Published in the *Proceedings of the Second International Conference on Information and Knowledge Management*, Washington, DC, 1993

Reformulating Query Plans For Multidatabase Systems*

Chun-Nan Hsu Craig A. Knoblock
Information Sciences Institute and Department of Computer Science
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292
(310)822-1511
{chunnan, knoblock}@isi.edu

Abstract

A practical heterogeneous, distributed multidatabase system must answer queries efficiently. Conventional query optimization techniques are not adequate here because these techniques are dependent on the database structure, and rely on limited information which is not sufficient in complicated multidatabase queries. This paper presents an automated approach to reformulating query plans to improve the efficiency of multidatabase queries. This approach uses database abstractions, the knowledge about the contents of databases, to reformulate a query plan into less expensive but semantically equivalent one. We present two algorithms. The first algorithm reformulates subqueries to individual databases, the second algorithm extends the first one and reformulates the entire query plan. Empirical results show that the reformulations can provide significant savings with minimal overhead. The reformulation approach provides a global reduction in the amount of the intermediate data as well as local optimizations on the subqueries. [1]

*The research reported here was supported by Rome Laboratory of the Air Force Systems Command and the Defense Advanced Research Projects Agency under contract no. F30602-91-C-0081. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policy of DARPA, RL, the U.S. Government, or any person or agency connected with them.

[1]

1 Introduction

An important and difficult problem is how to efficiently retrieve information from distributed, heterogeneous multidatabase systems (Sheth and Larson, 1990). Retrieving and integrating distributed data often requires processing and storage of large amounts of intermediate data, which can be very costly. This cost can be reduced in some cases by selecting the appropriate sites for processing and employing query optimization techniques (Apers, Hevner, and Yao, 1983; Jarke and Koch, 1984) to reduce the cost of individual queries. However, these techniques are often inadequate since they rely on limited information about the syntactic structure of the queries and databases. This information alone is not usually sufficient for reducing the cost for complicated distributed, heterogeneous multidatabase queries.

This paper addresses this problem of multidatabase retrieval by bringing to bear a richer set of knowledge about databases to optimize multidatabase queries. The idea is to use semantic knowledge of the contents of databases to reformulate queries into equivalent yet less expensive ones. Using the additional semantic knowledge, the potential cost reduction is significantly greater than can be derived from optimization based on the syntactical structure of queries alone. Since the knowledge required can be learned from any database, this approach is very general.

Consider the following hypothetical example. Suppose that there are two databases in a multidatabase system, one containing data about ports, and another about ships. A query is given to retrieve the data of ports that have a depth that can accommodate tankers. This query may be very expensive because the data about ports must be retrieved and compared with the draft of all the tankers. Suppose that the system learned from the ship database that if the ship type is tanker, then its draft is at least 10 meters. With this knowl-

edge, the original query can be reformulated so that the system only retrieves data about ports whose depth is greater than 10 meters. This additional constraint may significantly reduce the amount of data retrieved from the ship database and thus substantially reduce the cost of executing the query.

In this paper, we present an efficient algorithm to perform this type of semantic reformulation. We implement the algorithm in the context of the SIMS project (Arens and Knoblock, 1992; Arens, Chee, Hsu, and Knoblock, 1993). The SIMS project applies a variety of AI techniques and systems to build an integrated intelligent interface between users and distributed, heterogeneous multiple data/knowledge-bases systems. Given a multidatabase query, the planner of SIMS generates a partially ordered query plan to retrieve the data. The reformulation algorithm presented here is used to reformulate this initial query plan to reduce the cost of retrieval.

The query reformulation approach was initially proposed by (King, 1981) and (Hammer and Zdonik, 1980). Our approach differs from theirs and the following related work (Siegel, 1988; Chakravarthy, Grant and Minker, 1990) in that we do not rely on heuristics to guide the search in a hill-climbing manner, which often results in local optima. Moreover, we consider queries for data distributed over multiple sources, while they only consider single database queries.

The remainder of this paper is organized as follows. The next section describes the query planning in SIMS. Section 3 reviews the semantic query optimization and our reformulation algorithm for single database queries. Section 4 extends the idea to multidatabase queries. Section 5 shows our experimental results. We compare our approach with related work in Section 6. Section 7 reviews the contributions of the paper and describes directions for future work.

