_m, and the final transformed form by Q_f.

Theorem 1: The query forms Q_0, Q_m, and Q_f are semantically equivalent.

Proof: $Q_0 \Leftrightarrow Q_m$:

The semantic transformation from Q_0 to Q_m takes place in the expansion stage due to the addition of edges to G_c from the consequent atoms of the implication constraints. (Note that initial conversion from G_q to G_c does not alter the query semantics.) Addition of each such edge to G_c can be assumed to transform the graph to a new query form. The transformation from Q_0 to Q_m thus constitutes a chain, $Q_0 \rightarrow Q_{01} \rightarrow Q_{02} \rightarrow \cdots \rightarrow Q_m$.

Here, the difference between two consecutive forms Q_{0i} and $Q_{0i} + 1$ is at most one edge, say e_i (apart from any differences resulted by restoring the graph to its transitive reduction, which does not alter any semantics of the query). Introduction of e_i to Q_{0i} is due to the presence of a set of edges E_i and Q_i such that there exists an implication constraint $E_i \rightarrow e_i$. So addition of e_i to Q_{0i} does not alter the semantics of Q_{0i}. In other words, Q_{0i} and $Q_{0i} + 1$ are semantically equivalent. Extending the argument for the entire chain of transformations, it can be seen that Q_0 and Q_m are semantically equivalent.

Proof: $Q_m \Leftrightarrow Q_f$:

The transformation from Q_m to Q_f is accomplished in the elimination stage of the algorithm. As above, let us assume that the transformation from Q_m to Q_f can be rep-

resented by a chain, say $Q_m \rightarrow Q_{m1} \rightarrow Q_{m2} \rightarrow \cdots \rightarrow Q_f$.

The transformation from Q_{mi} to $Q_{mi} + 1$ can be due to elimination of a relation (join) or removal of an edge by the semantic reduction stage of the algorithm.

All the relations and edges qualified for elimination are the ones found semantically redundant, and hence their removal does not alter the semantics of the query. Hence, we conclude that Q_{mi} and $Q_{mi} + 1$ are semantically equivalent, implying the semantic equivalence of Q_m and Q_f.

C. Cost Comparison of Query Forms

Theorem 2: Cost (Q_f) < Cost (Q_0) provided the estimation of the selection–join sequence is valid.

If Q_f is different from Q_0, let this difference be represented by three components: 1) set of edges E_f^+ that are present in Q_f but not in Q_0; 2) set of edges E_0^+ that are present in Q_0 but not in Q_f; 3) set of relations R_0^+ that are present in Q_o but not in Q_f.

The edges in E_f^+ are syntactically or semantically redundant since they have been added by the algorithm to the initial graph during initial conversion (to G_c) or semantic expansion. The fact that they were not eliminated during the semantic reduction implies that they belong to the "profitable" category, provided the estimated selection-join sequence holds good. In this context they represent an additional profit for Q_f as compared to Q_0.

All the edges in E_0^+ are also syntactically or semantically redundant because otherwise they would have been retained in Q_f too. The reason for their removal by the semantic reduction stage was that they were not found to be profitable. In other words, the edges in E_0^+ represent an elimination of the nonprofitable part from the original query, if the estimations on the selection–join sequence holds good.

In short, as compared to Q_0, E_0^+ represents the edges lost, whereas E_f^+ represents the edges gained by Q_f. The strategy of adding and eliminating the edges always concentrates on adding profitable edges and removing nonprofitable ones. Both these components thus represent profit provided the sequence estimation of selections and joins are valid.

The set R_0^+ represents a clear profit for Q_f because Q_f does not have the corresponding joins.

To conclude, the cost advantage of Q_f over Q_0 can be represented by $C = a_1 * |E_f^+| + a_2 * |E_0^+| + a_3 * |R_0^+|$, where a_1, a_2, a_3 are scaling factors to reflect the relative importance of components as well as validity of the assumption of the selection–join sequence. If these assumptions are valid, C represents a positive quantity, and in that case the larger the sets E_f^+, E_0^+, R_0^+ are, the higher is the resulting cost advantage.

D. Implementation Architecture

The optimization algorithm has been implemented on a Vax 8530 running VMS (version 4). The core implementation language is C, supplemented with some used interfaces and screen utilities (ABF, OSL, Vifred) available with Ingres (version 5).

The optimizer consists of three main modules—specification module, maintenance module, and processing module.

