Query Caching and Optimization in Distributed Mediator Systems *

S. Adalı
University of Maryland
sibel@cs.umd.edu

K.S. Candan
University of Maryland
candan@cs.umd.edu

Y. Papakonstantinou
Stanford University
yannis@db.stanford.edu

V.S. Subrahmanian
University of Maryland
vs@cs.umd.edu

Abstract

Query processing and optimization in mediator systems that access distributed non-proprietary sources pose many novel problems. Cost-based query optimization is hard because the mediator does not have access to source statistics information and furthermore it may not be easy to model the source's performance. At the same time, querying remote sources may be very expensive because of high connection overhead, long computation time, financial charges, and temporary unavailability. We propose a cost-based optimization technique that caches statistics of actual calls to the sources and consequently estimates the cost of the possible execution plans based on the statistics cache. We investigate issues pertaining to the design of the statistics cache and experimentally analyze various tradeoffs. We also present a query result caching mechanism that allows us to effectively use results of prior queries when the source is not readily available. We employ the novel ***invariants*** mechanism, which shows how semantic information about data sources may be used to discover cached query results of interest.

1 Introduction

During the past few decades, the world has witnessed a spectacular explosion in the quantity of data available in one electronic form or another. This vast quantity of data has been gathered, organized, and stored by a small army of individuals, working for different organizations on varied problems in ways that were best suited to accomplish the task in question. Wiederhold [29] proposed the concept of a *mediator* as a way of formulating the semantic information necessary to integrate these heterogeneous sources and make sense out of a collection of potentially incomplete and inconsistent data and inherently incompatible programs. Intuitively, a mediator is a program that accesses and integrates multiple databases and/or software packages. In particular, the user of a mediated system sends queries to the mediator, which in turn sends appropriate subqueries to different software packages and/or databases in the mediated system. The HERMES project (short for HEterogeneous Reasoning and MEdiator System) at the University of Maryland [28, 4, 27, 20] and the TSIMMIS project at Stanford University [41] provide frameworks for handling different types of heterogeneity that exist between programs and databases.

In this paper we focus on issues related to query processing and optimization in mediator systems that access distributed non-proprietary information sources. We make the following contributions:

1. **Intelligent Caches:** We show how a mediator may maintain local caches consisting of the results of previous calls to external software packages (local or remote). Furthermore, we recall the notion of an *invariant* that provides knowledge about how to use a cache. In particular, invariants may be used to process calls to external packages *even if these calls were not previously stored explicitly in the cache.*
2. **Query Optimization:** We show how given any query Q to a mediator program M, we can rewrite the query and the mediator to a new query Q' and a new mediator M' respectively such that the answers to query Q w.r.t. mediated system M coincides with the answers to query Q' w.r.t. M' *and*

   Q' and M' make appropriate usage of

   - the cache and invariants,
   - existing, query rewriting techniques (e.g., pushing selections down, join reordering, etc.)

   In general given a query and a mediator, our rewriter constructs a number of viable rewritings of the query and the mediator. The rewritings represent possible *execution plans* and the optimizer has to choose one of them based on an estimation of their cost.
3. **Cost-Estimates:** Some of the external sources may have well-understood cost estimates for the queries

*This research was partially supported by the Air Force Wright Laboratory Aeronautical Systems Center under ARPA Contract F33615-93-1-1339, by the Army Research Office under grant DAAH-04-95-10174, by the Air Force Office of Scientific Research under grant F49620-93-1-0065, by ARPA/Rome Labs contract Nr. F30602-93-C-0241 (Order Nr. A716), and by an NSF Young Investigator award IRI-93-57756.

SIGMOD '96 6/96 Montreal, Canada
© 1996 ACM 0-89791-794-4/96/0006...$3.50

that are sent to them. For example, in relational DBMSs, cost models have well known characteristics [31, 32, 33].) However, in other cases, cost estimates may be hard to obtain – for example, in several domains that exist within HERMES (face recognition system, terrain reasoning system, transportation logistics planning system,video retrieval system) it is extremely difficult to develop a reasonable cost model. We have developed a *Domain Cost and Statistics Module (DCSM)* within which both kinds of domains (ones with good cost-estimation functions, as well as ones without) can be modeled based on actual performance. DCSM is based on storing statistics on previous calls to data sources, in order to estimate the cost of the calls that will be issued by a plan.

4. **Lossless and Lossy Summarizations:** If the size of the cached statistics becomes too large, we may encounter problems in maintaining them and efficiently accessing them. We show that statistics caches can be neatly "compacted" through the use of a special process called *summarization.* Two kinds of summarizations are introduced – *lossless* summarizations that reduce the size of a cache without losing any information that was found in the original cache, and *lossy* summarizations that compress cached statistics, but may lose some information in the process, thus compromising the quality of cost estimation. Our experiments compare the tradeoffs involved in lossy summarizations.

5. **Distributed Implementation and Experiments:** The algorithms described in this paper have been implemented in an experimental testbed on top of HERMES. We report on specific experiments that integrate data on 3-5 machines across the Internet (sites in Maryland, Cornell, Bucknell, and Italy). The experiments deal with the following packages – INGRES, flat files, and a special software package called AVIS for content-based video information (that has no well-understood cost estimation policies.) We will report on experiments comparing the use of caching with and without invariants, as well as the use of the DCSM, lossy and lossless compression schemes.

Note that HERMES uses a logic based language extended with special constructs to access external software. The details of the language can be found in [5, 28]. In HERMES, external programs are referred to as domains and they are viewed as black boxes that allow certain operations to be performed by outside sources. These operations are executed via domain calls of the form `d:f` where `d` is the name of the domain and `f` is a function name corresponding to a predefined operation that can be performed in this domain. Domain calls are expressed uniformly in HERMES with the help of a special predicate of the form `in(X,d:f(Args))` which can be read as: "Execute the domain function `d:f` on arguments `Args` and return the set of results in variable `X`."

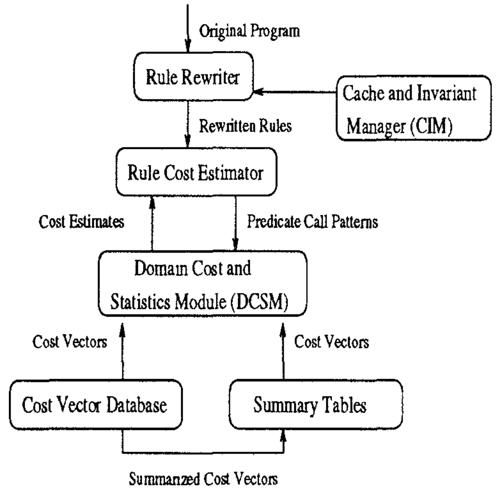

Figure 1: Architecture of the HERMES heterogeneous optimizer

2 An overview of the query optimizer and its architecture

This section provides an overview of the optimization architecture (Figure 1) and the function of all the modules. The complete description of the modules will be given in the following sections. The optimizer consists of four components described below.

First, the **rule rewriter** takes a program and a query as input, and finds different possible rewritings of the original program allowed by the possible adornments [35] of it. For simplicity we assume that domain calls are always ground, i.e. in a call of the form `in(X,d:f(Args))`, we require `Args` to be ground, but `X` can be either ground or variable. If `X` is a variable, then it is instantiated to an answer returned by `d:f(Args)`. Otherwise, we check if `X` is in the set of answers returned by the domain call. Hence, if `X` is ground, it can be used to prune the rest of the query. The rewriter also derives rewritings of the original query and program so as to use the cache and invariant manager module instead of actually calling the external domain.

