International Journal of Intelligent and Cooperative Information Systems
Vol. 2, No. 2 (1993) 107–125

DESIGN AND IMPLEMENTATION OF A KNOWLEDGE-BASED QUERY PROCESSOR

NABIL R. ADAM and ARYYA GANGOPADHYAY
MS/CIS Department,GSM, Rutgers University, Newark, NJ 07102, USA

JAMES GELLER
CIS Department, New Jersey Institute of Technology, Newark, NJ 07102, USA

Received 15 January 1993
Revised 1 May 1993

ABSTRACT

This paper deals with query processing using semantic knowledge in relational databases. The Select-Project-Join (SPJ) conjunctive class of queries are dealt with in this paper. We propose to optimize highly repetitive queries by using semantic transformations in addition to syntactic transformations. Thus, we generate a set of pre-optimized queries. This set contains queries that are semantically equivalent to, syntactically different from, and more efficient to process than the user queries that we started with. The issues we address in this paper are: how to map a user query to a query that is in the set of pre-optimized and already optimized queries, how to search efficiently through the set of pre-optimized queries and set of semantic rules, and how to incorporate new queries to the set of pre-optimized queries, so that the number of queries that can be optimized using this method increases with the passage of time. Furthermore, we suggest some ideas of handling queries that do not have any semantically equivalent counterpart in the set of pre-optimized queries. We have tested the performance of the proposed method. An algorithm for mapping is implemented in Prolog. A database schema is implemented in the INGRES database management system. We have adopted a database schema that is widely used for measuring performance in the semantic query optimization literature.

Keywords: Database systems, knowledge base, pre-optimized queries, semantic query optimization.

1. Introduction

Query optimization refers to the process of transforming a user query into an equivalent query that can be processed more *efficiently*. The transformation could be syntactic (as in the case of *conventional* query optimization) or semantic (as in the case of *semantic* query optimization). In conventional query optimization (e.g. [2,3,7,12,15,17,19,25]) the execution strategy that is implied by a user query is transformed into a more efficient one through the use of statistics on database usage, join algorithms, query decomposition, indices, and syntactic rearrangements. An execution strategy refers to a complete sequence of operations, e.g. join and project, that the database management system (DBMS) must perform

in order to answer the query. On the other hand, semantic query optimization (e.g. [1,4,6,9,10,11,13,14,18,21,20,22,23]) makes use of semantic knowledge about the database to transform a user query into a semantically equivalent and operationally more efficient one. Two queries are semantically equivalent if they result in the same answer under any state of the database that conforms to the semantic integrity constraints.

In semantic query optimization, the database is divided into two distinct parts, the intensional database (IDB) and the extensional database (EDB). The IDB represents the semantic knowledge about the database, and consists of semantic integrity constraints and deductive rules. The EDB is the actual database instance. When a query is posed to the DBMS, the query is transformed into a semantically equivalent form, by using the IDB and some logical inference rules. Since there could be several equivalent queries to the original one posed by the user, alternative queries are compared by a conventional query optimizer to determine the least costly query which is then processed.

The research on semantic query optimization has been motivated by the notion of utilizing the knowledge about the database in processing a user query. The research done in this area can be grouped into developing: semantic query optimizers (e.g. [13,14,21,22]), IDB (e.g. [9]), algorithms for efficient search of the IDB (e.g. [20]), and automatic knowledge acquisition and maintenance of the IDB (e.g. [24,25]). Studies such as [4], [6] and [11] have demonstrated the potential benefits of semantic query optimization.

Typically, a large percentage of user queries are preprogrammed and highly repetitive (e.g. [22, p. 36]). Conventional query optimizers allow for *precompilation* of such queries, i.e. the execution plan for each of these queries is stored in the DBMS and is accessed whenever the query is executed. As a result, the need to start the optimization process from scratch every time such a query is issued is eliminated. We propose to adopt the same concept in the context of semantic query optimization. That is, initially we generate a "set of pre-optimized" queries, Q_p, that is made up of the "optimized" version of each preprogrammed and highly repetitive query. The optimization is done *semantically* as well as *syntactically*. To illustrate, consider a hypothetical database system that consists of the relations in Fig. 1, and has the IDB shown in Fig. 2 (adapted from [14, p. 48–51]).

The following abbreviations are used in Figs. 1 and 2.

- SName: Ship's name.
- OName: Owner's name.
- SType: Ship's type.
- Dwt: Dead weight (the actual weight of cargo carried).
- PName: Port's name.
- FType: Port's facility type.
- Type: Cargo type.
- LNG: Liquefied natural gas.

SHIPS (SName,OName,SType,Draft,Dwt,Capacity,Registry)
PORTS (PName,Country,Depth,FType)
CARGOES (SName,PName,Shipper,CType,Quantity, Dollar,Policy)
OWNERS (OName,Location,Assets,Business)
POLICIES (Policy,Issuer,Coverage)
INSURERS (Issuer,Insurercountry,Capitalization)

Fig. 1. Relations.

R1: $(Dwt > 350) \Rightarrow (FType =$"offshore"$)$
R2: $(Dwt > 300) \Rightarrow (Business =$ "leasing"$)$
R3: $(Coverage \leq Dollar)$
R4: $(Quantity \leq Capacity)$
R5: $(CType \notin \{$"LNG", "refined" $\} \wedge (Dollar > 500) \Rightarrow (FType =$ "general"$)$
R6: $(Dwt > 150) \Rightarrow SType =$ "supertanker"
R7: $(Dollar > 3000) \wedge (SType =$ "supertanker" $) \Rightarrow (Issuer =$ "Lloyds"$)$
R8: $(Business =$ "petroleum" $) \Rightarrow (CType \in \{$ "LNG", "refined", "oil"$\})$
R9: $(CType =$ "LNG"$) \Rightarrow (SType =$ "Pressure vessel"$)$
R10: $(SType =$ "supertanker"$) \Rightarrow (Dwt > 150)$

Fig. 2. The intensional database.

Suppose that the following is among the highly repetitive queries. Throughout the rest of the paper we will express queries in SQL syntax.

q_1 : *select* $(SName)$ *from* $CARGOES$, $POLICIES$ *where* $(Coverage \leq Dollar)$.

Without semantic optimization, the processing of this query requires performing a join operation. Making use, however, of the rules included in the IDB, specifically $R3$ (Fig. 2), results in eliminating the constraint from the query q_1.

In this case, Q_p would include the following *transformed* version of q_1:

q_2 : *select* $(SName)$ *from* $CARGOES$.