The specification module is an interface for end users to specify queries in an interactive mode and get the semantically optimized query form back. For an ordinary user, this is the only interface to the optimization system. The frame associated with this module has two major sections, one for specifying the initial query form and the other for displaying the optimized form. In both sections, a query is defined to be a combination of two sets, namely, a set of target attributes and a set of qualifications. Both of these sets are entered and displayed in individual tabular fields capable of scrolling independently, thus allowing the handling of any number of target attributes and qualifications. This module also supports an exhaustive error management scheme. Target attributes and qualification specifications entered by the user are validated against the schema details, and any error is reported before passing the information to the processing module. The specification frame has another section for displaying the run time statistics from the processing module. This statistic includes relative time spent by the processing module on various components of the optimization algorithm. Fig. 7 illustrates the frame associated with the specification module. This module is implemented using the Ingres utilities Vifred, ABF, and OSL.

The maintenance module is a background module which is generally invisible to the normal user. A user with maintenance privilege can use this module to change the details of schema, index information, and semantics (i.e., relations and rules). The data dictionary containing relation names, their attributes, and index information as well as the semantic details containing implication constraints and subset constraints are stored in data structures similar to tables. Tabular fields with independent scrolling features are used to display them on the frame. This module also has comprehensive error checking mechanisms to make sure that all the semantics specified by the user are valid for the existing schema definition. The frame associated with the maintenance module is illustrated in Fig. 8. This module is also implemented using the Ingres utilities Vifred, ABF, and OSL.

The processing module implements the optimization part of the algorithm. The information entered by the user in the specification module is passed to this module after some syntactic analysis and preliminary error checking. This information (query) is analyzed by the processing module in the context of existing schema and semantic details (relations and rules). The processing module then transforms the query through various sequential stages of the algorithm, namely, transitive reduction, compressed graph formation, semantic expansion, relation elimination, restriction elimination, and spanning tree generation. Errors and contradictions detected at any stage are reported to the specification module, and processing is

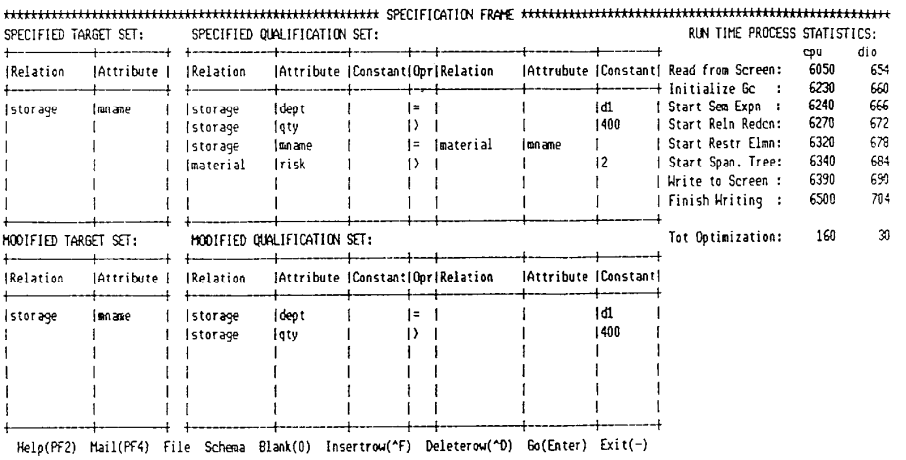


Fig. 7. Query specification frame.

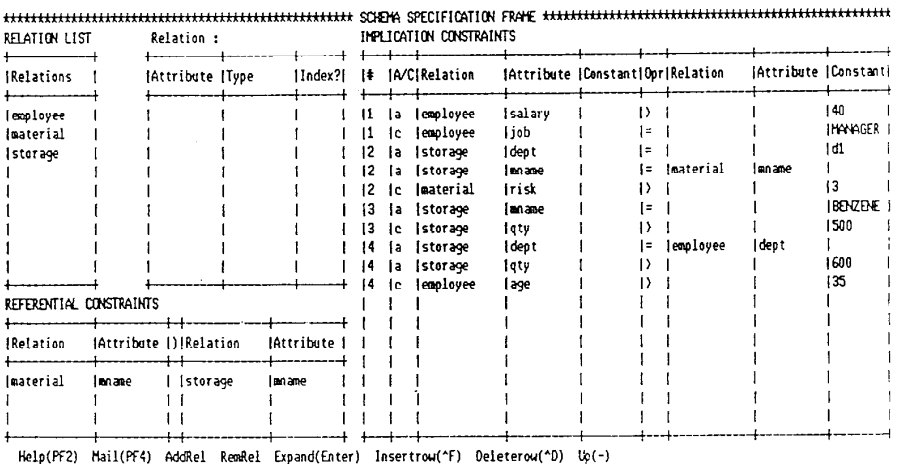


Fig. 8. Schema maintenance frame.

aborted in such cases. As the processing proceeds, the module also collects various run time statistics. The final form of the query as well as the collected statistics is passed to the specification module upon completion of the processing. The major portion of the processing module is implemented in C language. A small portion is written in Equel (query language supported by Ingres) to communicate with the storage tables where the schema and semantics details are stored.