2 Query Planning

Figure 1 shows an example SIMS semantic query. This query retrieves the name of the ports in Germany that have both railroad capabilities at the port and refrigerator storage. SIMS accepts queries in the form of a description of a class of objects about which information is desired. This description is composed of statements in the Loom knowledge representation language (Macgregor, 1990). The user is not presumed to know how information is distributed over the data- and knowledge bases to which SIMS has access — but he/she *is* assumed to be familiar with the application domain, and to use standard terminology to compose the Loom query. The interface enables the user to inspect the domain model as an aid to composing queries. SIMS proceeds to decompose the user's query into a collection of more elementary statements that refer to data stored in available information sources. SIMS then uses Prodigy (Carbonell, Knoblock, and Minton, 1991) to create a plan for retrieving the desired information, establishing the order and content of the various plan steps/subqueries. Figure 2 shows an example partially ordered multidatabase query plan generated by SIMS's query planner.

```
(retrieve (?name)
  (:and (port ?port)
        (port.rail ?port "Y")
        (port.refrig ?port ?refrig)
        (> ?refrig 0)
        (port.geocode ?port ?geocode)
        (port.name ?port ?name)
        (geoloc ?geoloc)
        (geoloc.geocode ?geoloc ?geocode)
        (geoloc.country_name ?geoloc
         "Germany")))
```

Figure 1: Example SIMS Semantic Query

Each node in the plan corresponds to a *subquery* to an individual data- or knowledge base. The edges indicate the data flow direction from one database to another. Data pertaining to this query is spread over two remote databases — one containing information about ports and the other about geographic locations. In the figure, the two `db-retrieve` subqueries are queries to each of these databases. They will be translated into their corresponding database query languages before being sent to the DBMSs. The `loom-retrieve` subquery contains the interaction constraints involving values from the different remote databases. To execute this query plan, the two `db-retrieve` subqueries will first be executed, and the retrieved data will be loaded into Loom and translated into objects of semantic classes. Loom then evaluates the constraints specified in the `loom-retrieve` to retrieve the resired answer from the sets of intermediate data.

Each subquery consists of conjunctions of constraints. In the upper subquery in Figure 2, the first clause, `(afsc_sea_port ?port)`, binds the variable `?port` to the set of port instances in the database `afsc_sea_port`. The second clause is a *range constraint* which restricts the attribute `afsc_port.rail` of `?port` to have value `Y`. This indicates that the port has railroad capability. The clause `(> ?refrig 0)` is another example of range constraint that constrains the ports to have non-zero refrigerator storage. Constraints that involves two or more variables are *interaction constraints*, such as the one in the `loom-retrieve` subquery.

The most expensive part of the query plan is often moving intermediate data from remote databases to the

Figure 2: Preliminary SIMS Plan for Example Query

local Loom system. Consider the example in Figure 2, the cost of pairwise comparison in the final subquery is proportional to the square of the amount of data items retrieved from the remote databases. If we can reformulate these subqueries such that the interaction constraints in the final subquery are also considered, the amount of intermediate data will be reduced.

3 Subquery Reformulation

We start with the subquery reformulation algorithm and then extend it to reformulate the entire query plan in the next section. The goal of the query reformulation is to use reformulation to search for the least expensive query from the space of semantically equivalent queries to the original one. Two queries are defined to be *semantically equivalent* (Chu and Lee, 1990; Siegel, 1988) if they return identical answer given the same contents of the database. The reformulation from one query to another is by logical inference using *database abstractions*, the abstracted knowledge of the contents of relevant databases. The database abstractions describe the databases in terms of the set of closed formulas of first-order logic. These formulas describe the database in the sense that they are true with regard to all instances in the database. We define two classes of formulas: *range information* are propositions that assert the ranges of the values of database attributes; and *rules* are of the form of implications with an arbitrary number of range propositions on the antecedent side and one range proposition on the consequent side. Figure 3 shows a small set of the database abstractions. In all formulas the variables are implicitly universally quantified.

The first two rules in Figure 3 state that for all instances, the value of its attribute country name is `"GERMANY"` if and only if the value of its attribute country code is `"FRG"`. With these two rules, we can reformulate the subquery `SUBQ1` in Figure 4 to the equivalent subquery `SUBQ2` by replacing the constraint on `geo_geoloc.country_name` with the constraint on `geo_geoloc.country_code`. We can inversely reformulate `SUBQ2` to `SUBQ1` with the same rules. Given a subquery Q, let $C_1, \ldots, C_k$ be the set of range and interaction constraints in Q, the following *reformulation operators* return a semantically equivalent query:

- **Range Refinement:** A range-information proposition states that the values of an attribute A are within some range R_d. If a range constraint of A in Q constrains the values of A in some range R_i, then we can refine this range constraint by replacing the constraining range R_i with $R_i \cap R_d$.
- **Constraint Addition:** Given a rule $A \rightarrow B$, if Q implies A then we can add constraint B to Q.
- **Constraint Deletion:** Given a rule $A \rightarrow B$, and Q implies A. If there exists C_i in Q and B implies C_i, then we can delete C_i from Q.
- **Subquery Refutation:** Given a rule $A \rightarrow B$, and Q implies A, if there exists C_i in Q and B implies $\neg C_i$, then we can assert that Q will return `NIL`.