The **cache and invariant manager** (`CIM` for short) module is used to maintain caches and to avoid actually calling an external domain when the answer to that call is physically present in the cache. The caches are usually scanned for exact matches. In addition to this, `CIM` uses expressions called invariants to find other acceptable entries in the cache – intuitively, an invariant specifies certain relationships between different calls. For example, if DC_1 and DC_2 are two different calls, and DC_1's answer is stored in the cache, and the invariants imply that all answers to DC_1 are also answers to DC_2, then we may use the cached answers to DC_1 to provide a partial answer for DC_2 - in a case of this kind, we may avoid the need to execute the domain call DC_2 altogether. Invariants were first introduced in [6]. In this paper, we concentrate on the performance issues related to invariants. The decision as to when to use `CIM` can be performed both online or offline. We investigate the conditions under which the cache is useful and how to use this information during optimization.

The next module is the **domain cost and statistics module** (DCSM for short). It is responsible for providing estimates of calls to external programs/sources. From now on, we will refer to these programs/sources as *domains*. The module keeps execution time and cardinality statistics in a database and provides cost estimates to the rule cost estimator. DCSM may keep very detailed tables of statistics information. Alternatively, it may maintain summarized tables.

DCSM is built as an extensible module. Hence, if a domain already provides a cost estimation module, the DCSM can be connected to it and avoid caching statistics for this domain. Hence, the estimates for calls to this domain will be directed to their respective domains.

Finally, the **rule cost estimator** takes the rewritten programs from the rule rewriter and computes the cost of each plan by obtaining the cost estimates of individual calls to the sources from DCSM and combining the results. The module then decides on the best plan for executing the given query. We will not give the details of rule rewriting and rule cost estimation due to space restrictions, [35] gives a detailed discussion on this subject.

In this paper we assume that there are two modes of operation for the mediator. The first mode is the *all answers mode* where the mediator calculates all the answers automatically. The second mode is the *interactive* mode, where the mediator calculates a first set of answers and presents them to the user. The mediator then asks the user if he wants to see more answers. If the answer is yes, the next set of answers is evaluated. The user has the choice of requesting all the remaining answers at any time.

3 Invariants and intelligent caching using invariants

We have seen above that domain functions are executed uniformly in the mediator through the use of the `in()` calls. Most of the time, however, these calls are costly operations. For example, the required domain may be located at a remote site, or the domain may charge an access fee per request, and so on. It is desirable to store the results of previous executions of costly operations. Note that caching only prevents making the same call more than once. However, in order to make better use of the caches, we plan to use specialized knowledge called "invariants."

Invariants are expressions that show possible substitutions for a domain call. Suppose we have a spatial index and we can perform range queries on it. The function **range** returns all the points within a given distance to a given point. Suppose we know that all points in the file "points" lie within a 100x100 square. Then we can write the following invariant:

```
Dist > 142 ⇒
      spatial:range(''points'',X,Y,Dist) =
            spatial:range(''points'',X,Y,142).
```

This says that given a very big range query, we can shrink it to the smallest admissible range query, i.e., a range of **142**. The equality in this invariant indicates that the two domain calls have identical answers. Next we write an invariant for a relational database, called **relation**), which supports a function, called **select** $<$, that given a table name, an attribute name, a comparison operator (e.g. $<$) and a value v, selects all the tuples from the given table where the given attribute stores a value smaller than v.

$$\mathtt{V1} <= \mathtt{V2} \Rightarrow \mathbf{relation}:\mathtt{select}(\mathtt{Table},\mathtt{Attr},<,\mathtt{V2}) \supseteq \mathbf{relation}:\mathtt{select}(\mathtt{Table},\mathtt{Attr},<,\mathtt{V1}).$$

This invariant says that given a call to **select**, we can replace it by another call to **select** with a smaller value. The relation between these two calls is not that of equality as in the previous example. Instead, it states that all the answers returned by `relation:select(File,Attr,<,V1)` will also be returned by `relation:select(File,Attr,<,V2)`. Hence, invariants are viewed as sound, but not necessarily complete rewrite rules, in our system. Invariants are intended to enhance the intelligent use of caches when processing a domain call. The query processor is expected to first check the cache to see if the answer for a domain call is already stored in it. Then, it will use the invariants to substitute domain calls and check if these calls are in the cache. If the invariants indicate that there is a domain call in the cache that provides a partial list of answers, then the actual domain call may need to be performed eventually. Even in this case, we expect to get a first set of answers quickly using the fast cache and invariant processing. In some cases, the user may not want the rest of the answers to his/her query and the actual domain calls may not need to be executed at all.

Formally, an invariant is an expression of the form:

$$\mathtt{Condition} \Rightarrow \mathtt{DomainCall}_1 \ \mathcal{R} \ \mathtt{DomainCall}_2$$

where $\mathcal{R}$ is one of $=, \supseteq$, and $\mathtt{DomainCall}_1, \mathtt{DomainCall}_2$ are two domain calls and `Condition` is a conjunction of atoms in the underlying language. We assume that the invariants only use simple conditions such as comparisons and contain no free variables in the invariants, i.e. all the variables in `Condition` appear either in $\mathtt{DomainCall}_1$ or in $\mathtt{DomainCall}_2$.

A cache consists of a list of ground domain calls of the form `domain:function(arg1,...,argN)` and the answer sets associated with each domain call. Hence, we may view the cache as a collection of pairs of the form (domain call, answer set). The domain call in this pair is used as the unique index to the answer set.

3.1 Query processing with caching and invariants

In this section we specify how domain calls are handled in the presence of caches and invariants. For this purpose, we are going to define a special program called "Cache and Invariant Manager" (`CIM`). During run-time (i.e., when we execute the plan) the **CIM** behaves like any other domain. Thus, no special operators are needed from the HERMES execution engine in order to retrieve data from the cache.

Suppose we execute the domain call **domain:function (arg1,...,argN)** in **CIM**. Then, the following operations take place in **CIM**:

- First **CIM** tries to match this call with one of the calls already in the cache. In this case, all the answers associated with the cached call are returned to the mediator and the cached entry replaces the actual domain call.
- In case there is no such entry in the cache, then **CIM** consults the invariants. Suppose the following is an invariant in **CIM**,

  Condition $\Rightarrow$ DomainCall_1 = DomainCall_2.

  and there exists a substitution θ where $\texttt{DomainCall\_1}\theta = \texttt{domain}:\texttt{function}(\texttt{arg1},...,\texttt{argN})$, $\texttt{Condition}\theta$ is true and $\texttt{DomainCall\_2}\theta$ is in the cache. In this case, the answer set for $\texttt{DomainCall\_2}\theta$ is passed on to the mediator and this set replaces the actual domain call.
- Finally, suppose both of the above two conditions are not satisfied, but **CIM** contains the following invariant,

  Condition $\Rightarrow$ DomainCall_1 $\supseteq$ DomainCall_2.

  and there exists a substitution θ where $\texttt{DomainCall\_1}\theta = \texttt{domain}:\texttt{function}(\texttt{arg1},...,\texttt{argN})$, $\texttt{Condition}\theta$ is true and there exists an entry $\texttt{DomainCall\_2}\theta$ in the cache. In this case, the answer set for $\texttt{DomainCall\_2}\theta$ is passed on to the mediator to provide a *subset* of the actual answer set for **domain:function (arg1,...,argN)**. If these answers are not sufficient, **CIM** must invoke the actual domain call.

