Logic-Based Approach to Semantic Query Optimization

UPEN S. CHAKRAVARTHY
Xerox Advanced Information Technology
JOHN GRANT
Towson State University
and
JACK MINKER
University of Maryland

The purpose of semantic query optimization is to use semantic knowledge (e.g., integrity constraints) for transforming a query into a form that may be answered more efficiently than the original version. In several previous papers we described and proved the correctness of a method for semantic query optimization in deductive databases couched in first-order logic. This paper consolidates the major results of these papers emphasizing the techniques and their applicability for optimizing relational queries. Additionally, we show how this method subsumes and generalizes earlier work on semantic query optimization. We also indicate how semantic query optimization techniques can be extended to databases that support recursion and integrity constraints that contain disjunction, negation, and recursion.

Categories and Subject Descriptors: H.2.0 [**Database Management**]: General; H.2.2 [**Database Management**]: Physical Design—*access methods*; H.2.4 [**Database Management**]: Systems—*query processing*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*logic programming*; *resolution*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods and Search—*heuristic methods*; *plan execution*; *formation*; *generation*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Deductive database, integrity constraint, query optimization, residue, semantic compilation, semantic query transformation, subsumption

1. INTRODUCTION

Relational formalism introduced a conceptual framework for databases which provided a clean separation between abstract concepts and their physical realizations. Early database systems, namely, hierarchical and network models, lacked such a clean separation; their query languages were procedural with record-at-a-

* Work on this paper was supported by the National Science Foundation under grants IRI 86-09170 and IRI-8714544, and by the Army Research Office under grant DAAL-0388-K0087.
Authors' addresses: U. S. Chakravarthy, CIS Department, University of Florida, Gainesville, FL 32611; J. Grant, Department of Computer and Information Sciences, Towson State University, Towson, MD 21204; J. Minker, Department of Computer Science and University of Maryland Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742.

time retrieval and the operators were intricately tied to the physical representation (e.g., navigation). Efficient evaluation, in this framework, involved *rewriting queries* using the knowledge of physical representation and their effect on the retrieval/update operations explicitly.

The advent of the relational model (and nonprocedural query languages) essentially introduced the need for a translation phase: a nonprocedural query had either to be interpreted or translated into a procedural equivalent that could be executed over the physical representations of data objects. This translation phase not only involved a correct translation from the higher level constructs to lower level operations, but also had to deal with choosing a sequence of operations (from among several alternatives) that translated the original query into an *efficient* program. Hence, query optimization developed as a very important area of research for relational database systems. Without query optimization, most relational systems would be highly inefficient. The symbiosis of higher level constructs and query optimization techniques helped the user to concentrate on the application at hand without having to worry about the details of physical representation and efficient retrieval/update.

Although a large body of knowledge has been developed encompassing various aspects of optimization, the primary focus, so far, has been on the translation of abstract operators using properties of the operators and their mappings to a variety of implementation alternatives. This body of knowledge can be characterized as *syntactic* query optimization. The theory developed so far has not meaningfully utilized the properties of the application domain which, in our opinion, is likely to play a key role during the optimization process (as the levels of abstraction increase). Dramatic improvements in performance can be achieved, beyond what has been done so far, by extending the theory to encompass the *semantics* of the applications. We consider as semantics any application specific knowledge that is declared as part of the database. The use of (any kind of) semantic knowledge for optimizing queries is called *semantic query optimization* (*SQO*).

Deductive databases provide a theory of relational databases as well as a query language which is more powerful than relational query languages. Furthermore, deductive databases provide a uniform representation (in terms of clause form) for expressing database components, namely, facts, deductive rules, as well as semantic knowledge. In addition, logic provides the necessary formalism (in terms of satisfiability, provability and inference mechanisms) for developing a theory of semantic query optimization. Semantic query optimization provides the same kind of transparency with respect to semantic knowledge that relational optimizers provide with respect to physical representation. The user is motivated to concentrate on the application rather than forming queries with explicit semantic knowledge of the application. The techniques presented in this paper enhance the capabilities of extant query optimizers by meaningfully exploiting (application specific) semantic knowledge.

Deductive databases extend relational databases by allowing relations to be defined implicitly in terms of stored relations using rules. A survey of the use of logic for databases in general, and deductive databases in particular, can be found in Gallaire, Minker, and Nicolas [9]. A perspective on deductive databases dating

back to 1957 may be found in Minker [21]. In our paper, we discuss semantic query optimization in the framework of deductive databases; however, our methodology can be applied to standard relational databases also, as we will show. For a theory on which the above approach is based, refer to Chakravarthy, Grant, and Minker [5].

Jarke and Koch [13] present a survey of query optimization. The process is divided into four steps: (1) find an internal query representation, (2) apply logical transformations to the query representation, (3) obtain a set of alternative sequences of elementary operations, (4) find the cheapest element in 3 and execute it.

In this paper we use clause form for representing queries. The bulk of the material presented deals with step 2, which is divided in Jarke and Koch [13] into three parts: standardization (placing the query into some standard form), simplification (eliminating redundancy in the query), and amelioration. According to this classification, semantic query optimization falls into the amelioration category along with other query transformation heuristics, such as the combining and reordering of operations. A set of alternative (semantically equivalent) queries are produced as a result of applying step 2.

The strategy chosen for optimization and its cost are important considerations. In order to reduce the optimization overhead, the work is partitioned into two categories: processing that can be performed statically once and processing that has to be performed at run time. The approach presented in this paper applies the above partitioning, by using the so-called compiled approach, where appropriate, to compute statically the information required during run time. In the compiled approach, nonrecursive deductive rules for the intensional relations are applied repeatedly, until each such relation becomes formulated entirely in terms of extensional relations. Semantic knowledge is merged with the compiled form prior to the evaluation of any query. Then, a query containing implicitly defined relations can be transformed (at run time) to (possibly several) queries using only extensional relations and the compiled semantic knowledge. Such queries can be optimized and solved by applying various known methods.

We generalize the work of other researchers on semantic query optimization in several ways. Our method applies to a larger class, namely, deductive databases, thereby making it applicable to standard relational databases. We deal with a very general class of integrity constraints. We use a general-purpose theorem proving method, rather than one that considers special cases separately. Also, we compile the semantic knowledge only once in a preprocessing step, thereby reducing the amount of computation required at the time of query processing. This paper consolidates results from several of our earlier papers, Chakravarthy, Fishman, and Minker [4], Chakravarthy, Minker, and Grant [7], and Chakravarthy, Grant, and Minker [5], and contains additional material to demonstrate the applicability and effectiveness of SQO techniques for relational databases.

The outline of the paper is as follows. In Section 2 we review the database literature on and closely related to semantic query optimization, including our previous papers. We present a brief background on logic and deductive databases, as well as our assumptions and a general overview of our method along with a list of potentially useful transformations in Section 3. We discuss semantic

compilation in Section 4. The semantic transformation phase is the subject of Section 5. In Section 6 we indicate the generality of our approach by casting examples from earlier work in our formalism and indicating the transformations used. We show, in Section 7, how our approach can be extended to recursion and integrity constraints that contain disjunction, negation, and recursion. Section 8 has the summary and conclusions.

2. LITERATURE REVIEW

In this section we review the database literature on semantic query optimization and discuss work closely related to it. The idea of semantic query optimization is to use semantic knowledge (we use the terms semantic information, semantic knowledge, and integrity constraints interchangeably throughout this paper) about the database to transform a query into one which is semantically equivalent to the original query and which can be executed efficiently (that is, using less resources, whatever they may be). Informally, semantic equivalence means that the transformed query has the same answer as the original query on all databases satisfying the integrity constraints.

There are several aspects to semantic query optimization. One aspect involves the type of database under consideration and the type of integrity constraints allowed. Another aspect concerns the generation of semantically equivalent queries and their correctness. Yet another aspect is the filtering, or pruning, of information that is not useful for semantic optimization. Another important question is how to control the generation process to limit it to "promising" candidate queries. And finally, the integration of semantic query optimization with conventional query optimization is also an important issue.

McSkimin [19] and McSkimin and Minker [20], use a semantic network model and typing of attributes in a deductive search process. Such typing can be used to inhibit the search when clauses cannot be unified because of conflicting types. A semantic well-formedness condition is also applied to queries and data inputs to check for no solutions. Additionally, the system checks for cases where the maximum number of solutions has already been found.

The knowledge-based query process (KBQP) was developed in Hammer and Zdonik [11] and Zdonik [28]. KBQP uses a knowledge base about the application domain and the database to perform query transformations. A pattern matcher and rewriter replace subexpressions of a query by equivalent expressions based on semantic information in the hope of yielding an expression that can be evaluated more efficiently. The control problem is also discussed: query transformations are not done exhaustively; heuristics, such as domain refinement, are used for predetermined cases to guide the transformations. Both the knowledge base and the queries are expressed in a language based on the lambda calculus.

King discusses semantic query optimization in general and the QUIST (Query Improvement through Semantic Transformation) system in particular in King [14, 15]. The query language is a subclass of Select–Join–Project queries for relational databases and is somewhat limited in expressive power as only a fixed single join is allowed between relations. The constraints are on values only; no dependencies are used. The notions of semantic equivalence between queries and

merging an integrity constraint into a query are formally defined. Various heuristics, based on knowledge of query processing and file structures, guide the transformation process. This transformation process is data-directed, in contrast to the goal-directed process of Hammer and Zdonik [11].

Xu [27] discusses query optimization for Select–Join–Project queries for relational databases. The integrity constraints are domain rules, dependency rules, and production rules (a class of constraints involving two relations and a join condition between them). Examples of transformations that preserve semantic equivalence are presented for each type of integrity constraint. The most efficient access path to each relation specified in the original query is obtained by using various heuristics, including join elimination and index application. The transformation yields a single semantically optimized query that can then be passed to a conventional query optimizer.

Jarke et al. [12] deal with semantic query optimization in the context of an optimizing PROLOG front-end to a relational database system. Function-free conjunctive queries are considered. Three types of integrity constraints are allowed: value bounds, functional dependencies, and referential constraints. A query graph is constructed to deal with inequalities; tableaux are used with functional dependencies to semantically optimize queries.

Kohli and Minker [16] develop a control strategy for function-free logic programs using integrity constraints and an analysis of failures. The idea of an integrity constraint or part of an integrity constraint subsuming a goal clause is introduced. Portions of proof tree are pruned using subsumption as well as creating and propagating new constraints for failed nodes. Our work expands on this concept in the database context by transforming a query to a semantically equivalent form.

Shenoy and Ozsoyoglu [25] provide a detailed investigation of the use of two important types of integrity constraints in semantic query optimization. These types are subset constraints and implication integrity constraints, where the latter are essentially domain constraints, although they may be interrelational as well. A query is represented by a query graph. Semantic query optimization is achieved by using the integrity constraints to modify the query graph.

This paper consolidates results from our previous papers on semantic query optimization (Chakravarthy, Fishman, and Minker [4], Chakravarthy, Grant, and Minker [5], and Chakravarthy, Minker, and Grant [7]) emphasizing the techniques and their applicability to query optimization in relational databases. Many of the examples and results in these three papers, as well as the present one, are based on Chakravarthy [3]. In Chakravarthy, Fishman, and Minker [4] we introduce our two-step method of semantic query optimization using the modified subsumption algorithm and the application of residues for deductive databases. This method is presented primarily through examples in a somewhat informal manner. Then, Chakravarthy, Minker, and Grant [7] is a follow up to Chakravarthy, Fishman, and Minker [4] using an example with an integrity constraint that contains a function symbol. It also contains a more detailed discussion of the use of different types of residues as well as control strategies. The formal definitions and algorithms for our approach to semantic query optimization are contained in Chakravarthy, Grant, and Minker [5]. Detailed proofs of the correctness of our approach can also be found there.

3. OVERVIEW OF SQO

In this section we provide an overview of the two-phased approach to semantic query optimization and the utility of the approach by indicating the types of transformations that can be effected by semantic knowledge. Before providing an overview, we need to set the context by introducing the components of a deductive database and the assumptions used in this paper.

This section is organized as follows. We first present background information on first-order logic and deductive databases. We then state the assumptions used in the paper followed by an overview of our two-phased approach to semantic query optimization. Finally, we present the main types of transformations that can be accomplished using semantic query optimization. Detailed database examples of these transformations are given later.

3.1 Deductive Databases

Our presentation of logic and deductive databases is rather sketchy; additional material and references may be found in Gallaire, Minker, and Nicolas [9]. We discuss the notion of deductive databases in a Prolog-like notation that we use for representing facts, deductive rules (intensional axioms), integrity constraints, and queries.

A *literal* is either an atomic formula or the negation of an atomic formula. A *clause* is a disjunction of literals which has the form S1 $\vee \cdots \vee$ Sm $\vee \neg$R1 $\vee \cdots \vee \neg$Rn, where each Si and Rj represents an atomic formula. Clauses can be written in an equivalent form using implication, such as R1 $\& \cdots \&$ Rn $\rightarrow$ S1 $\vee \cdots \vee$ Sm for the clause given above. We use a Prolog-like notation throughout the paper and write this clause as S1, ..., Sm $\leftarrow$ R1, ..., Rn. In particular, a literal is written as S $\leftarrow$ (i.e., with the arrow to the right) if positive and $\leftarrow$ R (i.e., with the arrow to the left) if negative. All the variables in a clause (in all notations) are assumed to be universally quantified.

In logic programming, a program consists of clauses. Prolog restricts clauses to be Horn. It turns out that Horn clauses (used for expressing facts, rules, as well as integrity constraints) are also adequate for specifying a large class of database applications. A clause is *Horn* if m$\leq$1 and *disjunctive* otherwise. In the representation given above, the Ri form the *body* of the clause and the Sj form the *head*. Thus, a *Horn clause* is one whose head consists of at most a single atom. A *goal clause* has a null head. The *null clause* has both a null body and a null head; it represents a contradiction. A clause is *definite* if the head of the clause contains exactly one atom. A *unit clause* is a definite clause with a null body. A *ground unit clause* is a unit clause all of whose arguments are constants.

We use infix operators for denoting relations corresponding to widely used arithmetic operations, such as: $>$ (greater than), $<$ (less than), $=$, $\neq$, $\leq$, $\geq$. These relations are termed evaluable (or built-in) relations in contrast to (nonevaluable) relations defined as part of the database. A clause is said to be *range-restricted* if every variable in the head also appears in the body as arguments of nonevaluable relations. A clause is called *recursive* if the same relation symbol appears both in the head and the body. A relation R may be defined recursively by a recursive clause or by mutual recursion where R is defined (in the head) in terms of S1 (in the body), S1 is defined in terms of S2, ..., and Sk is defined in terms of R.

The notion of deductive database is based on the proof-theoretic approach to databases. We limit our discussion to the basic components of a deductive database: facts, deductive rules and integrity constraints. Database facts are expressed as ground unit clauses. For example, the fact that a tuple ⟨a1, . . . , an⟩ is a member of the relation R is expressed by the atomic literal R(a1, . . . , an) ←. Such facts comprise the extensional database, *EDB*, and such a relation R is called an extensional (stored) relation. The intensional database, *IDB*, contains deductive rules. Relations defined by deductive rules are called intensional relations. For example, the Horn clause

Emp-Dept(emp-name,dept-name)
← Emp(emp-id,emp-name,dept-id), Dept(dept-id,dept-name)

is a rule defining the (intensional) *Emp-Dept* relation in terms of the (extensional) Emp and Dept relations. In general, a rule $S \leftarrow R1, \ldots, Rn$ defines a relation S in terms of the relations R1, . . . , Rn. In relational database terminology, universally quantified nonrecursive rules are expressed as view definitions.