Replacing constraints is treated as a combination of addition and deletion. Note that these reformulation operators do not always lead to more efficient versions of the subquery. Knowledge about the access cost of

attributes is required to guide the search. For example, suppose the only index is placed on the attribute geo_geoloc.country_name, then reformulate SUBQ2 to SUBQ1 will reduce the cost from $O(n)$ to $O(k)$, where n is the size of the database and k is the amount of data retrieved. However, if either geo_geoloc.country_name and geo_geoloc.country_code are not indexed, then we will prefer the lower cost short string attribute geo_geoloc.country_code. In this case, reformulating SUBQ1 to SUBQ2 becomes more reasonable. Figure 5 shows our subquery reformulation algorithm. We explain the algorithm below by showing how SUBQ-REFORMULATION reformulates the subquery SUBQ1, the lower query in the query plan in Figure 2.

Range Information:

```
1:(geo_geoloc.country_name ∈
 ("Taiwan" "Italy" "Denmark" "Germany"
  "Turkey"))
2:(afsc_port.geocode ∈
 ("BSRL" "HNTS" "FGTW" "VXTY" "WPKZ" "XJCS"))
3:(0 ≤ afsc_port.refrig_storage ≤ 1000)
```

Rules:

```
1:(geo_geoloc.country_name = "GERMANY")
  ⟹ (geo_geoloc.country_code = "FRG")
2:(geo_geoloc.country_code = "FRG")
  ⟹ (geo_geoloc.country_name = "Germany")
3:(geo_geoloc.country_code = "FRG")
  ⟹ (47.15 ≤ geo_geoloc.latitude ≤ 54.74)
4:(afsc_port.rail = "Y" )
  ⟹ (afsc_port.geocode ∈
   ("BSRL" "HNTS" "FGTW"))
5:(6.42 ≤ geo_geoloc.longitude ≤ 15.00)
  ∧ (47.15 ≤ geo_geoloc.latitude ≤ 54.74)
  ⟹ (geo_geoloc.country_code = "FRG")
```

Figure 3: Example of Database Abstractions

There are three input arguments in this algorithm. The first argument Subquery is the subquery to be reformulated. Another argument DB-Knowledge contains the set of range information and rules that describe the database queried by the input subquery. And Cost-Model contains the knowledge to decide the execution cost of constraints. Initially, all the range constraints are refined by applying the **range refinement** operator. The reason why we want to refine the constraining ranges is to make the subquery more likely to match many rules. This is because after range refinement, the constraining ranges are smaller and more likely to imply the antecedent of a rule. Range refinement also reduces comparisons in evaluating constraints on string type attributes. The only range constraint in

SUBQ1:

```
(retrieve (?geoloc ?geocode2)
 (:and (geo_geoloc?geoloc)
       (geo_geoloc.geocode ?geoloc ?geocode2)
       (geo_geoloc.country_name ?geoloc
        "Germany")))
```

SUBQ2:

```
(retrieve (?geoloc ?geocode2)
 (:and (geo_geoloc?geoloc)
       (geo_geoloc.geocode ?geoloc ?geocode2)
       (geo_geoloc.country_code ?geoloc
        "FRG")))
```

SUBQ3:

```
(retrieve (?geoloc ?geocode2)
 (:and (geo_geoloc?geoloc)
       (geo_geoloc.geocode ?geoloc ?geocode2)
       (geo_geoloc.country_code ?geoloc "FRG")
       (geo_geoloc.latitude ?geoloc ?latitude)
       (>= ?latitude 47.15)
       (<= ?latitude 54.74)))
```

Figure 4: Equivalent Subqueries

SUBQ1 is on geo_geoloc.country_name, and its constrained value GERMANY is within the range of possible values (see the first formula of range information). Thus, this constraint is unchanged.

The second step is to match all applicable rules from the set of database abstractions using the reformulation operators defined above. If a **Subquery Refutation** rule is found then the subquery is refuted and the algorithm halts immediately. When a **Constraint Deletion** rule is found, then some constraints in the subquery are redundant and can be deleted from the subquery without changing the semantics. We only put the constraint in the Inferred-Set instead of actually deleting it from the subquery. This is because, its redundancy is due to the *logical* reason, not the performance consideration. More knowledge and analysis is required to decide whether it should be actually deleted. In the case that a **Constraint Addition** rule is found, we add the constraint to the subquery and also put it in the Inferred-Set. The first rule in Figure 3 is matched and fired for SUBQ1 and we get an additional constraint (geo_geoloc.country_code ?geoloc "FRG"), which is added to the Inferred-Set. Then the second and third rules are matched because of the additional constraint on country code. The constraints geo_geoloc.latitude and geo_geoloc.country_name are added to the Inferred-Set.