Currently, all the inter module communication uses stored tables as the main data structures. Access to the stored information is minimized in the processing module due to efficiency considerations. On the other hand, the other two modules assign importance to human factors and user friendliness rather than operating speed. Management of errors is uniform and exhaustive in all three modules, and the errors handled range from simple specification mistakes to complex semantic contradictions.

E. Implementation Results

As a basic test of the implementation of the semantic query optimizer, we selected the database scheme, semantic rules, and query set introduced in Section III-F, which are repeated below.

Schema and Relation Sizes (Indexed attributes are underlined):

employee (ssn, fname, lname, dept, job, salary, age) [size: 37043 tuples]
storage (dept, material, qty) [size: 1601 tuples]
material (material, risk) [size: 1801 tuples]

Semantics:

employee.salary > 40K → employee.job = "Manager".

storage.dept = d1, storage.material = material.material → material.risk > 3.

storage.material = "Benzene" → storage.qty > 500.

storage.qty > 600, storage.dept = employee.dept → employee.age > 35.

Original and Optimized Queries:

Restriction Elimination:

Q_1: retrieve (storage.dept) where
storage.material = "BENZENE" and
storage.qty > 400

Q_1': retrieve (storage.dept) where
storage.material = "BENZENE"

Index Introduction:

Q_2: retrieve (employee.lname, employee.fname) where employee. salary > 42K

Q_2': retrieve (employee.lname, employee.fname) where employee.salary > 42K and employee.job = "MANAGER"

Scan Reduction:

Q_3: retrieve (employee.lname. employee.fname) where employee.dept = storage.dept and storage.qty > 625

Q_3': retrieve (employee.lname. employee.fname) where employee.dept = storage.dept and storage.qty > 625 and employee.age > 35

Relation Elimination:

Q_4: retrieve (storage.material) where
storage.dept = "d1"and
storage.qty > 400 and
storage.material = material.material and
material.risk > 2

Q_4': retrieve (storage.material) where
storage.dept = "D1" and
storage.qty > 400

Test Procedure and Results: The employee table was populated by actual data from a payroll file, and then modified to ensure anonymity. The storage table was filled by a random number generator. The material table was loaded from a chemical data file. All the tables were then extensively modified manually to be consistent with the semantic rules. The relation "employee" had a primary index (B-tree) on the attribute "job", and "storage" had a secondary index on (isam) on "dept".

The four original queries (Q_1, . . . , Q_4) and their optimized counterparts (Q_1' . . . , Q_4') represent each of the four transformation heuristics. Several tests were performed to study the optimization costs and execution costs of the above query pairs. Both these costs were measured in terms of two resources consumed by the Ingres process, _cpu_ms (CPU time in milliseconds) and _dio_cnt (direct I/O requests). Since these parameters are highly influenced by hardware configuration, we do not stress any units. For us, the matter of relevance is only the relative magnitude of these parameters.

All four queries (Q_1, . . , Q_4) were fed to the semantic optimizer to generate the optimized forms (Q_1', . . , Q_4'). This process was repeated several times for each of the queries to measure the optimization cost. The optimization cost (in terms of _cpu_ms and _dio_cnt) did not seem to vary much from query to query and we obtained an average of about 180 _cpu_ms and 30_dio_cnt.

Then the four original queries and the corresponding four semantically optimized ones were run against the database tables repeatedly, about 15 times. During each execution, cost of the query was monitored using the above two parameters. Fig. 9 is a consolidation of the average values from these 15 tests.