Another component of a database, whether deductive or not, is the set of integrity constraints, IC. IC contains rules expressing restrictions that the database must satisfy. A large class of integrity constraints are expressible as Horn clauses. The following are examples on the extensional relation Supplier(Sno,Sname,Item).

(1) y1=y2 ← Supplier(x,y1,z1),Supplier(x,y2,z2).

(2) ← Supplier(x,y,gun),Supplier(x,z,butter).

The first constraint represents the functional dependency of supplier name on supplier number. The second constraint states that no supplier supplies both the object gun and the object butter. While integrity constraints play an important role in checking update validity, they are not needed in answering queries over definite deductive databases (see Reiter [23]). However, semantic query optimization is predicated on the premise that integrity constraints are useful for query evaluation and it is the purpose of this paper to show how integrity constraints can be used in transforming a query into a form that is more efficient to evaluate.

We use the following notations for representing various components of a database. All intensional relations are italicized. All extensional relations are in normal Roman with the first letter capitalized. Output variables are denoted by an asterisk ("*") following the variable name. We assume that constants can be distinguished easily from variables from the context. In abstract examples, we use letters from the beginning of the alphabet to denote constants whereas we use letters from the end of the alphabet to denote variables. Though we use a Prolog-like notation, we do not adhere to the Prolog convention for constants and variables.

3.2 Assumptions

The theory of a definite deductive database used in this paper consists of database facts, the EDB; deductive rules (also called axioms), the IDB; a set of integrity constraints, IC; and the inference rule, linear resolution for definite Horn clauses with negation as failure (SLDNF) (see Gallaire, Minker, and Nicolas [9]). All

clauses in the IDB, EDB, and IC are assumed to be range-restricted, although we will relax this restriction in Section 7. A query is a goal clause. For a query $\leftarrow$ Q(x1*, ..., xn*), an answer to Q is any tuple ⟨a1, ..., an⟩ such that Q(a1, ..., an) is provable. For example, $\leftarrow$ Supplier(x*,y*,gun) represents a query to find all supplier numbers and names for suppliers who supply the object gun. So, ⟨s1,remington⟩ is an answer to this query if Supplier(s1,remington,gun) is provable from the database.

The presentation and proofs of correctness are simplest under certain reasonable assumptions. We make such assumptions for the next two sections. Namely, we assume the following:

EDB: a set of function-free unit clauses with no variables;

IDB: a set of nonrecursive range-restricted function-free definite Horn clauses (i.e., with at least one atom in the body and exactly one atom in the head);

IC: a set of nonrecursive range-restricted Horn clauses.

In the case of IC we allow Skolem functions in the head of a clause. Skolem functions are introduced when first-order logic formulas contain existential quantifiers. For example, the formula $\forall$ssno $\exists$ name Person(ssno,name), stating that every person with a social security number has a name, is transformed to the clause Person(ssno,f(ssno)) $\leftarrow$. The function f is a Skolem function which indicates that the value f(ssno) is a function of the value ssno. An integrity constraint having Skolem functions will be given in Section 5.1; see also query Q9′ in Section 5.2 for the transformation of queries in the presence of Skolem functions.

We also assume that our deductive databases are structured. This means that every relation is either purely extensional or purely intensional and that all the integrity constraints are expressed using only extensional relations. As was shown in Chakravarthy, Grant, and Minker [5], it is always possible to transform a deductive database into an equivalent structured deductive database. Hence generality is not lost this way and it is more convenient to deal with structured deductive databases. We also note that a standard relational database is a special case of a (structured) deductive database. Thus, our techniques can be immediately applied to relational databases.

3.3 Two-Phased Approach to SQO

In (nonrecursive) deductive databases, query evaluation can be partitioned into a compilation phase followed by a modification and an evaluation phase. Figure 1 indicates the process of query evaluation in a nonrecursive deductive database using the so-called compiled approach. A query is evaluated by first modifying the query using the compiled (intensional) database and then optimizing the result using conventional techniques.

Our approach to using integrity constraints for optimizing a query over a deductive database generates *several* semantically equivalent queries as a result of the query modification stage in Figure 1, instead of a single query. This entails changes to both the compilation stage (now called semantic compilation phase)

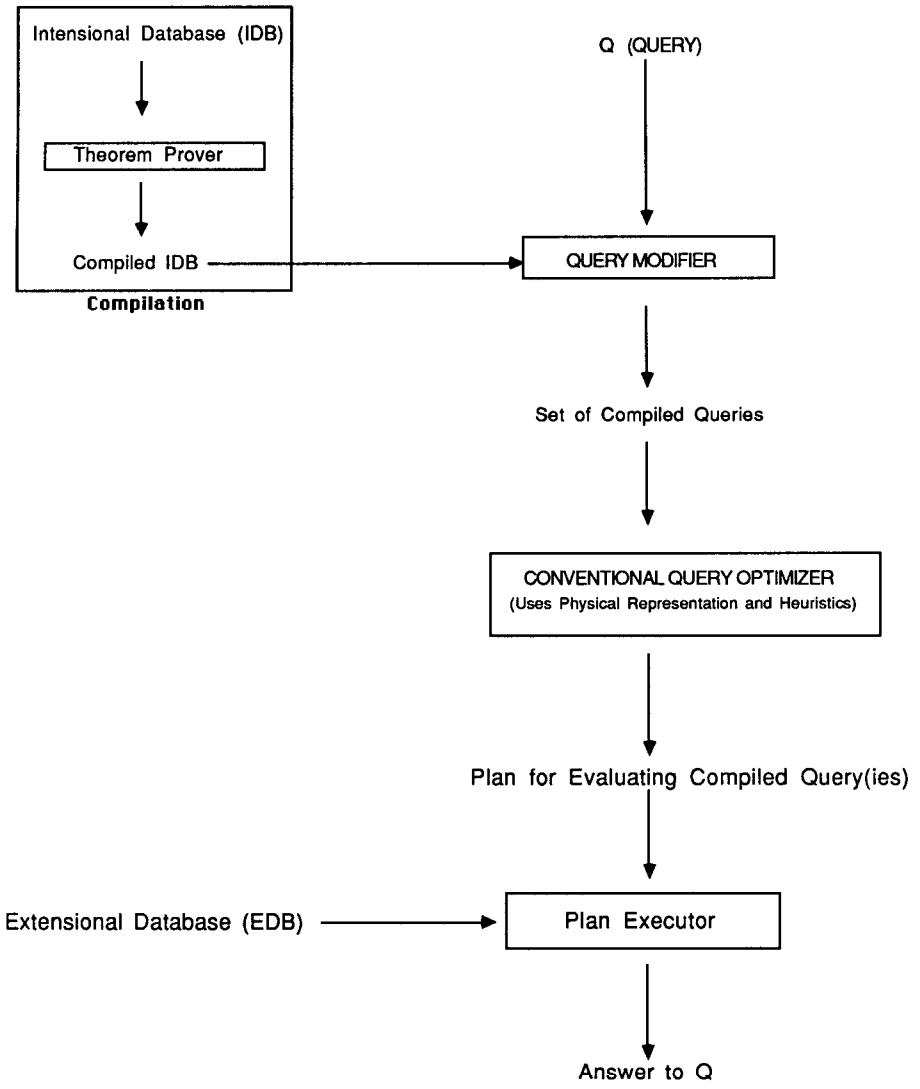


Fig. 1. Conventional query processing for nonrecursive deductive databases.

as well as the query modification stage (termed the semantic transformation phase) as shown in Figure 2.

During the semantic compilation phase, integrity constraint fragments, called residues, are computed and associated with deductive rules. The result is a set of semantically constrained axioms which are filtered and stored for later use. (The number of residues is problem dependent: it depends on the axioms and integrity constraints of the application.) When a query is given to the system, the semantic transformer uses these stored residues to generate semantically equivalent queries that may be processed faster than the original query.

The approach proposed in this paper has several beneficial features. First, semantic compilation is performed using the relatively stable components of the

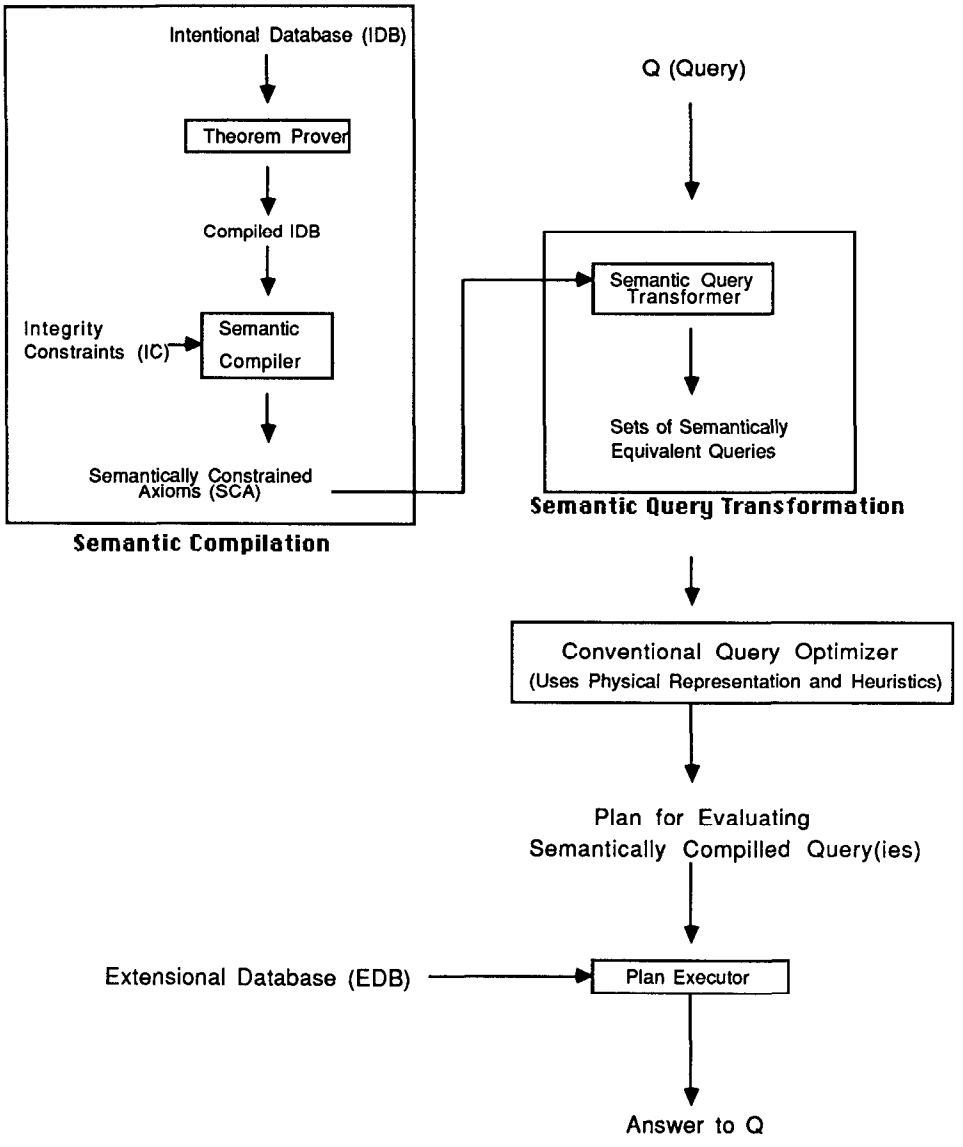


Fig. 2. Query processing using semantic query optimization.

database, is independent of the queries posed to the database, and hence is computed only once, prior to the processing of any query. Assuming that the nonextensional component of the database (the IDB and IC) is stable over a relatively long period of time, the time spent on semantic compilation is amortized over many queries. Second, though not presented in this paper, when the nonextensional portions of the database change, the new deductive rules and integrity constraints may be added *incrementally*, thus avoiding complete recompilation. Third, the transformations are at a higher level of abstraction (compared to the

choice of join algorithms, say) and hence heuristics are more appropriate at this level and are used instead of a cost model. Fourth, after the semantic compilation phase, the integrity constraints can be discarded, as they will not be required for query processing. Finally, our approach can be easily extended to include a feedback mechanism from a lower level of abstraction that can be used to prioritize residues in terms of their perceived effectiveness.

3.4 Types of Transformations

Transformations to queries are performed using various heuristics. The effectiveness of these transformations can only be evaluated at a stage where a detailed cost model is applicable. In this subsection we indicate the primary types of transformations that are deemed good in the literature and which are possible in our approach. In fact, as will be shown in Section 5, a correspondence can be established between residue types and applicable transformation types.

(1) Literal elimination: This transformation eliminates a literal and hence a join, which is an expensive relational operation.
(2) Restriction introduction/scan reduction: The introduction of a restriction (such as an equality or an evaluable range) may make it possible to use a range search, thereby reducing the cost of evaluation. Scan reduction, as introduced in King [14], also belongs to this class of transformations.
(3) Literal introduction: When a join (or a cartesian product) involves two large relations, the introduction of a small instantiated relation for an additional join can reduce the amount of computation drastically.
(4) Transformation that answers the query: It may be possible to answer a query if the transformations provide a unique answer (which may have to be verified in some cases).
(5) Detection of unsatisfiable conditions: It is possible to determine that the query does not have an answer if the existence of an answer would violate some integrity constraint in the database (McSkimin [19]).

4. SEMANTIC COMPILATION

Semantic compilation can be described informally as the process of retaining relevant (or processed) fragments of integrity constraints. It can also be viewed as a step in which all useful information (for the purpose of query processing) has been extracted from the integrity constraints, so that one can do away with the actual integrity constraints as far as query processing is concerned. As this process is designed to be used over the components of the database that are relatively stable (such as the database schema and integrity constraints), semantic compilation is performed only once for a given deductive (or relational) database. The primary objective of semantic compilation is to associate integrity constraint fragments, called residues, with compiled axioms. The residues represent the interactions of integrity constraints with relations and are used during the semantic transformation phase for generating semantically equivalent queries.