The third step is to select the constraints in Inferred-Set to delete from the subquery. The selection is based on the constraint's relative estimated execution cost which is computed by the type of the con-

```
SUBQ-REFORMULATION(Subquery, DB-Knowledge, Cost-Model)
 1.refine range constraints, if Subquery refuted, return  Nil;
 2.for all applicable rules A → B in DB-Knowledge:
    if Subquery refuted, return  NIL;
    else add B to Inferred-Set, add (B,A) to Dependency-List;
 3.for all B in Inferred-Set in the order of their cost:
    if B is not indexed and ∃ (B,A) in Dependency-List
       delete B from Subquery, delete (B,A) from Dependency-List;
       replace all (C,B) in dependency list with (C,A);
 4.return (reformulated Subquery, Inferred-Set)
END.
```

Figure 5: Subquery Reformulation Algorithm

straints (range constraint, or interaction constraint), the type of the attribute's values (integer, string, and their length), and whether they are indexed. The information required for this estimation is available from the input `cost-model` provided by SIMS. The constraints in the `Inferred-Set` are sorted into the partial order of their cost and then deleted in this order until the total cost of the remaining constraints is less than the original subquery. To preserve the semantics of the subquery, we keep a dependency list of the inferred constraints to avoid deleting all constraints in an implication cycle. In our example, the attribute `geo_geoloc.country_name` is deleted because its long string type is the most expensive. The next most expensive constraint is the one on attribute `geo_geoloc.country_code`. However, it should be preserved because the cause of its deletability (i.e., the constraint on `geo_geoloc.country_name`) was just deleted. Finally, the constraint on `geo_geoloc.latitude` is kept because it is an indexed attribute that will improve the efficiency of the subquery. The algorithm returns the reformulated subquery SUBQ3 as shown in Figure 4, as well as the `Inferred-Set`, which will be used for reformulating the succeeding subqueries in the query plan.

Not all rules are matched directly from the database abstractions. For interaction constraints, we have axioms for set inclusion and mathematical relations. For example, if there is an interaction constraint $(> ?Y\ ?X)$ and we have rules or range information which assert that $(> ?X\ 17)$, then we can add a new constraint $(> ?Y\ 17)$ because $(> ?X\ 17) \wedge (> ?Y\ ?X) \Rightarrow (> ?Y\ 17)$. These axioms are implemented as inference procedures for efficiency.

The worst case complexity of `SUBQ-REFORMULATION` is $O(R^2 N * max(M, \log N))$, where M is the maximum length of the antecedent of the rules, N is the greatest number of constraints in the partially reformulated query, that is, the number of original constraints plus the number of added constraints in `Inferred-Set` before final selection, and R is the size of `DB-Knowledge`. The worst case cost to match a rule is $O(MN)$. Suppose the system matches applicable rules linearly in the set of the database abstractions, all rules must be matched and this takes RMN. The complexity of Step 2 is thus $O(R^2 MN)$, in the case that only one rule is fired in every scan of the database abstractions, and every rule is eventually fired. The complexity of Step 3 is $O(N \log N)$, the cost of sorting. Deleting constraints in the `Inferred-Set` in their order of estimated cost takes $O(N)$.

Because the added constraints are range constraints of an attribute, the number of constraints will not exceed the number of the attributes of the relevant database tables.[2] Therefore, N is small compared to R. In the average case, the rule match cost is about $O(N)$, since the lengths of rules are usually less than 3. The database abstractions are normally scanned less than 3 times. Therefore, R dominates the complexity of the algorithm. With small values of R, this algorithm will not introduce significant overhead to the cost of query processing. To alleviate the impact of a large R on the system's performance, we can adopt sophisticated indexing and hashing techniques in rule matching, or restrain the size of the database abstractions by removing database abstractions with low utility.