Note that several decisions need to be taken when invoking the **CIM** module. For example, it is possible to make the actual domain call in parallel whenever a partial answer set is obtained. In this case, **CIM** is used to quicken the response time for the first set of answers. In the interactive mode, the partial set of answers may prove to be sufficient and the actual call may not need to be made. This may be accomplished since the query processor stops the execution of all the running external programs when they are no longer needed. The advantage of having a separate cache and invariant manager is that it is possible to build special purpose caches for different domains, hence making the overall system very flexible.

The run-time query processor for the mediator does not need to know of the existence of the caches and the invariants. All the processing of invariants is done in the **CIM** module. We only need to rewrite the mediator and direct the relevant calls to **CIM** instead of actual domains where **CIM** is expected to provide better performance. Such rewriting decisions are made prior to query execution. In addition, the rule rewriter and cost estimator explore the possibility of directing domain calls to **CIM** during the optimization process.

3.2 Maintaining Invariants

As discussed earlier, invariants are introduced into the system to enhance the use of caches. They specify which calls can be substituted with others. This information must be provided by someone who is knowledgeable about the domain in question, like the mediator author. It is possible to write very general invariants with very little information about specific domains. For example the invariant:

$$\Rightarrow \texttt{text}:\texttt{search}(\texttt{X},\texttt{Y}) \supseteq \texttt{text}:\texttt{title}(\texttt{X},\texttt{Y}).$$

specifies that **search** and **title** perform similar functions. This is probably due to the fact that **search** considers the whole document including the **title**. Note that both functions may be used in the mediator for efficiency reasons. This invariant just enables the system to use the caches for previous **title** searches in relation to a call to **search**. A more specific invariant is the following:

$$\Rightarrow \texttt{text}:\texttt{search}('\texttt{file1}',\texttt{Y}) = \texttt{text}:\texttt{search}('\texttt{file2}',\texttt{Y}).$$

In this case, we are stating that **file2** is a mirror image of **file1**. As invariants get more and more specific, deeper understanding of the operations and information stored in the domain may be necessary. Examples of such invariants were given earlier in this section. Generally, we expect that the mediator author will not allow the set of invariants to grow unmanageably large. Otherwise, the overall performance of the system may fall. In this case, it may be necessary to choose a subset of the invariants that are used frequently and found to be useful. To determine the use of invariants, the system may be set to observe how often an invariant is used successfully, the number of tuples provided and the execution time. A performance function may be invoked with these parameters and those invariants whose performance appears to be above a certain threshold are kept in the system. The details of this evaluation process is out of the scope of this paper.

We note that invariants are required to be correct statements about the domains they are referring to. Otherwise, it may not always be possible to verify the correctness properties since the set of possible values for their arguments may be infinite or unknown, especially for non-traditional domains.

4 Rule Rewriter

The **rule rewriter** (see Figure 1) transforms the rules of the program P, that contains the query and the mediator specification, into equivalent programs, that will reflect plans, by applying one of the following transformations:

1. Replace a subgoal G with a call to the cache and invariant manager.
2. Push selections to the source.
3. Rearrange the order of the subgoals of the rule, as long as it is compatible with the permissible adornments of every domain call.

Note that the rule rewriter processes only the rules that will be used for answering the query. Let us illustrate the rule rewriter's workings with the following example.

Example 4.1 Consider the following mediator (M1).

```
(M1) (R2) m(A,C) :- p(A,B), q(B,C).
     (R3) p(A,B) :-in(Ans, d1:p_ff()),
                    =(Ans.1, A), =(Ans.2, B).
     (R4) p(A,B) :- in(B, d1:p_bf(A)).
     (R5) q(B,C) :- in(Ans, d2:q_ff()),
                    =(Ans.1, B), =(Ans.2, C).
     (R6) q(B,C) :- in(C, d2:q_bf(B)).
```

Let us also consider the query (Q7)

```
(Q7) ?-m(a,C)
```

The query rewriter first adorns the predicates in a way that indicates the in-going and outgoing arguments of every predicate. The former are annotated with a `$b`, that stands for "bound", and the latter with `$f`, that stands for "free". Then, the subgoals of a rule are reordered in all possible ways, provided that there is a corresponding adornment. In our example, the query rewriter develops two programs that can compute the query. The first one, (P8) assumes that first we obtain all `B` bindings from domain `d1` and then we pass `B` bindings to `d2` and obtain corresponding `C` bindings. Note that the rewriter pushes the condition `A = a` to the source. Consequently, it projects out the attribute `A` of the `p` predicate. To avoid confusion, we replace `p` with $\mathtt{p}^{\mathtt{a},\$\mathtt{f}}$ where `a` is a reminder that we have projected the `A` attribute that would always be equal to `a`. In general, our query rewriter performs all the traditional algebraic optimizations (push selections and projections down) but we will not further deal with this set of optimizations in this paper. ([35] provides an extensive list of algebraic optimizations that can be applied.)

(P8) (R9) $\mathtt{m}^{\mathtt{a},\$\mathtt{f}}(\mathtt{a},\mathtt{C}):-\mathtt{p}^{\mathtt{a},\$\mathtt{f}}(\mathtt{B}),\mathtt{q}^{\$\mathtt{b},\$\mathtt{f}}(\mathtt{B},\mathtt{C}).$
(R10) $\mathtt{p}^{\mathtt{a},\$\mathtt{f}}(\mathtt{B}):-\mathbf{in}(\mathtt{B},\mathtt{d1}:\mathtt{p\_bf}(\mathtt{a})).$
(R11) $\mathtt{q}^{\$\mathtt{b},\$\mathtt{f}}(\mathtt{B},\mathtt{C}):-\mathbf{in}(\mathtt{C},\mathtt{d2}:\mathtt{q\_bf}(\mathtt{B})).$

The second plan, (P12), assumes that we first obtain `B` and `C` bindings from `d2` and then we pass `B` bindings to `d1`.

(P12) (R13) $\mathtt{m}^{\mathtt{a},\$\mathtt{f}}(\mathtt{a},\mathtt{C}):-\mathtt{q}^{\$\mathtt{f},\$\mathtt{f}}(\mathtt{B},\mathtt{C}),\mathtt{p}^{\mathtt{a},\$\mathtt{b}}(\mathtt{B}).$
(R14) $\mathtt{p}^{\mathtt{a},\$\mathtt{b}}(\mathtt{B}):-\mathbf{in}(\mathtt{X},\mathtt{d1}:\mathtt{p\_bb}(\mathtt{a},\mathtt{B})).$
(R15) $\mathtt{q}^{\$\mathtt{f},\$\mathtt{f}}(\mathtt{B},\mathtt{C}):-\mathbf{in}(\mathtt{B},\mathtt{C},\mathtt{d2}:\mathtt{q\_ff}()).$

Assuming that the rule rewriter derives more than one plan for a query (something expected in all but the most trivial mediators) we have to estimate the cost of each plan and select the best. This is the task of the DCSM module, which is presented in the next section.