Both q_1 and q_2 are semantically equivalent since they produce the same result under *all instances* of the database that satisfy the integrity constraints. This, however, does not stop the user from issuing the query in the form of q_1.

Thus, we end up with a set of pre-optimized queries that may be syntactically different from user queries. This necessitates a *mapping mechanism* to "map" a user query to a semantically equivalent pre-optimized one. In addition, we need to ensure that the set Q_p is kept up-to-date with respect to any changes in the IDB. The rules in the IDB are expressed in first-order Horn clause form.

In this paper, we address the generation, representation, and maintenance of the pre-optimized query set. We also present a methodology for utilizing the set of pre-optimized queries in the context of semantic query optimization.

1.1. Research Objectives

The goal of the paper is to develop and implement a methodology for utilizing the set of pre-optimized queries in the context of relational database systems. We restrict ourselves to the SPJ (select-project-join) conjunctive types of queries.

Specifically, our research goal is to deal with the following issues:

(i) Optimization of user queries using pre-optimized queries. This requires addressing the following:

- Generating the set of pre-optimized queries.
- Representing the IDB and the set of pre-optimized queries.
- Mapping a user query to a semantically equivalent pre-optimized query, thus extending the use of the pre-optimized query set to user queries that are syntactically different but semantically equivalent. A preliminary version of an algorithm that addresses this issue is discussed in Sec. 3.
- Searching efficiently through the IDB and the set of pre-optimized queries.
- Updating the set of pre-optimized queries against changes in the semantics of the database.

(ii) Optimizing queries that do not have any semantically equivalent counterpart in the set of pre-optimized queries.

(iii) Implementing and testing the proposed method.

(iv) Performance Analysis of the proposed method: We first develop a database model of an application area. Next, we test whether the benefit (measured by reduction in response time) of the algorithm is more than its overhead cost. This can be done as follows: Let the *net savings* resulting from the algorithm be $\delta(t)$, system response time with the algorithm be $C(t)$, and system response time without the algorithm be $R(t)$. Then, our aim is to ensure the following inequalities:

$$\delta(t) > 0, \text{ where },$$

$$\delta(t) = R(t) - C(t) .$$

The rest of the paper is organized as follows. A brief discussion of related previous work is presented in Sec. 2. The proposed methodology and analysis of its performance are discussed in Secs. 3 and 4 respectively. Our conclusion and some future research directions are discussed in Sec. 5.

2. Previous Work

2.1. Algorithms for Query Improvement

One of the early publications in this area is [9] where the authors take an object-oriented view of the database. They develop a framework for query improvement (optimization) and describe the details of a system for knowledge-based query processing (KBQP). The system consists of a knowledge base that contains constraints about the set of objects in the database. A query is viewed as a combination of a number of subexpressions. The system contains *techniques* which are programs that attempt to modify the subexpressions in such a way that a more efficient query is generated. Examples of such techniques are *domain refinement* and *mapping substitution*. New facts may be generated at the time of the modification of

the subexpressions. These are added to the knowledge base by a knowledge base compiler.

King [13,14] describes a system called QUIST (Query Improvement through Semantic Transformation), which is a heuristics based SQO that uses the IDB to generate semantically equivalent queries. The notion of semantic equivalence is formally defined in [14]. Semantically equivalent queries are generated by adding and/or deleting constraints from the original query. In order to restrict generation of queries to only potentially profitable ones, several heuristics like *index introduction join elimination, scan reduction, join introduction and detection of unsatisfiable conditions* are used.

Jarke *et al.* [11] describes a PROLOG front-end to a relational database system. The system translates a tuple-oriented PROLOG query into an intermediate set-oriented language of *database calls* that is referred to as DBCL. Syntactic query simplification techniques that attempt to reduce the number of joins in a query are applied within DBCL. Semantic integrity constraints (value bounds, functional dependencies and referential constraints) are next applied for semantic query simplification. The resulting "optimized" query expressed in DBCL is then translated into SQL.

In [10] Jarke describes an algorithm based on graph theory, where an inequality constraint is represented in the form of a directed query-graph, and functional and key dependencies are expressed in a FD/KD graph. Tableau and syntactic simplification techniques are used to optimize queries. The algorithm makes use of key dependencies, functional dependencies and value bounds constraints.

A graph-theoretic approach towards developing a SQO is proposed in [22]. A query is represented by a *query graph* which is a directed graph whose vertices represent either the attributes (attribute vertices) or constants (constant vertices), and edges represent the constraints imposed by the query. The algorithm derives the *canonical condensed form* of the query through *transitive reduction*. Next, the query graph is *semantically expanded* by adding constraints to it. Two types of constraints are used: subset constraints and implication constraints. The graph obtained after semantic expansion is reduced by eliminating redundant relations (*relation elimination*) and edges (*edge elimination*). Finally, the query graph is converted from the condensed form back to its original form. It has also been formally proven that the cost of processing the query after the transformation by the above algorithm is less than that of the original query. The implementation architecture and test results are presented.

In [4,5,6] Chakravarty *et al.* present a two-phase approach to semantic query optimization: semantic compilation and semantic query transformation. This approach is applicable to both deductive as well as relational databases. In the semantic compilation phase, a theorem prover generates compiled axioms from the IDB, EDB and the set of integrity constraints. These compiled axioms are associated with integrity constraint fragments, called residues, to form semantically constrained axioms (SCA). The first step in semantic query transformation is *query/residue* mod-

ification, which refers to adding residues to the query using the SCAs and some criteria for residue simplification. The next step consists of generating semantically equivalent queries. Three strategies are discussed: (1) generate all possible semantically equivalent queries, estimate the cost of each query, and select the least costly one, (2) use heuristics such as index introduction and join elimination, to produce only the promising queries and select the least costly one, and (3) combine query generation and cost estimation in one step to generate only the optimal query.

As evident from the above literature review, the notion of making use of pre-optimized queries in the context of semantic query optimization has not been explored. Our proposed work is an attempt towards this goal.

3. Methodology

3.1. Generating the Pre-optimized Query Set

The initial step of our research work calls for generating the set Q_p. To accomplish this step we first need to identify all preprogrammed and highly repetitive queries. These queries can then be optimized using semantic as well as conventional optimization. Since the optimization will be done *off-line*, response time is not much of a concern. Each preprogrammed and highly repetitive query is processed in terms of the following *two levels* (e.g. [20]).

At the *first* level, the entire space of semantically equivalent queries is identified by applying the various relevant rules in the IDB. This step can be done manually (e.g. [20]) or can be automated (e.g. the query generation part of QUIST [14]).

At the *second* level, a conventional query optimizer is used to determine the optimal execution plan for each of the queries identified at the first level. The query that results in the most efficient execution plan is then added to Q_p.