4 Query Plan Reformulation

We can reformulate each subquery in the query plan with the subquery reformulation algorithm and improve their efficiency. However, the most expensive aspect of the multidatabase query is often processing intermediate data. In the example query plan in Figure 2, the constraint on the final subqueries involves the variables `?geocode` and `?geocode2` that are bound in the preceding subqueries. If we can reformulate these preceding subqueries so that they retrieve only the data in-

[2] If there are two constraints on the same attribute, we can always apply the **range refinement** operator on them and merge them together.

```
QPLAN-REFORMULATION(Plan, DB-Knowledge, Cost-Model)
 1.KB ← DB-Knowledge;
 2.for all subqueries S in the order specified in Plan:
   (S',Inferred-Set) ← SUBQ-REFORMULATION(S,KB,Cost-Model);
   if S' refuted, return  Nil;
   else update KB with Inferred-Set; update Plan with S';
 3.for all subqueries S whose semantics are changed:
   SUBQ-REFORMULATION(S, DB-Knowledge, Cost-Model);
 4.return reformulated Plan
END.
```

Figure 6: Query Plan Reformulation Algorithm

stances possibly satisfying the constraint `(= ?geocode ?geocode2)` in the final subquery, the intermediate data will be reduced. This requires the query plan reformulation algorithm to be able to propagate the constraints along the data flow paths in the query plan. The query plan reformulation algorithm defined in Figure 6 achieves this by updating the database abstractions and rearranging constraints. We explain the algorithm below using the query plan in Figure 2.

The algorithm takes three input arguments. The argument `Plan` is the input query plan, `DB-Knowledge` and `Cost-Model` are defined as in `SUBQ-REFORMULATION`. After the initialization step, in the second step, the algorithm reformulates each subquery in the partial order (i.e., the data flow order) specified in the plan. The two `db-retrieve` subqueries are reformulated first. The database abstractions are updated with `Inferred-Set` which is returned from `SUBQ-REFORMULATION` to propagate the constraints to later subqueries. For example, when reformulating the upper subquery, the fourth rule is fired for adding the constraint on the variable `?geocode` which is bound to the attribute `afsc_port.geocode`. Although this long string type constraint is then selected to be deleted, it reveals the range of `afsc_port.geocode` in the output data of the upper subquery. This range information together with other inferred constraints in `Inferred-Set` replaces the original range information to update the initial database abstractions. In this example, the second formula of the initial range information is replaced by `(afsc_port.geocode ∈ ("BSRL" "HNTS" "FGTW"))`, the consequent condition of the fourth rule. The algorithm uses this updated range information to reformulate the final subquery and reduces the possible values from six to three. In addition, the constraint `(afsc_port.rail ?port "Y")` in the upper subquery is propagated along the data flow path to its succeeding subquery implicitly.

Now that the updated range information for `?geocode` is available, the subquery reformulation algorithm can infer from the constraint `(= ?geocode ?geocode2)` a new constraint `(member ?geocode2 ("BSRL" "HNTS" "FGTW"))` and add it to the final subquery. However, this constraint should be executed by the remote DBMS instead of by the local Loom system, because it does not involve interaction with different databases. In this case, when updating the query plan with the reformulated subquery, the algorithm locates where the constrained variable of each new constraint is bound, and inserts the new constraint in the corresponding subqueries. In our example, the variable is bound by `(geo_geoloc.geocode ?geoloc ?geocode2)` in the lower subquery in Figure 2. The algorithm will insert the new constraint on `?geocode2` in that subquery. In this way, the constraints `(afsc_port.rail ?port "Y")` and `(= ?geocode ?geocode2)` are propagated back along the data flow path to the lower subquery. This process of new constraint insertion is referred to as *constraint rearrangement*.

However, the semantics of the rearranged subqueries, such as the lower subquery in this example, are changed because of the newly inserted constraints. (Note, that the semantics of the overall query plan remain the same.) After all the subqueries in the plan have been reformulated, Step 3 of the algorithm reformulates these subqueries again to improve their efficiency. In our example, the reformulation algorithm is applied again to the lower subquery, but no reformulation is found to be appropriate. The final reformulated query plan is shown in Figure 7.

This query plan is more efficient and returns the same answer as the original one. In our example, the lower subquery is more efficient because of the new constraint on the indexed attribute `geo_geoloc.latitude` (by `SUBQ-REFORMULATION`). The intermediate data items are reduced because of the new constraint on the attribute `geo_geoloc.geocode`. The logical rationale of this new constraint is derived from the constraints in the other two subqueries: `(afsc_port.rail ?port "Y")` and `(= ?geocode ?geocode2)`, and the fourth rule in the database abstractions.

The worst case complexity of `QPLAN-REFORMULATION`

Figure 7: Reformulated SIMS Plan for Example Query

is $O(SR^2N * max(M, \log N))$, where S is the number of subqueries in the query plan, and $R^2N * max(M, \log N)$ is the cost of `SUBQ-REFORMULATION`. In the average case, S is less than 5, so the dominating factor is still the cost of the subquery reformulation $R^2N * max(M, \log N)$, in which R is the most important factor. Consequently, if R is relatively small, or we can match rules efficiently, this algorithm is efficient enough to be neglected in the total cost of query processing.