The architecture of a semantic compiler is presented in Figure 3. It includes the conventional compiler for a deductive database along with a number of

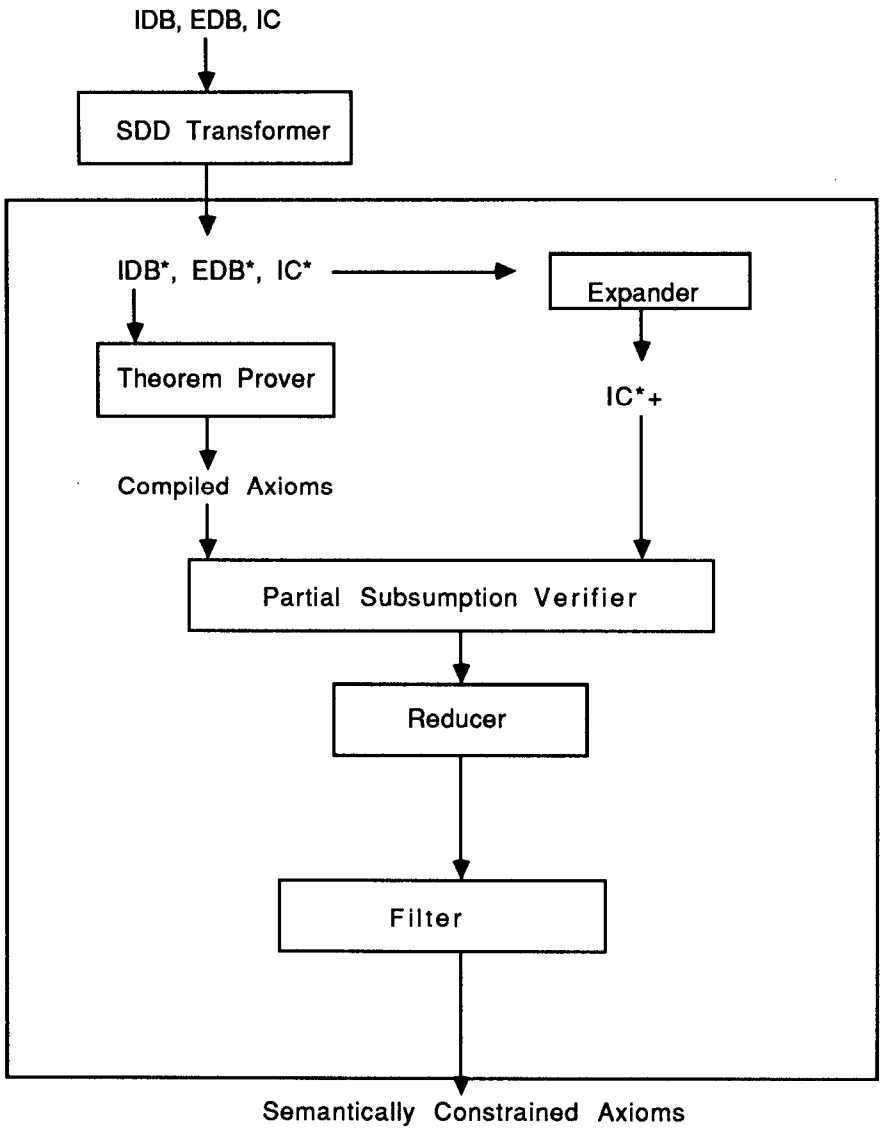


Fig. 3. Architecture of a semantic compiler.

components that are specific to the generation of semantically constrained axioms. The *Expander*, for example, changes the original integrity constraints into a form required by the partial subsumption verifier. The *Reducer*, on the other hand, compacts the residues into a form that is best suited for the filter as well as the semantic transformer. The notion of partial subsumption, a generalization of the notion of subsumption, is central to the process of semantic compilation and ties all the components together. The static filter is a post-processor which eliminates redundant residues as well as other residues that will not contribute to query transformation.

In this section we first motivate the study of semantic compilation with two database examples—one relational and one deductive. Then we present the role of partial subsumption and show how residues are generated. We provide algorithms for Expanding and Reducing integrity constraints and at the end indicate with an example the entire process of semantic compilation. We defer the discussion of residue elimination (both compile time and run time) to the next section where both are discussed. Recall that the database is assumed to be structured.

4.1 Motivating Examples

The following examples, though simple, motivate the use of integrity constraints for answering queries. More detailed examples are presented in Section 6. The first example uses relations from King [14]. (Section 6.1 discusses examples drawn from King [14] in more detail.)

Example 1

EDB: Ships(Shipname,Owner,Shiptype,Draft,Deadweight,Capacity, Registry), Owners(Ownername,Location,Assets,Business)

IC: (integrity constraints)

IC1: ← Owners(x1,iceland,x3,x4)

IC2: x2 = onassis ← Ships(x1,x2,supertanker,x4,x5,x6,x7)

The meaning of IC1 is that "There are no owners with iceland as their business location." The meaning of IC2 is that "All supertankers are owned by onassis."

We show now how these integrity constraints may be useful in query processing. First we note that IC1 is useful only if Owners appears in the query and IC2 is useful only if Ships appears in the query. Now suppose the query is Q1.

Q1: ← Owners(x1*,iceland,x3,x4*),(x3 > 1000000)

This query asks for the names and businesses of owners who are located in iceland and whose assets are greater than 1 million. A relational database system may solve such a query by a table lookup of the Owners relation or by using an index on the Owners relation. However, it follows from IC1 that Q1 cannot have any answers, so no search is needed at all.

Next we consider query Q2.

Q2: ← Ships(x1,x2,supertanker,x4,x5,x6,x7*)

This query asks for the registry information of all supertankers. Assuming that the relation is not indexed on shiptype, a table lookup is necessary. However, by using IC2 it is sufficient to look for x2 values of onassis and the corresponding registry information. Thus, IC2 introduces a selection criterion for the Owner attribute. Hence an index on the Owner attribute of Ships, if available, will dramatically speed up the retrieval.

Finally, we consider query Q3.

Q3: ← Owners(y1*,y2,y3,y4),Ships(x1,y1*,supertanker,x4,x5,x6,x7)

This query asks for the owner names of all ships of the type supertanker. Using IC2 one can establish the value of y1* as onassis. However, it is necessary to confirm the presence of such a tuple in both the relations Ships and Owner. An index is introduced on a join attribute, which also happens to be the output attribute in this example. In the presence of appropriate existential and inclusion integrity constraints, both the join and the lookup can be eliminated.

Now we consider a deductive database example.

Example 2

EDB: Emp(Ssno,Salary,Deptno,Age)
Dept(Deptno,Managerssno,Floorno)
Sales(Deptno,Item,Vol)

IDB:

Highsalesdept(x1,x2,x3,y2,y3) ←
Dept(x1,x2,x3),Sales(x1,y2,y3),y3 > 100000
Highsalesmgrprofile(x2,y2,y4) ←
Emp(x2,y2,y3,y4),*Highsalesdept*(x1,x2,x3,x4,x5)

IC:

IC3: ← Dept(x,y,2)

IC4: (y > 40000) ← Emp(x,y,z,u),(u > 50)

The meaning of IC3 is that "There are no departments on floor 2." The meaning of IC4 is that "All employees whose age is greater than 50 have salary greater than 40000."

Highsalesdept and *Highsalesmgrprofile* are intensionally defined relations; in fact, *Highsalesmgrprofile* is defined in terms of *Highsalesdept*. A standard method for dealing with such a database is to reduce all definitions of intensional relations to extensional relations. As long as the axioms (rules) in IDB are not recursive, the body of every axiom can be reduced to purely extensional relations using the compiled method (see Reiter [23]). For example, in this case, by substituting the definition of *Highsalesdept* in the body of the clause defininining *Highsalesmgrprofile*, we obtain

Highsalesmgrprofile(x1,x2,x4)
← Emp(x1,x2,x3,x4),Dept(y1,x1,y2),Sales(y1,z2,z3),z3 > 100000

and now *Highsalesmgrprofile* is defined completely in terms of extensional relations.

Next we consider some queries for this database.

Q4: ← *Highsalesdept*(x*,y,2,z*,u)

This query asks for the deptno and item values that are sold in large quantity (i.e., volume > 100000) by that department. A deductive database system may solve this query by transforming it using the rule for *Highsalesdept* to

Q4′: ← Dept(x*,y,2),Sales(x*,z*,u), (u > 100000)

and then solving for this query in the extensional database. This can be done by lookups, using a join, and possibly indexing on Dept and Sales. However, it follows from IC3 and Q4′ that there are no answers.

Now, consider the query

Q5: ← *Highsalesmgrprofile*(x1*,x2*,x3), (x3 > 50)

This query asks for the Ssno and Salary values in *Highsalesmgrprofile* for employees whose Age > 50 (senior employees). After solving for *Highsalesmgrprofile* we obtain

$$Q5': \quad \leftarrow \text{Emp}(x1^*,x2^*,z,x3),\text{Dept}(y1,x1^*,y2),\text{Sales}(y1,z2,z3),z3 > 100000,\ x3 > 50$$

By using IC4 we can restrict the search of x2* (Salary) to values greater than 40000.

Intuitively, these examples indicate the utility of integrity constraints during query evaluation. Semantic query optimization formalizes the use of integrity constraints in this endeavor. In order to explain our logic-based approach, we need to introduce the notion of partial subsumption.

4.2 The Role of Partial Subsumption

We start with the definition of subsumption and then proceed to introduce partial subsumption. Briefly, a *substitution* (σ) is a finite set of the form {t1/v1, ..., tn/vn}, where vi's are unique variables and ti's are terms. If σ is a substitution and C is a clause, then Cσ is the clause obtained by simultaneously replacing each occurrence of the variable vi in C which is also in σ by the term ti. Cσ is called an *instance* of C. A clause C is a *subclause* of D if every literal in the clause C is also in D.

Definition. A clause C *subsumes* a clause D if there is a substitution σ such that Cσ is a subclause of D.

For example, if

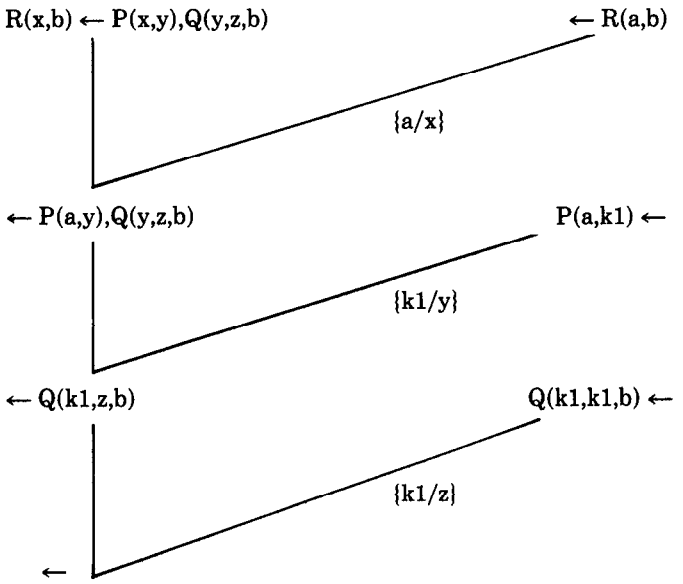
$$C = R(x,b) \leftarrow P(x,y),Q(y,z,b), \text{ and}$$
$$D = R(a,b) \leftarrow P(a,z),Q(z,z,b),S(a)$$

then C subsumes D by the substitution {a/x,z/y}.

Let us take a look at the queries that have no answers because of the integrity constraints in the above two examples. In Example 1 note how the integrity constraint IC1 subsumes the query Q1. The subsumption of a query by an integrity constraint is suggestive of an empty answer, but is not the way we want to proceed, because the subsumption test has to be carried out at run time. Instead, we wish to associate integrity constraints with relations before receiving any queries. In fact, we want to associate simpler integrity constraints than the given constraints. (An integrity constraint IC' is simpler than IC if it has fewer nonevaluable atomic literals.)

To understand partial subsumption, we need to look at the basic subsumption algorithm in Chang and Lee [8], testing to see if C subsumes D. We illustrate the subsumption algorithm on the example given above. First, D is instantiated to a ground clause by using new constants, not present in C or D. We will use k1, ..., kn for these constants and call the substitution θ. In the example, θ = {k1/z}, so that Dθ = R(a,b) ← P(a,k1),Q(k1,k1,b),S(a). Then Dθ is negated, and thereby a set of literals, $\neg$Dθ, is obtained. In this case, $\neg$Dθ = {← R(a,b), P(a,k1) ←, Q(k1,k1,b) ←, S(a) ←}. The algorithm tries to construct a linear refutation tree with C as the root, using at each step an element of $\neg$Dθ

in the resolution. C subsumes D if and only if at least one such refutation tree ends with the null clause. This is the case in this example.



In order to apply partial subsumption to extensional relations, we write the trivial axiom, R ← R, for each extensional relation. This is merely a notational device and is not meant as a recursive definition. Recall our remark about the integrity constraints subsuming queries in cases where no solutions exist because of the integrity constraints. As we stated earlier, the semantic compilation phase occurs before any queries are posed. Hence, we must use the integrity constraints and the axioms for the relations to anticipate later modifications to queries.

The essence of partial subsumption is to apply the subsumption algorithm to an integrity constraint and the body of an axiom. In general, the subsumption algorithm will not yield the null clause, because the integrity constraint does not subsume the body of the axiom. However, a subclause of the integrity constraint might subsume the body of the axiom. In that case we say that there is a partial subsumption of the body of the axiom by the integrity constraint. Instead of the null clause, a fragment of the integrity constraint remains at the bottom of a refutation tree. Such a fragment is called a *residue*. In order to obtain residues, as will be shown (see Section 4.3), it is necessary to transform an integrity constraint into an expanded form first. To illustrate this, let us return to Example 1 and suppose that we try to apply the subsumption algorithm to IC1 = ← Owners(x1,iceland,x3,x4) and the body of the axiom for Owners, ← Owners(y1,y2,y3,y4). Note that using our previous terminology, C = ← Owners(x1,iceland,x3,x4) and D = ← Owners(y1,y2,y3,y4). Thus, $\neg D\theta$ = {Owners(k1,k2,k3,k4) ←}, and no resolution is possible because the constants "iceland" and "k2" do not match, even though we would like to have a match there. In order to apply resolution a variable is needed in place of the constant "iceland". This is achieved by placing C into the expanded form, denoted by C+, where C and C+ are logically equivalent.

Definition. An integrity constraint IC *partially subsumes* an axiom A if IC does not subsume the body of A, but a subclause of IC+ subsumes the body of A.

Partial subsumption generalizes the notion of subsumption allowing the algorithm to generate fragments of an integrity constraint as residues instead of the empty clause.

4.3 Expanded and Reduced Forms

An integrity constraint is expanded by substituting variables in place of constants systematically, in order to apply subsumption and resolve as many literals as possible. This transformation is best explained by the individual transformations applied to the literals of an integrity constraint. We have shown in [5] that each individual transformation preserves logical equivalence.

Steps for Expansion

The expanded clause C+ is obtained from the clause C by modifying the body of C as follows:

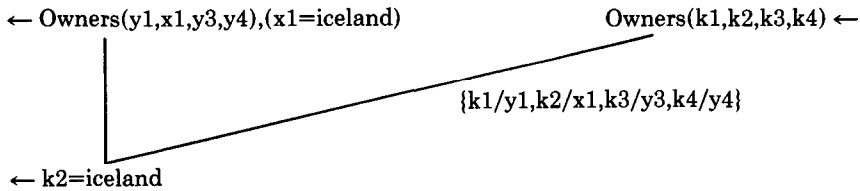
(1) An evaluable relation (predicate) that contains a constant and a variable is modified to two relations with the original relation containing two variables. For example, (u > c) is replaced by (u > x1),(x1 ≥ c). Similarly for other evaluable relations.

(2) An extensional relation that contains a constant or a variable that has occurred previously (that is, to its left in the clause) is modified by changing the constant or the variable to a new variable and adding an equality consisting of the constant or the variable and the new variable introduced. We use x followed by an integer for the new variables and assume that the integrity constraint does not already contain such variables.

Applying the above steps to IC1, we obtain

IC1+: ← Owners(y1,x1,y3,y4),(x1=iceland).