3.2. Representing the Pre-optimized Query Set

A given query q_i can be viewed as consisting of two parts [22, p. 352]: the target attributes part (hereafter referred to as the retrieval part), which is denoted by $q_{ia} = \{a_1, a_2, \ldots, a_k\}$, and a conjunction of join as well as restriction specifications (hereafter referred to as the constraint part), denoted by $q_{ic} = \{q_{ic_1}, q_{ic_2}, \cdots, q_{ic_n}\}$. Furthermore, for a given rule R_j we refer to its antecedent and consequent parts as R_{jA} and R_{jC} respectively.

In order to efficiently maintain and search through the set $Q_p = \{q_{p_1}, q_{p_2}, \ldots, q_{p_t}\}$, we propose to maintain a framework similar to the one depicted in Table 1, where we associate with each attribute a_i in the database four lists:

$Rquery_{a_i} = \{j \mid a_i \in q_{p_j a}\} \equiv$ indices of pre-optimized queries whose retrieval part includes a_i,

$Cquery_{a_i} = \{j \mid a_i \in q_{p_j c}\} \equiv$ indices of pre-optimized queries whose constraint part includes a_i,

$Arule_{a_i} = \{j \mid a_i \in R_{jA}\} \equiv$ indices of rules whose antecedent part includes a_i, and

$Crule_{a_i} = \{j \mid a_i \in R_{jC}\} \equiv$ indices of rules whose consequent part includes a_i.

Thus, given a user query q_u, the set of *potentially semantically equivalent queries* is given by:

$\{q_{p_j}, j \in \{Rquery_{a_1} \cap Rquery_{a_2} \ldots \cap Rquery_{a_k}\}$, where $a_i \in q_{ua}$, for $i = 1, 2, \ldots, k\}$.

For illustration purposes, we use the set of pre-optimized queries depicted in Fig. 3.

q_{p_1}: select (s.OName) from SHIPS s where (s.SType = "supertanker")
q_{p_2}: select (c.Shipper, c.Quantity) from CARGOES c where (c.CType = "LNG" and c.PName = "Marseilles")
q_{p_3}: select (s.OName, c.Quantity) from SHIPS s, CARGOES c, PORTS p where (s.Dwt < 60 and s.SName = c.SName and c.CType = "Refined petroleum" and c.PNAME = p.PNAME and p.country = "Denmark")
q_{p_4}: select (s.OName) from SHIPS s, OWNERS o where (o.Business = "leasing" and o.OName = s.OName and
s.SType = "supertanker" and s.Dwt > 350 and s.SName=c.SName and c.PName = "Marseilles")
q_{p_5}: select (s.OName) from SHIPS s where (s.SType= "bulk carrier" and s.Dwt > 200)
q_{p_6}: select (s.OName) from SHIPS s where (s.SType= "Refrigerated vessel")

Fig. 3. The pre-optimized query set Q_p.

Table 1 describes how the pre-optimized queries and the IDB can be represented in the framework discussed above.

Thus, the *Rquery* list of SHIPS.OName contains $\{1, 3, 4, 5, 6\}$, which means that pre-optimized queries q_1, q_3, q_4, q_5, and q_6 have this attribute in their retrieval parts. Similarly, the *Cquery* list of SHIPS.SType equals {1, 4, 5, 6}, indicating that the attribute SHIPS.SType is used in the constraint parts of pre-optimized queries q_1, q_4, q_5, and q_6. Note that the join conditions (e.g. s.SName = c.SName in query q_{p3}) are not included in the representation scheme. In a similar vein, the lists *Arule* and *Crule* corresponding to the attribute SHIPS.SType indicate that SHIPS.SType is used in the antecedent and consequent parts of rules R7 and R9, and R6 and R9 respectively.

To facilitate our discussion further, we use the $\prec$ (*succ*) relation between queries. The $\prec$ ($\succ$) relation is defined as follows: for any two queries A and B, $A \prec B$, if and only if all attributes mentioned in the constraint part of A are also present in the constraint part of B. As an illustration of this definition, consider the following example queries where $q_A \prec q_B$.[a]

q_A: select OName from SHIPS where Dwt = 100.

q_B: select OName from SHIPS where Dwt > 200 and ....

[a]Note that the $\prec/\succ$ relation between queries as used here is different from the notion in which a query is a subset of another if the data items retrieved by the former is a subset of the latter.

Table 1: Representing the pre-optimized queries and IDB.

Relation	Attribute	Rquery	Cquery	Arule	Crule
SHIPS	SName	{}	{}	{}	{}
	CName	{1,3,4,5,6}	{}	{}	{}
	SType	{}	{1,4,5,6}	{7,9,10}	{6,9}
	Draft	{}	{}	{}	{}
	Dwt	{}	{3,4,5}	{1,2,6}	{10}
	Capacity	{}	{}	{}	{4}
	Registry	{}	{}	{}	{}
PORTS	PName	{}	{2,4}	{}	{}
	Country	{}	{3}	{}	{}
	Depth	{}	{}	{}	{}
	FType	{}	{}	{}	{1,5}
CARGOES	SName	{}	{}	{}	{}
	PName	{}	{2,4}	{}	{}
	Shipper	{}	{}	{}	{}
	CType	{}	{2,3}	{5,9}	{8}
	Quantity	{2,3}	{}	{}	{4}
	Dollar	{}	{}	{5,7}	{3}
	Insurance	{}	{}	{}	{}
OWNERS	OName	{1,3,4,5,6}	{}	{}	{}
	Location	{}	{}	{}	{}
	Assets	{}	{}	{}	{}
	Business	{}	{4}	{8}	{2}
POLICY	Policy	{}	{}	{}	{}
	Issuer	{}	{}	{}	{7}
	Coverage	{}	{}	{}	{3}
INSURERS	Issuer	{}	{}	{}	{7}
	Insurercountry	{}	{}	{}	{}
	Capitalization	{}	{}	{}	{}

Now, consider the following two user queries:

q_{u_1}: *select* $(s.OName)$ *from Ships s where* $(s.Dwt > 200)$

q_{u_2}: *select* $(c.Shipper, c.Quantity)$ *from* $CARGOES$ c, $SHIPS$ s *where* $(s.SType =$ "Pressure vessel" *and* $c.PName =$ "Marseilles")

Using the above framework, we see that with respect to q_{u_1}, the set of potentially semantically equivalent queries = $\{q_{p_1}, q_{p_3}, q_{p_4}, q_{p_5}, q_{p_6}\}$, since $Rquery_{OName} = \{1, 3, 4, 5, 6\}$. Similarly, with respect to q_{u_2}, the set of potentially semantically equivalent queries = $\{q_{p_2}, q_{p_3}\}$, since $Rquery_{Quantity} = \{2, 3\}$.

