A Dynamic Query Scheduling Framework for Distributed and Evolving Information Systems [*]

Ling Liu
Department of Computing Science
University of Alberta
GSB 615, Edmonton, Alberta
T6G 2H1 Canada
email: lingliu@cs.ualberta.ca

Calton Pu
Dept. of Computer Science and Engineering
Oregon Graduate Institute
P.O.Box 91000 Portland, Oregon
97291-1000 USA
email: calton@cse.ogi.edu

Abstract

The rapid growth of the wide-area network technology has led to an increasing number of information sources available on-line. To ensure the query services to scale up with such dynamic open environments, an advanced distributed information system must provide adequate support for dynamic interconnection between information consumers and information producers, instead of just functioning as a static data delivery system. We develop a distributed query scheduling framework to demonstrate the feasibility and the benefit for supporting interoperability and dynamic information gathering across heterogeneous information sources, without relying on an integrated view pre-defined over the participating information sources. We outline the mechanisms developed for the main components of our distributed query scheduling framework, such as query routing and query execution planning services. We also provide a concrete example to illustrate the issues on how the information consumers' query requests are dynamically processed and linked to the heterogeneous information sources and how the query scheduling framework scales up as the number of information sources increases.

1 Introduction

With recent advances in world-wide high-speed networks, there is a rapid increase in the number of information sources available on line. The availability of various Internet information browsing tools further promotes the information sharing across departmental, organizational, and national boundaries. We view an advanced distributed information system as a dynamic interconnection between information consumers and information producers, instead of just functioning as a static data delivery system. Two issues that arise immediately are: (1) heterogeneity of information producers' data sources and information consumers' query requests, and (2) scalability of distributed query services in the presence of a growing number of information sources and the evolving requirements of both information producers and information consumers.

We have proposed the Distributed Interoperable Object Model (DIOM) [10] in the context of the Diorama project as a concrete mediator-based approach to interoperability through the cooperation among a network of specialized mediators. Our early work mainly focuses on the design of the DIOM object model which captures the requirements of advanced distributed information systems as the USECA properties [10]: *Uniform access* to heterogeneous multimedia information sources, *Scalability* to the growing number of information sources, *Evolution* and *Composability* of software and information sources, and *Autonomy* of participants, both information consumers and information producers. In this paper we develop a distributed query scheduling framework to support adaptive information gathering, while preserving the USECA properties.

The most salient features of the DIOM distributed query scheduling framework are summarized as follows. First, we provide facilities that allow information consumers to pose queries on the fly, i.e., without relying on the existence of any integrated view. These facilities include (1) creation of a user query profile for each consumer query to capture the semantic scope and context of what the user wants in this specific query and (2) generation of a set of virtual interface classes that describe the representation of the resulting objects of the query (see Section 3). Second, we describe each producer's information source in terms of its content and its query capability and capture them in the source profile. Each producer's data source profile is created independently at the source registration time to capture the usage and constraints of the available source data. By describing each information source independently

[*] This research is partially supported by NSERC grant OGP-0172859, NSERC grant STR-0181014 for the first author, and ARPA grant N00014-94-1-0845, NSF grant IRI-9510112, grants from the Hewlett-Packard Company and Tektronix for the second author.

