Semantic Query Optimization for Query Plans of Heterogeneous Multidatabase Systems *

Chun-Nan Hsu
Institute of Information Science
Academia Sinica
128 Yan-Jiu-Yuan Road Section 2
Nankang, Taipei City 115
TAIWAN
chunnan@iis.sinica.edu.tw
Ph: +886-2-27883799 Fax: +886-2-27824814

Craig A. Knoblock
Information Sciences Institute
Department of Computer Science
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292, USA
knoblock@isi.edu
http://www.isi.edu/~knoblock/

January 22, 1999

Abstract

New applications of information systems, such as electronic commerce and healthcare information systems, need to integrate a large number of heterogeneous databases over computer networks. Answering a query in these applications usually involves selecting relevant information sources and generating a query plan to combine the data automatically. As significant progress has been made in source selection and plan generation, the critical issue has been shifting to query optimization. This paper presents a semantic query optimization (SQO) approach to optimizing query plans of heterogeneous multidatabase systems. This approach provides global optimization for query plans as well as local optimization for subqueries that retrieve data from individual database sources. An important feature of our local optimization algorithm is that we prove necessary and sufficient conditions to eliminate an unnecessary join in a conjunctive query of arbitrary join topology. This feature allows our optimizer to utilize more expressive relational rules to provide a wider range of possible optimizations than previous work in SQO. The local optimization algorithm also features a new data structure called *AND-OR implication graphs* to facilitate the search for optimal queries. These features allow the global optimization to effectively use semantic knowledge to reduce data transmission cost. We have implemented this approach into the PESTO query plan optimizer as a part of the SIMS information mediator. Experimental results demonstrate that PESTO can provide significant savings in query execution cost over query plan execution without optimization.

Index terms: Semantic query optimization, heterogeneous multidatabase systems, relational rules, joins, information mediators.

*The research reported here was supported in part by the National Science Foundation under Grant No. IRI-9313993, and in part by Rome Laboratory of the Air Force Systems Command and the Defense Advanced Research Projects Agency under Contract No. F30602-94-C-0210. We wish to thank the SIMS project members: Yigal Arens, Wei-Min Shen, Andrew Philpot, Chin Y. Chee and José-Luis Ambite for their help on this work. This work was partly done while the first author worked at USC/Information Sciences Institute and Arizona State University.

Contents

1 Introduction

Integrating heterogeneous multidatabases is an important problem for the next generation of information systems [1, 2, 3, 4, 5, 6]. A wide-area health-care information system, for example, would require integrating many different types of information for physicians in the course of their work. Answering a query in these applications usually involves selecting appropriate information sources from which to retrieve data and generating an efficient plan to combine the data automatically. As significant progress has been made in source selection and plan generation, the critical issue has been shifting to query optimization. Source autonomy and heterogeneity preclude the use of traditional techniques for distributed databases [7, 8]. Recent research focuses on techniques to prune irrelevant source accesses [9, 10, 11], but it is still difficult to reduce unnecessary data retrieval and transmission from relevant sources. One reason for this difficulty is the lack of information about intermediate data at query planning time. To address this issue, researchers have proposed interleaving query planning and execution so that the query processor can use intermediate data to refine the part of the query plan that has not been completely executed [9, 12, 13].

A relatively unexplored area is the use of *semantic query optimization* (SQO) [14, 15, 16, 17, 18, 19, 20, 21] for multi-source query plan optimization. The advantage of SQO is that the optimizer can infer the information about intermediate data from semantic knowledge prepared prior to query execution time. Another reason is that SQO supports the extensibility of multidatabase systems because it minimizes the dependency on how individual sources execute a query. When a new information source is integrated into the system, the optimizer can still be used with minimal modification. Many algorithms are available for learning useful semantic knowledge [16, 22, 19, 23, 24, 25].

1.1 Query Plans

A *query plan* is a directed acyclic graph with its nodes as plan steps and its edges as the ordering constraints that specify data flow direction as well as the order in which the plan steps should be executed. Query plans generated by existing multidatabase query processing systems differ on the granularity of their plan steps. In this paper, we consider query plans where each plan step corresponds to a subquery that can be executed by a single database server.

For example, suppose we have a query as follows:

> **Example Query 1:** Retrieve the classes of active ships with container capability that can dock in the wharves with cranes at Long Beach seaport; list by ship class name and wharf id.

Suppose the data required to answer this query are spread over two remote databases: `Geo` for the data about geographical locations, seaports and wharves, and `Assets` for ships, ship classes,

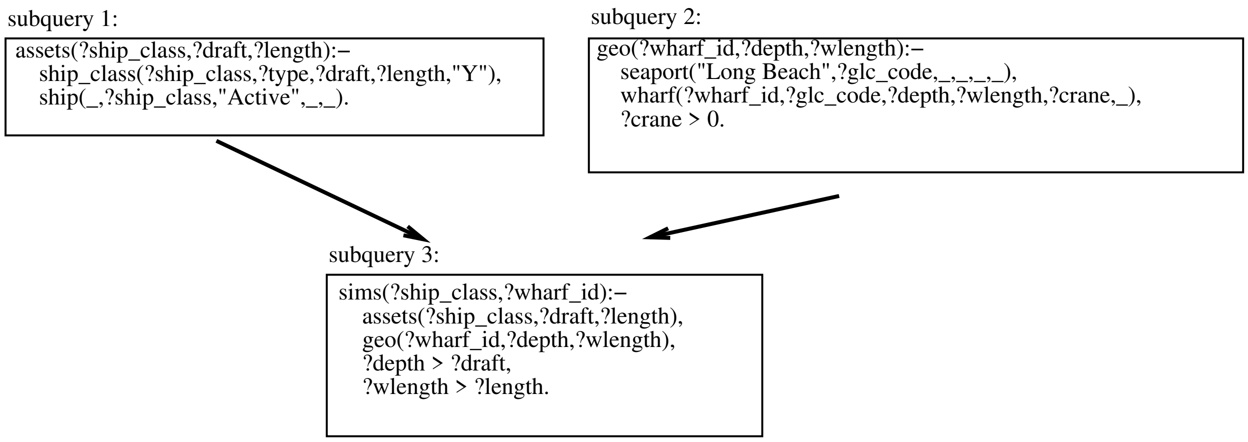


Figure 1: Example query plan that retrieves heterogeneous multidatabases

aircraft, etc. The source model of these databases is given in Table 1. Figure 1 shows a query plan that answers the given query.

In a query plan graph, subqueries without a predecessor retrieve data from a remote information source, while others combine or process the data transmitted from their preceding subqueries. In this example, the first subquery retrieves data about ships from the remote database `Assets`, the second subquery retrieves data about seaports and wharves from database `Geo`, and the third subquery compares data retrieved by the previous subqueries and joins the results.

This query plan is optimal in terms of the number of subqueries because the desired data are spread over two databases `Assets` and `Geo` and those subqueries cannot be merged. However, this plan could still be expensive because the system needs to retrieve and transmit a large amount of unnecessary ship class and wharf data that will eventually be discarded. Also in subquery 1, the system needs to execute a join over the large `ship` relation. Conventional query optimizers [7, 8, 26] can be applied to optimize individual subqueries, but still, the amount of unnecessary intermediate data would not be reduced.

1.2 Semantic Query Optimization

Semantic query optimization can help in providing local optimization to subqueries as well as reducing unnecessary data transmission. The essential idea of semantic query optimization is to use semantic rules about data, such as *all California seaports have railroad access*, to reformulate a query into a more efficient but *semantically equivalent* query. Two queries are defined to be semantically equivalent if they return identical answers from a database state that is consistent with the semantic knowledge. With the given rule, suppose we have a query

> **Example Query 2:** Find all California seaports with railroad access and 2,000,000 ft^3 of storage space.

The system can reformulate the query into a new query

> **Example Query 2 (optimized):** Find all California seaports with 2,000,000 ft^3 of storage space.

This optimized query is equivalent to the original query, because from the given rule there is no need to check the railroad access of seaports in California. Executing the optimized query is less expensive than executing the original query because the system saves the time for the unnecessary comparisons.

Semantic query optimization is not widely used in practice largely because it is difficult to encode useful semantic knowledge. Several algorithms have been designed specifically for this purpose previously [16, 22, 19]. Semantic knowledge used in our experiment is learned automatically by our knowledge discovery system [23, 24, 25].

The optimization approach described in this paper extends previous work in SQO, which focuses mainly on conjunctive queries in a stand-alone database. This approach provides global optimization for query plans as well as local optimization for subqueries that retrieve data from individual database sources. An important feature of our local optimization algorithm is that we prove necessary and sufficient conditions to eliminate an unnecessary join in a conjunctive query of arbitrary join topology. This feature allows our optimizer to utilize more expressive relational rules and thus provide a wider range of possible optimizations than previous work in SQO. The local optimization algorithm also features a new data structure called *AND-OR implication graphs* to facilitate the search for optimal queries. These features allow the global optimization to effectively use semantic knowledge to infer useful information about intermediate data to reduce data transmission cost.

1.3 Illustrative Example

We illustrate how our SQO approach can optimize the query plan in Figure 1. Suppose in the beginning, the optimizer possesses the following semantic knowledge about the relevant databases:

- Possible range of attribute values that binds the variables: `?depth`, `?draft`, `?length` and `?wlength`.
- Semantic rules about ship classes and ships:
  - if the maximum draft of a ship is less than 50 then its status is active.
  - if a ship class has container capability, then there exists at least one ship of that ship class.

Given the range information and semantic rules, our optimizer can reduce the intermediate data and optimize subqueries with the following steps.

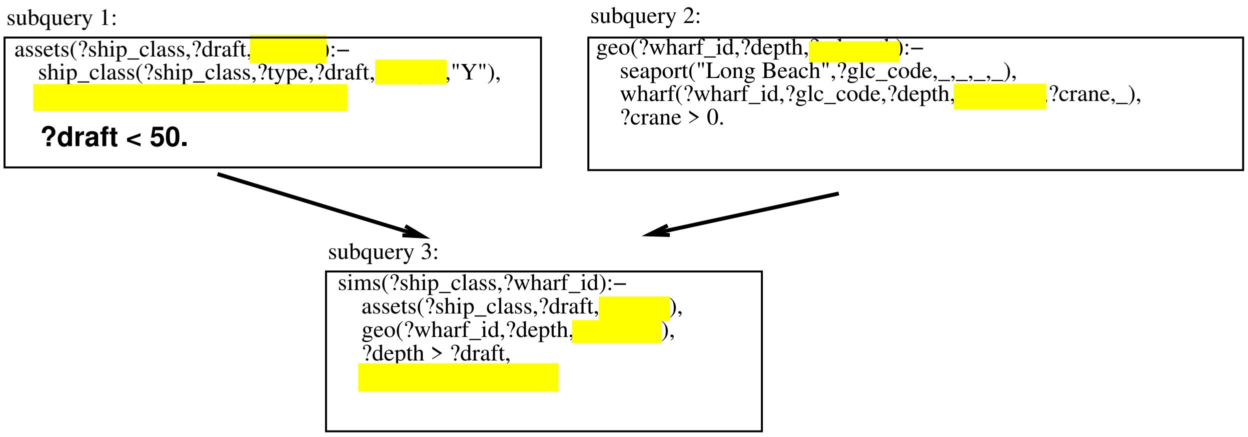


Figure 2: Optimized query plan

- From range information, inferring that `?wlength > ?length` is always true for this query. Therefore, this literal can be deleted from Subquery 3. Also, since they are not used as output, data about `?wlength` and `?length` are no longer necessary and can be deleted from the heads of Subqueries 1 and 2.
- From range information again, deriving a more restrictive range of `?draft`, which must be less than 50 to satisfy `?depth > ?draft`.
- Moving this newly derived literal `?draft < 50` to Subquery 2.
- Deriving that it is unnecessary to specify `"Active"` based on the semantic rules, a ship is always `"Active"` if `?draft < 50`.
- Removing the entire `ship` literal, because `"Active"` is redundant, and the semantic rule states that if a ship class with container capability (specified as `"Y"`), then there must exist at least one ship that belongs to that ship class in the database. That is, the join with `ship` is not necessary.