Qry #	Execution Cost, Original Form		Execution Cost, Optimized Form		% Gain in Exec. Cost		Opt+Exec Cost, Optimized Form		% Gain in Total Cost	
	_cpu	_dio	_cpu	_dio	_cpu	_dio	_cpu	_dio	_cpu	_dio
Q1	657	60	582	50	11.46	17.12	762	80	-15.98	-33.33
Q2	14461	2571	4227	368	70.77	85.69	4407	398	69.52	84.52
Q3	138295	4364	91209	3325	34.05	23.80	91389	3355	33.92	23.12
Q4	15167	88	626	49	95.87	44.25	806	79	94.69	10.22

Fig. 9. Test results of semantic optimization.

It can be observed from the above results that optimized versions of the queries take less execution time (in terms of CPU time and direct I/O requests) as compared to the original ones. The magnitude of saving depends on the size of the tables involved as well as the amount of potential optimization possible. This saving is partially offset by the optimization cost which, as reported above, is about 180 _cpu_ms and 30 _dio_cnt. In most circumstances these values are much smaller than the saving in execution cost, thus justifying semantic optimization. On the other hand, if the original query is already in the optimized form, the optimization cost becomes an addition to the execution cost. This is the situation with Q_1, where the execution cost of the optimized query added to the optimization cost is larger than the execution cost of the original query. In such cases, semantic optimization may not be worth the effort.

The savings in the execution cost due to semantic optimization grows with the data size involved in answering the query, whereas the optimization cost remains the same. Hence, it is reasonable to assume the optimizaton cost to be negligible as compared to the savings in the execution cost with very large data sets. Also, if the optimized query is expected to be executed a multiple number of times, the optimization cost can be assumed to be amortized over those executions.

If optimization cost becomes comparable to the execution cost, it becomes important to consider a compromise between those two. A detailed analysis of such a tradeoff between optimization and execution costs is reported in [33].

IX. Conclusion and Future Extensions

In this paper we have proposed and described a scheme for utilizing semantic constraints for optimizing a database query. We have tried to quantify the factors that decide the profit of a query and have illustrated how relations, rules, and query can interact to arrive at an optimum query form. The major contribution of the work is a scheme that dynamically selects from a large collection of rules only the profitable ones for a relation in a query context.

An algorithm is introduced to transform the initial query to a semantically equivalent one. The algorithm has its

best performance if the estimations of selection–join sequences holds good in reality. Certain major factors like elimination of redundant joins are independent of these assumptions anyway. Cases where a query can be answered just using semantic rules and the ones where query conditions and/or semantic constraints imply a null answer are also handled efficiently by the algorithm. In other cases, semantic rules aid the query processing by generating useful additional constraints or by eliminating existing redundant constraints.

The algorithm is implemented with necessary user interface modules and tested with real data of reasonable volume. The test results are very encouraging, thus revealing the potential savings a semantic optimizer can provide.

Also an algorithm is devised to maintain semantic implication constraints. The related maintenance scheme assures that the semantic rule set is free from contradiction and redundancy.

We are currently studying some additional types of constraints and optimization strategies to incorporate in the algorithm. Usage of conventional constraints like functional dependencies along with the semantic constraints requires further analysis. Methods like introducing an additional join to the original query (join introduction) as an optimizing scheme [18] also needs further investigation from an implementation point of view.

Two possible future extensions to the system currently under investigation are semantic categorization and partial optimization. Both these extensions are more relevant to programmed (repetitive) queries rather than interactive ones. Statistically, more than 80 percent of all the database queries are preprogrammed and highly repetitive. In such a situation, it is quite sensible to optimize the queries only once and save the optimized form for all future uses. But such saved forms are valid only when the relevant semantics (that used for optimization) remains unchanged. Any change in the semantics mandates a reprocessing of those queries that used the changed semantics for optimization.

In semantic categorization, semantic rules are given weights according to their volatility. This categorization is highly dependent on the nature of data and represents only an approximate stability of different rules. The assumption is that if a rule is more volatile, there are more chances for an optimized query form that used the rule to become invalid within a given time frame, thus mandating a reoptimization. If the rules are categorized, the semantic optimizer can then analyze them before using in an optimization, in terms of profitability and volatility. These factors can be weighed against each other to arrive at an appropriate selection of optimization rules.

The second extension, partial optimization, becomes useful in the case where a semantic information used to optimize a query is subjected to change. In such a situation, we are studying the possibilities of avoiding a total reprocessing of the user specified query. Reflecting the semantic changes into the previously processed queries without reprocessing them from the original form could be profitable especially if semantic processing is costly.

Also, we are conducting more experiments using the optimizer with various practical situations. Our studies involve different table sizes, various storage structures and secondary indexes, and different query patterns.