Illustrating partial subsumption, now we can try the subsumption algorithm using IC1+:



We obtain the clause ← k2=iceland at the bottom of the tree. Now we apply θ^{-1} (back substitution, defined below), because k2 stands for y2. The clause that remains is ← y2=iceland. This is the residue of IC1 and the axiom of Owners. We can interpret this clause as "y2 cannot be iceland for Owners(y1,y2,y3,y4)". Incorporating the result of partial subsumption, we rewrite the axiom for Owners as: Owners(y1,y2,y3,y4) ← Owners(y1,y2,y3,y4) {← y2=iceland} with the residue

in braces. Note how the residue anticipates the query Q1; we will discuss, in the next section, the usage of residues for processing queries.

The notion of back substitution is used to translate the constants introduced by θ into the original variables of an integrity constraint. Because of the way θ is defined, changing variables to the k constants, it is always possible to define θ^{-1}, the inverse of θ. For example, if θ = {k1/x,k2/y}, then θ^{-1} = {x/k1,y/k2}.

Note that back substitution merely undoes the substitution that was specifically introduced for the purpose of applying the partial subsumption algorithm. Even after back substitution, the resulting clause is likely to have the variables introduced during the expansion of the clause. Furthermore, the clause resulting from partial subsumption is likely to contain atoms (e.g., c=d, where c and d are constants) that can be eliminated without altering the meaning of the clause. The concept of a reduced clause is introduced for this purpose. Basically, reducing a clause is the opposite of expanding a clause. We write C- for the reduced form of C. Just as for expansion, C and C- are logically equivalent.

Steps for Reduction

The reduced clause C- is obtained from the clause C as follows:

(1) Delete the head if it is a false atom (example: c=d)
(2) In the body of C
    (a) delete every duplicate occurrence of an atom
    (b) delete every true atom (examples: c=c or x=x)
    (c) perform the reverse of step(1) in the Steps for Expansion
    (d) perform the reverse of step(2) in the Steps for Expansion

The reduction step essentially undoes the expansion that was performed during the expansion step. For an integrity constraint IC, IC-+- = IC-.

4.4 Residues

Here we explain the notion of residues informally and show how to generate them. Given an integrity constraint IC and an axiom A, apply the subsumption algorithm to IC+ and the body of A until no more resolutions are possible, and let B be the clause at the bottom of a refutation tree. Then (B-)θ^{-1}, that is, the clause obtained by the back substitution of the reduction of B is a *residue* of IC and A. Note that if a relation occurs more than once in the integrity constraint or the body of the axiom, then several residues may be generated when partial subsumption is applied.

Residues can be categorized in various ways. If IC subsumes the body of A, then we obtain the null clause as the residue: this is called the *null* residue. In this case the axiom adds no tuples to the intensional relation A; if the null residue is obtained for an extensional relation, it indicates that the relation is empty. When the residue is IC-, we call it a *maximal residue*; this indicates that no resolution was possible. A *redundant residue* is one that evaluates to true; for example a=a ←. Maximal and redundant residues are not useful for query optimization, because they show no interaction between the integrity constraint and the axiom. This leads to the notion of merge-compatibility.

Definition. An integrity constraint is *merge-compatible* with an axiom if at least one of the residues is nonredundant and nonmaximal.

Our definition of merge-compatibility differs from the definition given in [14] Section 3.5.1, because the latter involves an integrity constraint and a query, and it means that the universally quantified variables in the integrity constraint range over the same domains as the free variables in the query. Our definition involves an axiom and an integrity constraint.

Semantic compilation consists of generating all the residues for all the axioms. For each pair of integrity constraint and axiom, use the subsumption algorithm (along with the steps for obtaining expanded and reduced forms) for generating all the residues. Delete the redundant and maximal residues. For every redundant residue, go back one step in the refutation tree to find a new clause B. Add $(B\text{-})\theta^{-1}$ as a residue if it is not redundant or maximal. (We do so in this case because in a sense we have gone too far in the application of the subsumption algorithm by going to the bottom of the tree. As we will see in Section 5, the primary example for this situation occurs with functional dependencies.)

If an integrity constraint subsumes an axiom A, then A may be deleted. For each axiom, attach all residues, written in braces, as will be illustrated in the examples shown below. An axiom is called *semantically constrained* if all the residues applicable to it are associated with it. The semantic compilation phase transforms original axioms into semantically constrained axioms.

Before obtaining the residues (and thereby the semantically constrained axioms) in the two examples presented at the beginning of this section, we discuss the meaning of residues. A compiled intensional axiom $H \leftarrow$ P1, . . . , Pm (terms omitted) is transformed to its semantically constrained form $H \leftarrow$ P1, . . . , Pm {R1, . . . , Rn} where the Ri are residues. Let Ri have the form G ← F1, . . . , Fk and define Ri′ as not(F1, . . . , Fk,not(G)). In the case of a unit residue, Ri′ is G; for the empty residue, Ri′ is fail. Then the use of the residue Ri entails the addition of Ri′ to the definition of H. In particular, if all n residues are used, then we obtain $H \leftarrow$ P1, . . . , Pm,R1′, . . . , Rn′ as the meaning (declaratively and procedurally) of the semantically constrained axiom.

Example 3

Expressing the EDB of Example 1 as axioms and recalling the integrity constraints, we have

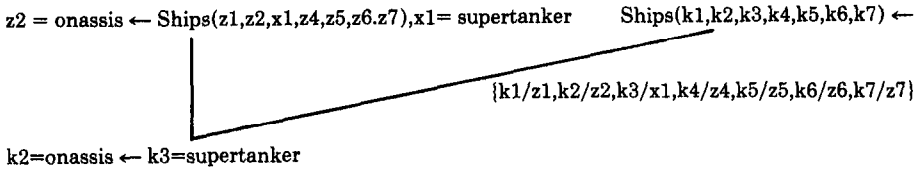
A1: Ships(z1,z2,z3,z4,z5,z6,z7) ← Ships(z1,z2,z3,z4,z5,z6,z7)
A2: Owners(y1,y2,y3,y4) ← Owners(y1,y2,y3,y4)

IC1: ← Owners(y1,iceland,y3,y4)
IC2: z2 = onassis ← Ships(z1,z2,supertanker,z4,z5,z6,z7)

It is clear that A1 is not merge-compatible with IC1 and A2 is not merge-compatible with IC2. For possible applications of IC1 and IC2 we must obtain the expanded forms.

IC1+: ← Owners(y1,x1,y3,y4),x1=iceland
IC2+: z2 = onassis ← Ships(z1,z2,x1,z4,z5,z6,z7),x1=supertanker

We have previously shown that the residue for A2 and IC1 is ← z2=iceland. In applying the subsumption algorithm to IC2 and A1 we obtain



After back substitution we obtain the residue: z2=onassis ← z3=supertanker. So the semantically constrained axioms are:

SCA1: Ships(z1,z2,z3,z4,z5,z6,z7) ← Ships(z1,z2,z3,z4,z5,z6,z7)
{z2=onassis ← z3=supertanker}
SCA2: Owners(y1,y2,y3,y4) ← Owners(y1,y2,y3,y4) { ← y2=iceland}

According to our earlier discussion, the meanings of these semantically constrained axioms are as follows:

for SCA1: Ships(z1,z2,z3,z4,z5,z6,z7) ←
Ships(z1,z2,z3,z4,z5,z6,z7),not(z3=supertanker, not(z2=onassis))
for SCA2: Owners(y1,y2,y3,y4) ← Owners(y1,y2,y3,y4),not(y2=iceland)

Example 4

Compiling the EDB and the IDB and expanding the integrity constraints of Example 2, we obtain

A3: Emp(x,y,z,u) ← Emp(x,y,z,u)
A4: Dept(x,y,z) ← Dept(x,y,z)
A5: Sales(x,y,z,u) ← Sales(x,y,z,u)
A6: *Highsalesdept*(x1,x2,x3,y2,y3) ← Dept(x1,x2,x3),Sales(x1,y2,y3), y3 > 100000
A7: *Highsalesmgrprofile*(x1,x2,x4) ← Emp(x1,x2,x3,x4),Dept(y1,x1,y2), Sales(y1,z2,z3),(z3 > 100000)
IC3+: ← Dept(x,y,z), (z = 2)
IC4+: (y > 40000) ← Emp(x,y,z,u),(u > 50)

The semantically constrained axioms are:

SCA3: Emp(x,y,z,u) ← Emp(x,y,z,u) {(y > 40000) ← (u > 50)}
SCA4: Dept(x,y,z) ← Dept(x,y,z) {← z=2}
SCA5: Sales(x,y,z,u) ← Sales(x,y,z,u) { }
SCA6: *Highsalesdept*(x1,x2,x3,y2,y3) ← Dept(x1,x2,x3),Sales(x1,y2,y3), (y3 > 100000)
{ ← x3=2}
SCA7: *Highsalesmgrprofile*(x1,x2,x4) ← Emp(x1,x2,x3,x4),Dept(y1,x1,y2), Sales(y1,z2,z3),(z3 > 100000)
{ (x2 > 40000) ← (x4 > 50); ← y2=2}

The meanings for the semantically constrained axioms are as follows:

for SCA3: Emp(x,y,z,u) ← Emp(x,y,z,u),not((u > 50),not(y > 40000))
for SCA4: Dept(x,y,z) ← Dept(x,y,z),not(z=2)
for SCA6: *Highsalesdept*(x1,x2,x3,y2,y3) ← Dept(x1,x2,x3),Sales(x1,y2,y3), (y3 > 100000),not(x3=2)
for SCA7: *Highsalesmgrprofile*(x1,x2,x4) ← Emp(x1,x2,x3,x4),Dept(y1,x1,y2), Sales(y1,z2,z3),(z3 > 100000),not((x4 > 50),not(x2 > 40000)),not(y2=2)

Finally, we mention that in some cases it may be useful to apply deduction to obtain derived integrity constraints and consequently additional residues. Functional dependencies provide such an example. For instance, if R(x,y,z) is a relation with the two functional dependencies $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$ follows by transitivity. The latter functional dependency may yield a useful residue unobtainable from the original functional dependencies.

5. SEMANTIC QUERY TRANSFORMATION

In the previous section we introduced semantic compilation and elaborated on the notion of partial subsumption. Residues, generated as a result of partial subsumption, contain all the necessary information that can potentially be used during actual query evaluation. These residues are associated with the deductive rules (compiled intensional axioms as well as trivial rules generated for extensional relations) to form semantically constrained axioms.

In this section we expand on the process that actually transforms a query into one or more semantically equivalent queries. Informally, a query is semantically equivalent to the original query if the two queries have the same answers for all databases that have the same IDB and IC. We preserve semantic equivalence during the transformation of a query (using the residues associated with it) by using a sound inference rule (SLDNF). Semantic query transformation is predicated upon the interaction of residues with the query; the process of harnessing this interaction into useful transformations is the theme of this section.

The functional components of a semantic transformer are presented in Figure 4. The query/residue modifier uses all applicable semantically constrained axioms to transform a query into (possibly) a set of queries, where the union of the answers to each query corresponds to the answers of the original query. As the name suggests, along with the modification of the queries (using unification or pattern matching), the residues are also modified appropriately. This is followed by a reduction phase as variables in the residues may become instantiated as a result of query/residue modification. At this stage, it is appropriate to filter and remove residues that will not be useful for transforming the modified query based upon the instantiations and literals in the original query. This filter essentially reduces the number of residues using various criteria. The strategiser incorporates heuristics (or works using a set of heuristics supplied to it) to prioritize the residues for consideration by the generator. For example, if restricting an indexed attribute is considered important, the strategiser will search for relations into which a restriction can be introduced on the required attribute.

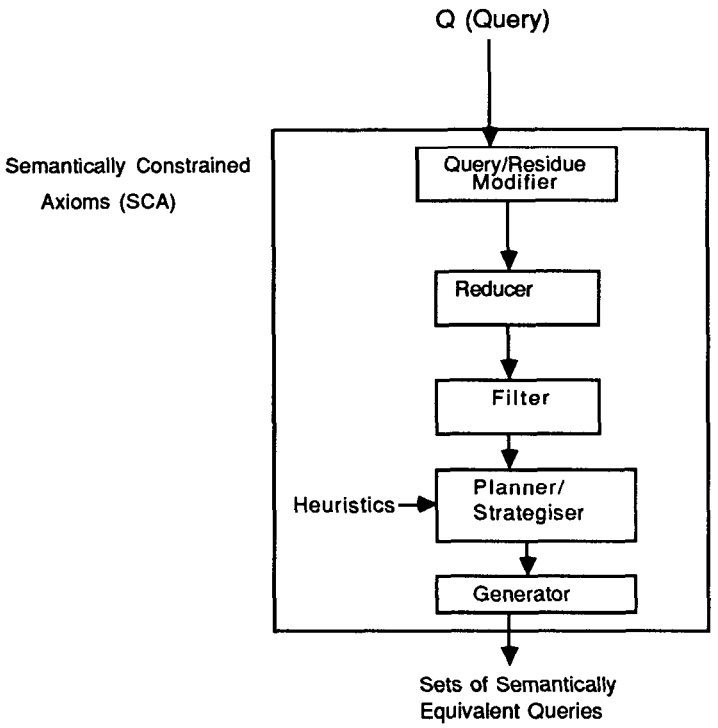


Fig. 4. Functional components of semantic query transformer.

The generator essentially uses the resolution principle in determining whether a required transformation is possible and generates queries that are semantically equivalent to the original one. The strategiser has been separated from the generator in order to emphasize their functionalities and make the architecture extensible. This separation also helps in allowing feedback from other levels of query evaluation for revising/changing the heuristics.

In this section we start with a set of semantically constrained axioms and demonstrate the processes of query/residue modification, reduction, filtering, and the generation of semantically equivalent queries. We elaborate on the characteristics of the residues and how they interact with the query during the generation of semantically equivalent queries. We do not elaborate on the strategiser apart from assuming that the strategiser orders the residues for transformation using known heuristics. We allude to the set of heuristics that are commonly used for semantic query transformation and indicate how they relate to the types of residues and predicates in our formalism. Since the strategiser is the component that needs to interact with the relational query optimizer (shown in Figure 2), we suggest strategies that can be used to interface the SQO to a conventional relational optimizer. This interaction, however, is not shown in Figure 4. Finally, the concepts and the process of semantic transformation are explained through intuitively understandable, simple examples.

5.1 Query/Residue Modification

Consider the following deductive database partly based on an example in [12]:

EDB:

E1: Employee(Name,Salary,Dname)
E2: Dept(Dname,Function,Mname)
E3: Emp-cars(Tagno,Ownedby,Color)
E4: Company-cars(Owner,Tagno,Dept,Color,Miles)

IDB:

A8: *Works_dir_for*(n1,n2) ← Employee(n1,s1,d),Dept(d,f1,n2)
A9: *Car_owners*(o,d,t,c) ← Employee(o,s,d),Emp-cars(t,o,c)
A10: *Car_owners*(o,d,t,c) ← Company-cars(o,t,d,c,mi)

IC:

"Salaries of all employees are less than 70000 and greater than 10000"
IC5a: (s1 > 10000) ← Employee(n1,s1,d1)
IC5b: (s1 < 70000) ← Employee(n1,s1,d1)

"Employee name functionally determines salary and department name"
IC6a: (s1=s2) ← Employee(n1,s1,d1),Employee(n1,s2,d2)
IC6b: (d1=d2) ← Employee(n1,s1,d1),Employee(n1,s2,d2)

"Every car in the Emp-cars relation is owned by an Employee"
IC7: Employee(o,f1(t,o,c),f2(t,o,c)) ← Emp-cars(t,o,c)
where f1 and f2 are Skolem functions.