The resulting plan is shown in Figure 2, where terms in bold font are newly inserted terms, and terms under shaded regions are deleted. This new plan will retrieve the same answer as the original plan because the optimization process is logically sound, given that the semantic knowledge is consistent with the database state. Since the system does not need to transmit `?wlength` and `?length`, and a more restrictive constraint `?draft < 50` will reduce the number of `?ship_class` data retrieved, the intermediate data is reduced. Furthermore, the system does not need to access and compute a join over the costly relation `ship`. Thus, Subquery 1 is also optimized. In our experiments, it takes 3.17 seconds to execute the original plan and 1.78 seconds to execute the optimized plan. This amounts to a 43.8 percent reduction. The execution time of the optimized plan includes 0.03 seconds of overhead optimization time. Therefore, the optimizer effectively optimizes this query and reduces the execution cost.

Assets database:

```
ship_class(class,type,max_draft,length,container_cap),
ship(name,class,type,status,fleet,year_built).
```

Geo database:

```
geoloc(name,glc_cd,country,latitude,longitude),
seaport(name,glc_code,storage,rail,road,anch_offshore),
wharf(wharf_id,glc_code,depth,length,crane_qty).
```

Table 1: Schema of example databases

1.4 Organization

The remainder of this paper describes our approach. The next section defines the terminology used throughout this paper. Section 3 describes how to exploit general relational rules to delete unnecessary joins. Section 4 describes AND-OR implication graphs and how they facilitate search in the local optimization algorithm for conjunctive subqueries. Section 5 presents the global optimization algorithm. The approach has been implemented into a system called PESTO and tested as a component of the SIMS information mediator [2, 3] in a heterogeneous multidatabase environment. Section 6 reports on the implementation and the experimental results. Section 7 compares our approach with related work in query optimization. The last section concludes with a discussion of the contributions and future work.

2 Terminology

2.1 Databases

In this paper, we consider information sources organized in the relational data model because it is well-defined and widely used in practice. However, it should be emphasized that the approach described in this paper applies to information sources in more expressive data models such as the object-relational model with minor extensions. Table 1 shows the schema of two example databases. In database `Assets`, the relation `ship_class` stores information about ship classes, and `ship` contains data about individual ships. The other database `Geo` provides geographic location information. We list the schema of two relations on seaports and wharves related to our examples.

2.2 Queries and Subqueries

We assume that *queries* are expressed in terms of some uniform language and a query processor will generate a query plan to answer an input query by decomposing an input query into subqueries to relevant information sources and determining a correct and efficient order to execute these subqueries. In our approach, the optimizer takes the query plan as input instead of the query.

Typically, *subqueries* are expressed in conjunctive Datalog queries, which correspond to the `select-from-where` subset of SQL.[1] A subquery is of the form

$$S(\bar{X}) \; :- \; C(\bar{Y}), E_1(\bar{X}_1), \ldots, E_k(\bar{X}_k).$$

The head S is the ID of the subquery. Usually we assign the ID of the database site from which the subquery retrieves data. The parameters $\bar{X}, \bar{Y}, \bar{X}_1, \ldots, \bar{X}_k$ denote tuples of variables. Variables always start with a `?` mark (e.g., `?x`), except anonymous variables "_," which represent variables that appear exactly once in the query and thus can be omitted. $C(\bar{Y})$ is a conjunction of built-in predicates on the variables of the query. The E_j's are relation names of the database from the source model, or IDs of the predecessor subqueries. In the latter case, the literal represents the tuples returned from the predecessor subquery. We refer to literals $E_j(\bar{X}_j)$ as *database literals*. We refer to literals on built-in predicates, such as `>` and `member`, between a single variable and one or more constants as *built-in literals* (e.g., `?crane > 0`). Literals on built-in predicates between two or more variables are *comparisons* (e.g., `?depth > ?draft`), however equalities are required to be replaced with common variables in database literals. We allow negations and disjunctions in built-in literals.

For example, `Q1` in Table 2 is a subquery that begins with the site name `assets` followed by arguments `?ship_class`,`?draft` and `?length`. The literals in line 2 and line 3 are database literals, while literals 4 to 6 are built-in literals. This subquery retrieves the ship classes and the maximal draft of the ships in those classes which satisfy the following conditions: the ships in the class are capable of carrying containers, their draft is less than 50 feet, and there is at least one active ship in this class.

A query plan may have subqueries for disjunctions and set operators (e.g., `union` and `intersect`). Since the cost to perform these operations is relatively expensive, usually a query processor will separate ordinary literals and the set operations, and push the latter down the plan graph so that they will be executed as late as possible. Therefore, we assume that the query processor will create subqueries specifically for the set operations.

2.3 Semantic Knowledge

Our approach uses two forms of semantic knowledge: *semantic rules* and *range facts*. Semantic rules, expressed in terms of Horn-clause rules, define the regularity of data in an individual database. We adopt standard Prolog terminology and semantics as defined in [28] in our discussion of rules. Semantic knowledge is interpreted under the closed-world assumption. That is, a database literal

[1]In our implementation, however, queries are expressed in the LOOM knowledge representation language [27]. It is also used as the representation language for database modeling. For the simplicity of presentation, we choose Datalog to express queries because modeling is not the subject of this paper and Datalog is well-known to both databases and AI research communities.

```
Q1: assets(?ship_class,?type,?draft):-
        ship_class(?ship_class,?type,?draft,_,?container),
        ship(_,?ship_class,?type,?status,_,_),
        ?status = "Active",
        ?container = "Y",
        ?draft < 50.
```

Table 2: Example subquery

Semantic Rules:

```
R1: If the maximum draft of a ship is less than 50 then its status is active.
  ship_class(?class,?type,?draft,_,_) ∧ ship(_,?class,?type,?status,_,_) ∧ ?draft < 50
     ⇒ ?status = "Active"
R2: If a ship class has container capability, then there must exist some
    ships that belong to that ship class in the database.
  ship_class(?class,?type,_,_,?container) ∧ ?container = "Y"
     ⇒ ship(_,?class,?type,_,_,_)
R3: If a ship is active, then it was built after 1945.
  ship(_,_,_,_,?status,?year-built) ∧ ?status = "Active"
     ⇒ ?year-built > 1945
R4: The depth of wharves at Long Beach is at most 50 feet.
  seaport(?name,?code,_,_,_,_) ∧ wharf(_,?code,?depth,_,_) ∧ ?name = "Long Beach"
     ⇒ ?depth ≤ 50
R5: The length of wharves with at least one crane at Long Beach is greater than 1200 feet.
  seaport(?name,?code,_,_,_,_) ∧ ?name = "Long Beach" ∧
  wharf(_,?code,_,?length,?crane) ∧ ?crane > 0
     ⇒ ?length ≥ 1200
R6: For all the geographic location codes of wharves, there is a seaport with the same code.
  wharf(_,?code,_,_,_) ⇒ seaport(_,?code,_,_,_,_)
R7: If two seaports share the same geographic location code, than their names are also identical.
  seaport(?name1,?code,_,_,_,_) ∧ seaport(?name2,?code,_,_,_,_)
     ⇒ ?name1 = ?name2
```

Table 3: Example semantic rules

is satisfiable with regard to a database if and only if in the database there exists a tuple in the corresponding relation. Semantic knowledge should be consistent with the database. To distinguish a rule from a query, we show queries using Datalog syntax and semantic rules in a standard logic notation. Table 3 shows some example semantic rules.

We make distinction between two classes of rules. The first class, referred to as a *range rule*, contains rules with their consequent a positive built-in literal (e.g., `R1`). The second class consists of rules with their consequent a database literal (e.g., `R2`), referred to as a *relational rule*. Relational rules may be recursive, that is, the relation of the consequent appears in the antecedent. The class of relational rules subsumes a variety of database integrity constraints. `R6` is an example of a referential integrity constraint, a special case of relational rule where only one condition in the antecedent and only one shared variable on both sides of implication are allowed. A functional dependency can also be expressed in a Horn-clause rule such as `R7`.

Range Facts:

```
F1: 12 ≤ ship_class.draft ≤ 72.
F2: 325 ≤ ship_class.length ≤ 950.
F3: ship_class.container_cap ∈ {"Y","N"}.
F4: ship.status ∈ {"Active","Inactive","Resigned"}.
F5: 7 ≤ wharf.depth ≤ 100.
F6: 580 ≤ wharf.length ≤ 2700.
F7: 0 ≤ wharf.crane_qty ≤ 7.
```

Table 4: Range facts state the range of attribute values

Range facts state the range of the values of a given database attribute. For numeral attributes, the range facts show the minimal value and the maximal value. For string-typed attributes, their range facts enumerate the possible values. In our implementation, range facts of a string-typed attribute will not be used if there are more than 20 possible values. Table 4 gives some examples.

3 Semantic Query Optimization Using Relational Rules

Semantic query optimization involves two major phases. In the first phase, the optimizer locates applicable semantic knowledge and proposes a sequence of one or more reformulation operations (e.g., to delete a literal or insert a new literal) that preserve the semantics of the query. The second phase is to evaluate the proposed reformulations and apply the best reformulation based on a cost model of query execution. Relational rules are useful for the optimizer to detect and eliminate redundant database literals in the first phase. This section first reviews the techniques for proposing reformulations based on range rules, and then presents the techniques for using relational rules.

3.1 Query Reformulation Based on Semantic Knowledge

A semantic query optimizer proposes reformulations depending on the applicable semantic rules. Possible reformulations are refining the range constraint specified in a built-in literal (e.g., refining `?crane > 0` into more restrictive `?crane > 5`), deleting a literal, inserting a new literal to the query, and refuting the entire query (e.g., asserting that it will return an empty set). A rule is considered *applicable* to a query if the antecedent part of the rule is a logical consequence of the query body. For conjunctive queries and Horn-clause rules, this can be determined by computing a *containment mapping* [26], which is a set of variable substitutions that unify the antecedent of an applicable rule and the query body. Once an applicable rule is matched, the optimizer will substitute the variables in the consequent of the rule using the variable substitutions.

If the applicable rule is a range rule, where the consequent is a built-in literal, then the optimizer may propose inserting the consequent if there is no built-in literal defined on the same variable as the consequent. Otherwise, the optimizer may propose deleting that literal from the query if

```
Q1.1: assets(?ship_class,?draft):-
       ship_class(?ship_class,?type,?draft,_,?container),
       ship(_,?ship_class,?type,?status,_,_),
       ?container = "Y",
       ?draft < 50.
Q1.2: assets(?ship_class,?draft):-
       ship_class(?ship_class,?type,?draft,_,?container),
       ship(_,?ship_class,?type,?status,_,?year-built),
       ?status = "Active",
       ?year-built > 1945,
       ?container = "Y",
       ?draft < 50.
Q1.3: assets(?ship_class,?draft):-
       ship_class(?ship_class,?type,?draft,_,?container),
       ?container = "Y",
       ?draft < 50.
```

Table 5: Equivalent queries of Q1 deduced from semantic knowledge

the consequent implies it, or propose refining the range constraint of the literal. Because range information is a special class of range rules whose antecedent is empty, the use of range information in SQO is identical to that of range rules.

Consider the conjunctive query `Q1` in Table 2 and the semantic knowledge given in Table 3. Some of the equivalent queries of `Q1` inferred from the semantic rules are shown in Table 5. The optimizer deletes literal 3 on the variable `?status` based on `R1` and generates `Q1.1`. This is an example of *constraint elimination* [15] reformulation. The optimizer can add `?year-built > 1945` to `Q1` from `R3` and yield another equivalent query `Q1.2`. Adding new literals could be useful in many situations. One of them is when the added literal exploits an indexed attribute.

The optimizer can also *refute* a query, when it infers that the query literals contradict a rule (or a chain of rules) and will not be satisfiable by the data. Sometimes the optimizer can assert the answer directly from semantic rules. In either case, there is no need to access the database to answer the query and we could achieve close to 100 percent savings.