In Sec. 3.4, we carry this example to completion.

3.3. Maintaining the Pre-optimized Query Set

For maintaining the pre-optimized query set, we have two concerns. The *first* is to ensure that the pre-optimized query set Q_p is updated when either the database schema is changed and/or new highly repetitive user queries are identified. We propose maintaining *metadata* about Q_p, which would include such information as which integrity constraints and deductive rules are related to which $q_p \in Q_p$. Whenever a constraint and/or rule is deleted, added, or modified the queries in Q_p that have constraints either directly or indirectly related to the newly deleted, added, or modified constraints are identified and each such query is re-optimized semantically and syntactically.

Typically, the majority of user queries can be predetermined for most database applications. It is on this premise that menu-driven systems are built. In a menu-driven system, the user interacts through a set of hierarchically organized menus, where each menu has a set of predetermined options. A user can formulate a query by selecting a path through the menu structure. Thus, the queries that can be supported are implicitly represented in the menu structure. Determining user requirements is also a fundamental step in the design of database systems. The repetitiveness of user queries has also been noted in the literature on semantic query optimization (e.g. [22, p. 360]).

Thus, only a small percentage of the user queries might not have a semantically equivalent counterpart in Q_P. A statistical record of the frequency of failure to map a user query to any pre-optimized query in Q_p is kept. A predetermined cutoff point can be selected such that any query whose frequency of occurrence exceeds this cutoff point will be semantically optimized and the new semantically optimized query will be added to Q_p. This would account for such situations as adding one or more new attributes to existing relations.

In case of the deletion of an attribute from a relation, the pre-optimized queries for which that attribute appears in the retrieval part are taken out of Q_p. Those pre-optimized queries for which the deleted attribute appeared in the constraint part are reoptimized.

Statistical data of the frequency of usage of each pre-optimized query could be kept so that those pre-optimized queries that have reduced usage patterns can be identified and subsequently deleted.

The *second* concern is to ensure that the set Q_p is free from redundancy. The pre-optimized query set is *initially* generated from the highly repetitive queries. This is done by the database administrator.[b] Once Q_p is formed, we repeatedly try

[b]The definition of "highly repetitive" queries has to be developed empirically. On the one hand,

to semantically map each query in Q_p to another query in Q_p. If a map is found, one of the queries is kept and the others are removed from Q_p. This would ensure that Q_p is free from redundancy.

3.4. An Algorithm for Mapping a User Query to a Pre-optimized One

The mapping algorithm presented here has two functions: (a) searching efficiently through the IDB and the Q_p to identify the *potentially* applicable rules and semantically equivalent queries respectively, and (b) establishing semantic equivalence between q_u and one of the queries identified in the search process. The search process makes use of the representation scheme presented in Sec. 3.2.

Given a q_u whose retrieval part is $q_{ua} = \{a_1, a_2, \ldots, a_k\}$ and the constraint part is $q_{uc} = \{q_{uc_1}, q_{uc_2}, \ldots, q_{uc_n}\}$, we first identify the set $Q_p^* \subset Q_p$, where the retrieval part of each query in Q_p^* is the same or a superset[c] of that of q_u.

The next step is to attempt to map a given user query to a semantically equivalent one $q_p \in Q_p^*$. If such a query exists, it is executed instead of the original user query q_u. There are three ways of mapping a user query q_u to a pre-optimized query q_p: (1) *eliminating* redundant constraints that are in q_u but not in q_p, (2) *adding* constraints in q_u that are present in q_p but not in q_u (provided these new constraints can be *derived* from those mentioned in q_u, and (3) *replacing* some constraints in q_u by some in q_p (provided after such replacements q_u remains semantically equivalent to its original form). The algorithm makes use of function *Implies*, which is described in Fig. 4. The detailed steps of our algorithm are given below:

Step 0: Identify the set of potentially semantically equivalent queries, $Q_p^* = \{q_{p_j}^* \mid j \in \{Rquery_{a_1} \cap Rquery_{a_2} \ldots \cap Rquery_{a_k}\}$, where $a_i \in q_{ua}$, for $i = 1, 2, \ldots, k\}$.

/* Q_p^* contains all queries in Q_p whose retrieval parts are either equal to or are supersets of that of q_u */

Step 1: Identify all queries $q_p' \in Q_p^*$ that satisfy the following property: $q_p' \succeq q_u$ as follows:

(i) Identify the set of queries
$Q_p^{**} = \{q_{p_j}^{**} \mid j \in \{Cquery_{c_1} \cap Cquery_{c_2} \ldots \cap Cquery_{c_n}\}$, where c_i is an attribute in q_{uc}, for $i = 1, 2, \ldots, n\}$.
(ii) Let $Q_p' = Q_p^* \bigcap Q_p^{**}$.
(iii) If Q_p' is empty, then goto Step 3, else continue.

Step 2: Let Found = *false*, and Temp = Q_p'.

/* Found is a boolean variable that returns *true* if a semantically equivalent query is found in Q_p', and *false* otherwise. Temp is a temporary set of pre-optimized queries that is initialized to Q_p'. The algorithm

inclusion of a query in Q_p would improve the chances of getting a semantically equivalent mapping of a user query. On the other hand, this would increase the search time for determining such a mapping.

[c]In this case projection is used to arrive at the user's desired attributes.

attempts to establish semantic equivalence between its first element and q_u. If it fails, that element is eliminated from the Temp set. The process continues until either Temp is reduced to a null set *or* a semantically equivalent query is found. */

while (not Found) and (Temp $\neq \emptyset$) do
{

(i) Let q'_p be the first query in Temp.
(ii) Let Found = Implies (q_{uc}, q'_{pc}). /* Does q_{uc} imply q'_{pc} ? */
(iii) If Found then let Found = Implies$(q'_{pc}, q_{uc} - q'_{pc})$. /* Check if any constraint in q_{uc} that are not in q'_{pc} can be derived from q'_{pc}. */
(iv) If (not Found) then Temp = Temp − $[q'_p]$. /* If q_{uc} *does not* imply q'_{pc} */

} /* **while** */
If (Found) then return q'_p and exit, otherwise continue.

Step 3: Identify all queries $q''_p \in Q^*_p$ that satisfy the following property: $q''_p \prec q_u$.