5 Domain Cost and Statistics Module (DCSM)

As discussed in the introduction, heterogeneous systems necessitate the development of different cost estimate strategies. In optimization of relational queries, typically we have extensive statistics about the relations (e.g., select/project selectivities, cardinalities, and so on) and we also understand the behavior of the basic operators (select, project,...) Hence, cost estimators can be customized for the specific domain. This is not however a reasonable assumption for a general purpose cost estimator of a heterogeneous system. A system like HERMES may integrate arbitrary domains whose internals we do not know in general. Furthermore, the domains may be non-proprietary and hence we may not be able to access statistics information even if it exists. Sometimes, even the developer of these systems may not know the appropriate cost functions. In addition, the access to a domain at a remote site may vary greatly from time to time because of network delays.

Recall that the mediator in the HERMES system only knows a set of functions for any given domain, their input/output types and how to invoke these functions. The mediator may not know the function that best characterizes the time it takes to evaluate the calls. Hence using curve fitting techniques [36] to approximate the costs may not be practical since we do not know the shape of the function. Also, cost functions do not easily adapt to abrupt and unexpected changes in the costs of domain calls. In this system, we provide a general cost estimation technique that can adapt to the behavior of the underlying system easily. We now explain the DCSM module in greater detail.

The `DCSM` module provides cost estimates for domain calls. In particular, it provides a single function called `cost` which given a domain call pattern, returns an estimate of the cost of executing the given domain call.

A *domain call pattern* is an expression of the form `domain:function(Arg1,...,ArgN)` where `ArgI` is either a constant or the special symbol `$b` which stands for `bound`. Whenever `$b` appears at position `I` of a domain call pattern, it means that we know `ArgI` is bound but its exact value is not available. For example, the call `DCSM:cost(d:f(5,$b))` is asking the `DCSM` module for cost estimates of a domain call `d:f` where the first argument is `5` and the second argument is some constant that we do not know yet. Recall that we assume all domain calls are ground before they are executed, hence there cannot be a free variable in a domain call pattern.

Similarly, we define *predicate patterns*. Predicate patterns may contain the symbol `$f` to indicate that a corresponding variable may be free when this predicate is evaluated. For example, a predicate pattern of the form : $\mathtt{p}^{\$\mathtt{f},\$\mathtt{b},\mathtt{a}}$ indicates that the first argument of a three place predicate is free, the second one is bound and the third one is a known constant `a`.

A **cost estimate** (associated with a domain-call pattern or a predicate pattern) is a cost vector of the form $[\mathtt{T_F},\mathtt{T_A},\mathtt{Card}]$ where $\mathtt{T_F}$ is the (estimated) time required to retrieve the first answer, $\mathtt{T_A}$ is the (estimated) time to find all answers, and `Card` is the (estimated) cardinality of the answer set. It is possible that a specific cost estimator is available for some domain but this estimator does not provide some of the parameters mentioned above. Then the missing parameters can still be provided by the DCSM module while getting the others from the better estimator easily. From now on, we will restrict our attention to domains with no cost estimation capabilities. We now start describing the basic components of the DCSM module.

5.1 Cost vector database

This database records cost information about domain calls as they get executed by the mediator. In the simplest version for each domain call it contains a triple of the form (`domain call, cost vector, record_time`), where `record_time` is the *actual* time the call was recorded in the database. For simplicity, we will ignore the `record_time` information for now. Hence, the cost vector database consists of tables for different domain calls, where the columns correspond to the time to compute the first answer, time to compute all the answers, the cardinality of the answer and the arguments to which this values correspond to. Some of this information may not be available for some domain calls since all answers may not have been obtained (e.g. the mediator may have been working in interactive mode and the user stopped the query execution). We now give some example tables that will be used throughout the rest of the paper to illustrate the working of the DCSM module.

Example 5.1 Let us reconsider the mediator (M1), the query (Q7) and the two candidate plans (P8) and (P12). In order to estimate the cost of the two plans we have to estimate the cost of the domain calls `d1:p_bf`, `d1:p_bb`, `d2:q_bf` and `d2:q_ff` that appear in the two plans. Let us assume that the tables (T16), (T17), (T18) and (T19) of Figure 2 describe the total execution time and the cardinalities of `d1:p_bf`, `d1:p_bb`, `d2:q_bf` and `d2:q_ff` calls that have been issued in the past. Note that the same value for an argument may appear more than once in the tables corresponding to different calls. Note also that for simplicity of presentation we include only the attributes `Card` and T_A while in general we also have the response time to obtain the first answer and the time when the call was issued.

Now, we may estimate the cost of a domain call, e.g., `d1:p_bf(a)`, for the execution time to all the answers, by taking the average of the two entries in the table (T16) of Figure 2, namely 2.00 and 2.20 to get 2.10. We may also estimate the cost of a domain call where we do not know one or more of the parameters. For example, consider the call `d1:p_bf($b)`. We can estimate its cost by taking the average, i.e. (2.00+2.20+2.80+2.84)/4.

5.2 Summary Tables

Though the tables of Figure 2 have the necessary information, there are three important problems regarding their use and maintenance:

- **fully detailed statistics information:** Keeping the full statistics data of all the calls puts a heavy burden on storage.
- **expensive aggregation functions:** We repeatedly apply computationally expensive aggregation functions – in our examples, the average function. Thus, the time required for calculating the cost may be prohibitively long.

(T16) d1:p_bf(A)

A	Card	T_A
a	4	2.00
a	5	2.20
c	8	2.80
c	8	2.84

(T17) d1:p_bb(A,B)

A	B	Card	T_A
a	g	0	2.50
a	d	1	2.70
c	g	1	2.68
c	d	0	2.65

(T18) d2:q_bf(B)

B	Card	T_A
g	40	50.0
g	41	51.0
g	39	49.0
d	30	48.0
d	35	52.0

(T19) d2:q_ff()

Card	T_A
100	50.0
95	48.0
105	52.0

Figure 2: Tables in the cost vector database

(T20) d1:p_bf(A)

A	Card	T_A	1
a	4.5	2.10	2
c	8	2.82	2

(T21) d2:q_ff()

Card	T_A	1
100	50.0	3

Figure 3: Summarization of tuples with identical values for the dimensions attributes

In the following subsections we will show how we solve the above problems using off-line *summarizations* of the statistics information stored in the cost vector database.

5.2.1 Loss-less Summarizations

As we have seen above, the cost vector database contains very detailed statistics that make it very hard for the DCSM module to analyze and maintain relevant cost information for domain calls. In many cases we may summarize the statistics tables without losing any information that may be useful during cost estimation, i.e., any statistics question posed by the cost estimator will have the same answer on the summarized table and the original table. We call these summarizations *loss-less.* For example, the summarization of the table (T16) of Figure 2 with the table (T20) of Figure 3 and the corresponding summarization of the table (T19) of Figure 2 with the table (T21) of Figure 3 are loss-less. In effect, the tuples with `A='a'` (or `A='c'`) have been aggregated into a single tuple. The `1` attributes indicate the number of original table tuples that correspond to the summarized table tuples.