3.2 Reformulation Based on Relational Rules

When the applicable rule is a relational rule, and the consequent (in this case a database literal) implies another database literal in the query, the optimizer may propose deleting that database literal. This is referred to as *join elimination* [15] reformulation. For example, from `R2`, we can infer that literal 2 of `Q1` is implied by literal 1 and thus is redundant in the sense that the value pairs `(?ship_class,?type)` that satisfy literal 2 subsume the pairs that satisfy literal 1. Therefore, there is no need to evaluate literal 2 because the system would not miss any desired value pair in the retrieved answer.

However, in general cases, locating an applicable relational rule is not sufficient to validate the deletion of a database literal because the resulting query may not be semantically equivalent to the input query. The optimizer needs to assure that the resulting query is *safe* as defined in [26, pages 104–105]. Queries are not safe unless all its variables are *limited* in the sense that their values can be finitely determined. Section 3.3 describes necessary and sufficient conditions to delete a database literal.

Relational rules also allow the optimizer to insert a new database literal into the query (i.e., the *join introduction* reformulation [15]). The use of relational rules for this class of reformulation is similar to that of using range rules to insert a new literal. However, this reformulation is seldom found beneficial in cost reduction. If a knowledge base contains recursive rules, the optimizer might infinitely introduce new database literals and never terminate. In this paper, we do not consider introducing new database literals.

3.3 Detecting Unnecessary Joins

Eliminating a redundant database literal implies eliminating equi-joins implicitly specified by the variables that appear in both the target database literal and any other database literal in the query. We call these common variables the *comparand variables* of a database literal. Previously, Sun and Yu [21] identified necessary and sufficient conditions for eliminating a database literal without changing the answer of a query. Their conditions are adequate for optimizers using referential integrity constraints but too strong for optimizers that can use general relational rules because those conditions apply only to database literals with exactly one comparand variable.

We present the revised conditions for eliminating a redundant database literal with one or more comparand variables using relational rules. Given a query, the problem is to determine whether a target database literal can be deleted without changing its answer based on the available relational rules. Formally, suppose the input query is of the form:

$$Q(\bar{X}) \ :- \ E_1(\bar{X_1}), \ldots, E_i(\bar{X_i}), \ldots, E_k(\bar{X_k}), C_e(\bar{Y_e}), C_i(\bar{Y_i}).$$

where $E_i(\bar{X_i})$ is the target database literal, $C_i(\bar{Y_i})$ is a conjunction of related built-in literals defined on the variables that appear only in $\bar{X_i}$, and $C_e(\bar{Y_e})$ is a conjunction of the other built-in literals. We use $\bar{Z} = \bar{X_i} \cap (\bigcup_{j \neq i} \bar{X_j})$ to denote the tuple of the comparand variables in the target literal. There may be one or more comparand variables. We also assume that Q is a safe query.

Let Q' be the query obtained by deleting $E_i(\bar{X_i})$ and the related built-in literals $C_i(\bar{Y_i})$ from Q:

$$Q'(\bar{X}) \ :- \ E_1(\bar{X_1}), \ldots, E_{i-1}(\bar{X_{i-1}}), E_{i+1}(\bar{X_{i+1}}), \ldots, E_k(\bar{X_k}), C_e(\bar{Y_e}).$$

Our problem is to determine under what conditions $Q'(\bar{X})$ and $Q(\bar{X})$ will return the same answer in all database states consistent with the semantic rules. Note that $C_i(\bar{Y_i})$ needs to be eliminated to make the query safe, because after $E_i(\bar{X_i})$ is deleted, the variables in $\bar{Y_i}$ will appear only in the built-in literals and become not limited.

A database literal is considered redundant if and only if it satisfies all three conditions to be specified next. The first condition states that the target database literal is implied by the other literals. That is, the tuples projected on the comparand variables that satisfy the target database literal must subsume the corresponding tuples that satisfy the remaining literals. This condition is satisfied if the optimizer can locate an applicable relational rule that implies $E_i(\bar{X_i})$.

$$\pi_{\bar{Z}}(\sigma_{C_e(\bar{Y_e})}(E_1(\bar{X_1}),\ldots,E_{i-1}(\bar{X_{i-1}}),E_{i+1}(\bar{X_{i+1}}),\ldots,E_k(\bar{X_k}))) \subseteq \pi_{\bar{Z}}(E_i(\bar{X_i})) \tag{1}$$

The second condition states that all related built-in literals must be implied by the other literals and thus can be deleted without changing the answer. This condition can be verified from range rules.

$$\pi_{\bar{Y_i}}(\sigma_{C_e(\bar{Y_e})}(E_1(\bar{X_1}),\ldots,E_k(\underline{a}rX_k))) \subseteq \pi_{\bar{Y_i}}(\sigma_{C_i(\bar{Y_i})}(E_i(\bar{X_i}))) \tag{2}$$

The third condition states that if a variable in the target literal appears in the head of the query, then it must also occur in some other database literals. Otherwise, deleting the target literal may result in variables that are not limited.

$$\forall ?x \in \bar{X_i}\ (?x \notin \bar{X} \vee ?x \in \bigcup_{j\neq i} \bar{X_j}) \tag{3}$$

Proposition 1 *$Q'(\bar{X})$ is semantically equivalent to $Q(\bar{X})$ if and only if $E_i(\bar{X_i})$ satisfies all three conditions (1), (2) and (3).*

Proof "If" $\Longleftarrow$: Let $\bar{X_J} = \bigcup_{j\neq i} \bar{X_j}$. By definition, $\bar{Z} \subseteq \bar{X_J}$. Since Q is safe and thus $\bar{X} \subseteq \bigcup_{allj} \bar{X_j}$, by condition (3), we have $\bar{X} \subseteq \bar{X_J}$.

Next, we show that conditions (1) and (2) guarantee that eliminating $E_i(\bar{X_i})$ and $C_i(\bar{Y_i})$ will not change the answer of Q. Condition (1) implies that

$$\pi_{\bar{Z}}(\sigma_{C_e(\bar{Y_e})}(E_1(\bar{X_1}),\ldots,E_{i-1}(\bar{X_{i-1}}),E_{i+1}(\bar{X_{i+1}}),\ldots,E_k(\bar{X_k}))) = \pi_{\bar{Z}}(\sigma_{C_e(\bar{Y_e})}(E_1(\bar{X_1}),\ldots,E_k(\bar{X_k}))) \tag{4}$$

Let μ be a tuple in the result of the projection onto $\bar{X_J}$ of the expression at the left-hand side of (4) before applying any projection. That is,

$$\mu \in \pi_{\bar{X_J}}(\sigma_{C_e(\bar{Y_e})}(E_1(\bar{X_1}),\ldots,E_{i-1}(\bar{X_{i-1}}),E_{i+1}(\bar{X_{i+1}}),\ldots,E_k(\bar{X_k}))).$$

Since the difference between the two sides of (4) is the join with $E_i(\bar{X_i})$ on $\bar{Z}$, and both projections are applied onto $\bar{Z}$, if $\pi_{\bar{Z}}(\mu)$ does not match any tuple in $\pi_{\bar{Z}}(E_i(\bar{X_i}))$, then $\pi_{\bar{Z}}(\mu)$ will not appear in the result of the expression at the right-hand side of (4). But the expressions at both sides are equivalent and $\pi_{\bar{Z}}(\mu)$ must be in the result of the left-hand side of (4). There must be a tuple in the result of the right-hand side that is equal to $\pi_{\bar{Z}}(\mu)$ and matches a tuple in $\pi_{\bar{Z}}(E_i(\bar{X_i}))$. This leads to a contradiction. We can therefore conclude that

$$\mu \in \pi_{\bar{X_J}}(\sigma_{C_e(\bar{Y_e})}(E_1(\bar{X_1}),\ldots,E_k(\bar{X_k}))).$$

Meanwhile, since the expression at the right-hand side is more restrictive, its projection onto $\bar{X}_J$ must be a subset of the projection onto $\bar{X}_J$ of the left-hand side of (4). As a result, we have

$$\pi_{\bar{X}_J}(\sigma_{C_e(\bar{Y}_e)}(E_1(\bar{X}_1), \ldots, E_{i-1}(\bar{X}_{i-1}), E_{i+1}(\bar{X}_{i+1}), \ldots, E_k(\bar{X}_k))) = \pi_{\bar{X}_J}(\sigma_{C_e(\bar{Y}_e)}(E_1(\bar{X}_1), \ldots, E_k(\bar{X}_k))).$$

Moreover, since $\bar{X} \subseteq \bar{X}_J$, we have

$$\pi_{\bar{X}}(\sigma_{C_e(\bar{Y}_e)}(E_1(\bar{X}_1), \ldots, E_{i-1}(\bar{X}_{i-1}), E_{i+1}(\bar{X}_{i+1}), \ldots, E_k(\bar{X}_k))) = \pi_{\bar{X}}(\sigma_{C_e(\bar{Y}_e)}(E_1(\bar{X}_1), \ldots, E_k(\bar{X}_k))). \tag{5}$$

Similarly, from condition (2), we have

$$\pi_{\bar{Y}_i}(\sigma_{C_e(\bar{Y}_e)}(E_1(\bar{X}_1), \ldots, E_k(\bar{X}_k))) = \pi_{\bar{Y}_i}(\sigma_{C_e(\bar{Y}_e)}(\sigma_{C_i(\bar{Y}_i)}(E_1(\bar{X}_1), \ldots, E_k(\bar{X}_k)))).$$

In a manner similar to the derivation of (5), we can show that

$$\pi_{\bar{X}}(\sigma_{C_e(\bar{Y}_e)}(E_1(\bar{X}_1), \ldots, E_k(\bar{X}_k))) = \pi_{\bar{X}}(\sigma_{C_e(\bar{Y}_e)}(\sigma_{C_i(\bar{Y}_i)}(E_1(\bar{X}_1), \ldots, E_k(\bar{X}_k)))). \tag{6}$$

From (6) and (5), we have $Q'(\bar{X}) \equiv Q(\bar{X})$.

"Only if" $\Longrightarrow$: We first show that if $Q \equiv Q'$ then condition (3) holds. This is the case because otherwise, there will be variables in $\bar{X}$ not limited in Q' while all variables are limited in Q.

Next we show that if $Q \equiv Q'$ then conditions (1) and (2) hold by reducing the problem to the case of a single join. Then we can apply Sun and Yu's Proposition 1 [21, page 141] to establish the case. Table 6 shows the construction of this proof. We construct two views and a new query Q_{new} such that our problem ($\Longrightarrow$ marked by a "?") can be reduced to the case of a single join. By establishing (a) and (b), we can complete the proof.

To concisely define the views that will be used in the proof, we define a predicate $\texttt{composite}(x, x_1, \ldots)$ such that a tuple satisfies this predicate if and only if $x = \langle x_1, \ldots \rangle$, where x has a composite value of the rest of the variables. For instance, $\texttt{composite}(\langle 1,2,3\rangle, 1, 2, 3)$ is true.

The views are defined as follows:

$$Z'(\bar{Z}, z_c) \;\text{:-}\; E_1(\bar{X}_1), \ldots, E_{i-1}(\bar{X}_{i-1}), E_{i+1}(\bar{X}_{i+1}), \ldots, E_k(\bar{X}_k), C_e(\bar{Y}_e), \texttt{composite}(z_c, \bar{Z}).$$

$$E_i'(\bar{X}_i, z_c) \;\text{:-}\; E_i(\bar{X}_i), \texttt{composite}(z_c, \bar{X}_i).$$

In the first view Z', a tuple consists of attribute values from a $\bar{Z}$ tuple that satisfies Q' and an additional composite attribute z_c. Each z_c is the combination of the rest of the attribute values in the tuple. The second view E_i' is a duplicate of E_i except for an additional composite attribute z_c.