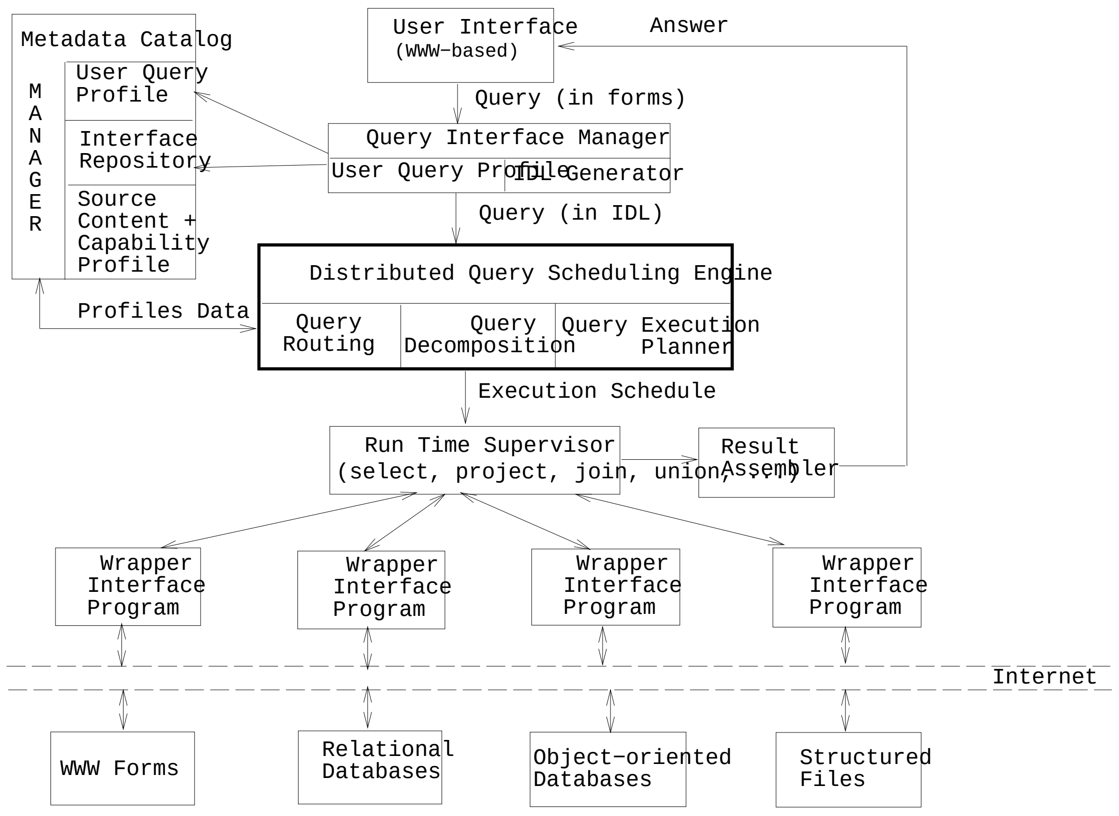

Figure 1: System Architecture

of the other sources and of the user queries, it enables us to incorporate the new sources into our query scheduling process dynamically and seamlessly, i.e., without affecting the way how queries are posed and processed (see Section 3). Third and most importantly, mechanisms are provided to dynamically relate information sources with a user query, based on the user's query profile and the producers' data source profiles. Thus, not only the new sources can be incorporated into the query scheduling process smoothly but also the information sources that do not contribute to a query can be pruned at early stage.

In the subsequent sections, we first give a brief overview of the system architecture currently being implemented in the Diorama project. Then we describe the DIOM distributed query scheduling framework and discuss each of the major steps in generating a relatively optimal schedule for distributed query execution. We also report our initial implementation experience in the Diorama project. Numerous other recent projects have similar goals and are discussed briefly at the end of the paper.

2 System Architecture: An Overview

The DIOM system architecture, first introduced in [10], is a two-tier architecture offering services at both mediator level and wrapper level. It conforms to the I^3 intelligent integration of information reference framework [6]. Figure 1 presents a sketch of the DIOM system architecture currently being implemented in the Diorama project.

A user may submit his/her query using the WWW fill-out forms. Query interface manager is responsible for (1) recording a form query in terms of the DIOM interface query language (DIOM IQL), a SQL-like language [11], and (2) generating virtual interface classes, described in terms of the DIOM interface definition language (DIOM IDL) [10], which are used as template holders for receiving and representing the resulting objects of the query. The interface manager also provides an interactive interface program to allow the user to provide the scope and context of what are to be expected in this specific query, such as images here are medical images like X-ray photos. This information will be used in the query routing module to restrain the search space of the query by isolating the query within a subset of information sources that are relevant to the query. The user query profile and the producers' data source profiles play an important role in discovering the relevance of information sources to a consumer's query. Once the relevant information sources are selected, the query decomposition module will rewrite the query into a group of subqueries, each targeting at one information source selected. The query execution planner is responsible for finding a relatively optimal schedule that makes use of the parallel processing potential and the useful dependencies between subqueries. The ultimate goal is to reduce the overall response time and the total query processing cost. The run-time supervisor executes the subqueries by communicating with wrappers. Wrappers in DIOM are software modules whose main task is to control and facilitate external access to the information sources, and translate the query request from the run-time supervisor to an executable query program (or a function call) that can run at the local source. The result assembly process involves two phases for resolving the semantic variations among the subquery results: packaging each individual subquery result into a DIOM object (done at wrapper level) and assembling results of the subqueries in terms of the consumers' original query statement (done at mediator level). The semantic attachment operations and the consumers' query profiles are the main techniques that we use for resolving semantics heterogeneity implied in the query results [11, 9].