5 Experimental Results

The reformulation algorithms are implemented in the context of the SIMS system, which, for the purpose of our experiments, is connected with two distributed Oracle databases. Table 1 shows the size of these databases. The queries used were selected from the set of SQL queries constructed by other users of the databases. Table 2 lists the features of these queries. The first three queries are single database queries. The remaining queries access both databases, so they have two database subqueries and one subquery for evaluating interaction constraints and performing joins in Loom. The number of constraints includes the number of range and interaction constraints. The number of answers may not equal the number of retrieved instances, because the answers are results of projection on specified attributes and all duplicates are removed. Query 3 and 6 are null queries.

The performance statistics are shown in Table 3. All entries are based on an average of 10 trial executions. The number of rules fired counts both range information and rules used in reformulation. Note that a rule may be fired twice or more in Step 2 and 3 of the `QPLAN-REFORMULATION` algorithm. The amount of intermediate data indicated for each multidatabase query is the total number of the data instances retrieved from both databases and transferred to the SIMS system.

Database	Contents	Instances	Size(MByte)
Geo	Geographical locations	56708 rows in 16 tables	10.48
Assets	Planes,ships other assets	5728 rows in 14 tables	0.51

Table 1: Database Size

The most noticeable cost reduction is achieved by reformulation when the system can determine the answers of queries from its knowledge. In these queries, the system can eliminate the corresponding database access. For example, the system refutes Query 3 and 6 and returns the answer `NIL` immediately. In Query 7, the system asserts the answer of a database subquery. This subquery is eliminated, and the query reduces to a single database query. Query 8, 9, and 10 are typical multidatabase queries, the system reformulates them and eliminates a large amount of intermediate data. Query execution time is thus reduced by about a factor of 2. Query 2 is an expensive single database query. The system reformulates it by introducing a constraint on an indexed attribute and saves a considerable amount of time.

There are cases where the reformulation did not achieve significant cost reduction. The first case is when the query is already very efficient. For example, in Query 1, the query execution time without reformu-

Query (short descriptions)	Database Accessed	Number of Subqueries	Number of Constraints	Number of Answers
1:Airports: runway≥8000, concrete surface	Geo	1	2	2
2:Locations: location code in state gsa code "TW"	Geo	1	2	147
3:Wharves: container cranes and rail track	Geo	1	4	0
4:Wharves: container/breakbulk ships	Geo,Assets	3	10	6
5:Ports: accommodate ship with code "1240"	Geo,Assets	3	4	2
6:Ports: accommodate ship "1207", mob "10C"	Geo,Assets	3	4	0
7:Ships: dock in channels of port in Long Beach	Geo,Assets	3	3	28
8:Ports & Ships: berths storage > ship capacity	Geo,Assets	3	1	9
9:Ports & Ships:ship length, fit berth type "TE"	Geo,Assets	3	4	20
10:Ports & Ships:Tunisia ports,frozen cargo unload	Geo,Assets	3	5	29

Table 2: Experiment Multidatabase Queries

query	1	2	3	4	5	6	7	8	9	10
planning time (sec)	0.5	0.3	0.6	2.1	1.1	0.7	0.7	0.5	0.5	0.8
reformulation time	0.1	0.1	0.0	0.5	0.1	0.0	0.0	0.1	0.1	0.3
rules fired (times)	37	18	11	126	63	8	17	15	19	71
intermediate data w/o Ref[a]	-	-	-	145	41	1	810	956	808	810
intermediate data w/ Ref[b]	-	-	-	145	35	0	28	233	320	607
query execution time w/o Ref	0.3	8.2	0.6	12.3	11.3	2.0	251.0	401.8	255.8	258.8
query execution time w/ Ref	0.3	1.5	0.0	11.3	11.1	0.0	0.3	207.5	102.9	195.2
total elapsed time w/o Ref	0.8	8.5	1.2	14.4	12.4	2.7	251.7	402.3	256.3	259.6
total elapsed time w/ Ref	0.9	1.9	0.6	13.9	12.3	0.7	1.0	208.1	103.5	196.3

[a] w/o Ref = Without reformulation.
[b] w/ Ref = With reformulation.

Table 3: Experimental Results

lation is very short, and reformulation appears to be unnecessary. Another case is when the system can not reduce the amount of intermediate data, as in Query 4 and 5. This is due to a lack of sufficient database abstractions, or it may just be impossible to reduce the cost for some particular queries and databases. However, as indicated in the experimental results, the reformulation time is so short that even when no significant cost reduction can be achieved, the overhead will not degrade the performance of retrieval. To sum up, the reformulation approach is effective and can achieve a substantial cost reduction.