Based on these new views, we can construct a new query Q_{new} as below:

$$Q_{new}(\bar{X}) \;\text{:-}\; E_1(\bar{X}_1), \ldots, E_{i-1}(\bar{X}_{i-1}), E_{i+1}(\bar{X}_{i+1}), \ldots, E_k(\bar{X}_k), C_e(\bar{Y}_e), Z'(\bar{Z}, z_c), E_i'(\bar{X}_i', z_c), C_i(\bar{Y}_i').$$

$$
\begin{array}{ccc}
(Q \equiv Q') & \overset{?}{\Longrightarrow} & Q \text{ and } E_i \text{ satisfies conditions (1) and (2)} \\
(a) \Downarrow & & \Uparrow (b) \\
(Q_{new} \equiv Q'_{new}) & \underset{\text{from [21]}}{\Longleftrightarrow} & Q_{new} \text{ and } E'_i \text{ satisfies (1),(2),(3) of Proposition 1 in [21]}
\end{array}
$$

Table 6: Constructions of the proof of Proposition 1.

where $\bar{X}'_i = \bar{X}_i\theta$ and $\bar{Y}'_i = \bar{Y}_i\theta$. θ is some variable substitution such that $\bar{X}'_i \cap \bar{X}_i = \emptyset$. Clearly, for E'_i, there is exactly one single join from E'_i to Z' over z_c in Q_{new}. If this join is unnecessary, we can remove the literals $E'_i(\bar{X}'_i, z_c)$ and $C_i(\bar{Y}'_i)$ to obtain a semantically equivalent query:

$$Q'_{new}(\bar{X}) \ :- \ E_1(\bar{X}_1), \ldots, E_{i-1}(\bar{X}_{i-1}), E_{i+1}(\bar{X}_{i+1}), \ldots, E_k(\bar{X}_k), C_e(\bar{Y}_e), Z'(\bar{Z}, z_c).$$

Now, we are ready to establish (a) and (b) in Table 6. For (a), the difference between Q' and Q'_{new} is the additional literal $Z'(\bar{Z}, z_c)$ in Q'_{new}, whereas for each $\bar{Z}$ tuple that satisfies Q', there is a corresponding tuple in Z'. Therefore, a $\bar{X}$ tuple satisfying Q' must also satisfies Q'_{new}, and vice-versa. Similarly, we can show that a $\bar{X}$ tuple satisfying Q must also satisfy Q_{new}, and vice-versa. This follows that if $Q \equiv Q'$ then $Q_{new} \equiv Q'_{new}$.

For (b), assuming that $Q_{new} \equiv Q'_{new}$, then from Sun and Yu [21], the following three conditions must be satisfied:

1. $\pi_{z_c}(E'_i) \supseteq \pi_{z_c}(Z')$,
2. $C_i(\bar{Y}'_i) = \emptyset$ or $C_i(\bar{Y}'_i)$ is redundant under the set of semantic rules,
3. $\bar{X} \cap (\bar{X}'_i \cup \{z_c\}) \subseteq \{z_c\}$.

The first condition implies that $\pi_{\bar{Z}}(E'_i) \supseteq \pi_{\bar{Z}}(Z')$. Replacing E'_i and Z' with their definitions and removing the predicate `composite` leads to our condition (1). From the second condition, $C_i(\bar{Y}'_i)$ is redundant in Q_{new}. Therefore, the set of E'_i tuples that satisfy the join condition with Z' must be a subset of those E'_i tuples that satisfy $C_i(\bar{Y}'_i)$. From the definitions of E'_i and Z', the set of E_i tuples that satisfy the literals defined on other literals in Q must be a subset of those E_i tuples that satisfy $C_i(\bar{Y}_i)$. This is equivalent to our condition (2). Consequently, (b) is established. □

An example redundant database literal is literal 2 of `Q1` in Table 2. This literal specifies an implicit equi-join from relation `ship` to `ship_class` over two comparand variables — `?ship_class` and `?type`. Literal 2 is redundant because it satisfies all three conditions. Condition (1) is satisfied because from the relational rule `R2`, the value pairs for `?ship_class` and `?type` that satisfy literal 1 must also satisfy literal 2. Since literal 3 is the only built-in literal that involves a variable in literal 2, and literal 3 is implied by other literals, as inferred from `R1`, (2) is also satisfied. For condition (3), though `?ship_class` in literal 2 is used in the query head, this variable also occurs in literal 1 and thus is limited by relation `ship_class`. Therefore, literal 2 satisfies all three conditions and the optimizer can delete literal 2 and the related built-in literal 3 to obtain the reformulated query shown as `Q1.3` in Table 5.

These new conditions are more general than Sun and Yu's [21] three conditions in two respects. First, our conditions can be applied to joins with an arbitrary number of comparand variables, while theirs can be applied only to joins with a single comparand variable. Second, our conditions allow the optimizer to use general relational rules while their conditions are restricted to referential integrity constraints. We note that general relational rules include recursive rules, and our three conditions allow the optimizer to detect unnecessary recursive joins [29, pages 140–143] using recursive rules. As a result, our optimizer can detect more opportunities to delete unnecessary joins for a wider range of queries.

3.4 Using Partially Matched Relational Rules

In general, verifying condition (1) requires the optimizer to locate a chain of one or more applicable rules entailing that a target literal is implied. In some case, a partially matched rule combined with functional dependency knowledge may also allow this inference. Suppose we have a relational rule `R8` in Table 7. This rule is similar to `R2` in Table 3 except that the variable `?type` is replaced by an anonymous variable "_." The antecedent of this rule implies `Q1` but its consequent partially matches literal 2 of `Q1` at the variable `?class` but misses `?type` because of the anonymous variables. We call this rule a *partially matched* rule.

This rule may become fully matched if we bring to bear the functional dependency rules. Suppose we know that the ship classes functionally determines the ship types as shown in Table 7, and that the functions underlying these two dependencies are identical, then the example rule will become applicable to `Q1`. This is because we can rewrite the anonymous variables for ship types as `foo(?class)` for some function `foo`. This entails that the consequent of the example rule implies literal 2 of `Q1`.

This inference enhances the utility of a relational rule, but to avoid increasing optimization overhead, we can compute this inference for all relational rules prior to the query optimization.

4 Local Optimization of Subqueries

The local optimization algorithm in our approach is an improved version of the state-of-the-art SQO algorithms for optimizing a conjunctive query in a stand-alone database environment [14,

```
R8: ship_class(?class,_,_,_,?container) ∧ ?container = "Y"
      ⇒ ship(_,?class,_,_,_,_)

    ship_class.class ⟶ ship_class.type
    ship.class ⟶ ship.type
```

Table 7: Partially matched rule and functional dependencies

15, 16, 17, 18, 19, 20, 21]. In addition to the capability of exploiting general relational rules, the improvement includes new features to facilitate the search and produce useful information for global optimization. This section describes these features and presents the complete local optimization algorithm.

4.1 Implication Closures

An intuitive search algorithm for SQO is to repeat proposing and applying reformulations to the query until the optimal equivalent query is found [15, 17, 16]. However, such a generate-and-test algorithm might miss applicable rules and hence miss optimization opportunities because an applicable rule to a query may become inapplicable if some literals are deleted from the query. To address this problem, when the optimizer detects a redundant literal, instead of deleting the literal immediately, it should retain the literal and delay the deletion until all applicable rules have been located. This can be achieved by computing an *implication closure* [21, 30, 18, 19] of semantic knowledge to propagate the results of rule applications. An implication closure contains all redundant literals in an input query that are implied by other literals and all of the new built-in literals derived from semantic rules.

For example, the implication closure of `Q1` given `R1`, `R2` and `R3` is indicated by the literals underlined in `Q1.2` (see Table 5). `R3` may not be considered applicable if the optimizer commits the deletion when it matches `R1` to generate `Q1.1`. Implication closures allow the optimizer to consider all possible literal deletions and insertions implied by the semantic rules. This way, the optimizer will not miss any optimization opportunities.

Another advantage of implication closures is that since an implication closure contains all literals implied by other query literals given the semantic knowledge, we can extract the most restrictive ranges of variables from the implication closure and use it to reduce the data transmission cost in the global optimization.

4.2 Detecting Circular Implications

After proposing reformulations based on semantic knowledge in the first phase, the second phase of SQO is to search for a subset of literals in the implication closure that is provides the greatest cost

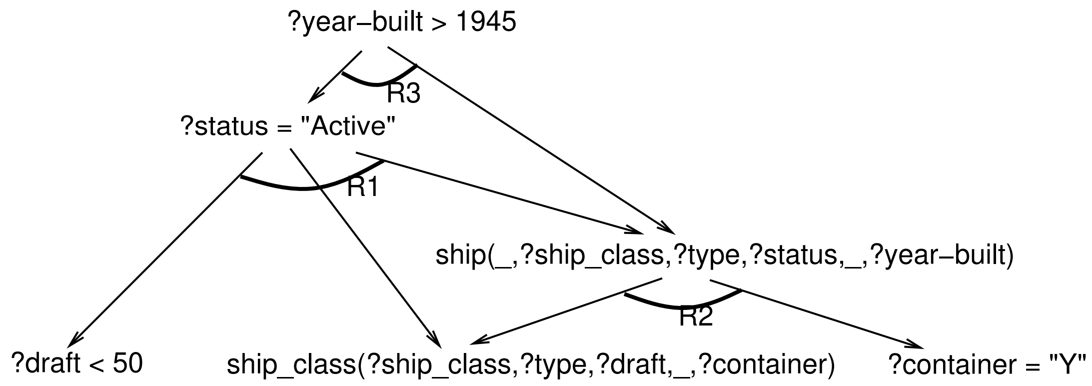


Figure 3: AND-OR implication graph

reduction. Literals not in the implication closure cannot be deleted because they are not implied by other literals, but not all literals in the implication closure can be deleted. For database literals, the optimizer must also verify if they satisfy the three conditions of redundancy. Other than that, literals in an implication closure might imply each other (based on the semantic knowledge) and form a cycle. If this is the case, and the optimizer has committed deleting all literals but one in the cycle, then the remaining literal might not be implied by any literal in the resulting query and thus cannot be deleted. To deal with this problem, we introduce a data structure called *AND-OR implication graphs* to keep track of the implication of literals after deletions.

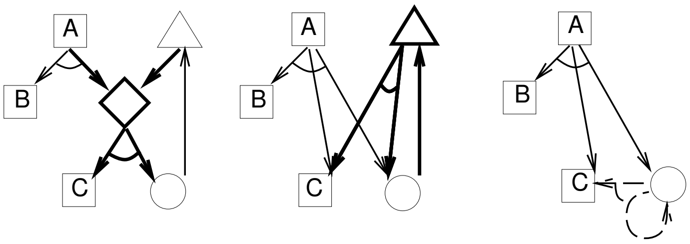
An AND-OR implication graph (V, A) for a given query and semantic knowledge is a pair of vertices V and AND-arcs A. V is the set of the literals in the query including its implication closure. An AND-arc $(u, \{v_1, \ldots, v_k\})$ that points out of u is in A if and only if there exists a semantic rule in the knowledge base entailing that $v_1, \ldots, v_k$ imply u. It follows that a literal u is in the implication closure if the out-degree of u is greater than zero. Figure 3 shows the AND-OR implication graph of `Q1` given the semantic rules `R1`,`R2`, and `R3`.

Algorithm 1 updates an AND-OR implication graph when a literal u is eliminated. This algorithm maintains the implication chains of literals by computing the inheritance of AND-arcs if the inheritance does not lead to a reflexive AND-arc, or returns fail if the out-degree of u is zero. Figure 4 illustrates how Algorithm 1 works. Figure 4(I) shows the initial graph of a query where the diamond is about to be eliminated. The diamond implies the triangle and jointly implies box A with box B. The triangle and box A inherit the AND-arc of the diamond as shown in Figure 4(II). Suppose next the optimizer decides to delete the triangle which is implied by the circle. In this case, no inheritance will be constructed because otherwise there will be a reflexive AND-arc to the circle, as indicated in dash lines in Figure 4(III).