(i) Let $Q^*_p = Q^*_p - Q'_p$.
(ii) Identify the set of queries, Q^{**}_p:
$Q^{**}_p = \{q^{**}_{p_j} \mid j \in \{Cquery_{c_1} \cup Cquery_{c_2} \ldots \cup Cquery_{c_n}\}$, where $c_i \in q_{uc}$, for $i = 1, 2, \ldots, n\}$.
(iii) Let $Q''_p = Q^*_p \bigcap Q^{**}_p$.
(iv) If Q''_p is empty, then goto Step 5, else continue.

Step 4: Identify a query q''_p that is semantically equivalent to q_u from the queries in Q''_p. We proceed as follows:

(i) Let Found = *false*, and Temp = Q''_p.
(ii) **while** (not Found) and (Temp $\neq \emptyset$) do
{

(a) Let q''_p be the query corresponding to the first query in Temp.
(b) Let Found = Implies (q''_{pc}, q_{uc}). /* Does q''_{pc} imply q_{uc} ? */
(c) If Found then let Found = Implies$(q_{uc}, q''_{pc} - q_{uc})$. /* Check if any constraint in q''_{pc} that are not in q_{uc} can be derived from q_{uc}. */
(d) If (not Found) then Temp = Temp − $[q''_p]$. /* If q''_{pc} *does not* imply q_{uc} ? */

} /* **while** */

(iii) If (Found) then return q''_p and exit, otherwise continue.

Step 5: Identify a pre-optimized query q'''_p from the rest of the pre-optimized queries in the set Q^*_p.

(i) Let $Q'''_p = Q^*_p - Q''_p$,

(ii) Let Found = *false*, and Temp = Q''_p.
(iii) **while** (not Found) and (Temp $\neq \emptyset$) do
{

(a) Let q'''_p be the query corresponding to the first index in Temp.
(b) Let Found = Implies (q'''_{pc}, q_{uc}).
(c) If (Found) then let Found = Implies (q_{uc}, q'''_{pc}).
(d) If (not Found) then Temp = Temp − [q'''_p].

} /* **while** */
(iv) If (Found) then return q'''_p otherwise return NIL. Exit the mapping algorithm.

The mapping algorithm, as discussed above, can handle the Select-Project-Join (SPJ) (conjunctive) class of queries. As shown in [16], establishing equivalence between two arbitrarily complex boolean expressions is an NP-complete problem. The algorithm is thus restricted in its ability to handle queries with deeply nested constraints.

The function *Implies* can handle transitive implications of any level. Thus, if we have rules in the IDB such that, $A \Rightarrow B$, $B \Rightarrow C$, then it concludes $A \Rightarrow C$. We avoid getting into infinite loops by stopping when any constraint is revisited.

Function Implies ($q_{premise}$, $q_{derived}$)**: boolean;**
{

/* $q_{premise}$ and $q_{derived}$ are two sets of constraints. The function attempts to *derive* the constraints in $q_{derived}$ from those in $q_{premise}$, using the rules in the IDB. */

(i) **If** $q_{derived} = \emptyset$ then Implies = true. Return.
(ii) **Else** {

(a) **Identify** the set of rules
(b) $\{R_p \mid p \in \{Arule_{c_1} \cup Arule_{c_2} \cup \ldots Arule_{c_l}\}$, where $c_i \in q_{premise}$, for $i = 1, 2, \ldots l$, l being the number of attributes mentioned in $q_{premise}\}$.

/* R_p contains all rules in the IDB whose *antecedent* parts refer to some attribute(s) that are mentioned in $q_{premise}$. */

(c) Let $q_{derived-only} = q_{derived} - q_{premise}$, with $q_{derived-only} = \bigcup_{k=1}^{r} q_{derived-only_k}$, where r is the number of constraints in $q_{derived-only}$.
(d) Let Flag = true, k = 1.

/* Flag is a boolean variable that returns *true* if all constraints $q_{derived-only_k}$ can be derived from $q_{premise}$, and *false* otherwise. */

(e) **While (Flag) and (k ≤ r)** do
{

i. Identify the set of rules
$\{R_d \mid d \in \{Crule_{c_1} \cup Crule_{c_2} \cup \ldots Crule_{c_n}\}$, where $c_i \in q_{derived-only_k}$, for $i = 1, 2, \ldots, n$. n being the number of attributes mentioned in $q_{derived-only_k}\}$.

/*R_d contains all rules in the IDB whose *consequent* parts refer to some attribute(s) mentioned in $q_{derived-only_k}$. */

Fig. 4. Function *Implies*

ii. Let $R_{pd} = R_p \bigcap R_d$.

/*R_{pd} contains all rules in the IDB whose *antecedent* parts refer to some attribute(s) mentioned in $q_{premise}$, and *consequent* parts refer to some attribute(s) mentioned in $q_{derived-only_k}$. Since we are interested in deriving $q_{derived-only_k}$ from $q_{premise}$, we focus only on these rules. */

iii. Find a rule from R_{pd}, that has the following form: $q_{premise_i} \Rightarrow \cdots \Rightarrow q_{derived-only_k}$, where $q_{premise_i}$ is any constraint in $q_{premise}$. If no such rule is found, then set Flag to *false*.

iv. k = k + 1.
} /* While */
Implies = Flag.

}

} /* Implies */

Fig. 4. (*Continued*)

3.4.1. Examples

In this section, we explain the above algorithm with the help of some examples.

Example 1: In this example, $q_p \succ q_u$.

q_u: *select* (*s.OName*) *from SHIPS s where s.SType* = "supertanker" *and s.Dwt* > 350 *and c.PName* = "Marseilles".

Step 0: From Table 1, *Rquery* of s.ONAME = {1,3,4,5,6}. Thus Q_p^* = {1,3,4,5,6}.

Step 1: From Table 1, $Cquery_{s.SType}$ = {1,4,5,6}, $Cquery_{s.Dwt}$ = {3,4,5}, and $Cquery_{c.PName}$ = {2,4}. Thus, Q_p^{**} = {1,4,5,6} $\bigcap$ {3,4,5} $\bigcap$ {2,4} = {4}, $Q_p' = Q_p^* \bigcap Q_p^{**}$ = {4}. Since Q_p' is not empty we continue to Step 2.