Information sources at the bottom of the diagram are either *well structured* (e.g., RDBMS, OODBMS), or *semi-structured* (e.g., WWW forms, structured files). Currently DIOM provides access to *unstructured* data (e.g., technical reports) through external WWW keyword based browsers such as Altavista, Infoseek, Yahoo.

3 The Query Scheduling Framework

3.1 The Main Components

The main components of the DIOM distributed query scheduling manager are query routing, query decomposition, and query execution planner (recall Figure 1).

- **Query routing**
  The main task of query routing is to select relevant information sources from a large collection of available information sources which can contribute to answering a query. This is done by relating the content and query capability descriptions of the sources with the query using the IDL virtual interface description and the user query profile created by the query interface manager (recall Figure 1).
- **Query decomposition**
  The main goal of this module is to rewrite an IQL query, posed by the information consumer on the fly, into a group of subqueries, each targeted at a single data source. In addition to the number of subqueries, the query decomposition process will also provide a set of dependencies among subqueries, which are required to be met by the query execution planner to ensure that the execution schedule generated is semantically correct, i.e., producing the desired answer as asked by the original query.
- **Query execution planner**
  The goal of query execution planner is to generate a query execution schedule that is optimal in the sense that it utilizes the potential parallelism and the useful execution dependencies between subqueries to restrain the search space, minimize the overall response time, and reduce the total query processing cost.

3.2 An Application Scenario

Consider the following medical insurance related application scenario: an insurance agent wants to construct `ClaimFolder` objects by combining data from the following three types of disparate data sources:

- a patient's insurance agreement and claims for special medical treatments stored in a relational data base of the relevant insurance company,
- a collection of X-ray images associated with each claim of a patient, each is stored in a image-specific file system of some radiological lab, and
- a doctor's diagnosis report stored in a document repository (such as a C++ based one) at doctor's office.

Assume that we have a query that asks for the patients who have a claim for insurance of the special "dental bridge" treatment and who have paid the insurance premium for less than three months (90 days). The desired query answer form is a collection of patients, each patient is presented by the name and the insurance number of the patients and the set of corresponding X-ray images. Obviously, this query requires to perform an inter-source join over multiple data repositories such as the Insurance database, the Document repositories, and the image repositories. Suppose the query is posed in terms of DIOM IQL as follows:

```
Q: SELECT I.patient-name, I.insur-no, F.X-ray-pictures
   FROM   ClaimFolder F, Insur-Agreement I
   WHERE  F.report.search("dental" && "bridge")
          AND (F.claims.claim-date - I.start-date) < 90;
```

Note that this query was posed on the fly, i.e., without relying on the existence of any integrated view. We express the query in terms of what we would like the result of this query to be delivered.

3.3 Query Routing

Once a query is posed, the DIOM query interface manager (also referred to as IQL preprocessor) will first check with the DIOM interface repository where the IDL interface definitions are stored. If the IDL specification of the target interfaces given in the FROM clause does not exist, an appropriate IDL specification will be generated [10]. For instance, assume that there is no interfaces previously generated, and from the user query profile, we know that, to create ClaimFolder objects, we need to link the patients' claims, stored in some relational databases, with the corresponding X-ray images and doctors' diagnosis reports. Thus, the IQL preprocessor will create two compound interfaces, one named ClaimFolder and another named Insur-Agreement, using the DIOM interface composition meta operation [10] before passing this query to the distributed query routing module. Based on the user query profile, the interface class ClaimFolder is generated using the interface aggregation meta operator over three component interface classes: Claims, Images, and DiagnosisRep; and the interface class Insur-Agreement is generated using the interface generalization meta operator. A sketch of the IDL specification generated for these two interfaces are shown below. The IDL definitions for Claims, Images, and DiagnosisRep can be created in a similar way.

```
CREATE INTERFACE ClaimFolder
( EXTENT ClaimFolders )
{
  AGGREGATION OF Claim, Image, DiagnosisRep;
    RELATIONSHIPS
         Claim        claims;
         Set<Image>   X-ray-photos;
         DiagnosisRep report;
}

CREATE INTERFACE Insur-Agreement
( EXTENT Insur-Agreements  )
{
  GENERALIZATION OF
    select interface-name from Interface-Repository
    where  description = '*agreement*' OR
           description = '*contract*';
  ATTRIBUTES
      Integer insur-no;
      String  client-name;
      Data    start-date;
}
```