The condition at line 7 excludes the construction of reflexive AND-arcs so that the optimizer can always assure that in an AND-OR implication graph, if the out-degree of a literal is greater

Algorithm 1 (Updating AND-OR implication graph)

1 INPUT $G = (V, A)$ an AND-OR implication graph; $u \in V$ a target literal to be eliminated;
2   IF no (u, S) is found in A THEN u is not implied RETURN FAIL;
3   FOR all $(u, S) \in A$
4   LET $A = A - \{(u, S)\}$;
5   FOR all $(w, U) \in A$ such that $u \in U$
6   LET $A = A - \{(w, U)\}$;
7   IF $w \notin S$ THEN LET $A = A \cup \{(w, (S \cup U - \{u\}))\}$;
8   LET $V = V - \{u\}$; RETURN G;



(I) (II) (III)

Figure 4: Updating AND-OR implication graph after literal elimination

than zero, it is implied by the other literals in the current reformulated query. This is useful because the optimizer can efficiently determine whether a literal is implied without the need to repeatedly match the knowledge base of semantic rules.

The number of iterations at lines 3 and 5 is bounded by the maximal number of AND-arcs of a literal, and this number is bounded by the number of applicable rules. Hence the time complexity of Algorithm 1 is polynomial with regard to the number of applicable rules. The number of AND-arcs usually shrinks after consecutive invocations but may grow moderately in some cases. In our experiments, we have not encountered any performance problem due to the size of AND-OR implication graphs.

AND-OR implication graphs allow an optimizer to separate rule matching and optimization search so that there is no need to repeatedly match applicable semantic knowledge. The optimization is therefore more efficient than previous work [18, 19, 21]. This data structure also allows the optimizer to abort the optimization and return an executable correct query at any time if necessary. This feature is important in a dynamic heterogeneous database environment.

4.3 The Local Optimization Algorithm

Algorithm 2 lists the top level steps of our local optimization algorithm of conjunctive subqueries. The algorithm computes the implication closure and builds the AND-OR implication graph, and then searches for an optimal subset of literals in the implication closure to delete from the query.

Algorithm 2 (Local optimization)

```
1 INPUT Q = conjunctive query; KB = semantic knowledge;
2   LET I = implication closure; G = AND-OR implication graph (both initially empty);
3   derive the most restrictive ranges of the variables in Q using the range facts in KB;
4   update I and Q with the derived range constraints;
5   IF a literal in Q contradicts a range fact THEN RETURN NULL;
6   FOR all applicable rules in KB
7     LET A → B be the applicable rule after variable substitution;
8     IF Q refuted THEN RETURN NULL;
9     ELSE add B to I; add AND-arc (B,A) to G; LET Q = Q ∪ {B};
10  search for a subset D of literals in I, so that Q' = Q − D is optimal;
11  RETURN Q' and I;
```

```
Q1.4: assets(?ship_class,?draft):-
        ship_class(?ship_class,?type,?draft,?length,?container),
        ship(_,?ship_class,?type,?status,_,?year-built),
        ?length >= 580,
        ?length <= 950,
        ?status = "Active",
        ?year-built > 1945,
        ?draft >= 12,
        ?container = "Y",
        ?draft < 50.
```

Table 8: Equivalent query with implication closures

We illustrate how this algorithm works using the semantic knowledge in Table 3 and the example query `Q1` in Table 2 as input. Initially, the optimizer uses range facts and the literals in the query to derive the most restrictive ranges of the variables in the query (lines 3 to 5). For example, from the range fact `F1` and the literal `?draft < 50`, the optimizer derives that the most restrictive range of the variable `?draft` is the interval $[12, 50)$ and inserts a new literal `?draft >= 12` to the query. Similarly, the optimizer derives ranges of other variables and inserts two new literals on `?length` to the query. The optimizer also saves the derived literals in the implication closure I.

Next, from the semantic rules, the optimizer derives new literals as well as redundant literals and saves the results in the implication closure (lines 6 to 9). In this example, the optimizer matches applicable rules `R1`, `R2` and `R3` in Table 3, and updates the implication closure I and reformulates the query as described in the previous section. As the optimizer matches rules, it builds a corresponding AND-arc in the AND-OR implication graph. The resulting query is given as `Q1.4` in Table 8, where the underlined literals are those derived from the semantic knowledge and saved in the implication closure.

When the optimizer applies semantic knowledge, it also checks whether there exists any literal that is not satisfiable. If this is the case, the optimizer can conclude that the entire query is not satisfiable and return `NULL` as the answer of the query without accessing the database at all (line 5 and 8).

The optimizer then searches for a subset of the literals in the implication closure to delete from the partially reformulated query (line 10). The search starts from the reformulated query and proceeds to assess literals for deleting from the query one at a time until a more efficient query is obtained.

The search procedure is outlined as follows. The optimizer selects a target literal with a none-zero out-degree in the AND-OR implication graph G. This literal is implied by the other literals and thus can potentially be deleted from the query. If this literal is a built-in literal, then the optimizer proceeds to delete it from the query, and updates G using Algorithm 1. If this literal is a database literal, then the optimizer checks whether it satisfies the three conditions described in Section 3.3. Since the target literal is implied, we can infer that condition (1) is verified. To check condition (2), the optimizer makes a copy of G and iteratively invokes Algorithm 1 to simulate deleting the related built-in literals of the target literal. The optimizer then checks if the invocations complete successfully. If this is the case, the built-in literals do not form a circular implication and thus are implied as a conjunction. The optimizer then checks the variables in the target literal to verify condition (3). If all three conditions are satisfied, the optimizer commits deleting this database literal and its related built-in literals. Otherwise, the target literal will be retained. The search continues as the optimizer selects the next literal.

Assuming that literals are equally expensive, then the optimal query is the one with a minimal number of literals. This search problem is equivalent to the minimum axiom set problem, which is shown to be $\mathcal{NP}$-complete [31, pages 263]. Searching for equivalent queries with a minimal number of joins is also $\mathcal{NP}$-complete [21]. The search problem would be at least equally hard for more complex cost model. To constrain the optimization cost, we use a greedy search procedure which applies an evaluation heuristic to guide the deletion of literals. One source of the heuristics is the information on the execution cost for various query operations from the technical manuals of DBMS. In our implementation, we use a set of heuristics derived from Table 13-1 of [32] for ORACLE$^{\text{TM}}$ databases.

Returning to our example, since there is no circular implication, the optimizer can commit deleting all of the implied literals and optimize the input query into `Q1.3` in Table 5. The algorithm also returns the implication closure that contains the most restrictive ranges of the variables in the query. The global optimization algorithm relies on this information to reduce data transmission cost, as described in the next section.

Algorithm 3 (Global optimization)

```
 1 INPUT P = query plan; KB_d = semantic knowledge learned from databases;
 2   LET KB = KB_d , S_b = an empty stack;
 3   LET S_f = a stack of subqueries in the data flow order specified in P
 4   WHILE S_f is not empty DO
 5     LET s = pop(S_f);
 6     optimize s by calling Algorithm 2 with input s and KB;
 7     update KB with newly inferred more restrictive ranges of variables;
 8     push optimized s into S_b;
 9   WHILE S_b is not empty DO
10     LET s = pop(S_b);
11     IF s is a subquery that combines or manipulates intermediate data THEN
12       determine required variables V;
13       extract and move newly inferred literals to L;
14     ELSE (s is a subquery that retrieves data from a remote database)
15       remove variables not in V from the parameter list of s;
16       insert literals in L to s if they are defined on variables in s;
17       IF s is changed THEN
18       optimize s by calling Algorithm 2 with input s and KB_d;
19     push s onto S_f;
20   update P with S_f;  RETURN P;
```

5 Global Optimization of Query Plans

The global optimization algorithm reduces irrelevant data retrieval and transmission in query plan execution. A query plan may have conjunctive subqueries as well as comparisons, set operations, and disjunctions. Algorithm 3 gives the top level algorithm for the global optimization. The algorithm takes a query plan and semantic knowledge about relevant databases as input and traverses the input query plan twice in their data flow order. The optimizer maintains two stacks of subqueries for the traversal. In the first traversal, the algorithm optimizes each subquery and propagates inferred range information forward in the data flow order. The second traversal propagates backward the insertions and deletions of query literals made by the optimizer in the first traversal to perform global optimization. We explain the algorithm for the example query plan in Figure 1 and the semantic knowledge shown in Table 3 and Table 4 as input.

5.1 Forward Propagation

In the first traversal, that is, the loop from line 4 to line 8, the plan graph is traversed forward in the data flow order. During the traversal, each conjunctive subquery is optimized by calling Algorithm 2, the local optimization algorithm, which also infers the most restrictive ranges of the variables in the head of the subquery from the implication closure. The inferred ranges will be propagated forward for the optimization of the succeeding subqueries (line 7).

This step is illustrated in Figure 5. Subquery 1 is optimized but no reformulation is found to

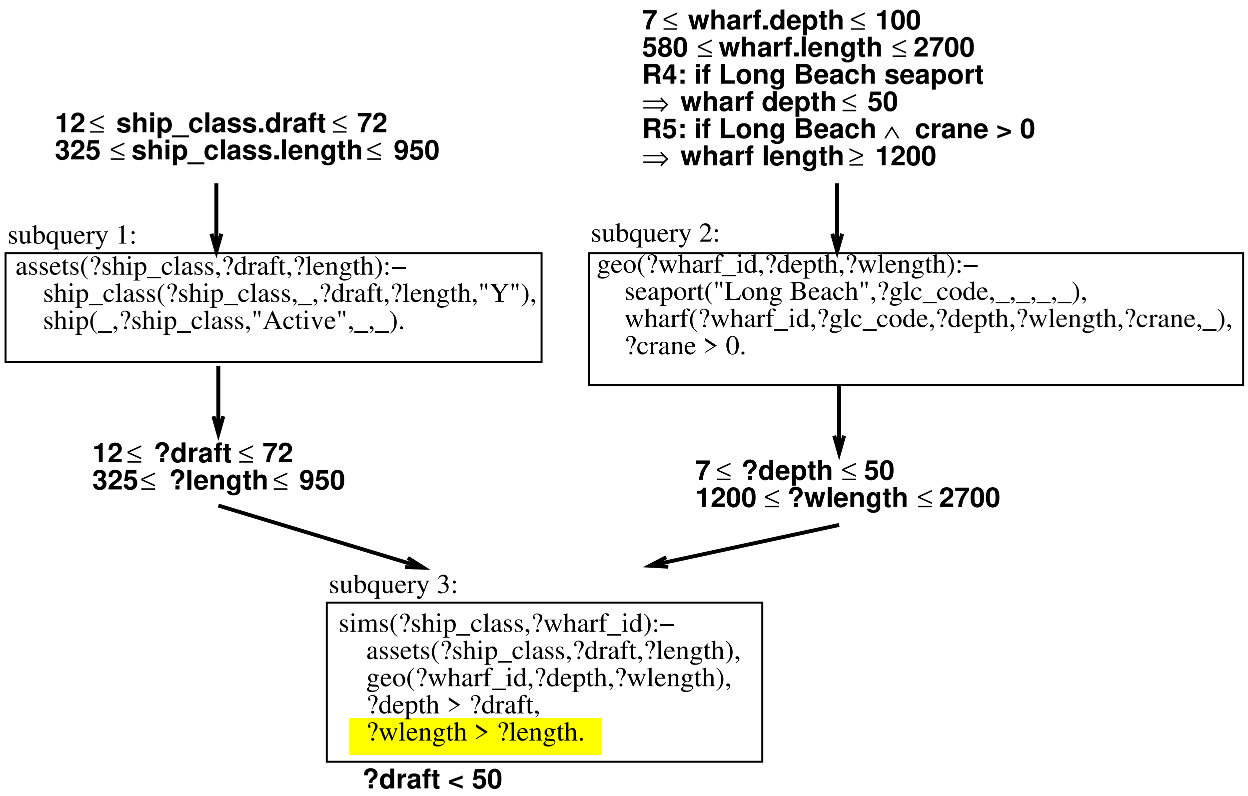


Figure 5: Propagating inferred range information forward to optimize subqueries

be appropriate. However, based on the range facts, the ranges of variables `?draft` and `?length` are inferred and propagated to subquery 3. Similarly, the optimizer does not reformulate subquery 2, but it uses semantic knowledge and the constraints specified in the subquery to infer the most restrictive ranges of `?depth` and `?wlength`, and propagates the results to optimize subquery 3.