The above integrity constraint expresses the subset relationship between the Ownedby attribute in the Emp-cars relation and the Name attribute in the Employee relation.

Not considering integrity constraints, for a query such as

"List the names of owners of red cars who work in a department with publicity function"

Q6: ← *Car-owners*(o*,d,t,red),Dept(d,publicity,m)

the following queries are generated when the query modifier elaborates intensionally defined relations into extensionally defined relations using the compiled axioms (in this example the compiled axioms are the same as the axioms given).

Q6′: ← Employee(o*,s,d),Emp-cars(t,o,red),Dept(d,publicity,m)
Q6″: ← Company-cars(o*,t,d,red,mi),Dept(d,publicity,m)

The union of the answers to the above two queries is the answer to Q6. Q6′ and Q6″ may be solved individually by a relational database processor; alternately, a deductive database query processor may do global optimization or multiple query optimization (see [9, 24, and 6]).

Next we present the semantically constrained axioms for the above deductive database.

SCA8: Employee(n,s,d) ← Employee(n,s,d)
{(s > 10000) ←; (s < 70000) ←;
(s=s2) ← Employee(n,s2,d2); (d=d2) ← Employee(n,s2,d2)}
SCA9: Dept(d,f,m) ← Dept(d,f,m) { }

SCA10: Emp-cars(t,o,c) ← Emp-cars(t,o,c)
{Employee(o,f1(t,o,c),f2(t,o,c)) ←}

SCA11: Company-cars(o,t,d,c,mi) ← Company-cars(o,t,d,c,mi) { }

SCA12: *Works_dir_for*(n1,n2) ← Employee(n1,s1,d),Dept(d,f1,n2)
{ (s1 > 10000) ←; (s1 < 70000) ←;
(s1=s2) ← Employee(n1,s2,d2); (d=d2) ← Employee(n1,s2,d2)}

SCA13: *Car_owners*(o,d,t,c) ← Employee(o,s,d),Emp-cars(t,o,c)
{Employee(o,f1(t,o,c),f2(t,o,c)) ←;
(s > 10000) ←; (s < 70000) ←;
(s=s2) ← Employee(o,s2,d2); (d=d2) ← Employee(o,s2,d2)}

SCA14: *Car_owners*(o,d,t,c) ← Company-cars(o,t,d,c,mi) { }

Note that the residues shown above have been statically filtered as indicated in Figure 3. Residues that are the same up to renaming of variables (in the residue only) have been eliminated. For example, IC6a generates

(s=s2) ← Employee(n,s2,d2), and
(s=s1) ← Employee(n,s1,d1)

for the base relation Employee as well as the intensional relation *Car_owners*. One of them is eliminated.

Residue modification is tightly coupled with the elaboration of queries using compiled axioms. When a literal in a query is unified with the head of a compiled axiom, the unifier generated is applied to the body of the compiled axiom as well. Residue modification expands the scope of the unifier to include the residues associated with the compiled axioms. For instance, given the query

"List the employees in the marketing department with salary 20000 who work directly for smith"

Q7: ← *Words_dir_for*(n*,smith),Employee(n*,20000,marketing)

when modified using the axioms SCA8–SCA14 results in

← Employee(n*,s1,d),Dept(d,f1,smith)
{ (s1 > 10000) ←; (s1 < 70000) ←;
(s1=s2 ← Employee(n*,s2,d2); d=d2 ← Employee(n*,s2,d2)},
Employee(n*,20000,marketing)
{ (20000 > 10000) ←; (20000 < 70000) ←;
(20000=s) ← Employee(n*,s,d2);
(marketing=d2) ← Employee(n*,s,d2)}

The residues have been instantiated with constants from the query. As a result, it may be possible to evaluate some of the residues. For example, the evaluable relations (20000 > 10000) and (20000 < 70000) can be evaluated to be true, and hence can be eliminated. The reduction step shown in Figure 4 essentially evaluates residues that have been sufficiently instantiated. Formally, if q is a literal in a query Q and sca is a semantically constrained axiom with hsca as the head and bsca as the body and if θ is the unifier that unifies q and hsca, then $(bsca)\theta \cup (Q-q)\theta$ is the modified query.

It is clear that the reduction step eliminates residues that have become instantiated in such a way that they can be evaluated immediately. Apart from these, a variety of characteristics can be used to eliminate residues that will not be useful for query transformation. One such criterion is the presence of a literal

in the body of a residue that does not appear in the query at all. For instance, if the query is

Q8: "List all the employees in the marketing department and their managers' names"

the elaborated query will be

← Employee(n*,s,marketing)
{ (s > 10000) ←; (s < 70000) ←;
(s=s2) ← Employee(n*,s2,d2);
(marketing=d2) ← Employee(n*,s2,d2)},
Dept(marketing,f,m*) { }.

Here, the third and fourth residues are not useful, because there is no other Employee occurrence in the query.

The examples shown above illustrate how queries obtain their residues from the constrained axioms as well as some of the criteria used in the simplification of residues. We now want to apply the residues for query transformation. We will discuss strategies for actually dealing with several residues later, but first we discuss the use of a single residue at a time. We will use the theorem proved in [5] that every residue in SCQ must be true in the database.

5.2 Residue Categorization and Equivalent Query Generation

Though residues are clauses in general, it helps to categorize them according to their clause form in order to appreciate their interaction with the literals in a query. This categorization also guides the strategiser in looking for the types of residues that help perform a specific transformation. We classify residues into:

(1) the null clause,
(2) a goal clause,
(3) a unit clause (empty body, single atom in the head), and
(4) a Horn clause with nonempty body and nonempty head.

It is also helpful to further classify the last three categories with respect to evaluable predicates (such as a unit clause with an evaluable predicate, etc.). We briefly discuss the above residue types and their transformation capabilities.

The residue is the null clause. When the null residue is obtained at any stage of semantic query optimization (this can happen during semantic compilation, query/residue modification, or reduction), the axiom/query has no answers. This is so because the axiom/query and the integrity constraints logically imply a contradiction for any possible answer. If a null residue is obtained during the semantic compilation phase, then the axiom, for which a null clause is generated, does not generate any tuples for the intensional relation. If it happens for an extensional relation, then the extensional relation is not consistent. Assuming that the database is consistent with respect to the integrity constraints, we are interested in the case where the null clause for a query is derived based on the instantiations in the query. For instance, a modified version of Q8 requesting the set of all employees in the marketing department earning 5000 will generate a

null clause as a residue. From this, one can infer that there are no employees satisfying the salary criterion.

← Employee(n*,5000,marketing)
{←; . . . }

The residue is a goal clause. There are two possibilities here. If the residue subsumes the query, then the query cannot have any answers. In this case, the query cannot be satisfied by the database, as the corresponding integrity constraint (which generated the residue) generates a null clause. In other words, the answers to the query violate the integrity constraint and hence there are no answers for the query. Even if the residue does not subsume the query, it may be reduced to a goal clause of evaluable relations, which can still be useful. For instance, consider the residue ← (z < 200). This residue states that z cannot be less than 200, hence z must be at least 200. This information may be helpful, if the relation in which z is an attribute is ordered on the attribute z, or if it has an index on the attribute z. Alternatively, a goal clause consisting only of an evaluable predicate can be transformed to a unit subclause. For example, the goal clause, ← (z < 200) can be changed to, (z ≥ 200) ←, a unit clause, by changing it to the opposite evaluable relation and thereby avoiding negation.

The residue is a unit clause. This unit clause can be asserted in the database. The effect of this assertion depends on whether or not it is an evaluable relation. This residue form can be used in several ways. It can be used to eliminate a literal (join elimination) from the query, if there is a literal in the query that can be resolved with the unit residue. It can also be used to introduce a literal (join introduction) by adding a nonevaluable relation to the query. Finally, it can be used to introduce various forms of restrictions into the query.

Care must be taken in performing join elimination: no output variable should be eliminated, nor should the terms be changed for other atoms in the query. For instance, the query

Q9: "Find all car owners who own a red car"

on the database of Section 5.1 reduces to

Q9′: ← Employee(o*,s,d)
{(s > 10000) ←; (s < 70000) ←;
(s=s2) ← Employee(o*,s2,d2); (d=d2) ← Employee(o*,s2,d2)},
Emp-cars(t,o*,red)
{Employee(o*,f1(t,o*,red),f2(t,o*,red)) ←}

Query Q9′ can be simplified to ← Emp-cars(t,o*,red), as the relation Employee can be eliminated from the query using the unit residue

Employee(o*,f1(t,o*,red),f2(t,o*,red)) ←.

Note, however, that if the query was "Find all car owners in the toy department who own a red car", then the elimination could not be performed, because the Employee predicate with a constant (toy) could not be resolved with the above Employee residue containing Skolem functions.

The following condition formalizes a rule for join elimination, where the semantically constrained query is

SCQ: $\leftarrow Q1, \ldots, Qn$ {U←, other residues}

Join elimination is possible if U← can be resolved with any of the Q's (say Qi) and the resulting resolvent contains both the original output variables and the original terms in all Qj, $j \neq i$. The resolvent is then substituted for the query. For example, the semantically constrained query ← Q1(a,x*,y),Q2(x*,z) {Q1(u,v,b)←} can be transformed to ← Q2*(x*,z) whereas, the semantically constrained query ← Q1(a,x*,y), Q3(x*,y,z) {Q1(u,v,b)←} cannot be transformed as the variable y in Q3 is changed to a constant.

On the other hand, join introduction, though not intuitive, can be used beneficially to transform a query under certain situations. If the query calls for the cartesian product of two relations, and if a third relation can be introduced, thereby replacing the cartesian product by two joins, then join introduction is likely to be beneficial. Also, when a query involves joining two large relations using sequential scan (for lack of an index) the introduction of a small (compared to the other two) relation with a restriction on join attributes can be extremely useful. For example, if

SCQ: ← R1(x*,y,z),R2(z,a) {R3(b,z) ←}

and R1 and R2 are large, while R3 is small, the new query ← R1(x*,y,z),R2(z,a), R3(b,z) may be evaluated faster by doing the join of R2 and R3 first.

There are other ways in which a unit clause can be used. If the unit clause is an evaluable relation, then a restriction can be introduced into the query. This is what, in effect, is achieved when an evaluable goal residue such as, ← ($z < 200$) is changed to ($z \geq 200$) ← and added to the query. We call this restriction introduction. In some cases, an evaluable relation in a residue subsumes the corresponding evaluable relation in a query. For example, ← R(x*,y,z),($z < 100$) {($z < 50$) ←} allows us either to change the query to ← R(x*,y,z),($z < 50$) by substituting the stronger restriction or to eliminate the relation altogether and obtain ← R(x*,y,z), since the restriction must be true anyway.

The residue is a Horn clause with both a nonempty body and a nonempty head. We deal with two subcases. First, consider the case where the body of the residue subsumes the query, and let θ be the substitution for subsumption. Then, if C is the head of the residue, $C\theta$ must be true for the query, so it can be treated as if it were a unit clause residue. For example, if the query is ← R3(x*,z),R1(a,x*,y),R2(x*,z) {R2(x*,z) ← R1(a,x*,w)} then the new query is ← R3(x*,z),R1(a,x*,y) by join elimination, using the residue as a unit clause, after obtaining the residue, R2(x*,z) ←.

The second subcase occurs when the residue is used to limit the search in evaluating a query. Probably the most important case here is the use of functional dependencies. Recall that the functional dependency R: $X \rightarrow Y$ for R(X,Y,Z) yields R(x,y,z) ← R(x,y,z) {$y=y'$ ← R(x,y',z')}. Now consider the query ← R(a,y*,z). The residue is interpreted as saying that a second y' value must be the same as any y* value already obtained. Therefore, the search can be limited to a single value.

A note on the difference between existentially quantified variables and universally quantified variables in integrity constraints is in order here. Integrity constraints that have only universally quantified variables may be vacuously true on the database, because the body of the integrity constraint may not be true and hence the consequence need not be verified. So, when an output variable is instantiated using the values in residues, it does not obviate the need for a lookup. The following example illustrates this point. Let R(x,y,z) ← R(x,y,z) {(y < 100) ← x=a} be a constrained axiom and let ← R(a,y*,z) be the query. The elaborated query is ← R(a,y*,z) {(y < 100) ←}. By restriction introduction we can limit the search to y values satisfying the condition y < 100. Suppose we have R(x,y,z) ← R(x,y,z) {y=b ← x=a} as the constrained axiom with the query ← R(a,y*,z). The elaborated query is ← R(a,y*,z) {y*=b ←}. Although we can instantiate y* to b at this point, we cannot provide b as the answer. We can only say that b is the only possible answer, but it has to be verified by a lookup in the R relation. If, however, there is an existential integrity constraint asserting the presence of a tuple with b as the second attribute value, then there is no need for a lookup.

5.3 Heuristics and Interface Issues

Up to this point we have dealt with a single residue at a time in generating residues for an axiom and in instantiating a residue for a query. When there are numerous integrity constraints associated with a database, many residues may be generated. This could lead to a large number of residues associated with a given query. Our residue elimination strategies for avoiding the generation of useless residues have already been given at the beginning of Section 4.4: we eliminate redundant and maximal residues during the semantic compilation phase. Sometimes residues are obtained which do not contain in their body (if nonempty) or in their head (if nonempty) any variables present in the axiom defining an extensional relation. Such a residue cannot be useful and may be eliminated. One other method for eliminating residues can be used if two or more residues are identical up to the renaming of variables that appear only in the residues; in this case, all except one of these residues may be eliminated. Note that these strategies to reduce the number of residues are applied during the process of semantic compilation, prior to query processing.

We briefly introduced some of the heuristics used for query transformation in Section 3.4. The previous subsection established the relationship between the heuristics (proposed in the literature) and the residues generated using integrity constraints. Though semantic query optimization can be formulated as a separate task, it is ultimately used in the context of a large deductive/relational database. Towards that goal, we suggest various ways of interfacing a semantic query optimizer with a conventional relational query optimizer. Three distinct combinations of the query transformer and the plan generator can be envisaged.

One possibility is for query transformation and plan generation to be independent tasks. The query transformer generates all the semantically equivalent queries exhaustively; the plan generator estimates the cost of processing each query and decides on the optimal version. This approach has been implemented on top of an existing plan generator (see [18]).

A second possibility is for the query transformer to be equipped with heuristics, based on the availability of indexes and cardinality information, to guide the query transformation process, so that only promising semantically equivalent queries are generated. Such heuristics include join elimination, join introduction, and restriction introduction. This approach is especially interesting as it permits information from the plan generator to be fed back to the strategiser. This feedback can be used to revise the heuristics to look for transformations that are *known* to be beneficial.

The third approach allows the two tasks to be tightly integrated. This approach permits the generation and modification of a plan dynamically and performs query transformation along the answer computation process. Also, statistics gathered for intermediate computations can be used to guide query transformation. Such an approach may be particularly suitable for databases with recursive intensional relations, which we will discuss in Section 7.

6. APPLICATIONS OF THE LOGIC-BASED APPROACH

Having presented our approach to semantic query optimization, we justify, in this section, our earlier claim that the approach presented in this paper is very general and encompasses earlier work on semantic query optimization. We do so by showing how the semantic query optimization problems given by other papers, are solved using our method. For this purpose we take examples from those papers, transform them to our notation, and apply our query transformation techniques to them in order to obtain semantically equivalent queries. These examples are taken from [12], [14], [25], and [27]. We concentrate here on the representation of integrity constraints and the generation of semantically equivalent queries, rather than the heuristics used in deciding which semantically equivalent query is best based on some cost model. The remainder of this section may be skipped by those readers who are not interested in the rigorous application of SQO techniques, discussed so far, on concrete examples. Each subsection is self-contained which is reflected in the numbering scheme used.