One of the most important benefits of using interface composition mechanisms is to minimize the impact of source schema changes over the application programs working with the existing interoperation interfaces. For example, consider the ClaimFolder scenario. Suppose a new image source, called CAT-Scan becomes available, the use of generalization abstraction to build the IDL interfaces for the example query allows this new source be seamlessly incorporated into the ongoing query scheduling process and be used in the processing of subsequent queries involving images. This is because the Image interface we refer to is defined as a generalization of all available repositories that contain images, i.e.,

```
CREATE INTERFACE Image
( EXTENT Images  )
{
  GENERALIZATION OF
    select interface-name from Interface-Repository
    where  description = '*image*' OR
           description = '*Image*';
  ATTRIBUTES
      String image-no;
      String annotation;
      *FILE  raw_image;
  METHODS
      *Image display();
}
```

Suppose we have access to the on-line information sources shown in Figure 2, among many others. Each source is described by its source profile, consisting of both content description, category information, and its query capability description. Some of the sources are obviously not contributing to answer Q. We can immediately determine that Source 2 and Source 6 are not relevant to answering this query, because they have no information about medical insurance. We can also conclude that Source 1 is not able to contribute to the answer of Q. The reasoning here is more subtle: our query requires the selection on a special range of insurance period, but there is no date or related information provided as parameters in Source 1. We are left with sources 3, 4, 5, denoted as S3, S4, S5. These three sources, selected by the first step of the query scheduling process, will be passed on to the next step, the query decomposition module.

Due to the space restriction, we omit the detailed query routing algorithms in this paper.

3.4 Query Decomposition

The goal of query decomposition is to break down an information consumer's query, expressed in IQL, into a collection of IQL subqueries, each targeted at a single source. The query decomposition module takes as input the IDL query expression modified (reformulated) by the dynamic query source selection module, and performs the query decomposition through the *target split* process. The process of target split results in a set of independent subqueries and a query decomposition plan that is semantically equivalent to the original query expression. This query decomposition plan also presents how to merge the results of the decomposed subqueries to produce the answer for the original consumer query.

Note that although the set of subqueries resulting from the query decomposition phase can be executed independently,

Source 1: SunLife Insurance Company **Category**: medical insurance; **Content**: Agreements, Claims; **Query Capabilities**: Accept as input ins_no or customer_name and ins_category, and optionally annual_price. Output is the qualifying insurance agreement, with ins_no, ins_category, annual_price, and contact information.
Source 2: Car Insurance Databases **Category**: Car insurance; **Content**: Clients, InsAgreements; **Query Capabilities**: Accept as input client_name or Insur_no, and optionally experience or price or category or company name. Output is the qualifying InsAgreement or Clients, with client_name, Insur_no, price, company name, contact info.
Source 3: Image Database **Category**: Medical Image; **Content**: Images; **Query Capabilities**: Accept as input category or image_type, and optionally description, patient_insur_no, Output may include raw_image, description, creation_date, creation_place, and method display().
Source 4: Med Insurance Company Repository **Category**: Medical Insurance Company Database; **Content**: Agreements, Company, Claims; **Query Capabilities**: Accept as input insur_no or customer_name optionally ins_category, starting_date, price. Output is the qualifying agreement, with insur_no, ins_category, starting_date, annual_payment, contact info.
Source 5: Medical Document Database **Category**: Doctors' Diagnosis Reports; **Content**: DiagReport; **Query Capabilities**: Accept as input ins_num or method keyword_search(String), and optionally patient_name. Output is the diagnosis reports (text) for that ins_num, patient_name.
Source 7: TechReport Database **Category**: Technical Report Database; **Content**: TechReport; **Query Capabilities**: Accept as input title or authors or organization, and optionally year. Output is the technical reports for that title or authors or organization or year.

Figure 2: Example information sources

very often it is useful to follow the execution dependencies between these subqueries to find some ordering so that results of some subqueries may be used to drastically reduced the search space of other subqueries (see Section for more detail).

Consider our example query. Suppose, after passing through the query routing phase, the necessary connection paths (e.g., `I.insur-no == F.claims.insur#` and the set of candidate information sources (e.g., `S3, S4, S5`) that are relevant to the answering of the example query are added into the original query expression. The original query `Q` is now reformulated as follows:

```
TARGET S3, S4, S5
SELECT I.patient-name, I.insur-no, F.X-ray-pictures
FROM   ClaimFolder F, Insur-Agreement I
WHERE  F.report.search("dental" && "bridge") AND
       (F.claims.claim-date - I.start-date) < 90
       AND I.insur-no == F.claims.insur-no;
```