Step 2: Found = *false*, Temp = {4}. We enter the *while* loop. q_p' = q_{p4}. Function *Implies* is invoked. $q_{premise}$ = {s.SType= "supertanker",s.Dwt > 350, c.PName= "Marseilles"}. From Fig. 3, $q_{derived}$ = {o.Business = "leasing", s.SType = "supertanker", s.Dwt > 350, c.PName = "Marseilles"}. $q_{derived-only}$ = {o.Business = "leasing"}, and r = 1. From Table 1, $Arule_{s.SType}$ = {7,9,10}, $Arule_{s.Dwt}$ = {1,2,6}, and $Arule_{c.PName}$ = {}. R_p = {7,9,10} $\cup$ {1,2,6} $\cup$ {} = {1,2,6,7,9,10}. Flag = true, we enter the *While* loop. k = 1. From Table 1, $Crule_{o.Business}$ = {2}. Since there is only one attribute mentioned in $q_{derived-only_k}$, R_d = {2}. R_{pd} = {1,2,6,7,9,10} $\bigcap$ {2} = {2}. From Fig. 2, R2 = (Dwt > 300) $\Rightarrow$ (Business = "leasing"). $q_{premise_2}$ = s.Dwt > 350, and this is a stronger constraint on the same field (namely s.Dwt). Hence we conclude that rule R2 can be applied to derive the constraint o.Business = "leasing". k = 2. Since $k > r$, we exit the *While* loop. Function *Implies* returns *true*. Thus, Found is assigned *true*. This causes an exit from the *While* loop, and we return q_{p4} as the semantically equivalent query and stop.

Example 2: In this example, $q_p \prec q_u$.

q_u: *select* (*c.Quantity*) *from CARGOES c where* *c.CType* = "LNG" *and c.PName* = "Marseilles" *and s.SType* = "Pressure Vessel".

Step 0: From Table 1, *Rquery* of c.Quantity = {2,3}. Since, the retrieval part of q_u contains only one attribute, $Q_p^* = \{2,3\}$.

Step 1: From Table 1, $Cquery_{c.CType} = \{2,3\}$, $Cquery_{c.PName} = \{2,4\}$, and $Cquery_{s.SType} = \{1,4,5,6\}$. $Q_p^* = \{2,3\} \cap \{2,4\} \cap \{1,4,5,6\} = \{\}$. Consequently, $Q_p^{'}$ is empty, and we go to Step 3.

Step 3: $Q_p^* = \{2,3\} - \{\} = \{2,3\}$. From Table 1, $Cquery_{c.CType} = \{2,3\}$, $Cquery_{c.PName} = \{2,4\}$, and $Cquery_{s.SType} = \{1,4,5,6\}$. Thus, $Q_p^{**} = \{2,3\} \cup \{2,4\} \cup \{1,4,5,6\} = \{1,2,3,4,5,6\}$. $Q_p^{''} = \{2,3\} \bigcap \{1,2,3,4,5,6\} = \{2,3\}$. Consequently, $Q_p^{''}$ is not empty, and we go to Step 4.

Step 4: Found = *false*, Temp = {2,3}. We enter the *while* loop. Function *Implies* is invoked. $q_{derived}$ = {c.CType = "LNG",c.PName = "Marseilles", s.SType = "supertanker"}. The first query in Temp is q_{p2}. Thus $q_p^{''} = q_{p2}$, and $q_{pc}^{''} = q_{premise}$. From Fig. 3, $q_{premise}$ = {c.CType = "LNG",c.PName = "Marseilles"}. Thus, $q_{derived-only}$ = {s.SType = "Pressure Vessel"}. From Table 1, $Arule_{c.CType} = \{5,9\}$, and $Arule_{c.PName}$ {}. Thus, $R_p = \{5,9\}$. Flag = true, we enter the *While* loop. $k = 1$. From Table 1, $Crule_{s.SType} = \{6,9\}$. Since there is only one attribute mentioned in $q_{derived-only_k}$, $R_d = \{6,9\}$. R_{pd} = $\{5,9\} \bigcap \{6,9\} = \{9\}$. From Fig. 2, R9 = (*CType* = "LNG")⇒ (*SType* = "Pressure vessel"). Since $q_{premise_1}$ = {c.CType = "LNG"}, we use R9 to conclude $q_{derived-only_1}$. $k = 2$. Since $k > r$, we exit the *while* loop. Found is assigned *true*. We return q_{p2} as the semantically equivalent query and stop.

Example 3: In this example, $q_p \not\succ q_u$, and $q_p \not\prec q_u$.

q_u : *select* (*s.OName*) *from SHIPS s where s.Dwt* > 150.

Step 0: From Table 1, $Rquery_{s.SName} = \{1,3,4,5,6\}$. Since, the retrieval part of q_u contains only one attribute, $Q_p^* = \{1,3,4,5,6\}$.

Step 1: From Table 1, $Cquery_{s.Dwt} = \{3,4,5\}$. Since there is only one attribute in q_{uc}, $Q_p^{**} = \{3,4,5\}$. $Q_p^{'} = Q_p^* \bigcap Q_p^{**} = \{3,4,5\}$. Since $Q_p^{'}$ is not empty we continue to Step 2.

Step 2: Found = *false*, Temp = {3,4,5}. We enter the *while* loop. Function *Implies* is invoked. q_{uc}, and hence, $q_{premise}$ = {s.Dwt > 150}. The first query in Temp is q_{p3}. Thus $q_p^{'} = q_{p3}$, $q_{pc}^{'} = q_{derived}$. From Fig. 3, $q_{derived-only}$ = {s.Dwt < 60, c.CType = "Refined petroleum", p.country = "Denmark"}, and $r = 3$. From Table 1, $Arule_{s.Dwt} = \{1,2,6\}$. Thus, $R_p = \{1,2,6\}$. Flag = true, we enter the *While* loop. $k = 1$. From Table 1, $Crule_{s.Dwt} = \{10\}$. Since there is only one attribute mentioned in $q_{derived-only_k}$, $R_d = \{10\}$. R_{pd} = $\{1,2,6\} \bigcap \{10\} = \{\}$ Since R_{pd} is empty, we set Flag to *false*. *Implies* returns *false*. Since Found is *false*, we reduce Temp by its first element. Thus Temp

$=\{4,5\}$. In a similar manner, we can test queries q_{p4} and q_{p5} for semantic equivalence. We are not giving the detailed steps here, but it can be verified that neither q_{p4} nor q_{p5} will satisfy the requirements for semantic equivalence. Thus, Temp will eventually be reduced to $\emptyset$, and we will exit the *while* loop. Found will be still *false*, and thus we continue to Step 3 of the algorithm.

Step 3: $Q_p^* = \{1,3,4,5,6\} - \{3,4,5\} = \{1,6\}$. $q_{uc} = \{\text{s.Dwt} > 150\}$. From Table 1 $Cquery_{s.Dwt} = \{3,4,5\}$. Thus, $Q_p^{**} = \{3,4,5\}$, and $Q_p'' = \{1,6\} \bigcap \{3,4,5\} = \{\}$. Since Q_p'' is empty, we go to Step 5.