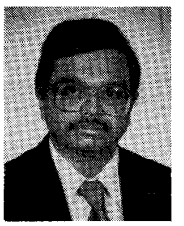
Acknowledgment

The authors are grateful to Prof. G. Ozsoyoglu for his suggestions on constraint maintenance.

References

[1] A. V. Aho, Y. Sagiv, and J. D. Ullman, "Equivalences among relational expressions," *SIAM J. Comput.*, vol. 8, pp. 218–246, 1979.
[2] M. M. Astrahan *et al.*, "SYSTEM R: A relational approach to database management," *ACM TODS*, vol. 1, June 1976.
[3] P. A. Bernstein *et al.*, "Query processing in systems for distributed databases (SDD-1)," *ACM TODS*, vol. 6, pp. 602–625, Dec. 1981.
[4] M. W. Blasgen and K. P. Eswaran, "Storage access in relational databases," *IBM Syst. J.*, vol. 16, no. 4, 1977.
[5] U. S. Chakravarthy, D. H. Fishman, and J. Minker, "Semantic query optimization in expert systems and database systems," in *Proc. 1st Int. Conf. Expert Database Syst.*, Kiawah Isl., SC, Oct. 1984, pp. 326–341.
[6] U. S. Chakravarthy and J. Minker, "Multiple query processing in deductive databases," *Univ. Maryland*, Tech. Rep. TR-1154, Aug. 1985.
[7] U. S. Chakravarthy, J. Minker, and J. Grant, "Semantic query optimization: Additional constraints and controil strategies," in *Proc. 1st Int. Conf. Expert Database Syst.*, Charleston, SC, L. Kershberg, Ed., Apr. 1986, pp. 259–269.
[8] R. Epstein, M. Stonebraker, and E. Wong, "Distributed query processing in database systems," in *Proc. ACM SIGMOD*, Austin, TX, June 1978, pp. 169–180.
[9] G. Gentzen, "Untersuchugen über das logische schliessen," in *The Collected Papers of Gerhard Gentzen*, M. E. Szabo, Ed. Amsterdam; The Netherlands: North-Holland, 1934, pp. 68–132.
[10] J. Grant and J. Minker, "Optimization in deductive and conventional database systems," in *Advances in Database Theory*, Vol. 1, H. Gallaire, J. Minker, and J. M. Nicolas, Eds. New York: Plenum, 1980.
[11] L. R. Gotlieb, "Computing joins of relations," *ACM SIGMOD Int. Symp. Management Data*, 1975, pp. 55–63.
[12] M. Hammer and S. B. Zdonik, Jr., "Knowledge based query processing," in *Proc. VLDB*, 1980, pp. 137–147.
[13] H. B. Hunt and D. J. Rosenkrantz, "The complexity of testing predicate locks," in *Proc. ACM SIGMOD*, 1979, pp. 127–133.
[14] A. R. Hevner and S. B. Yao, "Query processing in distributed database systems," *IEEE Trans. Software Eng.*, vol. 5, pp. 177–187, May 1979.
[15] M. Jarke, "External semantic query simplification: A graph-theoretic approach and its implementation in Prolog," in *Proc. 1st Int. Conf. Expert Database Syst.*, Kiawah, Isl., SC, Oct. 1984, pp. 467–482.
[16] M. Jarke, J. Clifford, and Y. Vassiliou, "An optimizing Prolog front end to a relational query system," in *Proc. ACM SIGMOD*, Boston, 1984, pp. 296–306.
[17] W. Kim, "Relational database systems," *ACM Comput. Surveys*, vol. 3, pp. 185–212, Sept. 1979.
[18] J. J. King, "QUIST: A system for semantic query optimization in relational databases," in *Proc. VLDB*, 1981, pp. 510–517.
[19] A. Klug, "On conjunctive queries containing inequalities," *JACM*, vol. 35, pp. 146–160, Jan. 1988.
[20] R. Kowalski, "Logic for problem solving," *Artificial Intelligence Series*, The Computer Science Library, 1983.
[21] R. Krishnamurthy, H. Boral, and C. Zaniolo, "Efficient optimization of nonrecursive queries," in *Proc. VLDB*, 1986.
[22] R. Kung, E. Hanson, Y. Ioannidis, T. Sellis, L. Shapiro, and M. Stonebraker, "Heuristic search in database systems," in *Proc. 1st Int. Conf. Expert Database Syst.*, Kiawah Isl., SC, Oct. 1984, pp. 96–107.
[23] D. Maier, *The Theory of Relational Databases*. New York: Computer Science, 1983.