In this experiment, the system uses a set of database abstractions consisting of 203 rules about range information and 64 implication rules for every query plan. These database abstractions were prepared by compiling the databases. For range information, the compiling procedure summarizes the range of each attribute of the database by extracting the minimum and maximum values for numerical attributes, and enumerating the possible values for string type attributes. If the number of possible values exceeds a threshold, this range information is discarded. The implication rules were prepared by a semi-automatic learning algorithm similar to the KID3 (Piatetsky-Shapiro, 1991). This algorithm takes the user input condition A, and learns a set of rules of the form $A \rightarrow B$ from the database. The algorithm retrieves the data that satisfy the condition A, then compiles the data for the conclusions B.

6 Related Work

The semantic query optimization approach has been studied extensively in previous work (Chakravarthy, Grant, and Minker, 1990; Siegel, 1988; King, 1981; Hammer and Zdonik, 1980). These systems demonstrate the benefit of using knowledge of database contents to optimize queries. The most significant difference between our approach and theirs is that they rely on heuristics, and search for the optimal equivalent query in a hill-climbing manner, while our approach adopts a *delayed-commitment* strategy. Their systems search for the optimal query in the space of equivalent queries of the given query. Whenever a rule is fired, their systems will generate a new equivalent query, until an optimal one is found. This leads to a combinatorial explosion of equivalent queries among which

the system needs to select. To overcome this problem, they use heuristics and hill-climbing to prune the search space, but as a consequence, the reformulated query is usually only locally optimal. Sometimes, this process causes infinite loops that require more heuristics to resolve (Siegel, 1988).

To illustrate the problem of previous work, consider the following situation. Suppose there are two rules in the set of database abstractions, $A \rightarrow B$, and $B \rightarrow C$. Suppose we are given a query Q which implies A, and the rule $A \rightarrow B$ is the only applicable rule. Rule $B \rightarrow C$ will be applicable if B is added to Q by firing $A \rightarrow B$. Suppose further that Q and C are contradictory, and B is a costly constraint. For hill-climbing systems, $A \rightarrow B$ will never be fired since adding B will increase the cost. Thus, $B \rightarrow C$ will not be applicable. As a result, the system can not figure out that the answer of the query is null, unless it can backtrack. But backtracking requires the system to maintain a large set of equivalent queries. This overhead will make the system impractical.

In contrast, our subquery reformulation algorithm does not generate queries each time a rule is fired. Instead, we fire all applicable rules at once and collect the candidate constraints in a list `Inferred-Set` and then select only those that will contribute to the cost reduction. In the example above, our algorithm can consider both rules and refute the query without maintaining a large set of equivalent queries. This approach is a delayed-commitment strategy because the system delays the reformulation until it has enough information to make a decision. Although the algorithm fires all applicable rules, it is still polynomial. The empirical results show that it is efficient. Moreover, it is flexible because no additional specific heuristics are required. The list `Inferred-Set` turns out to be the information needed to propagate constraints among subqueries. Subsequently, the subquery reformulation is easily extended to query plan reformulation.

Compared to conventional syntactical optimization techniques for distributed database systems, our approach differs in both the knowledge brought to bear and the way queries are optimized. (Apers, Hevner, and Yao, 1983; Jarke and Koch, 1984; Ullman, 1988) describe approaches that use the *semi-join* operation to join two database relations in distributed databases. The semi-join techniques propagate constraints by computing a semi-join before performing the actual join. Our approach propagates constraints from knowledge of database abstractions without accessing remote databases, and thus has less overhead then the semi-join. The semi-join techniques may reduce intermediate data when the result of semi-join is significantly smaller than the entire relevant relations. However, there are situations when semi-join degrades the performance. The system needs to know the *reduction factors* of each semi-join to decide a semi-join schedule that will save execution cost. To compute reduction factor requires knowledge of the size of relevant relations and their joining path. It is usually difficult to estimate the size of an intermediate relation when the query is complicated. Some semi-join approaches assume a unrealistically simplified model to reduce the overhead, but to make semi-join approach effective, the system still need to bring to bear extensive statistical knowledge to estimate relation sizes (Jarke and Koch, 1984).

Another difference between our approach and the conventional distributed query optimization techniques is that they assume a homogeneous environment. They can transfer data from one site to another without any transformation. They can also distribute a relation into fragments and store them in different sites. Data distribution strategy and execution order scheduling are their major concerns. We assume a heterogeneous environment, so we focus on flexible reformulation on the semantic aspects of queries. In the future, we also intend to include size and system configuration information in our planning and reformulation algorithm to optimize query plans on the execution order.