For subqueries with special literals that Algorithm 2 cannot optimize, such as comparisons between two variables (e.g., `?depth > ?draft`), set operators (e.g., `intersection` and `union`), and other data manipulation operators, the optimizer applies a set of axioms to optimize these operators. Table 9 gives the axiom for the operator $>$. Given the ranges of the variables occurring in a special literal, the optimizer will propose one of the following four reformulations using the axioms:

- **deleting the literal** when the literal is found redundant,
- **adding new built-in literals** when more restrictive range is inferred,
- **refuting the literal** when the given ranges show that the literal is unsatisfiable,
- **no action**.

In our example, when optimizing subquery 3, the optimizer infers that `?wlength` is always greater than `?length` because the minimal value of `?wlength` is greater than the maximal value of `?length`.

Given $L = ?x_1 > ?x_2$ in a query:

1 IF $min(?x_1) > max(?x_2)$ THEN delete L.
2 IF $(max(?x_1) = max(?x_2)) \wedge (max(?x_1) > min(?x_2) > min(?x_1))$
THEN insert $?x_1 > min(?x_2)$.
3 IF $(max(?x_2) > max(?x_1) > min(?x_1)) \wedge (min(?x_1) = min(?x_2))$
THEN insert $?x_2 < max(?x_1)$.
4 IF $max(?x_2) > max(?x_1) > min(?x_2) > min(?x_1)$
THEN insert $?x_1 > min(?x_2)$ and insert $?x_2 < max(?x_1)$.
5 IF $max(?x_2) > min(?x_2) > max(?x_1) > min(?x_1)$
THEN refute L.
6 IF $max(?x_2) > max(?x_1) > min(?x_1) > min(?x_2)$
THEN insert $?x_2 < max(?x_1)$.
7 IF $max(?x_1) > max(?x_2) > min(?x_2) > min(?x_1)$
THEN insert $?x_1 > min(?x_2)$.
8 IF $(max(?x_2) > min(?x_2) > min(?x_1)) \wedge (min(?x_2) = max(?x_1))$
THEN refute L.
9 IF $(max(?x_1) = min(?x_1) = max(?x_2)) \wedge (min(?x_1) > min(?x_2))$
THEN insert $?x_2 < max(?x_1)$.
10 IF $(max(?x_1) = max(?x_2) = min(?x_2)) \wedge (max(?x_1) > min(?x_1))$
THEN refute L.
11 IF $(max(?x_1) = min(?x_1) = min(?x_2)) \wedge (max(?x_2) > min(?x_2))$
THEN refute L.
12 IF $(max(?x_2) > min(?x_2) > min(?x_1)) \wedge (max(?x_1) > min(?x_1))$
THEN insert $?x_1 > max(?x_2)$.
13 IF $(max(?x_1) > min(?x_1)) \wedge (max(?x_2) > max(?x_1) > min(?x_2))$
THEN insert $?x_2 < max(?x_1)$.
14 IF $(max(?x_2) > min(?x_2)) \wedge (max(?x_1) > max(?x_2) > min(?x_1))$
THEN insert $?x_1 > max(?x_2)$
15 IF $max(?x_1) = max(?x_2) = min(?x_1) = min(?x_2)$ THEN refute L.
16 OTHERWISE no action.

Table 9: Axioms for $?x_1 > ?x_2$

Therefore, the literal `?wlength > ?length` is redundant and can be deleted (from Axiom 1 in Table 9). Meanwhile, for the literal `?depth > ?draft`, the optimizer inferred a new literal `?draft < 50`, because the maximal value of `?depth` is 50. This inferred literal is inserted into the subquery (from Axiom 13).

5.2 Backward Propagation

After the optimizer finishes optimizing a subquery and inferring ranges of variables, it pushes the subquery into another stack S_b for the second query plan traversal (line 9 to line 19). The optimizer pops out subqueries from S_b during the traversal, which corresponds to a backward traversal in the data flow order of the query plan. Each subquery is processed differently depending on whether it is to retrieve data from a remote database site or to process intermediate data. This step is illustrated in Figure 6.

The first case (line 11 to 13) is when the subquery is to combine or process intermediate data (e.g., subquery 3). The optimizer examines the optimized subquery and determines the required

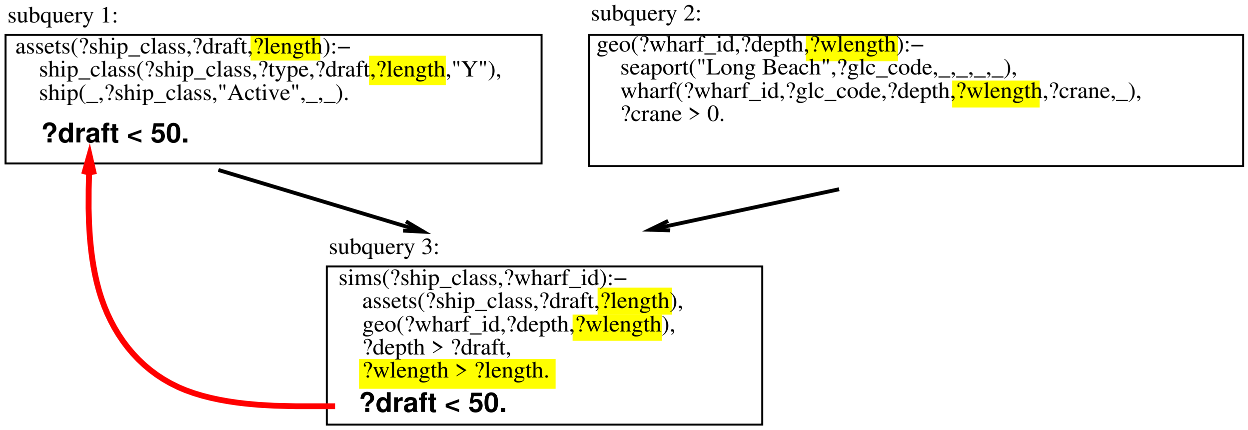


Figure 6: Propagating newly derived literals backward to optimize query plan

variables (line 12). Since some literals were deleted by the optimizer during the forward traversal, there is no need to retrieve or compute the values of the variables occurring in those literals. The optimizer can propagate this information to optimize the preceding subqueries that retrieve data values of those variables. In our example, the optimizer takes optimized subquery 3 and determines that in this subquery, all the variables are required except `?wlength` and `?length`, because the literal `?wlength > ?length` has been deleted from subquery 3 and neither variable is used for output or in any other constraints. The optimizer saves the required variables in V and removes the references to the two unnecessary variables from the database literals in subquery 3.

Next, the optimizer extracts newly inserted built-in literals from the optimized subquery if the values of their variables are retrieved in one of preceding subqueries. If a built-in literal satisfies this condition, then the optimizer will move the literal from the subquery to a set L temporarily so that later in the traversal, the optimizer can insert it into an appropriate preceding subquery. This allows the system to evaluate the literal as early as possible to reduce intermediate data. In subquery 3 there is a newly inserted built-in literal `?draft < 50`, which involves a variable initially defined in subquery 1. The optimizer thus moves this literal to L for further processing.

The second case (line 14 to 18) is when the subquery is to retrieve data from a remote database. The optimizer first removes any variable in the head not in the set V of the required variables, and then inserts literals collected in L previously into the subquery if the literals involve variables generated in this subquery. In our example, when the optimizer encounters subquery 2 — a subquery that retrieves data from a remote database, the optimizer finds that the variable `?wlength` is not a required variable and removes it from the head, as well as the reference to this variable in the database literal on `wharf`. The optimizer continues its traversal and encounters subquery 1. Similarly, it finds that `?length` in this subquery is not required and removes the variable from the subquery. The optimizer also finds that in the set L, there is a literal `?draft < 50` involving the variable `?draft` that is defined initially in subquery 1. Therefore, the optimizer moves the literal to this subquery and completes the traversal.

If a subquery is modified in the previous steps (line 15 and 16), the optimizer invokes the basic semantic query optimization algorithm to optimize this subquery again to see if there are additional optimization opportunities. Since both subquery 1 and 2 are modified, the optimizer will invoke Algorithm 2 to optimize both of them. At this point, subquery 1 is the same as the example query `Q1` in Table 2. The optimization for subquery 1 is also similar to that for `Q1` as we have discussed in the previous section. The resulting plan is the one already shown in Figure 2 in Section 1.3.

5.3 Analysis of the Query Plan Optimization

The rationale of the forward-propagation is to use literals specified in subqueries to specialize the semantic knowledge as much as possible for more effective optimization on succeeding subqueries. The backward-propagation step attempts to detect unnecessary data retrieval and move newly derived literals to subqueries that retrieves data from remote databases so that the literals can be evaluated as early as possible in a query plan and reduce intermediate data.

Since the loop for each traversal repeats in time proportional to the number of subqueries, and in each repetition, the time required for the basic SQO is the dominant factor, the time complexity of Algorithm 3 is $O(2 \cdot n \cdot T_{\text{local}})$, where n is the number of subqueries in the input query plan, and T_{local} is the maximal time required to perform the local optimization for each subquery.

The correctness of Algorithm 3 can be verified as follows. In each step, the inferred ranges of variables propagated forward are always larger than the ranges of variables in the answers; and therefore, no data will be lost due to the modifications to subqueries. Since the inferred ranges of variables are always at least as restrictive as the literals specified in the query, no undesired data will be retrieved due to the modifications to subqueries. We can also establish that moving literals backward will not change the semantics of a query plan. The proof is similar to the proof for the correctness of the predicate push-down techniques [26, 33]. Consequently, the resulting query plan of Algorithm 3 will return the same answer as an input query plan, as long as the given database state is consistent with the semantic knowledge.

6 Experimental Results

This section describes the empirical evaluation on our optimization approach. For this purpose, the optimization approach was implemented in a query optimization system and incorporated with an information mediator that integrates heterogeneous databases. The evaluation consists of two experiments. The first experiment evaluates the effectiveness of the optimization approach. The second experiment is to demonstrate the utility of relational rules in semantic query optimization.

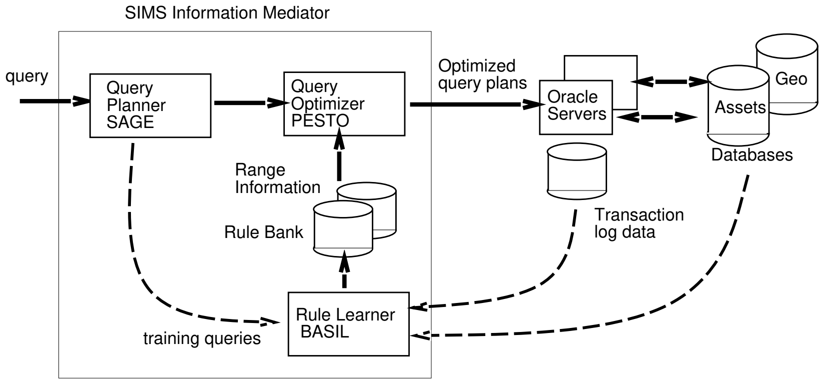


Figure 7: Complete organization of SIMS with BASIL and PESTO

6.1 Setup of the Experiments

The query plan optimizer PESTO[2] is an implementation of our optimization approach. PESTO uses semantic rules learned by the rule induction system BASIL[3] [25] to optimize query plans for an information mediator. These systems were developed to empirically evaluate the approaches developed in this research. They are incorporated into the SIMS information mediator [2, 3, 10]. SIMS invokes PESTO to optimize query plans, and PESTO in turn invokes BASIL to discover the required semantic rules. Figure 7 shows the organization of SIMS with the query plan optimizer and the learner.

SIMS takes as input a query expressed in the LOOM knowledge representation language [27], which is also used as the representation language to build the domain model and the source model of databases. To optimize queries for SIMS, PESTO has a component to translate a LOOM subquery into an internal representation similar to Datalog to facilitate optimization, and a component to translate the result back to LOOM. The semantic rules are expressed in the same internal representation. By attaching a different translation component, PESTO can optimize queries in other query languages.