[24] Z. Manna, *Mathematical Theory of Computation*. New York: McGraw-Hill, 1974, pp. 109-111.
[25] E. Mendelson, *Introduction to Mathematical Logic*. Princeton, NJ: Van Nostrand, 1964.
[26] M. Morgenstern, "The role of constraints in database, expert systems, and knowledge representation," in *Proc. 1st Int. Conf. Expert Database Syst.*, Kiawah Isl., SC, Oct. 1984, pp. 207-223.
[27] F. P. Palermo, "A database search problem," in *Information Systems COINS IV*, J. T. Tou, Ed. New York: Plenum.
[28] J. M. Nicholas and K. Yazdanian, "Integrity checking in deductive databases," in *Logic and Databases*, H. Galliere and J. Minker, Eds. New York: Plenum, 1978.
[29] X. Qian and D. R. Smith, "Reformulation: an approach to efficient constraint validation," in *Proc. VLDB*, 1987.
[30] D. J. Rosenkrantz and H. B. Hunt, "Processing conjunctive predicates and queries," in *Proc. VLDB*, 1980, pp. 64-72.
[31] T. K. Sellis, "Global query optimization," in *Proc. ACM SIGMOD*, May 1986, pp. 191-205.
[32] S. T. Shenoy and Z. M. Ozsoyoglu, "A system for semantic query optimization," in *Proc. ACM SIGMOD*, May 1987, pp. 181-195.
[33] S. Shekar, J. Srivastava, and S. Dutta, "A formal model of trade-off between optimization and execution costs in semantic query optimization," in *Proc. VLDB*, Aug. 1988, pp. 457-467.
[34] M. Siegel, "Automatic rule derivation for semantic query optimization," Boston Univ., Tech. Rep. 87-012, 1987.
[35] ——, "Automatic rule derivation for semantic query optimization," in *Proc. 2nd Int. Conf. Expert Database Syst.*, Tysons Corners, VA, 1988, pp. 371-385.
[36] M. R. Stonebraker, E. Wong, P. Kreps, and G. Held, "The design and implementation of INGRES," *ACM TODS*, vol. 1, Sept. 1976, pp. 189-222.
[37] L. Sterling and E. Shapiro, *The Art of Prolog*. Cambridge, MA: M.I.T. Press, 1986.
[38] X. Sun, N. Kamel, and L. M. Ni, "Solving implication problems in database applications," in *Proc. ACM SIGMOD*, May 1989, pp. 185-192.
[39] J. D. Ullman, *Principles of Database Systems*, 2nd ed. New York: Computer Science, 1978.
[40] E. Wong and K. Youssefi, "Decomposition: A strategy for query processing," *ACM TODS*, vol. 1, pp. 223-241, Sept. 1976.
[41] S. B. Yao, "Optimization of query evalulation algorithms," *ACM TODS*, vol. 4, no. 2, pp. 133-155.
[42] T. C. Yu and Z. M. Ozsoyoglu, "On determining tree query membership of a distributed query," in *Proc. INFOR*, Aug. 1983, pp. 261-281.



Sreekumar T. Shenoy was born in Quilon, India. He received the Bachelor of Engineering degree in electronics and communication from the College of Engineering, Trivandrum, India, the M.Tech degree in computer science from the Indian Institute of Technology, Madras, India, in 1982, and is completing the Ph.D. degree in computer science from Case Western Reserve University, Cleveland, OH, in 1989.

Currently, he is a Database Consultant to British Petroleum America, Cleveland, OH. Previously, he was a member of the Software Development Team at Keltron, Trivandrum, India. His research interests include semantic query processing, fourth generation database languages, software engineering, and intelligent user interfaces.

Zehra Meral Ozsoyoglu received the B.Sc. degree in electrical engineering, and the M.Sc. degree in computer science from the Middle East Technical University, Ankara, Turkey, in 1973 and 1975, respectively, and the Ph.D. degree in computer science from the University of Alberta, Alberta, Canada, in 1980.

In 1980, she was an Assistant Professor in the Department of Computer Engineering and Science, Case Western Reserve University, Cleveland, OH. Currently, she is an Associate Professor at the same university. Her research interests include nested relations and complex objects in database systems, query optimization, and logic based systems.

Dr. Ozsoyoglu is a member of the Association for Computing Machinery and an Editor of IEEE Database Engineering, and was a recipient of the IBM Faculty Development Award in 1983.