The example suggests a rather straightforward summarization procedure:

1. Split the attributes of every statistics table into a set of *dimensions* that consist of all attributes of the corresponding call, and the set of *metrics* that reflects the response time of the call, and the cardinality of the result. (Note that in general we may have more metrics attributes than response time and cardinality.) In our running example, the set of dimensions of table (T16) is {`A`} and the set of metrics is {`Card`, T_A}.
2. For all tuples that have identical values $d_1, d_2, \ldots, d_m$ on the dimension attributes, aggregate the metrics

(T22)

d1:p_bb(A,$b)			
A	Card	T_A	1
a	0.5	2.60	2
c	0.5	2.665	2

(T23)

d1:p_bf($b)		
Card	T_A	1
37	50.0	5

Figure 4: Dropping dimensions attributes whose bindings we can not predict

attributes into a single pair of average response time T_A and average cardinality Card and create a single tuple $(d_1, d_2, \ldots, d_m, \mathtt{Card}, T_A, 1)$, where 1 is the number of original table tuples that have been aggregated into the specific tuple.

5.2.2 Lossy Summarizations

The summarization described allows us to avoid the expensive average aggregation only when all the arguments of a domain call are set to constants. When, however, some constant is known to be bound, but its specific value is not known, we still need aggregation. Suppose for example, we want to estimate the time it will take to execute the call d1 : p_bf($b) based on table (T20) in figure 3. Then, again the most general conclusion we can draw is the average of all the tuples for this table. Hence, we could have precomputed this average and store it as shown in (T23). (T22) demonstrates the summarization of (T17) that removes the dimension B.

Indeed, there are cases where dropping a dimension attribute does not cause information loss because we can not have during optimization values of this attribute. Let us motivate this idea by the following example.

Example 5.2 Recall the mediator given in example 5.1. The tables (T17) and (T18) of Figure 2 contain in their dimensions set the attribute B, i.e., they provide the expected response times and cardinalities for specific values of the B attribute. However, if we assume that the predicates p and q of example 5.1 are "hidden" from the user, then it is impossible that the cost estimator will ever ask the response time for a specific B value. The reason is that we can not know the specific B values until we start executing the program and obviously by that time it will be too late to undo our decisions. Thus, we can remove the B attribute from the dimensions set and derive the summarized tables of Figure 4.

The intuition that allowed us to remove B from the dimensions set can be implemented by a procedure that inspects the given mediator program and decides which attributes may ever be instantiated to a specific constant during the rewriting phase. All attributes that can never be instantiated to a specific constant are dropped from the dimensions sets. Similarly, we can watch the access patterns for the tables and decide which tables are needed very frequently and decide to create these tables. Alternatively, drop the tables that are not accessed very often.

Summarization has a dual purpose; first, it *reduces the storage space needed for statistics.* Second, it provides *fast responses* to the questions of the cost estimator. We have the option of maintaining either summary tables and providing for rough and out-of-date estimates but saving time and space, *or* using the cost vector database for all purposes which is very time and space consuming. In Section 7 we give the experimental results for the utility of the DCSM module. We note here that, it is possible to perform the summaries in a more biased fashion, especially for the remote domain calls by observing the load of the network, by giving precedence to more recent statistics. Currently we are exploring these possibilities.

5.3 Cost Estimation using Cost Vector Database and Summary Tables

Now, given a domain call to the DCSM module, we describe how we can make use of the cost vector database and the summary tables to estimate the cost of the given call. At any given time we may have a couple of different tables for a domain call d:f. Having these tables does not guarantee that we can estimate the cost of the given call pattern without any calculations. The following example illustrates this point.

Example 5.3 Suppose we have a three place domain call d:f(A,B,C). For this domain call, we have three tables; namely d:f(A,B,C), d:f($b,B,C), d:f($b,$b,C), and d:f($b,$b,$b). Now, we want to estimate the cost of the call d:f(a,$b,2) by a simple table lookup. (Note that the table d:f($b,B,C) means that the variables B and C are set to known constants where the first argument is only known to be bound. Similarly, d:f(A,B,C) means that all the arguments are set to known constants.)

The table d:f(A,B,C) in the cost vector database can be used for this call, but it involves performing aggregate operations and hence we may want to avoid using it. Then, we look to see if there is a table for d:f(A,$b,C). We see that there is no such table. Next, we relax our call and look to see if we have table d:f($b,$b,C) or d:f(A,$b,$b). We see that we have d:f($b,$b,C), hence we look in the table for the entry d:f($b,$b,2). Suppose now that there is no entry for C=2 in this table. Then, we relax the call one more time and look to see if we have the table d:f($b,$b,$b) which is the average of all the information for this domain call. Since we have it, we look up the only tuple and get our estimate.

The complete procedure for estimating a domain call's cost in the most lossless way, given a collection of possibly summarized tables, is given by the following steps. Let us assume without any loss of generality that the call has the form $\mathtt{p}(c_1, \ldots, c_n, \$b, \ldots, \$b)$.

1. Set $i = 0$.
2. Find a table s whose set of dimension attributes contains the first $(n - i)$ columns of p.
3. Find the specific tuple $s(c_1, \ldots, c_n)$ of s. If it is found, return it to the cost estimator.
4. Set $i = i + 1$.

5. Nondeterministically replace i of the constants $c_1, \ldots, c_n$ with symbol `$b` and execute steps 2 and 3, until a match is found or there exists another way of replacing i constants with the symbol `$b`.
6. If $i < n$, go to step 4.

An important observation to make is that lossy tables are not created during the optimization and cost estimation process. Similarly, they are not updated every time the cost vector database is updated. They are simply consulted for estimates. At any given point in time, two tables must be present in the DCSM module for every domain function `f`; the cost vector database and most lossy summarization table, i.e. all arguments of `f` set to bound. As subsequent optimization operations are performed, certain less lossy tables may be requested frequently. Then, DCSM may choose to create those tables. The decision is made based on a quality function that takes into account the space and offline time required maintain this table and the refinement in the estimates provided by them. Different heuristics can be used to develop this function.

6 Rule Cost Estimator

The **rule cost estimator** associates cost and statistics information with every rule that was output from the rule rewriter starting from the query. The rule cost estimator invokes a function f_e, that, given the cost vectors of the subgoals of a rule, estimates the cost vector of the head.

Example 6.1 Recall the mediator given in example 5.1 and the plans generated for this mediator in example 4.1. Let us assume that the mode of operation is all answers to the query `?-m(a,C)` as given in example 5.1. Now, we have access to the following pieces of information:

- The expected time to all the answers ($\mathtt{T_A}(\mathtt{p}^{\mathtt{a},\$\mathtt{f}})$) for computing $\mathtt{p}^{\mathtt{a},\$\mathtt{f}}$, or equivalently $\mathtt{in(B, d1 : p\_bf(a))}$, and the expected cardinality (`Card(d1:p_bf(a))`) of `d1:p_bf(a)` tuples.
- The estimated time $\mathtt{T_A}(\mathtt{q}^{\$\mathtt{b},\$\mathtt{f}})$ for $\mathtt{q}^{\$\mathtt{b},\$\mathtt{f}}$, or equivalently for $\mathtt{in(C, d2 : q\_bf(B))}$.