7 Conclusion

This paper presented a problem reformulation approach to reducing the cost of domain-modeled multidatabase queries. The reformulation is based on logical inferences from database abstractions. This simple, efficient algorithm reduces the cost of the query plan by reducing intermediate data and refining each subquery. This is achieved without database implementation dependent heuristic control. Empirical results demonstrate that this algorithm can provide significant reductions in the cost of executing query plans.

One of the limitations of our implementation is that the rule match algorithm is linear to the size of the database abstractions. A very large set of database abstractions could make the reformulation costly. To avoid this problem, we plan to adopt a more sophisticated rule match algorithm, such as the RETE algorithm (Forgy, 1982), or its more efficient variations, to reduce this impact.

One important issue not addressed in this paper is how to automatically acquire the database abstractions for reformulation (Siegel, 1988). We are now developing a learning algorithm that is driven by example queries (Hsu and Knoblock, 1993). We plan to use inductive learning (Cai, Cercone, and Han 1991; Haussler, 1988; Michalski, 1983) to identify the costly aspects of the example subqueries and propose candidate rules. The candidate rules will then be refined and learned by the

system. After the system has learned a set of database abstractions, it needs to monitor their utility and validity to maintain the system's performance. We will address this issue in the future work.

Acknowledgements

We would like to thank the SIMS project leader Yigal Arens and the anonymous reviewers for their comments and suggestions on earlier drafts of this paper. Chin Y. Chee, another member of the SIMS project, is responsible for much of the programming that has gone into SIMS. Thanks also to the LIM project for providing us with the databases and the queries used for our work.

References

Apers, P., Hevner, A., and Yao, S.B., 1983. Optimizing algorithms for distributed queries. *IEEE Trans. on Software Engineering*, 9:57–68.

Arens, Y., Chee, C., Hsu, C.-N., and Knoblock, C.A., 1993. Retrieving and integrating data from multiple information sources. *International Journal on Intelligent and Cooperative Information Systems*. In press.

Arens, Y. and Knoblock, C.A., 1992. Planning and reformulating queries for semantically-modeled multidatabase systems. In *Proceedings of the First International Conference on Information and Knowledge Management*, Baltimore, MD.

Cai, Y., Cercone, N., and Han, J., 1991. Learning in relational databases: An attribute-oriented approach. *Computational Intelligence*, 7(3):119–132.

Chakravarthy, U., Grant, J., and Minker, J., 1990. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, 15(2):162–207.

Carbonell, J., Knoblock, C.A., and Minton, S., 1991. PRODIGY: An integrated architecture for planning and learning. In Kurt VanLehn, editor, *Architectures for Intelligence*, pages 241–278. Lawrence Erlbaum, Hillsdale, NJ.

Chu, W.W., Lee, R.-C., 1990. Semantic query processing via database restructuring. In Proceedings of the Eighth International Congress of Cybernetics and Systems. New York, NY.

Forgy. C.L., 1982. RETE: A fast algorithm for the many pattern/many object pattern matching problem. *Artificial Intelligence*, pages 17–37.

Haussler, D., 1988. Quantifying inductive bias: AI learning algorithms and Valiant's learning framework. *Artificial Intelligence*, 36:177–222.

Hsu, C.-N., and Knoblock, C.A., 1993. Learning database abstractions for query reformulation. In *Proceedings of AAAI-93 Workshop on Knowledge Discovery in Databases*, Washington, DC.

Hammer, M., and Zdonik, S., 1980. Knowledge-based query processing. In *Proceedings of the Sixth VLDB Conference*, pages 137–146, Washington, DC.

Jarke, M. and Koch, J., 1984. Query optimization in database systems. *ACM Computer Surveys*, 16:111–152.

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

MacGregor, R., 1990. The evolving technology of classification-based knowledge representation systems. In John Sowa, editor, *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Morgan Kaufmann.

Michalski, R.S., 1983. A theory and methodology of inductive learning. In *Machine Learning: An Artificial Intelligence Approach*, volume I, pages 83–134. Morgan Kaufmann Publishers, Inc., Los Altos, CA.

Piatetsky-Shapiro, G., 1991. Discovery, analysis, and presentation of strong rules. In G. Piatetsky-Shapiro, editor, *Knowledge Discovery in Databases*, pages 229–248. MIT Press.

Sheth, A.P., and Larson, J.A., 1990. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22:183–236.

Siegel, M.D., 1988. Automatic rule derivation for semantic query optimization. In Larry Kerschberg, editor, *Proceedings of the Second International Conference on Expert Database Systems*, pages 371–385. George Mason Foundation, Fairfax, VA.

Ullman, J.D., 1988. *Principles of Database and Knowledge-base Systems*, volume II. Computer Science Press, Palo Alto, CA.