Step 5: $Q_p''' = \{1,6\}-\{\} = \{1,6\}$. Found = *false*, Temp = $\{1,6\}$. We enter the *while* loop. Function *Implies* is invoked. The first query in Q_p''' is q_{p1}. Thus $q_p''' = q_{p1}$. From Fig. 3, q_{pc}''', and hence $q_{premise}$= {s.SType = "supertanker"}, $q_{derived}$ = {s.Dwt > 150}, and $r = 1$. From Table 1, the $Arule_{s.SType}$ = {7,9,10}. Thus, $R_p = \{7,9,10\}$. We set Flag to *true* and enter the *While* loop. $k = 1$. $q_{derived-only_1}$ = {s.Dwt > 150}. From Table 1, the $Crule_{s.Dwt} = \{10\}$. Thus, $R_d = \{10\}$. $R_{pd} = \{7,9,10\} \bigcap \{10\} = \{10\}$. From Fig. 2, we find that the antecedent of rule R10 is s.Stype = "supertanker", which is $q_{premise_1}$, and the consequent part of rule R10 is s.Dwt > 150, which is $q_{derived-only_1}$. $k = 2$. Since $k > r$, we exit the *While* loop. Since Flag is *true* function *Implies* returns *true*. Function *Implies* is invoked a second time. Now, $q_{premise} = q_{uc}$, and $q_{derived} = q_{pc}'''$. From Table 1, $Arule_{s.Dwt} = \{1,2,6\}$. Thus, $R_p = \{1,2,6\}$. $q_{derived-only}$ = {s.SType = "supertanker"}. Flag = *true*, and $s = 1$. We enter the *While* loop. $k = 1$. $q_{derived-only_1}$ is s.SType = "supertanker". From Table 1, $Crule_{s.SType}$ is {6,9}. Thus, $R_p = \{6,9\}$. $R_{pd} = \{1,2,6\} \bigcap \{6,9\} = \{6\}$. From Fig. 2, the antecedent part of rule R6 is $q_{premise_1}$ and the consequent part of rule R6 is $q_{derived-only_1}$. $k = 2$. Since $k > r$, we exit the *While* loop. Since Flag is *true* function *Implies* returns *true*. Thus, we set Found to *true*. Since Found is *true*, we exit the *while* loop. Since Found is *true*, we return q_{p1} as the semantically equivalent query, and stop.

3.4.2. Correctness of the algorithm

Theorem 3.1. *If q_u is mapped onto q_p by the above algorithm, then q_u is semantically equivalent to q_p.*

Proof. 3.1. *Step 0 of the algorithm ensures that the set of target attributes of q_u is a superset of that of q_u. This means that all attributes requested by q_u will also be retrieved by q_p. The additional attributes retrieved by q_p but not requested by q_u can be eliminated in subsequent processing.*

The constraint parts of any two queries, q_u and q_p, can have the following general forms:

q_{uc}: c_{u_1} *and* c_{u_2} *and* ... c_{u_n}.

q_{pc}: c_{p_1} *and* c_{p_2} *and* ... c_{p_m}.

n and m are the number of constraints in q_{uc} and q_{pc} respectively, and c_i is a constraint of the form <attribute rel-op expression>, where rel-op could be any relational operator ($>, \geq, =, \neq$, etc.), and expression could be an attribute, or a numeric or string constant. For example, c_i could be (dwt > 150).

If the algorithm maps q_u to q_p, the following must also be true:

$\forall c_i \mid c_i \in q_{uc}$ *and* $c_i \notin q_{pc}$, $\exists c_j \mid c_j \in q_{pc}$ *and* $c_j \notin q_{uc}$, *and* $c_j \Rightarrow \ldots \rightarrow c_i$.

$\forall c_k \mid c_k \in q_{pc}$ *and* $c_k \notin q_{uc}$, $\exists c_l \mid c_l \in q_{uc}$ *and* $c_l \notin q_{pc}$, *and* $c_l \Rightarrow \ldots \rightarrow c_k$.

Given the above, any constraint that is in q_u but not in q_p can be derived to from those in q_u, and vice versa, *by* modus ponens *[8]. Thus, q_{pc} and q_{uc} are logically, and thus, semantically equivalent.*

3.5. Queries that have No Match in the Set of Pre-optimized Queries

It is conceivable that there will be instances when a user query will have no semantically equivalent counterpart in the set of pre-optimized queries. In processing some of these queries, we can make use of the knowledge gained by attempting to find a match in the pre-optimized query set. In this section, we propose some heuristics for handling such cases. In general, the user query has to fall into one of the following categories:

(i) The user query differs from a pre-optimized query in the retrieval parts but matches in the constraint parts. Thus the selection conditions of the two queries are the same. Since the tuples selected by the two queries are identical, we can replace the retrieval part of q_p with that of q_u and process the modified query.

(ii) The user query is a superset of one or more pre-optimized queries. If there are several such queries, a workable heuristic is to select the one that is the largest subset of the user query. The following steps are then performed.

- Process the following query: *select* $*$ *where* q_{pc}.
- Process the following query on the tuples selected from the previous step: *select* q_{ur} *where* $q_{uc} - q_{pc}$.

(iii) The user query is a union of two pre-optimized queries, that is, $q_{uc} = q_{pc_1} \bigcup q_{pc_2}$: Here there are two possibilities:

- q_{pc_1} and q_{pc_2} are mutually exclusive: Process q_{pc_1} and q_{pc_2} sequentially.
- q_{pc_1} and q_{pc_2} are not mutually exclusive:
  - Process q_{pc_1}.
  - Process $q_{pc_2} - q_{pc_1}$

(iv) The only other possibility is as follows: q_{uc} is either a subset, or neither subset nor superset of q_{pc}. In such cases, the user query has to be optimized using one of the extant semantic query optimization techniques. However, we could reduce the size of IDB for optimizing the query. Since the semantic equivalence is established by using rules that involve the constraints in the user query, we could optimize q_{uc} by considering *only* those rules mentioned in the Arules corresponding to the constraints in q_{uc}.

4. Performance Analysis

In this section we present some results of performance testing of the mapping algorithm presented in Sec. 3.4. The schema presented in Fig. 1 was used for the experiments. There were 15 rules in the IDB and 34 queries in the pre-optimized query set. The relations had the following sizes (number of tuples):

(i) SHIPS: 2000
(ii) PORTS: 500
(iii) CARGOES: 2000
(iv) OWNERS: 200
(v) POLICIES: 400
(vi) INSURERS: 400

Four queries were processed, first in unoptimized forms, then in optimized forms. The INGRES database management system was used to store the relations and run the queries. The algorithm for mapping, Q_p, and IDB were implemented in Quintus Prolog. The following timings were measured.