We may estimate the cost of executing (P8) by the following formula, that considers that we first execute `in(B, d1:p_bf(a))` that takes time $\mathtt{T_A}(\mathtt{p}^{\mathtt{a},\$\mathtt{f}})$ and then we issue `Card(d1:p_bf(a)) in(C, d2:q_bf(B))` calls that each one takes $\mathtt{T_A}(\mathtt{q}^{\$\mathtt{b},\$\mathtt{f}})$. Thus, the cost is given by formula 1.

$$\mathtt{T_A}(\mathtt{p}^{\mathtt{a},\$\mathtt{f}}) + (\mathtt{Card(d1 : p\_bf(a))})(\mathtt{T_A}(\mathtt{q}^{\$\mathtt{b},\$\mathtt{f}})). \tag{1}$$

Similarly, we may estimate the cost of executing (P12) by the formula 2.

$$\mathtt{T_A(d2 : q\_ff())} + (\mathtt{Card(d2 : q\_ff())})(\mathtt{T_A(d1 : p\_ab(a,B))}). \tag{2}$$

where $\mathtt{T_A(d2 : q\_ff())}$ is the time needed for executing `in(Ans, d2:q_ff())`, $\mathtt{Card(d2 : q\_ff())}$ is the number of `Ans` tuples that we receive from the `in(Ans, d2:q_ff())` call, and $\mathtt{T_A(d1 : p\_ab(a,B))}$ is the time to execute an `in(X, d1:p_ab(a,B))` call.

Consider a query $\mathtt{p}(\mathtt{t_1}, \ldots, \mathtt{t_m})$ that involves using a rule R having head $\mathtt{p}(\mathtt{s_1}, \ldots, \mathtt{s_m})$. The processing of this query causes certain arguments in the head of R (i.e. certain $\mathtt{s_i}$'s) to become bound, while others are free. Suppose we wish to estimate the cost vector of rule R with respect to this particular call (and hence with respect to the bindings generated by this call). Suppose the body of R (suitably instantiated w.r.t. this call) is $\mathtt{g_1}, \ldots, \mathtt{g_n}$ (in that order). Our estimation procedure uses the following steps:

1. If $\mathtt{g_i}$ is of the form $\mathtt{in(X, d : f(arg1, \ldots, argk))}$, then we convert $\mathtt{g_i}$ into a domain-call pattern $\mathtt{d : f(a_1, \ldots, a_k)}$ where $\mathtt{a_i} = \mathtt{argi}$ if $\mathtt{argi}$ is a constant and $\mathtt{a_i} = \$\mathtt{b}$ otherwise. Then, we invoke the call $\mathtt{DCSM : cost(d : f(a_1, \ldots, a_k))}$ and obtain the cost vector for this call pattern.
2. If $\mathtt{g_i}$ is an IDB predicate then compute the cost vector of $\mathtt{g_i}$ by recursively invoking the described procedure for the rules defining $\mathtt{g_i}$ and then adding up the cardinalities and the execution times of the results produced by each rule.
3. Assuming that
   (a) we implement the join of the subgoals using nested-loops with left to right order, and
   (b) we perform no duplicate elimination, i.e., for every result we receive from $\mathtt{g_i}$ we issue a call to $\mathtt{g_{i+1}}$ regardless of whether we have issued again this call[1]

   we can associate with the body of the rule the cost vector

$$[\mathtt{T_{A_t}}, \mathtt{T_{F_t}}, \mathtt{Card_t}] = [\Sigma_{i=1,\ldots,n} \mathtt{T_{A_i}} \Pi_{j=1,\ldots,i-1} \mathtt{Card_j}, \Sigma_i \mathtt{T_{F_i}}, \Pi_i \mathtt{Card_i}]$$

   Assuming that we do no duplicate elimination we can write

$$[\mathtt{T_A}, \mathtt{T_F}, \mathtt{Card}] = [\mathtt{T_{A_t}}, \mathtt{T_{F_t}}, \mathtt{Card_t}]$$

7 Implementation and Experimental Results for the Hermes Optimizer

The HERMES system currently integrates 3 relational DBMSs (Paradox, DBase and Ingres), one object-oriented DBMS (ObjectStore), multimedia packages (MACS and AVIS), a US Army path planning package, a face recognition package, as well as flat file data, text databases (in particular a USA Today news-wire corpora), and a spatial database. It runs on the Unix/Xwindows platform as well as on the

[1] Note, caching gets around the disadvantages of combining duplicate elimination and pipelined nested-loops.

PC/Windows platform. The system is currently capable of accessing data distributed at several selected sites across the Internet (in USA, Europe and Australia).

In order to determine the performance of the algorithms described here, we ran a number of experiments. For space reasons, we report below only a small set of experimental data that is representative of the totality of the experimental results obtained. All timing values are given in milliseconds and they show the "query initialization + wait for response + display the results" times.

Executing Remote Calls with Caching and/or Invariants: Figure 5 shows a small representative sample of the times obtained when running queries that required accessing data/operations in a video retrieval package called AVIS [3]. It is easy to see from these figures that using caches always leads to savings in time when the software/data is located at remote sites. Furthermore, using invariants is useful when the query is not explicitly cached – in such cases both partial invariants and equality invariants lead to significant savings in time over actually making the call. We found partial invariants to be always useful, unless the size of the partial answer returned plays a significant role. CIM must keep the answers from the cache in memory and compare them with the answers from the actual call. We also found the overhead of checking the cache and the invariants without success and making the actual call to be negligible in our experiments.

The Utility of DCSM: The table in figure 6 shows our results on the utility of the DCSM. In particular, we show, for a representative set of queries that interoperate between AVIS and INGRES data located across the network, the times taken to compute the first answer and all answers. In each of these two cases, three times are shown: (1) the actual running time of the query, (2) the running time of the query as predicted by the DCSM using Lossless Tables, and (3) the time taken for the query as predicted by the DCSM using Lossy Tables obtained by dropping all the attributes of the cached functions. In the table below, each of queries i and i' are "equivalent" in the sense that query i' is a rewriting of query i. The actual queries are listed in the appendix. The cost vector database (lossless) contains about 20 different instantiations for the arguments of a domain call in the corresponding tables.

	First Answer			All Answers		
Query	Actual Time	Lossless DCSM	Lossy DCSM	Actual Time	Lossless DCSM	Lossy DCSM
1	2245	2487	2666	2439	2647	3033
1'	2384	2487	2666	9958	10825	14346
2	14054	2681	2622	55432	52725	61185
2'	3834	2681	2622	14213	27861	32233
3	2620	1378	1319	4651	4479	4520
4	3187	1335	1276	10485	9269	9515

Figure 6: The Utility of DCSM

There are several points to be noted when examining the above tables. First, when we look at the times taken to compute All Answers, the Lossy and the Lossless DCSM predictions closely match the actual running times (though it is certainly not perfect in its predictions, e.g. the case of query2'). The DCSM errs both ways, sometimes over-predicting the time taken, and sometimes under-predicting the time taken, Lossy tables do worse mainly as a result of the discrepancy between the expected and the real cardinalities of the outputs.

When looking at the figures for computing the "first" answer, DCSM's predictions are often good, yet in some cases, it can vastly under-predict the actual times taken. These are cases when it is hard to predict the amount of "backtracking" that the HERMES system might take in actually processing a derived query. The rule cost estimator calculates the cost of calculating predicates as if the first answer is going to be found by combining the first answers returned for the calls made to compute it. In reality, the amount of time spent on backtracking cannot be neglected as our experiments have shown. One way to remedy this solution can be to cache, especially the time for the first answer of predicates in the same way we cache statistics for domain calls.

Our experience, supported by the experimental figures shown above also implies that when $Q1$ is a rewriting of $Q2$:

1. If we want all answers, and DCSM predicts Q1 is better than Q2, then we have found that Q1 almost always runs much faster than Q2. Furthermore, the predicted values and the real values are quite close to one another.
2. The situation is slightly stranger when first answers are being computed. If DCSM predicts Q1 is better than Q2 by at least a 50% margin, then Q1 is usually runs faster than Q2. However, if DCSM predicts Q1 is better than Q2 by a small margin, then the results are unpredictable; in some cases Q1 executes faster, while in others Q2 may do much better.