6.1 Examples from QUIST

We start by rewriting the major example from [14] using our notation.

EDB: Ships(Shipname,Owner,Shiptype,Draft,Deadweight,Capacity,
Registry)
Ports(Portname,Country,Depth,Facilitytype)
Cargoes(Ship,Destination,Shipper,Cargotype,Quantity,Dollarvalue,
Insurance)
Owners(Ownername,Location,Assets,Business)
Policies(Policy,Issuer,Coverage)
Insurers(Insurer,Insurercountry,Capitalization)

IC:

IC1: z4=offshore ← Ships(x1,x2,x3,x4,x5,x6,x7),
Cargoes(x1,y2,y3,y4,y5,y6,y7),
Ports(y2,z2,z3,z4),(x5 > 350)

IC2: y4=leasing ← Ships(x1,x2,x3,x4,x5,x6,x7),Owners(x2,y2,y3,y4),
(x5 > 300)

IC3: (y3 ≤ x6) ← Cargoes(x1,x2,x3,x4,x5,x6,x7),Policies(x7,y2,y3)
IC4: (y5 ≤ x6) ← Ships(x1,x2,x3,x4,x5,x6,x7),
Cargoes(x1,y2,y3,y4,y5,y6,y7)
IC5: y4=general ← Cargoes(x1,x2,x3,x4,x5,x6,x7),Ports(x2,y2,y3,y4),
(x6 > 500),x4≠lng,x4≠refined
IC6: x3=supertanker ← Ships(x1,x2,x3,x4,x5,x6,x7),(x5 > 150)

(Note: This is the modified version of IC6 as given on page 73 of [14].)

IC7: z2=lloyds ← Ships(x1,x2,supertanker,x4,x5,x6,x7),
Cargoes(x1,y2,y3,y4,y5,y6,y7),Policies(y7,z2,z3),
(y6 > 3000)
IC8: y4=lng,y4=refined,y4=oil ← Ships(x1,x2,x3,x4,x5,x6,x7),
Cargoes(x1,y2,y3,y4,y5,y6,y7),
Owners(x2,z2,z3,petroleum)

We now indicate the semantically constrained axioms noting that we have eliminated several residues in accordance with the discussion in Section 5.1.

SCA1: Ships(x1,x2,x3,x4,x5,x6,x7) ← Ships(x1,x2,x3,x4,x5,x6,x7)
{(y5 ≤ x6) ← Cargoes(x1,y2,y3,y4,y5,y6,y7);
x3=supertanker ← (x5 > 150)}
SCA2: Ports(x1,x2,x3,x4) ← Ports(x1,x2,x3,x4)
{x4=offshore ← Ships(y1,y2,y3,y4,y5,y6,y7),
Cargoes(y1,x1,z3,z4,z5,z6,z7),(y5 > 350);
x4=general ← Cargoes(y1,x1,y3,y4,y5,y6,y7),
(y6 > 500),y4≠lng,y4≠refined}
SCA3: Cargoes(x1,x2,x3,x4,x5,x6,x7) ← Cargoes(x1,x2,x3,x4,x5,x6,x7)
{(y3 ≤ x6) ← Policies(x7,y2,y3);
(x5 ≤ y6) ← Ships(x1,y2,y3,y4,y5,y6,y7);
x4=lng,x4=refined,x4=oil ← Ships(x1,y2,y3,y4,y5,y6,y7),
Owners(y2,z2,z3,petroleum)}
SCA4: Owners(x1,x2,x3,x4) ← Owners(x1,x2,x3,x4)
{x4=leasing ← Ships(y1,x1,y3,y4,y5,y6,y7),(y5 > 300)}
SCA5: Policies(x1,x2,x3) ← Policies(x1,x2,x3)
{ (x3 ≤ y6) ← Cargoes(y1,y2,y3,y4,y5,y6,x1)
x2=lloyds ← Ships(y1,y2,supertanker,y4,y5,y6,y7),
Cargoes(y1,z2,z3,z4,z5,z6,x1),(z6 > 3000)}
SCA6: Insurers(x1,x2,x3) ← Insurers(x1,x2,x3) { }

King discusses six types of query transformation. We deal with each case to show how these are performed in our approach.

(1) *Index Introduction*

Q1 : ← Ships(x1,x2*,x3,x4,x5,x6,x7), (x5 > 200)
SCQ1: ← Ships(x1,x2*,x3,x4,x5,x6,x7)
{(y5 ≤ x6) ← Cargoes(x1,y2,y3,y4,y5,y6,y7);
x3=supertanker ← (x5 > 150)},
(x5 > 200)

In this case it is possible to use the second residue for index introduction to obtain

Q1′: ← Ships(x1,x2*,supertanker,x4,x5,x6,x7), (x5 > 200)

The body of the residue subsumes an evaluable literal in the query; this yields a unit residue that is added to the query.

(2) *Join Elimination*

Q2: ← Cargoes(x1,marseille,x3*,lng,x5*,x6,x7),Ships(x1,y2,pressurized_tanker,y4,y5,y6,y7)

In this case the following additional constraint is used.

IC9: y3=pressurized_tanker ← Cargoes(x1,x2,x3,lng,x5,x6,x7), Ships(x1,y2,y3,y4,y5,y6,y7)

Although two residues are obtained from IC9 (one for Cargoes and one for Ships), they cannot be used for join elimination as claimed in [14], because the Ships relation may be empty, for instance. However, in the presence of a subset constraint such as

Ships(x1,f2,pressurized_tanker,f4,f5,f6,f7) ←
Cargoes(x1,x2,x3,lng,x5,x6,x7)

where the fi, i = 2 to 7, are Skolem functions (variables omitted), the relation Ships can be eliminated. (A similar idea, called structural constraint, is mentioned in Appendix B of [14].)

(3) *Scan Reduction*

Q3: ← Ships(x1,x2*,x3,x4,x5,x6,x7),Ports(y1,denmark,y3,y4), Cargoes(x1,y1,z3,refined_petroleum,z5*,z6,z7)

In this case also, an additional constraint is used

IC10: (x5 < 60) ← Ships(x1,x2,x3,x4,x5,x6,x7), Cargoes(x1,z2,z3,refined_petroleum,z5,z6,z7)

Using IC10, the residue (x5 < 60) ← is obtained, and we can add it to the query Q3 to obtain

Q3′: ← Ships(x1,x2*,x3,x4,x5,x6,x7),Ports(y1,denmark,y3,y4), Cargoes(x1,z2,z3,refined_petroleum,z5,z6,z7),(x5 < 60)

Again a unit evaluable predicate is used to restrict the value of an attribute.

(4) *Join Introduction*

Q4: ← Ships(x1,x2*,supertanker,x4,x5,x6,x7),Ports(y1,france,y3,y4), Cargoes(x1,y1,z3,z4,z5,z6,z7),(x5 > 350)

King introduces a join with the Owner relation in this example. If the relevant rule: "Only leasing companies own vessels that exceed 300 thousand tons", means that for every ship with more than 300 thousand tons, there exists a leasing company that owns it, the constraint is

Owners(x2,f1,f2,leasing) ← Ships(x1,x2,x3,x4,x5,x6,x7),(x5 > 300),

where f1 and f2 are Skolem functions for which we have omitted the variables. (As stated above, this is similar to the notion of structural constraint as defined in [14].) This gives rise to the constrained query

← Ships(x1,x2*,supertanker,x4,x5,x6,x7),Ports(y1,france,y3,y4), Cargoes(x1,y1,z3,z4,z5,z6,z7),(x5 > 350) {Owners(x2, f1, f2, leasing) ← }

which can be used to introduce the literal Owners into the query. The Skolem functions are essentially treated as variables. By using SCA2 we can obtain

Q4′: ← Ships(x1,x2*,supertanker,x4,x5,x6,x7),
Ports(y1,france,y3,offshore),
Cargoes(x1,y1,z3,z4,z5,z6,z7),(x5 > 350)

which may increase efficiency.

(5) *Detection of Unsatisfiable Conditions*

Q5: ← Ships(x1,x2*,bulk_cargo,x4,x5,x6,x7),(x5 > 200)
SCQ5: ← Ships(x1,x2*,bulk_cargo,x4,x5,x6,x7)
{(y5 ≤ x6) ← Cargoes(x1,y2,y3,y4,y5,y6,y7);
← (x5 > 150)},
(x5 > 200)

In this case, the second residue introduces a contradiction with the second literal, so there are no answers.

(6) *Absence of Opportunities for Cost Reducing Semantic Transformations*

Q6: ← Ships(x1,x2*,refrigerated,x4,x5,x6,x7)
SCQ6: ← Ships(x1,x2*,refrigerated,x4,x5,x6,x7)
{(y5 ≤ x6) ← Cargoes(x1,y2,y3,y4,y5,y6,y7);
← (x5 > 150)}

It is observed in [14] that there is no opportunity for a useful transformation for the above query. However, the second residue might be used to obtain

Q6′: ← Ships(x1,x2*,refrigerated,x4,x5,x6,x7),(x5 ≤ 150)

which introduces a restriction on the attribute Deadweight.

6.2 Examples from the Relational Algebraic Approach of [27]

We start by showing that the integrity constraints of this paper can be represented in our notation. Three types of integrity constraints are considered: domain rules, dependency rules, and production rules. We give an example of each.

Domain rule.	(xi < a) ← R(x1, . . . , xn)
Dependency rule.	(x3 < c) ← R(x1,x2,x3,x4),(x1 > a),(x2 > b)
Production rule.	(x4 > c) ← R1(x1,x2,x3),R2(x4,x5),(x4 < x3)

In Section 6 of [27] comparisons are made with the methods of [14]. Some of this comparison is specific to various physical considerations not considered in [14].

Example 5

EDB: Applicant(Ssno,Jobtitle,Officeno)
IC: y=programmer ← Applicant(x,y,17)

We obtain

SCA: Applicant(x,y,z) ← Applicant(x,y,z) {y=programmer ← z=17}

The query and its variations are

Q: ← Applicant(x*,y,17)
SCQ: ← Applicant(x*,y,17) {y=programmer ←}
Q′: ← Applicant(x*,programmer,17)

A constant is introduced into the query.

Example 6
EDB: Emp(Ename,Job,Dno)
Dept(Dno,Dname,Loc)
IC: Dno=15 ← Dept(Dno,Dname,a)

We obtain

SCA1: Emp(x,y,z) ← Emp(x,y,z) { }
SCA2: Dept(x,y,z) ← Dept(x,y,z) {x=15 ← z=a}

The query and its variations are

Q: ← Emp(x*,secretary,z),Dept(z,u,a)
SCQ: ← Emp(x*,secretary,z),Dept(z,u,a) {z=15 ←}
Q′: ← Emp(x*,secretary,15),Dept(15,u,a)

The claim in [27] is incorrect: the query Q′ cannot be simplified further (and reduced to the query ← Emp(x*,secretary,15)) by eliminating the Dept relation using only the integrity constraint IC. Additional constraints are required to simplify Q′ further.

Queries with disjunctive selection predicates (e.g., ← Applicant(x*, y, z), (y = programmer V z = 17) from [27] on the database of Example 5, above) can be expressed as union compatible queries (← Applicant(x*,programmer,z), ← Applicant(x*, y, 17) for the above example). These can be simplified, as shown in [27], using residues. However, we do not deal with simplifying multiple queries using residues.

6.3 Examples from [12]

Only a portion of this paper deals with semantic query optimization. Three types of integrity constraints are considered: value bounds, functional dependencies, and referential integrity constraints. In our notation, the example database is:

Example 7
EDB: Empl(Eno,Nam,Sal,Dno)
Dept(Dno,Fct,Mgr)
IDB: *Works_dir_for*(x,y) ←
Empl(z1,x,z2,z3),Dept(z3,z4,z5),Empl(z5,y,z6,z7)
IC:
IC1: (x3 ≥ 10000) ← Empl(x1,x2,x3,x4)
IC2: (90000 ≤ x3) ← Empl(x1,x2,x3,x4)
IC3: x1=y1 ← Empl(x1,x2,x3,x4),Empl(y1,x2,y3,y4)
IC4: x2=y2 ← Empl(x1,x2,x3,x4),Empl(x1,y2,y3,y4)
IC5: x3=y3 ← Empl(x1,x2,x3,x4),Empl(x1,y2,y3,y4)
IC6: x4=y4 ← Empl(x1,x2,x3,x4),Empl(x1,y2,y3,y4)
IC7: x2=y2 ← Dept(x1,x2,x3),Dept(x1,y2,y3)

IC8: x3=y3 ← Dept(x1,x2,x3),Dept(x1,y2,y3)
IC9: x1=y1 ← Dept(x1,x2,x3),Dept(y1,y2,x3)
IC10: Dept(x4,f(x1,x2,x3,x4),g(x1,x2,x3,x4)) ← Empl(x1,x2,x3,x4)
IC11: Empl(x3,f(x1,x2,x3),g(x1,x2,x3),h(x1,x2,x3)) ← Dept(x1,x2,x3)

IC1–IC2 are value bounds; IC3–IC9 are functional dependencies; IC10–IC11 are referential integrity constraints.

In this approach, value bounds are used in two ways: (1) to replace a weaker inequality, and (2) to obtain a contradiction. Both cases were discussed in Section 5.2 under the first three types of residues. Functional dependencies are used to simplify queries by applying the chase [1] method. In order to compare this method with ours, implications of functional dependencies need to be obtained resulting in derived residues. We have remarked at the end of Section 4.4, as to how the derived residues can be obtained. Referential integrity constraints may be used for join elimination as unit residues; that is also discussed as the third type of residue in Section 5.2.

6.4 Examples from [25]

This paper allows implication integrity constraints and subset integrity constraints. We give the main example using our notation.

Example 8

EDB: Supplier(Sname,Status,City)
Material(Mname,Risk)
Department(Dept,City,Manager)
Shipment(Sname,Mname,Qty)
Storage(Dept,Mname,Qty)
Employee(Ename,Age,Sal,Dept)

IC:

IC1: Material(x1,f(x1,x2,x3)) ← Storage(x1,x2,x3)
IC2: x1=d3 ← Storage(x1,x2,x3),(x3 > 450)
IC3: (y2 > 4) ← Storage(d1,x2,x3),Material(x2,y2)
IC4: (y2 > 4) ← Storage(d3,x2,x3),Material(x2,y2)
IC5: (x3 > 500) ← Shipment(x1,benzene,x3)
IC6: (y2 > 25) ← Shipment(x1,x2,x3),Supplier(x1,y2,y3),(x3 > 400)
IC7: x4=d1 ← Employee(x1,x2,x3,x4),(x2 > 40)
IC8: x2=london ← Department(d1,x2,x3)
IC9: x1=s1 ← Shipment(x1,x2,x3),Storage(y1,x2,y3),(x3 > y3)

The semantically constrained axioms are the following.