For the purpose of our experiments, SIMS was connected with two remote ORACLE™ relational databases via the Internet. These databases originally were part of a real-world transportation logistic planning application. Table 10 summarizes the contents and the sizes of these databases. Together with the databases, there were 29 sample queries written by the users of the databases. We also had 3 queries written for the purpose to test different functionalities of the SIMS query planner, and 4 queries to test PESTO, especially to test its ability to detect null queries (i.e., queries that return an empty set). That is a total of 36 queries. Among these 36 queries, 18 are multidatabase queries that require access to multiple databases to retrieve the answer. Table 11 lists

[2]Plan Enhancement by SemanTic Optimization.
[3]BAyesian Speedup Inductive Learning.

Databases	Contents	Relations	Tuples	Size(MB)	Server
Geo	Geographical locations	15	56124	10.48	HP9000s
Assets	Air and sea assets	16	4881	0.51	Sun SPARC 4

Table 10: Sample databases in a transportation logistic planning domain

ID	Short description	Number of Subqueries	Query Length	Size of Answer
205	T-countries' airports, by name, where C-5 aircraft can land at wartime	3	22	7
206	T-countries' airports, by name, where fully loaded DC-8-61s can takeoff at wartime	3	22	1
213	wharves with container cranes at Long Beach, by pier name and berth ID, where slow Container/Breakbulk ships can dock	3	51	108
214	wharves at Long Beach with RORO ramps, by pier name and berth ID where METEOR ships can dock	3	43	54
215	For each ship class-subclass with LASH cargo capacity, list all wharves at Long Beach, by pier name and berth ID, where such class of ship can dock	3	51	864
216	ships which can handle RORO cargo and can dock in T-country	3	30	3
217	ship classes, by ship class name, seaport name, and berth-type name which can handle container and can dock at S-port or T-port	3	25	272
218	ship classes, by ship class name, seaport name, and berth-type name which can handle container and can dock at S-port or T-port	3	27	360
226	low-altitude airports where a C5 can land	3	21	4
227	airports in T-country where a C5 can land	3	19	9

Table 11: Multidatabase queries used in the experiments

some properties of the multidatabase queries. To train the learner, BASIL, 23 queries were selected to serve as the training queries. The selection is based on the similarity of queries. Because we found that BASIL learns nearly identical sets of rules using similar queries, to save experimentation time, we removed some similar queries from the training set. PESTO used about 110 semantic rules and 271 range facts compiled from the databases for the optimization. SIMS, PESTO and BASIL were running on a Sun SPARC-20 workstation during the experiments.

6.2 Performance Report

This experiment evaluates whether PESTO can produce significant cost reduction using machine learned semantic rules. This experiment applied a *k-fold cross validation* [34] so that each training query may have a chance to be used for training as well as testing. In this procedure, queries are divided into k equal-sized subsets. In each iteration of learning and testing, one subset is chosen

	Test 1	2	3	4	Average savings	hand-coded rules
All	28.99%	31.60%	33.94%	29.86%	31.07% s=2.20%	25.84%
Multidb	39.43%	42.51%	42.61%	39.63%	41.05% s=1.75%	36.19%
# of Rules	101	119	106	118	111 s=91	112
opt time (s)	0.038	0.047	0.041	0.054	0.045 s=0.007	0.044

Table 12: Percentage savings performance of learned rules and hand-coded rules

as the test set and the union of the other $k-1$ subsets serves as the training set. In this case, the 23 training queries were randomly divided into four sets, three of them contain 6 queries, and one contains 5 queries. For each set of queries, BASIL took the remaining three sets of queries as training queries and learned a set of semantic rules. The selected set of queries was combined with the 13 additional queries to form the test set of queries. Next, SIMS took the test set as input and invoked PESTO to optimize the queries using the learned semantic rules. After collecting performance data, the learned rules were discarded and the process repeated for the next set of queries. The experiment thus generated four sets of performance data.

Prior to this experiment, we hand-crafted a set of 112 semantic rules for the purpose of comparison. These rules were carefully designed after several iterations of debugging and modifications to allow the optimizer to explore as many optimization opportunities as possible for the sample queries.

The performance data contains the total elapsed time of each query execution, which includes the time for database accesses, network latency, as well as the overhead for semantic query optimization. To reduce inaccuracy due to the random latency time in network transmission, all elapsed time data were obtained by executing each query 10 times and then computing their medians. Then for each query, the percentage time savings were obtained by computing the ratio of the total time saved due to the optimization over the total execution time without optimization.

Table 12 shows the average of the savings for all queries, the average of savings for multidatabase queries and the standard deviations. The data shows that PESTO can produce a significant savings on the test queries, with a ten percent higher savings for multidatabase queries. The data also shows that the learned rules outperform hand-coded rules in all four tests. This result is significant given that the standard deviations are low, and because all tests use about the same number of rules and optimization time.

Table 13 shows the detail performance data of the selected best, median and the worst cases. For some expensive multidatabase queries, the savings can reach as high as 70 to 90 percent. Some

	Query ID	exec. time w/o opt (s)	exec. time w/ opt. (s)	% savings	opt. time (s)
	304	0.74±0.00	0.02±0.00	97.0%	0.02±0.00
Best	215	8.53±0.07	2.46±0.06	71.0%	0.09±0.04
	308	1.00±0.09	0.58±0.02	44.0%	0.03±0.02
	2	1.04±0.01	0.71±0.05	32.0%	0.04±0.04
Medium	201	0.87±0.02	0.77±0.03	12.0%	0.03±0.00
	32	0.66±0.03	0.61±0.04	8.0%	0.03±0.04
	200	0.67±0.02	0.69±0.02	-3.0%	0.03±0.02
Worst	233	0.67±0.02	0.77±0.03	-15.0%	0.04±0.00
	216	1.15±0.04	1.49±0.06	-30.0%	0.05±0.00

Table 13: Detail performance data in the best, medium and worst cases

	exec. time w/o opt. (s)	exec. time w/ opt. (s)	opt. time (s)	Avg Savings	# of rules used
range rule	1.34	1.18	0.03	13.22%	49
relational	1.34	0.95	0.03	26.28%	52

Table 14: Performance data for range rules and relational rules

of our test queries are already very cost-effective, and there is not much room for optimization for those queries, The worst case where we have -30 percent savings is because the cost of data transmission cut down by the new literal inserted by PESTO is not enough to compensate the cost to evaluate that new literal. This shows the importance to include data distribution information in the cost model. We also found that ORACLE$^{\mathrm{TM}}$ contributes to some negative savings because it fails to optimize some sequence of joins in some cases.

6.3 Utility of Relational Rules and Range Rules

One of the important features of PESTO and BASIL is their capability to learn and use relational rules. The second experiment compared the utility of relational rules and range rules. This experiment aimed to verify our claim that in general, relational rules are more widely applicable and produce higher savings. This section describes an empirical comparison between relational rules and range rules to verify our assumption.

Table 14 shows the average performance data and the standard deviations. The average savings data were obtained using a k-fold cross-validation as described in Section 6.2, except that before PESTO optimized the test queries, a filter was used to remove range rules or relational rules from the rule bank. The data shows that using only relational rules yields about twice as much savings as using only range rules.

7 Related Work

The query plan optimization approach described in this paper is elaborated from our prototype approach described previously in [30]. This section compares our approach with related work in intelligent information mediators, the most closely related work on traditional semantic query optimization, predicate move-around, and semi-joins.

7.1 Query Optimization for Information Mediators

A promising approach to integrating heterogeneous databases is through the use of *information mediators*(or *brokers*) [1, 2, 3, 4, 5, 6]. Clients of information mediators can access heterogeneous databases without knowing their implementation details such as their locations, query languages, platforms, etc. A variety of work on query optimization for information mediators has been developed for optimizing different aspects of query plans. Levy et al. [9, 11] provided an approach to pruning irrelevant and redundant source selection based on view integration. Kwok and Weld [35] generalized the work of Levy et al. to support binding patterns in the source selection process. In contrast to their work, our approach covers more aspects of query plans. Our global and local algorithms can prune redundant and irrelevant source (by asserting that a subquery will return empty), reduce unnecessary data transmission from relevant sources, as well as provide local optimization to subqueries.

Another line of research is interleaving query planning and execution. The idea is to allow the query planner to gather useful run-time data to guide the planning process. There are two important uses of run-time data. First, run-time data can be used to retrieve information from one source and that information is then used to formulate subqueries to other sources. Second, run-time data can be used to retrieve information which is then used in the selection of the most appropriate information sources. Levy et al. [9] presented a specialized algorithm for the first accompanying their source selection algorithm. Knoblock and Levy [12] described the second use.

An issue that arises in the use of run-time data is that until desired information is available, the planning may have to be postponed or a plan with all possible contingencies will have to be produced in order to deal with the possible returned values. Knoblock [13] has developed a flexible planning algorithm called SAGE that supports parallel planning and execution to address this issue.

Semantic knowledge is analogous to run-time data because it allows the optimizer to infer useful information to reformulate query plans. Since semantic knowledge is prepared prior to the run-time, inference results of semantic knowledge can be propagated forward and backward along the data flow order, while run-time data can only be propagated forward. However, since it is difficult to store semantic knowledge that covers all possible queries, run-time data are useful to compensate for missing semantic knowledge. In addition, interleaving planning and execution also supports replanning in case of a source failure, which is crucial in dynamic environments where information sources may become temporarily unavailable. An interesting direction of research on information mediators is to interleave planning, execution and semantic query optimization.

Recently, Stonebraker et al. proposed a new architecture called *Mariposa* [6] which supports an economic paradigm for query and storage optimization. The idea is to use market forces to achieve high performance while maintaining the autonomy of each database site. In the architecture described in [6], like most of the information mediators, there is a query planner that generates a query plan for an input query, but the query plan will not be executed directly. Instead, a *broker* is used to send out subqueries in the plan as bids to each remote database server, which contains a *bidder* to respond to requests for bids. The broker then decides which remote servers win the bids and assigns them to execute the subqueries. In other words, their system optimizes a query plan by optimizing the selection of servers rather than performing global and local optimization for query plans as in our approach. It would be interesting to investigate whether the market economic paradigm can be applied to perform other types of optimization.

7.2 Semantic Query Optimization of Conjunctive Queries

Traditional semantic query optimization has focused mainly on optimizing conjunctive subqueries in a stand-alone database environment. Our local optimization algorithm is also aimed at this problem. The most closely related work of our algorithm is the SQO algorithm developed by Yu and Sun [19, 21]. Yu and Sun's algorithm makes two critical improvements over the previous work in SQO. The first improvement is the introduction of necessary and sufficient conditions to delete a single join. The second is that their algorithm can match all applicable rules in optimization by computing a closure of restrictions.

Their algorithm starts by generating a *restriction closure* of all implied built-in literals. Meanwhile it also generates a *core query* that consists of the database literals that cannot be deleted. Then their optimizer selects the implied database literals to insert back to the core query until the resulting query is semantically equivalent to the input query.

The restriction closures are similar to our implication closures, but they do not include database literals that are implied. To test the equivalence of a partially optimized query, their algorithm needs to repeatedly match applicable rules to compute the restriction closures. This could be expensive if we have a large number of semantic rules. The use of the AND-OR implication graphs in our algorithm avoids this unnecessary overhead.

The main weakness of Yu and Sun's algorithm is that it does not allow variables in their queries and semantic rules. As a result, their algorithm works only for *tree queries* [21], where joins form a tree graph. Moreover, all equi-joins of two relations are required to be on a pair of single attributes. The same weakness precludes their algorithm from using general relational rules to detect unnecessary joins. Since our local optimization algorithm can optimize conjunctive queries of any join topology using general relational rules, it can detect more unnecessary joins and achieve higher savings.

7.3 Predicate Move-Around

Optimizing a query plan by moving literals has been previously studied in *predicate push-down* [26], *performing group-by before join* [36], and *predicate move-around* [33].