8 Related Work and Conclusions

There is now a great deal of work in mediated systems techniques. For example, there have been several efforts to integrate multiple relational DBMSs[10, 21] and relational DBMSs, object-oriented DBMSs and/or file systems [11, 15, 24, 16, 17]. Our approach in this paper differs from the above approaches in the following ways: first, in most of the above approaches, there are well-developed cost models for evaluating the behavior of queries. In contrast, in our framework, we wish to mediate between arbitrary "non-traditional" databases (including face databases, video repositories, databases of plans for transportation logistics, etc.) where such cost models are not always available. Furthermore, when cost models are available, we would like to take maximal advantage of them as well. Second, our notion of an invariant is unique and applies in a uniform way to heterogeneous data "exchanged" during computation of complex queries that apply to multiple data sources. Third, we have, presented experimental results that apply not only to heterogeneous databases consisting of "traditional" sources, but also a number of "non-traditional" sources.

Query	Type	Time for First Ans.	Time for All Ans.	Comments (sites)
Find all actors	no cache	1776	2581	USA
in "The Rope"	no inv.	48374	49039	Italy
	cache only	300	1021	USA & Italy
	cache + equality inv.	873	1646	USA & Italy
result: 6 tuples (421 bytes)	cache + partial inv.	501	2490	USA
(22 bytes from partial inv.)		321	73175	Italy
Find all frames in	no cache	1459	2756	USA
"The Rope" in which	no invar.	11023	12158	Italy
Phillip appeared.	cache only	351	1405	USA & Italy
	cache + equality inv.	1807	2775	USA & Italy
result: 20 tuples (3108 bytes)	cache + partial inv	1983	2073	USA
(421 bytes from partial inv.)		1941	16553	Italy
Find the objects that appear	no cache, no invar.	1420	2319	USA
between frames 4 and 47	cache only	326	1153	
in "The Rope"	cache + equality inv.	504	1386	
	cache + partial inv.	578	2989	USA
result: 19 tuples (182 bytes)	no cache, no inv.	6600	7526	Italy
(130 bytes from partial inv.)	cache + partial inv.	439	7795	Italy
Find the objects that appear	no cache, no inv.	1178	2426	USA
between frames 4 and 127	cache only	357	1450	
in "The Rope"	cache + equality inv.	709	1960	
	cache + partial inv.	431	4092	Italy
result: 24 tuples (247 bytes)	no cache, no invar	3926	4941	Italy
(130 bytes from partial inv.)	cache + partial inv.	447	7273	Italy

Figure 5: Executing Remote Calls with Caching and/or Invariants

Cost based optimization in mediated systems is a novel problem that is different from traditional distributed query optimization. An extensive discussion of the differences and the need for novel research in the area of optimization in mediated systems appears in [44]. The most important difference is the absence of statistics of non-proprietary sources. [42, 43] find out the performance behavior of a non-proprietary source by probing it with carefully organized sample queries and applying regression methods for estimating various parameters of a predetermined cost model. Their method is very effective but it is inapplicable when we do not have a predetermined cost model. This is the case with many unconventional sources. For example, it is very difficult to generate a cost model for the face recognition or the video retrieval or terrain reasoning/path planning sources of HERMES.

Work on caching in databases has been done extensively through the notion of a *materialized view* [1, 2, 8, 9, 12, 13, 22, 23, 25, 26]. These papers show how views (and their materializations) may be defined for different kinds of databases such as relational DBMSs, object-oriented DBMSs, and object-relational systems. However, it is only recently that materialized views were studied in the context of mediated systems [19]. Consequently, very little work has been done on how to effectively use such materialized mediated views to effectively process queries [33, 34, 39, 40]. A materialized mediated view may be viewed as a domain cache and hence, all the algorithms in this paper deal with how to effectively use such caches to process queries (and optimize them) in a distributed heterogeneous database management system. In addition to this work, there has been work on caching in the deductive database community through the use of OLDT-resolution [37, 38]. Our work effectively shows how such caches may be defined when views access non-logical data representations and software packages and furthermore, through the use of invariants, shows how such caches may be effectively used.

References

[1] S. Abiteboul and A. Bonner. (1991) *Objects and view.*, In Proc. of the ACM SIGMOD Conf. on Management of Data, pp. 238–247, 1991.

[2] S. Abiteboul, S. Cluet, and T. Milo. (1993) *Querying and updating the file.*, In Proc. Int. Conf. on Very Large Data Bases (VLDB), pp. 73–84, 1993.

[3] S. Adalı, K.S. Candan, Su-Shing Chen, K. Erol and V.S. Subrahmanian. (1995) *Advanced Video Information System: Data Structures and Query Processing.*, Accepted for publication in the ACM-Springer Multimedia Systems Journal.

[4] S. Adalı and V.S. Subrahmanian. (1994) *Amalgamating Knowledge Bases, III: Algorithms, data structures and query processing.* Technical Report CS-TR-3124, Computer Science Department, University of Maryland, Aug 1993. Accepted for publication in Journal of Logic Programming. (http://www.cs.umd.edu/projects/hermes/ publications/abstracts/akbiii.ps)

[5] S. Adalı and R. Emery. (1995) *A Uniform Framework For Integrating Knowledge In Heterogeneous Knowledge Systems,* Proc. of the Eleventh International Conference on Data Engineering, pp. 513-520. (http://www.cs.umd.edu/projects/hermes/ publications/abstracts/cons.ps)

[6] S. Adalı and V.S. Subrahmanian. (1995) *Intelligent Caching in Heterogeneous Reasoning and Mediator Systems,* Proc. of the Second International Conference on Building and Sharing of Very Large-Scale Knowledge Bases (ed. N. Mars), pps 247–256, IOS Press, Twente, The Netherlands, May 1995.

[7] J. Blakeley, N. Coburn, and P.-A. Larson. (1989) *Updating derived relations: Detecting irrelevant and*

autonomously computable updates., ACM Trans. on Database Systems, 14(3):369–400, 1989.

[8] Stefano Ceri and Jennifer Widom. *Deriving Production Rules for Incremental View Maintenance.*, Proc. of the 17th VLDB Conference, 1991.

[9] U. Dayal. (1989) *Queries and views in a object-oriented databases.*, In Int. Workshop on Database Programming Languages, 1989.

[10] U. Dayal and H. Hwang. (1984) *View definition and generalization for database integration in a multi-database system.*, IEEE Trans. Software Eng., SE-10(6):628–644, 1984.

[11] N. Gehani, H. Jagadish, and W. Roome. (1994) *OdeFS: A file system interface to an object-oriented database.*, Proc. Int. Conf. on Very Large Databases (VLDB), pp. 249–260, 1994.

[12] Ashish Gupta, Dinesh Katiyar, and Inderpal Singh Mumick. (1992) *Counting Solutions to the View Maintenance Problem.*, In Workshop on Deductive Databases, JICSLP, 1992.

[13] A. Gupta, I.S. Mumick and V.S. Subrahmanian. (1993) *Maintaining Views Incrementally.*, Proc. 1993 ACM SIGMOD Conf. on Management of Data, Washington, DC.

[14] E. Hanson. (1987) *A performance analysis of view materialization strategies.*, In Proc. of the ACM SIGMOD Conf. on Management of Data, pp. 440–453, 1987.