SCA1. Supplier(x1,x2,x3) ← Supplier(x1,x2,x3)
{(x2 > 25) ← Shipment(x1,y2,y3),(y3 > 400)}

SCA2. Material(x1,x2) ← Material(x1,x2)
{(x2 > 4) ← Storage(d1,x1,y3);
(x2 > 4) ← Storage(d3,x1,y3)}

SCA3. Department(x1,x2,x3) ← Department(x1,x2,x3)
{x2=london ← x1=d1}

SCA4. Shipment(x1,x2,x3) ← Shipment(x1,x2,x3)
{(x3 > 500) ← x2=benzene;
(y2 > 25) ← Supplier(x1,y2,y3),(x3 > 400);
x1=s1 ← Storage(y1,x2,y3),(x3 > y3)}

SCA5. Storage(x1,x2,x3) ← Storage(x1,x2,x3)
{Material(x2,f(x1,x2,x3)) ←;
x1=d3 ← (x3 > 450);
(y2 > 4) ← Material(x2,y2),x1=d1;
(y2 > 4) ← Material(x2,y2),x1=d3;
y1=s1 ← Shipment(y1,x2,y3),(y3 > x3)}

SCA6. Employee(x1,x2,x3,x4) ← Employee(x1,x2,x3,x4)
{x4=d1 ← (x2 > 40)}

Now we consider the query examples illustrating various uses of semantic query optimization. In each case, we start by indicating the semantically constrained query.

Restriction elimination

SCQ1: ← Shipment(x1*,benzene,x3*),(x3* > 300)
{(x3* > 500) ←;
(y2 > 25) ← Supplier(x1*,y2,y3),(x3* > 400);
x1*=s1 ← Storage(y1,benzene,y3),(x3* > y3)}

By the unit residue, the restriction (x3* > 300) must be true and hence can be eliminated to obtain

Q1′: ← Shipment(x1*,benzene,x3*)

Index introduction

SCQ2: ← Department(d1,x2,x3*) {x2=london ←}

By the residue, we obtain

Q2′: ← Department(d1,london,x3*)

Scan reduction

SCQ3: ← Supplier(x1*,x2,x3*) {(x2 > 25) ← Shipment(x1*,y2,y3),
(y3 > 400)},
Shipment(x1*,y2,y3)
{(y3 > 500) ← y2=benzene;
(x2 > 25) ← Supplier(x1*,x2,x3*),(y3 > 400);
x1*=s1 ← Storage(z1,y2,z3),(y3 > z3)},
(y3 > 400)

By the first residue, we can change Q3 to

Q3′: ← Supplier(x1*,x2,x3*),Shipment(x1*,y2,y3),(y3 > 400),(x2 > 25)

Q3′ is given incorrectly in [25].

Join elimination

SCQ4: ← Storage(d1,x2*,x3)
{Material(x2*,f(d1,x2*,x3)) ←;
← (x3 > 450);
(y2 > 4) ← Material(x2*,y2);
z1=s1 ← Shipment(z1,x2*,z3),(z3 > x3)},
Material(x2*,y2)
{(y2 > 4) ← Storage(d1,x2*,x3);
(y2 > 4) ← Storage(d3,x2*,x3)},
(y2 > 3)

Here, by resolving the first residue with the second literal in the query, we obtain

Q4′: ← Storage(d1,x2*,x3)

The last example is a more involved query that illustrates several techniques.

SCQ5: ← Storage(d1,x2*,x3)
{Material(x2*,f(d1,x2*,x3)) ←;
← (x3 > 450);
(y2 > 4) ← Material(x2*,y2);
y1=s1 ← Shipment(y1,x2*,y3),(y3 > x3)},
(x3 > 300),
Material(x2*,y2)
{(y2 > 4) ← Storage(d1,x2*,z3);
(y2 > 4) ← Storage(d3,x2*,z3)},
(y2 > 3),
Shipment(z1,x2*,z3)
{(z3 > 500) ← x2*=benzene;
(y2 > 25) ← Supplier(z1,y2,y3),(z3 > 400);
z1=s1 ← Storage(y1,x2*,y3),(z3 > y3)},
(z3 > x3)

By the residues for Storage, we can eliminate the term Material(x2*,y2), and then (y2 > 3) (as (y2 > 4) must be true). The last residue for Storage allows us to replace z1 by s1. We obtain

Q5′: ← Storage(d1,x2*,x3),(x3 > 300),Shipment(s1,x2*,z3),(z3 > x3)

The only difference between Q5′ and the modified query in [25] is that they include the extra term (z3 > 300) in their transformed query.

7. EXTENSIONS

In Sections 4 and 5 we presented our approach to semantic query optimization cast in the logic formalism. We had imposed a variety of restrictions on the components of the database. Although the assumptions may seem restrictive, Horn databases, and integrity constraints (even with the assumptions imposed in this paper), cover a large class of useful and practical databases.

In this section we show how our approach to semantic query optimization can be extended to databases in which the earlier assumptions of this paper are relaxed. In particular, we show how to handle nonrange-restricted clauses, disjunctive integrity constraints, negation, and recursion. We mostly use abstract examples for the purpose of illustration.

7.1 Nonrange-Restricted Integrity Constraints

The purpose of range-restriction is to make sure that clauses are meaningful and are independent of the domains. However, our method works even if the range-restricted qualification is removed. Partial subsumption and the generation of residues does not change in this case. Consider the following simple relational example.

Example 9

EDB: Supplier(Sno,Sname)
SP(Sno,Pno)
IDB: ∅
IC: SP(x,z) ← Supplier(x,y)

The integrity constraint here is not range-restricted; it says that whenever a tuple is in the Supply relation, that Sno value appears with each Pno value in the SP relation, that is, each supplier whose name appears in Supplier supplies all parts. (We need to assume that the domain of all possible Pno values is known.) The compiled axioms are:

SCA1. Supplier(x,y) ← Supplier(x,y) {Sp(x,z) ←}
SCA2. Sp(x,z) ← Sp(x,z) { }

Now consider the query

Q: ← Supplier(S8,y*),SP(S8,P5)
The semantically constrained query is
SCQ: ← Supplier(S8,y*),SP(S8,P5) {SP(S8,z) ←}

Hence, using our standard method for the case of the unit clause residue, we can eliminate the join and obtain the semantically equivalent query

Q′: ← Supplier(S8,y*)

It is illustrative to point out the differences in literal elimination when dealing with existentially quantified variables (which become Skolem functions) and universally quantified variables. The above residue is SP(S8,z) ← so that literal elimination is valid for *any* value of z. This is not the same as the case of existentially quantified variables (refer to the *Car_owners* example in the discussion of the unit clause residue in Section 5.2).

7.2 Disjunctive Integrity Constraints

We consider disjunctive integrity constraints in this subsection. Although it is reasonable to restrict the EDB and IDB to Horn clauses, there are some cases where integrity constraints are not Horn. Consider the following.

Example 10

EDB: Dept, Stock, Supply
IDB: ∅
IC: Dept(y),Stock(y) ← Supply(x,y)

The integrity constraint may be interpreted as meaning that any supplied item is either in a department or in stock or both. So the union of Dept and Stock contains the projection of Supply on the second attribute. (If we wish to have a partitioned union, then an additional constraint such as ← Dept(y),Stock(y) can be added.)

As far as partial subsumption and residue generation are concerned, it does not matter whether the integrity constraint is Horn or not. What is different in this case is that residues generated are likely to be disjunctive, and therefore they are not likely to be particularly useful for optimizing standard queries. Thus, in

the above example, we would have the semantically constrained axiom

Supply(x,y) ← Supply(x,y) {Dept(y),Stock(y) ←}

Another way of handling disjunctive residues is to reduce them to the Horn case by introducing negation. Recall that we used this for evaluable predicates in earlier sections. We enlarge the class of queries by allowing negated literals. For example, suppose that we allow the query:

Q: ← Supply(x*,y),not(Dept(y)),not(Stock(y))

asking for all suppliers who supply an item that is neither in Dept nor in Stock. Our disjunctive integrity constraints, when rewritten using negation, should help us infer that there are no such suppliers. For instance, the residue {Dept(y),Stock(y) ←} can be rewritten as (← not(Dept(y)),not(Stock(y))}, in which case the residue subsumes the query; so there are no answers, and no lookup is needed. (Note that this transformation is correct by the logical equivalence of the two clauses.) Therefore, we now consider the case with negations allowed in the body of an integrity constraint and a query; by the above residue transformation, this will encompass the disjunctive case.

7.3 Negated Literals

Here we consider the case of negated literals in a query. (It is not necessary to consider negated literals in the body of an integrity constraint, since we allow disjunctive integrity constraints now.) One must be careful in writing such queries in order to avoid floundering. Floundering occurs when a negated atom in a query contains a variable that is not bound to a constant at the time of its evaluation. The more useful case is one where the negated literal in a query refers to an extensional relation. We need to have residues for negated literals in order to perform semantic query optimization.

We present two methods. The first method involves the rewriting of all residues with nothing in the head. This can always be done. In the general case, the residue B1, ... Bm ← A1, ..., An is modified to ← A1, ..., An,not(B1), ...,not(Bm). Again, since the two clauses are logically equivalent, this transformation is correct. The question is how to use such a residue for semantic query transformation. Here, we just treat the new residue as a goal clause. So, if the residue subsumes the query, then there are no answers. This is exactly what we did in Example 10 given above. It may also be possible to apply our rule for the case where the residue is a definite implicational clause. For suppose that the residue is ← R1(x,y),not(R2(y)),not(R3(x)). This residue can be transformed to R2(y) ← R1(x,y),not(R3(y)). Now, if the query contains R1(x,y),not(R3(y)), we can add R2(y) to the query. This is advantageous in cases where join introduction is useful. If the query contains R1(x,y),not(R3(y)),R2(y), then R2(y) can be eliminated.

The second method for handling negation is more elaborate; it involves essentially the compilation of residues for negated relations. That is, in order to anticipate queries involving negated relations, we compile residues for them. We can think of this method as using the rule (axiom) not(R) ← not(R) instead of

R ← R. The difference in the partial subsumption algorithm is to move the R to the right of the arrow from the left.

We illustrate this method using our running example. The compiled axioms are as follows:

Supply(x,y) ← Supply(x,y) {← not Dept(y), not Stock(y)}
Stock(y) ← Stock(y)
Dept(y) ← Dept(y)
not(Supply(x,y)) ← not(Supply(x,y))
not(Stock(y)) ← not(Stock(y)) {← Supply(x,y),not(Dept(y))}
not(Dept(y)) ← not(Dept(y)) {← Supply(x,y),not(Stock(y))}

We show how to obtain the residue for not Stock(y)

The reverse substitution yields the residue after moving the head of the clause to the right-hand side. Note the symmetry that we have obtained. Whether we consider the residue for Supply(x*,y), or not(Dept(y)), or not(Stock(y)), we find that the residue subsumes the query Q of Section 7.2.

For this example, both approaches to handling negation worked. However, the second approach, which includes the first, is really more general. Consider the following example.

Example 11
EDB: Object(x), Stock(x), Backorder(x)
IC: Stock(x) ∨ Backorder(x) ←

The EDB relations represent all the objects, the objects in Stock, and the objects that are Backordered, respectively. Object is used to avoid floundering. The integrity constraints may be interpreted as meaning that every item is either in stock or backordered, or both. There are no residues here for Stock or Backorder, so the IC is not merge-compatible with the relations Stock and Backorder. Therefore, applying the first method to the query

Q: ← Object(x), not(Stock(x)),not(Backorder(x))

there would be no semantic query optimization. However, we obtain residues for not(Stock(x)), and not(Backorder(x)) as follows

not(Stock(x)) ← not(Stock(x)) {← not(Backorder(x))}
not(Backorder(x)) ← not(Backorder(x)) {← not(Stock(x))}

Now we see that the residue subsumes the query, hence there are no solutions. Now consider the query

Q′: ← Object(x), not(Stock(x)), Backorder(x)

The semantically constrained query is

SCQ′: ← Object(x), not(Stock(x)) {Backorder(x) ←}, Backorder(x)
(changing the residue to a unit clause)

Using our method of semantic query optimization for clauses, we obtain the semantically equivalent query

Q'': $\leftarrow$ Object(x), not(Stock(x))

in which a join has been eliminated. This example illustrates that, in general, a residue can be written in different forms and the most appropriate form can be chosen for the purposes of optimization.

Recapitulating, the general method for handling queries that may involve negated extensional relations consists of writing axioms for all such negated relations, then semantically compiling these axioms. We also remark that the complexity of query processing does not change when negation, including the disjunctive case, is handled this way, because the residues are generated in a manner that is analogous to the Horn case. Although we may have disjunctive integrity constraints, the theory that we are dealing with remains Horn. The disjunctive integrity constraint assures us that information must be in one or more of the relations. However, if the extensional or intensional database were disjunctive, the situation would be different.

Handling queries with negated intensional relations is more problematic. For suppose that we have

H1 (x,y) $\leftarrow$ R1(x,z),R2(z,y)
H1 (x,y) $\leftarrow$ R3(x,y,z),R2(z,y)

Now consider the query

Q: $\leftarrow$ R3(x*,y*,u),not(*H1* (x*,y*))

In terms of extensional relations, we obtain

$\leftarrow$ R3(x*,y*,u),not(R1(x*,z),R2(z,y*)),not(R3(x*,y*,z),R2(z,y*))

which can be written as

$\leftarrow$ R3(x*,y*,u),(not(R1(x*,z)) $\vee$ not(R2(z,y*))),(not(R3(x*,y*,z)) $\vee$
not(R2(z,y*)))

Note that all the axioms for *H1* must be included. The presence of the "or"'s makes the evaluation of such queries difficult. Sometimes, such a query can be simplified, however. Suppose, for instance that we have a residue such as R1(x*,z) $\leftarrow$. Then we can rewrite the query as

$\leftarrow$ R3(x*,y*,u),not(R2(z,y*)), (not(R3(x*,y*,z)) $\vee$ not(R2(z,y*)))

Actually, in this example the z's must be restricted in some way in order to avoid floundering. One way of doing this is by introducing a new predicate, such as an Object in Example 11, and binding the unbound variable to Object. In some cases, queries transformed from intensional to extensional relations can be simplified. This occurs when the right-hand side of the axiom contains no variables not on the left-hand side. Suppose for example that we have *H1* (x,y) $\leftarrow$ R1(x,y),R2(y) as the axiom defining *H1*. Now consider the query

Q: $\leftarrow$ R3(x*,y*,u),not(*H1* (x*,y*))

Transforming *H1* to its definition, we obtain

$\leftarrow$ R3(x*,y*,u), not(R1(x*,y*),R2(y*))).

This query can be transformed into two queries ← R3(x*,y*,u),not(R1(x*,y*)) and ← R3(x*,y*,u),not(R2(y*)) by logical equivalence. The union of the answers to these two queries is the answer to the original query. At this point, we have reduced the query to a form that might be optimized semantically using the method presented earlier.

7.4 Handling Recursion

Now we consider extending our work on semantic query optimization to the case where recursion is allowed for the IDB or the IC or both. A great deal of work has been done in recent years on query evaluation for recursive deductive databases. We refer the reader to [2] for a survey and comparison of various strategies. Our interest is in optimizing any such strategy by considering relevant integrity constraints.

We start by noting that in the presence of recursive axioms we may not be able to obtain a structured deductive database that is equivalent to a specific deductive database. Recall from Section 3.2 that in a structured deductive database every relation is purely extensional or purely intensional and all integrity constraints are expressed using only extensional relations. The problem now is that when integrity constraints are defined in terms of intensional relations which are recursive, it is not possible to express them using extensional relations only.