Predicate push-down is a commonly used query optimization technique. By pushing data selection predicates down the hierarchical access graph of a query, predicate push-down allows the selections to be applied as early as possible during query execution. A similar idea is to push the group-by operation past one or more joins in order to reduce the amount of data participating in joins. Yan and Larson [36] proved necessary and sufficient conditions for deciding when this transformation is valid. Predicate move-around is a generalization of predicate push-down. This technique optimizes queries that involve views by moving predicates across subqueries in a query graph. Similar to the forward and backward propagations in our global optimization algorithm, predicate move-around moves predicates up in a query graph as an intermediate step before pushing them down.

Our approach differs from those techniques in the use of semantic knowledge. Since semantic knowledge may enlarge the search space of optimization, the potential savings of their knowledge-free techniques may not be as large as what our optimizer can achieve. Though predicate move-around does not apply semantic rules in optimization, it can apply functional dependencies of attribute values to infer new literals to reduce intermediate data. Our approach uses functional dependencies differently, as described in Section 3.4. Since Horn-clause rules can express functional dependencies, it should be straightforward to extend our approach to apply functional dependencies in this manner.

7.4 Semi-Joins

Compared to the traditional query optimization techniques for distributed database systems, such as *semi-joins* [7, 8, 26], our approach is more appropriate in a heterogeneous environment. Semi-joins requires the optimizer to move data from one remote database to the other. However, in many applications of heterogeneous information systems, database servers might have write-protection against external data and prohibit an optimizer from computing semi-joins. Even if there is no write-protection, in a heterogeneous environment, servers need *wrappers* for communication. This implies that we need to build wrappers between each pair of databases integrated, which amounts to n^2 translators if there are n databases. Our approach does not move data between database servers and requires only n wrappers for n databases. The overhead at run time for our approach is thus smaller. By the same token, our approach applies to the applications that involve databases with write-protection.

8 Conclusions

This paper presented a novel query optimization approach to reducing the cost of query plans generated by an information mediator. Our approach optimizes a query plan by modifying subqueries in the query plan using semantic knowledge about data. To efficiently execute a complex multidatabase query, it is important to reduce unnecessary intermediate data. The approach presented here can use rich semantic knowledge to infer the ranges of intermediate data accurately and yield arbitrarily large additional savings for complex multidatabase queries.

In addition to its effectiveness, the approach is more general and flexible than previous work in semantic query optimization in many aspects. This approach optimizes a larger class of queries, exploits more expressive semantic knowledge, and detects more optimization opportunities than previous work. This optimization approach can be implemented on top of existing query optimizers in a heterogeneous environment and hence supports the extensibility of information mediators.

Limitations of this approach include that to optimize aggregate operators such as `group-by`, the optimizer may need additional information other than the ranges of variables to derive effective cost-reducing reformulations. Also, it may be necessary to extend the cost models. An interesting future direction is to extend this approach to optimize query plans in an information mediator that integrates semi-structured information sources such as Web pages, text and multimedia data.

References

[1] G. Wiederhold, "Mediators in the architecture of future information systems," *IEEE Computer*, March 1992.

[2] Y. Arens, C. Y. Chee, C.-N. Hsu, and C. A. Knoblock, "Retrieving and integrating data from multiple information sources," *International Journal of Intelligent and Cooperative Information Systems*, vol. 2, no. 2, pp. 127–159, 1993.

[3] C. A. Knoblock, Y. Arens, and C.-N. Hsu, "Cooperating agents for information retrieval," in *Proceedings of the Second International Conference on Intelligent and Cooperative Information Systems (Coopis-94)*, (Toronto, Ontario, Canada), 1994.

[4] T. Kirk, A. Y. Levy, Y. Sagiv, and D. Srivastava, "The Information Manifold," in *Working Notes of the AAAI Spring Symposium on Information Gathering in Heterogeneous, Distributed Environments,Technical Report SS-95-08*, (Menlo Park, CA), AAAI Press, 1995.

[5] J. Hammer, H. Garcia-Molina, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom, "Information translation, mediation, and mosaic-based browsing in the TSIMMIS system," in *Proceedings of the ACM SIGMOD Internation Conference on Management of Data*, (San Jose, CA), 1995.

[6] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu, "Mariposa: A wide-area distributed database system," *The VLDB Journal*, vol. 5, no. 1, pp. 48–63, 1996.

[7] P. M. Apers, A. R. Hevner, and S. Yao, "Optimizing algorithms for distributed queries," *IEEE Trans. on Software Engineering*, vol. 9, pp. 57–68, 1983.

[8] M. Jarke and J. Koch, "Query optimization in database systems," *ACM Computer Surveys*, vol. 16, pp. 111–152, 1984.

[9] A. Y. Levy, D. Srivastava, and T. Kirk, "Data model and query evaluation in global information systems," *Journal of Intelligent Information Systems, Special Issue on Networked Information Discovery and Retrieval*, vol. 5, no. 2, 1995.

[10] Y. Arens, C. A. Knoblock, and W.-M. Shen, "Query reformulation for dynamic information integration," *Journal of Intelligent Information Systems, Special Issue on Intelligent Information Integration*, vol. 6, no. 2/3, pp. 99–130, 1996.

[11] A. Y. Levy, A. Rajaraman, and J. J. Ordille, "Querying heterogeneous information sources using source descriptions," in *Proceedings of the 22nd VLDB Conference (VLDB-96)*, (Bombay, India), 1996.

[12] C. A. Knoblock and A. Levy, "Exploiting run-time information for efficient processing of queries," in *Working Notes of the AAAI Spring Symposium on Information Gathering in Heterogeneous, Distributed Environments*, (Palo Alto, CA), 1995.

[13] C. A. Knoblock, "Planning, executing, sensing, and replanning for information gathering," in *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, (Montreal, Quebec, Canada), 1995.

[14] M. Hammer and S. B. Zdonik, "Knowledge-based query processing," in *Proceedings of the Sixth VLDB Conference*, (Washington, DC), pp. 137–146, 1980.

[15] J. J. King, *Query Optimization by Semantic Reasoning.* PhD thesis, Stanford University, Department of Computer Science, 1981.

[16] M. D. Siegel, "Automatic rule derivation for semantic query optimization," in *Proceedings of the Second International Conference on Expert Database Systems* (L. Kerschberg, ed.), pp. 371–385, Fairfax, VA: George Mason Foundation, 1988.

[17] S. Shekhar, J. Srivastava, and S. Dutta, "A formal model of trade-off between optimization and execution costs in semantic query optimization," in *Proceedings of the 14th VLDB Conference*, (Los Angeles, CA), 1988.

[18] S. T. Shenoy and Z. M. Ozsoyoglu, "Design and implementation of a semantic query optimizer," *IEEE Trans. Knowledge and Data Engineering*, vol. I, no. 3, pp. 344–361, 1989.

[19] C. T. Yu and W. Sun, "Automatic knowledge acquisition and maintenance for semantic query optimization," *IEEE Trans. Knowledge and Data Engineering*, vol. I, no. 3, pp. 362–375, 1989.

[20] U. S. Chakravarthy, J. Grant, and J. Minker, "Logic-based approach to semantic query optimization," *ACM Transactions on Database Systems*, vol. 15, no. 2, pp. 162–207, 1990.

[21] W. Sun and C. T. Yu, "Semantic query optimization for tree and chain queries," *IEEE Trans. Knowledge and Data Engineering*, vol. 6, no. 1, pp. 136–151, 1994.

[22] S. Shekhar, B. Hamidzadeh, A. Kohli, and M. Coyle, "Learning transformation rules for semantic query optimization: A data-driven approach," *IEEE Transactions on Knowledge and Data Engineering*, vol. 5, no. 6, pp. 950–964, 1993.

[23] C.-N. Hsu and C. A. Knoblock, "Rule induction for semantic query optimization," in *Machine Learning, Proceedings of the 11th International Conference (ML-94)*, (San Mateo, CA), Morgan Kaufmann, 1994.

[24] C.-N. Hsu and C. A. Knoblock, "Using inductive learning to generate rules for semantic query optimization," in *Advances in Knowledge Discovery and Data Mining* (U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, eds.), ch. 17, AAAI Press/MIT Press, 1996.

[25] C.-N. Hsu, *Learning Effective and Robust Knowledge for Semantic Query Optimization.* PhD thesis, Department of Computer Science, University of Southern California, 1996. Available as USC/ISI Technical Report RR-96-451, or ftp://ftp.isi.edu/isi-pubs/rr-96-451.ps.Z.

[26] J. D. Ullman, *Principles of Database and Knowledge-base Systems*, vol. I,II. Palo Alto, CA: Computer Science Press, 1988.

[27] R. MacGregor, "The evolving technology of classification-based knowledge representation systems," in *Principles of Semantic Networks: Explorations in the Representation of Knowledge* (J. Sowa, ed.), Morgan Kaufmann, 1990.

[28] J. W. Lloyd, *Foundations of Logic Programming.* Berlin, Germany: Springer-Verlag, 1987.

[29] E. F. Codd, *The Relational Model for Database Management, Version 2.* Addison-Wesley, 1990.

[30] C.-N. Hsu and C. A. Knoblock, "Reformulating query plans for multidatabase systems," in *Proceedings of the Second International Conference on Information and Knowledge Management (CIKM-93)*, (Washington, D.C.), 1993.

[31] M. R. Garey and D. S. Johnson, *Computers and Intractability.* New York: W. H. Freeman and Company, 1979.

[32] ORACLE, *ORACLE7 Server Concepts Manual.* Redwood, CA: Oracle Cooperation, December 1992.

[33] A. Y. Levy, I. S. Mumick, and Y. Sagiv, "Query optimization by predicate move-around," in *Proceedings of the 20th VLDB Conference*, (Santiago, Chile), 1994.

[34] P. R. Cohen, *Empirical methods for artificial intelligence.* Cambridge, MA: The MIT Press, 1995.

[35] C. T. Kwok and D. S. Weld, "Planning to gather information," in *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, (Menlo Park, CA), AAAI Press, 1996.

[36] W. P. Yan and P.-A. Larson, "Performing group-by before join," in *Proceedings of the 10th International Conference on Data Engineering*, (Houston, TX), pp. 89–100, IEEE Computer Society, 1994.

Technical Biographies of Authors

Chun-Nan Hsu Chun-Nan Hsu is an Assistant Research Fellow at Institute of Information Science, Academia Sinica in Taiwan and an Adjunct Assistant Professor in the Department of Computer Science and Information Engineering of National Chiao-Tung University, Taiwan. He was an Assistant Professor in the Department of Computer Science Engineering at Arizona State University from 1996 to 1998. He received his B.S. in Computer Engineering from National Chiao-Tung University, Taiwan in 1988, and both his M.S. and Ph.D. in Computer Science from the University of Southern California, USA in 1992 and 1996, respectively. His current research interests include machine learning, knowledge discovery and data mining, databases and intelligent Internet agents. He is a member of ACM, AAAI, and the Taiwanese Association for Artificial Intelligence. He was on the Program Committee of the 1998 National Artificial Intelligence Conference (AAAI-98). URI of his personal homepage is `http://www.iis.sinica.edu.tw/~chunnan`.

Craig A. Knoblock is a Project Leader at the USC Information Sciences Institute and a Research Associate Professor in both the Computer Science Department and the Integrated Media Systems Center at the University of Southern California. He has been at the University of Southern California since 1991. He received both his M.S. and Ph.D. in Computer Science from Carnegie Mellon University in 1988 and 1991, and his B.S. in Computer Science from Syracuse University in 1984. His research interests are on information gathering and integration, automated planning, machine learning, knowledge discovery, and knowledge representation.

He has published over 30 articles, book chapters, and conference papers in planning, machine learning and information integration, as well as the book *Generating Abstraction Hierarchies: An Automated Approach to Reducing Search in Planning* (Kluwer Academic Publishers, 1993). He received the Best Paper Award at the 1994 Canadian Artificial Intelligence Conference. He is a member of the editorial board for the Journal of Artificial Intelligence Research and was also on the Senior Program Committee of the 1997 and 1998 National Artificial Intelligence Conferences. URI of his personal homepage is `http://www.isi.edu/~knoblock`.