[15] A. Kemper, C. Kilger, G. Moerkotte. (1994) *Function Materialization in Object Bases: Design, Realization, and Evaluation.*, IEEE Transactions on Knowledge and Data Engineering, Vol.6, No.4, August 1994.

[16] Laks V.S. Lakshmanan, F. Sadri amd I.N. Subramanian. (1993) *On the logical foundations of schema integration and evolution in Heterogeneous Database Systems.*, Proc. DOOD-93, Phoenix, Arizona.

[17] Laks V.S. Lakshmanan, F. Sadri amd I.N. Subramanian. (1995) *Logic and Algebraic Languages for Interoperability in Multidatabase Systems*, submitted to Journal of Logic Programming.

[18] S. Leach and J. Lu. (1994) *Computing Annotated Logic Programs.*, Proceedings of the 11th International Conference on Logic Programming (ed. P. Van Hentenryck), MIT Press, pps 257-271.

[19] J. Lu, G. Moerkotte, J. Schue, V.S. Subrahmanian. (1995) *Efficient Maintenance of Materialized Mediated Views*, Proc. 1995 ACM SIGMOD Conf. on Management of Data, San Jose, CA, May 1995.

[20] J. Lu, A. Nerode and V.S. Subrahmanian. (1993) *Hybrid Knowledge Bases*, Accepted for publication in: IEEE Trans. on Knowledge and Data Engineering.

[21] A. Motro. (1987) *Superviews: Virtual integration of multiple databases.*, IEEE Trans. Software Eng., 13(7):785–798, 1987.

[22] I. S. Mumick. (1991) *Query Optimization in Deductive and Relational Databases.*, Ph.D. Thesis, Stanford University, CA 94305, 1991.

[23] M. Scholl, C. Laasch, and M. Tresch. (1991) *Updatable views in object-oriented databases.*, In Proc. Int. Conf. on Deductive and Object-Oriented Databases (DOOD), 1991.

[24] A. Sheth and J. Larson. (1990) *Federated database systems for managing distributed, heterogeneous and autonomous databases.*, ACM Computing Surveys, 22(3):183–236, 1990.

[25] Oded Shmueli and Alon Itai. (1984) *Maintenance of Views.* In Sigmod Record, 14(2):240-255, 1984.

[26] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. (1990) *On rules, procedures, caching and views in data base systems.*. In Proc. ACM SIGMOD Int. Conf. on Management of Data, pp. 281–290, 1990.

[27] V.S. Subrahmanian. (1994) *Amalgamating Knowledge Bases.*, ACM Trans. on Database Systems, 19, 2, pps 291–331, 1994.

[28] V.S. Subrahmanian, S. Adalı, A. Brink, R. Emery, J. Lu, A. Rajput, T.J. Rogers, R. Ross. (1994) *HERMES: A Heterogeneous Reasoning and Mediator System*, submitted for publication. (http://www.cs.umd.edu/projects/hermes/overview/paper)

[29] G. Wiederhold. (1992) *Mediators in the Architecture of Future Information Systems*, IEEE Computer, March 1992, pps 38–49.

[30] G Wiederhold, S. Jajodia, and W. Litwin. (1993) *Integrating temporal data in a heterogeneous environment.*, In Temporal Databases, Benjamin/Cummings, Jan 1993.

[31] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie and T. Price. (1979) *Access Path Selection in a Relational Database Management System.*, Proc. of the 1979 ACM SIGMOD International Conference on Management of Data, pp. 22-34.

[32] D. Chimenti, R. Gamboa and R. Krishnamurty. (1989) *Towards an open architecture for LDL.*, Proc. of the 15th International VLDB Conference, pp. 195–203, Amsterdam, The Netherlands, August 1989.

[33] S. Chaudhuri and K. Shim. (1993) *Query Optimization in the Presence of Foreign Functions.*, Proc. of the 19th VLDB Conference.

[34] S. Chaudhuri, R. Krishnamurty, S. Potamianos and K. Shim. (1995) *Optimizing Queries with Materialized Views*, Proc. of International Conference on Data Engineering, pp. 109–200, 1995.

[35] J. D. Ullman. *Principles of Database and Knowledge Base Systems*, volume 2. Computer Science Press, 1989.

[36] C.M. Chen and N. Rousopoulos. (1994) *Adaptive Selectivity Estimation Using Query Feedback.*, Proc. of the 1994 ACM-SIGMOD Conference on Management of Data, pp. 161 – 172.

[37] H. Tamaki and T. Sato. (1986) *OLD Resolution with Tabulation*, Proc. 3rd Intl. Conf. on Logic Programming (ed. E. Shapiro), pps 84–98, Springer.

[38] D.S. Warren. (1992) *Memoing for Logic Programs*, Comm. of the ACM, 35, 3, pps 94–111.

[39] X. Qian. (1995) *Query folding.*, To appear in the Proc. of the 1996 IEEE Data Engineering Conf., Technical Report SRI-CSL-95-09, Computer Science Laboratory, SRI International, July 1995.

[40] S. Adalı and X. Qian. (1995) *Query Transformation in Heterogeneous Reasoning Systems.*, Submitted for publication.

[41] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. (1994) *The TSIMMIS Project: Integration of Heterogeneous Information Sources.*, In Proceedings of IPSJ Conference, Tokyo, Japan, October 1994. (Also available via anonymous FTP from host db.stanford.edu, file /pub/chawathe/1994/tsimmis-overview.ps.)

[42] W. Du and R. Krishnamurthy and M.-C. Shan. (1992) *Query Optimization in Heterogeneous Database Management Systems.*, In Proc. VLDB Vonference, pp. 277–291, Vancouver, Canada, 1992.

[43] Q. Zhu and P.-A. Larson. (1994) *A Query Sampling Method for Estimating Local Cost Parameters in a Multidatabase System.*, Proc. IEEE Data Engineering Conf., pp. 144–153, 1994.

[44] H. Lu and B.-C. Ooi and C.-H. Goh. (1993) *Multidatabase Query Optimization: Issues and Solutions.*, Proc. RIDE-IMS '93, pp. 137–143, 1993.

Appendix: List of Queries Used in 2nd Experiment

```
query1(First,Last,Object,Size) :-
    in(Size,video:video_size('rope')) &
    in(Object,video:frames_to_objects('rope',First,Last)).

query1'(First,Last,Object,Size) :-
    in(Object,video:frames_to_objects('rope',First,Last) &
    in(Size,video:video_size('rope')).

query2(First, Last,Object,Frames,Actor) :-
    in (Object, video:frames_to_objects ('rope', First, Last)) &
    in(Frames,video:object_to_frames('rope',Object)) &
    in (Actor, relation:equal ('cast', role, Object)).

query2'(First, Last,Object,Frames,Actor) :-
    in (Object, video:frames_to_objects ('rope', First, Last)) &
    in (Actor, relation:equal ('cast', role, Object)) &
    in(Frames,video:object_to_frames('rope',Object)).

query3(First, Last,Object,Actor) :-
    in (Object, video:frames_to_objects ('rope', First, Last)) &
    in (Actor, relation:equal ('cast', role, Object)).

query4(First, Last,Object,Actor) :-
    in (P, relation:all ('cast')) &
    =(P.name, Actor) &
    =(P.role, Object) &
    in (Object, video:frames_to_objects ('rope', First, Last)).
```