Let us consider the relevant portion of an example with *H1* a nonrecursive relation and *H2* a recursive relation. Suppose that the IDB contains

H1(x,y) ← R1(x,z),R2(z,y)
H2(x,y,z) ← R1(x,w),*H2*(w,y,z)

and the IC contains

IC1: ← *H1*(u,a)
IC2: ← *H2*(u,a,v)

It is possible to reduce IC1 to a constraint consisting only of extensional relations (IC1′: ← R1(u,z),R2(z,a)), but not IC2.

Now we go back to consider recursive intensional axioms. Although there are substantial differences among the various query evaluation strategies, they all rely on a loop for computing the recursive relation in terms of extensional relations. Consider the standard *Ancestor* relation, defined as

Ancestor(x,y) ← Parent(x,y)
Ancestor(x,y) ← Parent(x,z),*Ancestor*(z,y)

and the query

← *Ancestor*(jim,x*)

We illustrate the use of integrity constraints on the naive evaluation method from [2]. We number the steps for easier reference. The program is

begin
(1) initialize *Ancestor* to the empty set;
(2) evaluate (*Ancestor*(x,y) ← Parent(x,y));

(3) insert the result in *Ancestor*;
while "new tuples are generated" do
begin
(4) evaluate (*Ancestor*(x,y) ← Parent(x,z),*Ancestor*(z,y))
using the current value of *Ancestor*;
(5) insert the result in *Ancestor*
end;
(6) evaluate (*Ancestor*(jim,x*)) using the current value of *Ancestor*
end

It is clear that the evaluation of a recursive query involves the evaluation of several sub-queries inside a loop. It is for the evaluation of these sub-queries that one can use the techniques proposed in this paper.

Now let us suppose that there is a functional dependency in the Parent relation. Then, in both steps 2 and 4 we can use semantic query optimization on the Parent relation, having obtained the residue earlier, to stop the search after one value is found. For the functional dependency

y1=y2 ← Parent(x,y1), Parent(x,y2), we would modify
step 2 to 2(SC) evaluate
(*Ancestor*(x,y) ← Parent(x,y) {y=y2 ← Parent(x,y2)});
and step 4 to 4(SC) evaluate
(*Ancestor*(x,y) ← Parent(x,z) {z=y2 ← Parent(x,y2)}, *Ancestor*(z,y))

using the current value of *Ancestor*; similar evaluations occur in all recursive query evaluation strategies and appropriate integrity constraints may be used in the evaluation steps. Note that the implementation must recognize the difference between the case where x is bound in a query and where y (or z) is bound. In the former case, the search can stop after one answer has been found for Parent; in the latter case, the residue is not useful.

Now we consider the problems that may occur in the case of recursive integrity constraints for semantic query optimization. Such integrity constraints have a non-empty head, as long as we write them without using "not". We now show that care must be taken in this case when join elimination is called for by semantic query optimization. We give an example to illustrate the problems in eliminating literals in the presence of recursive integrity constraints.

Example 12

EDB: Parent, Sibling
IDB: ∅
IC: Sibling(x,y) ← Sibling(y,x)

Note that this integrity constraint is recursive. Now consider the query

Q′: ← Parent(w*,x),Sibling(x,y),Sibling(y,x)

The semantically compiled query is

SCQ: ← Parent(w*,x),Sibling(x,y) {Sibling(y,x) ←},
Sibling(y,x){Sibling(x,y) ←}

If we use both residues, then we obtain

Q': $\leftarrow$ Parent(w*,x)

which is not semantically equivalent to Q.

The problem is caused by the recursive nature of the integrity constraint which makes literal eliminations interdependent. In other words, the elimination of the literal Sibling(x,y) assumes the presence of the literal Sibling(y,x) and vice versa; hence only one literal can be eliminated from the query. There are several ways to get around this problem. One method involves eliminating each residue when its corresponding atom is eliminated. Thus, when Sibling(y,x) is eliminated, so is the residue Sibling(x,y) $\leftarrow$, and there is no problem. Note that it is not necessary to do this type of association for non-recursive integrity constraints.

We note that our method of incorporating integrity constraints with deductive rules is applicable to any problem that contains either recursive or nonrecursive rules. In the case of nonrecursive rules, we described in Section 4 precisely how to compile the rules and the integrity constraints. In the case of recursion, the integrity constraints are incorporated with the rules, and any technique used to solve questions involving recursion may then be applied.

So far we have considered recursion separately for intensional axioms and integrity constraints. If recursion appears in both, then we can combine the methods applied for recursion in those two cases. A recent paper, [17], contains some additional results and examples involving recursion.

8. SUMMARY AND CONCLUSIONS

Relational databases were not considered as serious commercial possibilities until query optimization made them practical. We believe that for the database systems of the future, particularly expert and deductive database systems, the present syntactic optimization techniques will not suffice. These databases tend to have many integrity constraints, rules that should be taken into consideration by an intelligent query processor. One aspect of such a processor will be semantic query optimization, the use of integrity constraints to limit the search for answers.

Semantic query optimization has been studied by several researchers in the last decade. Different authors concentrated on different types of integrity constraints, applied different methods, and used different notation. Our approach uses first-order logic formulas for describing the entire deductive database and the queries. A modified version of the subsumption algorithm for automated theorem proving provides a uniform method for associating relevant fragments of integrity constraints, called residues, with extensional relations and deductive rules for intensional relations, before any queries are posed. During query processing, the residues may be used to simplify the search.

As we have demonstrated in this paper, our method is more general than the other techniques that essentially consider special cases only. Also, our method applies both to conventional and deductive databases. In addition, our approach consolidates earlier scattered work by providing a uniform framework with a formal basis in first-order logic. An implementation for the semantic compiler is described in [18]. This implementation uses a metainterpreter and is written in

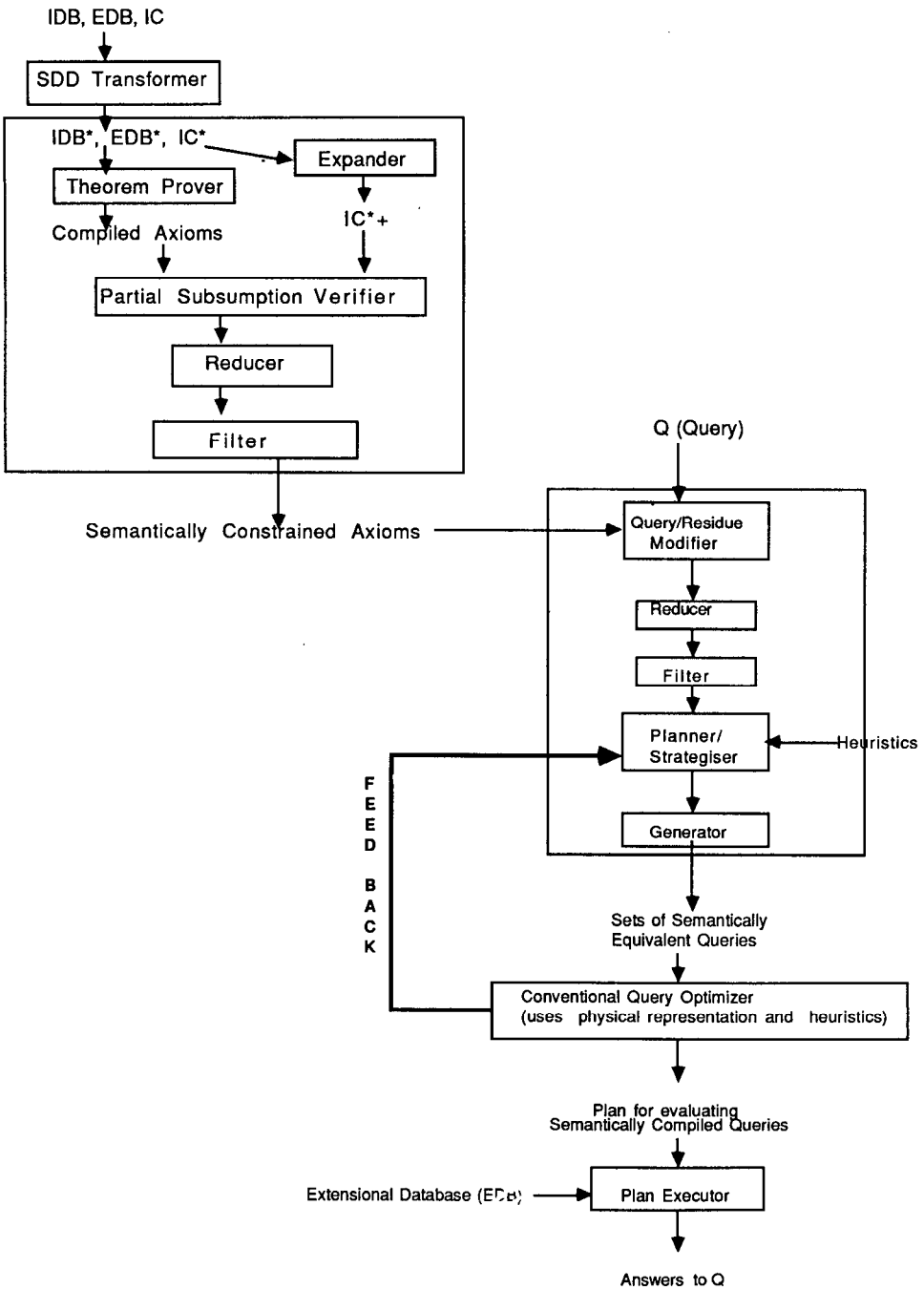


Fig. 5. Detailed architecture of semantic query processor with feedback to the planner/strategizer.

PROLOG. (The metainterpreter is a variant of the procedure that defines the SOLVE predicate in [26]).

If we use the procedural/declarative meaning of the semantically compiled queries, we can evaluate a query directly in a PROLOG deductive database system, such as MU-Prolog [22]. We can also interface our approach with a conventional relational database system. We have presented such a functional architecture for semantic query optimization in this paper; it is summarized in Figure 5. This architecture also indicates the feedback from a conventional query optimizer. Since the axioms are compiled, a query is transformed into one which contains only evaluable and extensional relations. A metainterpreter can then be written in PROLOG to translate the query to a query in a language such as SQL (see [13] for instance), for which many implementations are available.

ACKNOWLEDGMENTS

We wish to thank the editor, Dr. Umeshwar Dayal, for providing many helpful comments and the anonymous referees for their valuable suggestions.

REFERENCES

1. AHO, A. V., SAGIV, Y., AND ULLMAN, J. D. Efficient optimization of a class of relational expressions. *ACM Trans. Database Syst. 4* (1979), 435–454.
2. BANCILHON, F., AND RAMAKRISHNAN, R. An amateur's introduction to recursive query processing strategies. In *Proceedings of the 1986 ACM-SIGMOD Conference*, 16–52.
3. CHAKRAVARTHY, U. S. Semantic query optimization in deductive databases. Ph.D. thesis, Dept. of Computer Science, Univ. of Maryland, College Park, 1985.
4. CHAKRAVARTHY, U. S., FISHMAN, D. H., AND MINKER, J. Semantic query optimization in expert systems and database systems. In *Expert Database Systems*, L. Kerschberg, Ed., Benjamin/Cummings, 1985, 659–675.
5. CHAKRAVARTHY, U. S., GRANT, J., AND MINKER, J. Foundations of semantic query optimization for deductive databases. In *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed., Morgan Kaufmann, 1987.
6. CHAKRAVARTHY, U. S., AND MINKER, J. Multiple query processing in deductive databases using query graphs. In *Proceedings of the 12th VLDB Conference*, 1986, 384–391.
7. CHAKRAVARTHY, U. S., MINKER, J., AND GRANT, J. Semantic query optimization: additional constraints and control strategies. In *Expert Database Systems*, L. Kerschberg, Ed., Benjamin/Cummings, Inc., 1987, 345–379.
8. CHANG, C. L., AND LEE, R. C. T. *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.
9. GALLAIRE, H., MINKER, J., AND NICOLAS, J.-M. Logic and databases: a deductive approach. *ACM Comput. Surv. 16* (1984), 153–185.
10. GRANT, J., AND MINKER, J. Optimization in deductive and conventional relational database systems. In *Advances in Data Base Theory, Vol. 1*, H. Gallaire, J. Minker, and J. M. Nicolas, Eds., Plenum Press, 1981, 195–234.
11. HAMMER, M. M., AND ZDONIK, S. B. Knowledge based query processing. In *Proceedings of the Sixth VLDB Conference*, 1980, 137–147.
12. JARKE, M., CLIFFORD, J., AND VASSILIOU, Y. An optimizing Prolog front-end to a relational query system. In *Proceedings of the ACM-SIGMOD Conference*, 1984, 296–306.
13. JARKE, M., AND KOCH, J. Query optimization in database systems. *ACM Comput. Surv. 16* (1984), 111–152.
14. KING, J. J. Query optimization by semantic reasoning, Ph.D. thesis, Dept. of Computer Science, Stanford Univ., Palo Alto, 1981.
15. KING, J. J. QUIST: a system for semantic query optimization in relational databases. In *Proceedings of the 7th VLDB Conference*, 1981, 510–517.

16. KOHLI, M., AND MINKER, J. Intelligent control using integrity constraints. In *Proceedings of AAAI*, 1983, 202–205.
17. LEE, S., AND HAN, J. Semantic query optimization in recursive databases. In *Proceedings of the Fourth International Conference on Data Engineering*, 1988, 444–451.
18. LOBO, J., AND MINKER, J. A metainterpreter to semantically optimize queries in deductive databases. In *Proceedings of the Second International Conference on Expert Database Systems*, 1988, 387–420.
19. MCSKIMIN, J. R. Techniques for employing semantic information in question-answering systems. Ph.D. thesis, Dept. of Computer Science, Univ. of Maryland, College Park, 1976.
20. MCSKIMIN, J. R., AND MINKER, J. The use of a semantic network in a deductive query answering system. In *Proceedings of the Fifth IJCAI*, 1977, 50–58.
21. MINKER, J. Perspectives in deductive databases. *J. Logic Program. 5* (1988), 33–60.
22. NAISH, L. The MU-Prolog 3.2 reference manual. Tech. Rep. 85/11, Dept. of Computer Science, Univ. of Melbourne, Australia, 1985.
23. REITER, R. Deductive query-answering on relational databases. In *Logic and Databases*, H. Gallaire, and J. Minker, Eds., Plenum Press, 1978, 149–178.
24. SELLIS, T. Multiple-query optimization. *ACM Trans. Database Syst. 13* (1988), 23–52.
25. SHENOY, S. T., AND OZSOYOGLU, Z. M. A system for semantic query optimization. In *Proceedings of the ACM-SIGMOD Conference*, 1987, 181–195.
26. STERLING, L. Meta-interpreters: the flavors of logic programming. In *Proceedings of the Workshop on the Foundations of Deductive Databases and Logic Programming* (Washington, D.C., 1986), J. Minker, Ed., 1986, 163–175.
27. XU, G. D. Search control in semantic query optimization. Tech. Rep. 83-9, Dept. of Computer Science, Univ. of Massachusetts, Amherst, 1983.
28. ZDONIK, S. B., JR. On the use of domain specific knowledge in the processing of database queries. M.S. thesis, Massachusetts Institute of Technology, Cambridge, 1980.

Received September 1987, revised August 1988 and April 1989, accepted April 1989