(i) Time to map q_u to q_p (T_m).
(ii) Time to process q_u (T_u).
(iii) Time to process q_p (T_p).

The results are shown in Table 2. As can be seen from Table 2, $T_m + T_p < T_u$ in all cases, with time savings ranging from 17% to 29%.

Table 2. Experimental results.

Timings in milliseconds				
Q #	T_m	T_p	T_u	Savings
1	217	5920	7400	17 %
2	184	13880	19440	28 %
3	50	3580	5120	29 %
4	50	5840	7700	24 %

5. Conclusion and Future Research

In this paper we have described a method for using the concept of precompilation of queries in the context of semantic query optimization. We discussed the issues of generating the set of pre-optimized queries, representing the intensional database and the set of pre-optimized queries, identifying a semantically equivalent

counterpart of a user query, searching through the intensional database for potentially applicable rules, and maintaining the pre-optimized query set. Performance issues of the mapping algorithm were also discussed.

Furthermore we plan to study the performance of the system when applied to three example databases which have been discussed in [12], [14] and [22] as well as two real world example databases that we have access to. In order to ensure anonymity, the name of the organizations will be withheld and some of the actual data will be modified.

Our study would entail a detailed analysis of the trade-off between optimization and execution costs. Further we would test the individual and joint effects of the following factors on the performance of the algorithm: the sizes of Q_p and IDB, and the characteristics of q_u and q_p (specifically, the number of attributes in the retrieval and the condition parts).

Ways of integrating our proposed system with currently available algorithms (e.g. [5,14,22]), will be explored. This integration means that if our proposed system concludes that for a given user query, q_u, no semantically equivalent query is available in the set Q_p, then a selected query optimization algorithm would then begin generating the optimal semantically equivalent query (since, for any given query q_u, there could be more than one semantically equivalent queries) by making use of the information so far gathered by our system.

References

[1] N. R. Adam, A. Gangopadhyay and J. Geller, Knowledge based query processing using preoptimized queries, in *Proc. First Int. Conf. on Information and Knowledge Management*, Nov. 1992.

[2] P. Apers, A. Henver and S. Yao, Optimization algorithms for distributed queries, *TSE* **9**, 1 (1983) 57–68.

[3] P. Bernstein, N. Goodman, E. Wong, C. Reeve, and J. Rothnie, Query processing in SDD-1: A system for distributed databases, *ACM Trans. Database Systems* **6**, 4 (1981) 602–625.

[4] U. S. Chakravarty, D. H. Fishman and J. Minker, Semantic query optimization in expert systems and database systems, *Expert Database Systems: Proc. First Int. Workshop*. ed. L. Kershberg (Benjamin/Cummings, 1986) pp. 659–674.

[5] U. S. Chakravarty, J. Grant and J. Minker. Logic-based approach to semantic query optimization, *ACM Trans. Database Systems.* **15**, 2 (1990) 162–207.

[6] U. S. Chakravarty, J. Minker and J. Grant, Semantic query optimization: Additional constraints and control strategies, *Expert Database Systems: Proc. Second Int. Workshop*, ed. L. Kershberg (Benjamin/Cummings, 1987) 345–379.

[7] J. Chen and V. Li, Optimizing joins in fragmented database systems on a broadcast local network, *IEEE Trans. Softw. Engin.* **12**, 1 (1989) 27–38.

[8] M. R. Genesereth and N. J. Nilsson, *Logical Foundations of Artificial Intelligence* (Morgan Kaufmann, 1987).

[9] M. Hammer and S. B. Zdonik, Knowledge based query processing, *Proc. VLDB*, Montreal, Canada, 1980 pp. 137–147.

[10] M. Jarke, External semantic query simplification: A graph theoretic approach and its implementation in PROLOG, *Expert Database Systems: Proc. First Int. Workshop* ed. L. Kershberg (1986) pp. 675–690.

[11] M. Jarke, J. Clifford and Y. Vassiliou, An optimizing PROLOG front-end to a relational query system, in *Proc. ACM-SIGMOD Conf.*, 1984, pp. 296–306.

[12] M. Jarke and J. Koch, Query optimization in database systems, *Compt. Surv.* **16**, 2 (1984) 112–152.

[13] J. King, QUIST: A system for semantic query optimization in relational databases, in *Proc. VLDB*, Cannes, France, 1981, pp. 510–517.

[14] J. King, Query optimization by semantic reasoning (UMI Research Press, 1984).

[15] M. Mannino, P. Chu and T. Sager, Statistical profile estimation in database systems, *ACM Comput. Surv.* **20**, 3 (1988) 191–221.

[16] B. Nebel, Computational complexity of terminological reasoning in BACK, *Artif. Intell.* **34** (1988) 371–383.

[17] S. Pramanik and D. Vineyard, Optimizing join queries in distributed databases, *IEEE Trans. Softw. Engin.* **14**, 9 (1988) 1319–1326.

[18] X. Qian and D. R. Smith, Integrity constraint reformulation for efficient validation, in *Proc. VLDB* (Morgan Kaufman, 1987) pp. 417–425.

[19] T. Selis, Multiple-query optimization, *ACM Trans. Database Systems*, **13**, 1 (1988) 23–52.

[20] S. Shekhar, J. Srivastava and S. Dutta, A formal model of trade-off between optimization and execution costs in semantic query optimization, in *Proc. VLDB* (Morgan Kaufman, 1988) 457–467.

[21] S. T. Shenoy and Z. M. Ozsoyoglu, A system for semantic query optimization, *ACM SIGMOD*, 1987, pp. 181–195.

[22] S. T. Shenoy and Z. M. Ozsoyoglu, Design and implementation of a semantic query optimizer, *IEEE Trans. Knowledge and Data Engineering*, **I**, 3 (1989) 344-361.

[23] M. Siegel, E. Sciore and S. Salveter, A method for automatic rule derivation to support semantic query optimization, *ACM Trans. Database Systems* **17**, 4 (1992) 563–600.

[24] M. D. Siegel, Automatic rule derivation for semantic query optimization, *Proc. Second Int. Conf. on Expert Database Systems*, ed. L. Kerschberg, 1989, pp. 669–698.

[25] S. Yu, K. Ghu, D. Brill and A. Chen, Partition strategy for distributed query processing in fast local networks, *IEEE Trans. Softw. Engin.* **15**, 6 (1989) 781–793.