Recall that `ClaimFolder` is an interface defined based on the aggregation of the three interfaces: `Claim`, `Image`, `DiagnosisRep`, and `Insur-Agreement` is a generalization interface. Thus, we first replace the interface `ClaimFolder` by an aggregation-based join, namely **D-AggregationJoin**, over the three interfaces `Claim`, `Image`, and `DiagnosisRep`. An aggregation-based join refers to an object join which maintains the nested aggregation structure while traversing through the objects following the aggregation path [8]. Then we replace each generalization interface by the corresponding base interfaces. In our example, `Claims` is replaced by `S4:Claim`, `Insur-Agreement` is replaced by `S4:Agreements`, `Image` is replaced by `S3:Images`, and `DiagnosisRep` is replaced by `S5:DiagReport`. Thus the query is reformulated into inter-site joins of three subqueries, with insurance number as the join attribute, each targeting at one of the three sources `S3, S4, S5`. The following are the three independent IQL subqueries:

```
SubQ1: TARGET S4
       SELECT I.patient-name, I.insur-no
       FROM   Claim C, Insur-Agreement I
       WHERE  (C.claim-date - I.start-date) < 90
              AND I.insur-no == C->insur#;

SubQ2: TARGET S5
       SELECT D.insur-no
       FROM   DiagnosisRep D
       WHERE  D.search("dental" && "bridge");

SubQ3: TARGET S3
       SELECT P.insur-no, P.X-ray-pictures
       FROM   Image P;
```

3.5 Query Execution Planner

The goal of generating a query execution plan for a group of subqueries is to select an "optimal" distributed query schedule that has, relatively speaking, the most efficient overall query response time and/or the minimal overall query processing cost. In DIOM, the overall processing cost of a query includes the communication cost, the local processing cost, and the result assembly cost. We allow users who wish to tune the performance of particular type of queries to use weight functions to assign different co-efficiency to each cost computation component. Techniques for utilization of parallel processing during the search for query answers from multiple information sources are used for reducing the overall query response time.

3.5.1 General Strategies and Techniques

A parallel access plan, constructed from the query decomposition plan, consists of three main components: (1) a set of subqueries (each posed against exactly one local export schema), (2) the *data-shipping* operations that move the result of a subquery to the chosen site for processing of another subquery or for result assembly, and (3) the post-processing operations that assemble results of previously decomposed subqueries in terms of the information consumer's original query statement. In the development of the DIOM distributed query optimizer, we explicitly distinguish two types of query decomposition plans:

1. The query decomposition plans which contains no aggregation-based joins (**D-AggregationJoin**). In most cases, the subqueries contained in this type of decomposition plans are union-compatible.
2. The query decomposition plans which require **D-AggregationJoin** in their query decomposition trees generated by the query decomposition module. Subqueries in this type of decomposition plans are often not union-compatible.

For the first type of query decomposition plans, if the response time is the most critical concern for the performance tuning, then the optimal schedule for distributed query execution is the one that explores the full scale of parallelism, that is, all the subqueries resulting from the query decomposition plan are submitted to the corresponding data sources in parallel. The query optimization process is relatively simple. For example, assume the response time of each subquery executed at site i is denoted by $LRt(i)$ $(1 \leq i \leq n)$ which is composed of both the local execution cost and the subquery result packaging cost. The overall response time is basically the result of $MAX(LRt(1), ..., LRt(n))$.

The second types of query decomposition plans, however, involve more sophisticated situations. This is mainly because the parallel submission of all the subqueries may not be the best query schedule in the cases where some subqueries have much smaller set of results and thus can be used to reduce the search space of other subqueries drastically. In these cases, any combination of *parallel* "$\|$" and *sequential* "$\mapsto$" synchronization operations may produce a possible execution schedule for the given set of subqueries. For each consumer's query, if the number of subqueries resulting from the query decomposition phase is k, then it is easy to see that the number of execution ordering alternatives is combinatorial and increases rapidly when the number of subqueries (i.e., k) becomes large. Not surprisingly, many of the possible synchronization alternatives will never be selected as the "optimal" execution schedule for the given query. For example, assume that, for any two subqueries, the independent evaluation of a subquery subQ_1 is more expensive than the independent evaluation of subquery subQ_2. Thus, the synchronization scheme that executing subQ_1 first and then ship the result of subQ_1 to another site to join with the subquery subQ_2 is obviously an undesirable access plan because of the poor performance comparing with other execution schedules. Therefore, it is important to find the useful subquery execution dependencies among the combinatorial number of alternative schedules for ordering the execution of the set of decomposed subqueries. Useful dependencies are those that can be used to minimize the overall response time and reduce the total processing cost of a consumer query by at least a factor. We use heuristic-based method to make a first cut from all the possible subquery submission/execution schedules. Those obviously undesirable synchronization alternatives will be ruled out immediately. A detailed discussion on the collection of heuristics will be reported in another paper. Here are some examples of such heuristics applicable to the subqueries whose results are not union-compatible:

- The selectivity level of a *Fetch* condition with one of the following operators "<=, <, >, >=" is higher than the selectivity level of a *Fetch* condition with the operator "=", but lower than the selectivity level of a *Fetch* condition with "<>" operator.
- Subqueries of the lowest level of selectivity are executed earlier. If there is more than one subquery of lowest level of selectivity, then those subqueries should be executed in parallel.
- Subqueries of the highest level of selectivity are executed later.

The idea of applying these simple heuristics is to use different selectivity levels of different query predicates (*fetch conditions*) to estimate the cost difference among various subqueries. When the selectivity level of a subquery is lower, the size of the subquery result tends to be relatively small. Thus it is always a recommended tactic to execute the subqueries of lower selectivity level earlier.

It is well known that the quality of a query processor relies on the efficiency of its query processing strategies and the performance of its query execution plans. Furthermore, the performance of a distributed query processing plan is not only determined by the response time of the local subqueries but also affected by the synchronization scheme chosen for synchronizing the execution of subqueries. A good synchronization scheme may utilize the results of some subqueries to reduce the processing cost of the other subqueries whenever it is beneficial.

In the early design of DIOM query processing framework [11], we considered only the simple cases where a consumer query are dispatched into a number of independent subqueries, which should be executed independently. In the current development, We include the cases where the set of subqueries generated from query decomposition of a consumer query may need to be synchronized in a certain schedule to improve the overall response time and the total cost of distributed query processing (communication cost + local processing cost + result assembly cost). For large-scale distributed and cooperative information systems, another important dimension is the synchronized delivery of subquery results to the consumer. This is an active topic of research and beyond the scope of this paper.

3.5.2 A Walk-through Example

Recall our example query reformulated by query decomposition, the possible execution dependencies that may be used for synchronization of the executions of subqueries SubQ1, SubQ2, SubQ3, in addition to those sequential alternatives, are the following:

1. SubQ1 $||$ SubQ2 $||$ SubQ3.
2. SubQ1 $\mapsto$ SubQ2 $||$ SubQ3.
3. SubQ2 $\mapsto$ SubQ1 $||$ SubQ3.
4. SubQ3 $\mapsto$ SubQ1 $||$ SubQ2.
5. SubQ1 $||$ SubQ2 $\mapsto$ SubQ3.
6. SubQ2 $||$ SubQ3 $\mapsto$ SubQ1.
7. SubQ1 $||$ SubQ3 $\mapsto$ SubQ2.

Each dependency alternative represents a possible synchronization plan for subquery executions. Based on the selectivity estimation heuristics given at the beginning of Section , we observe the following facts: First, the result of SubQ1 and SubQ2 can be used to restrict the search space of SubQ3. Second, SubQ2 may have a much lower level of selectivity than SubQ1 does. Therefore, based on the first observation we may drop those execution dependencies with SubQ3 at the start of the execution sequence. Namely, the execution dependencies 4, 6, 7 are dropped. Based on the second observation, those execution dependencies which do not include SubQ2 at the start of the execution sequence are disregarded too. Thus the execution dependency 7 is dropped. As a result, the useful subquery execution dependencies being recorded are the following three:

- (1) SubQ1 $||$ SubQ2 $||$ SubQ3.
- (3) SubQ2 $\mapsto$ SubQ1 $||$ SubQ3.
- (5) SubQ1 $||$ SubQ2 $\mapsto$ SubQ3.

Since in our example query, SubQ1 and SubQ2 are considered to have similar selectivity factor, and SubQ3 has much higher selectivity factor, if all the three sources have similar network accessibility, then the plan (3) is obviously optimal with respect to the others. When using selectivity factors is not enough for making a final decision, we will use the distributed cost estimation model that is being developed within the Diorama project. The result will be reported in a forthcoming paper.

Assume that SubQ1 $||$ SubQ2 $\mapsto$ SubQ3 is the execution schedule generated by the query execution planner. Let Temp2 denote the result of subquery SubQ2. We need to reformulate the subquery SubQ3 accordingly, as follows:

```
SubQ3: TARGET S3
       SELECT P.insur-no, P.X-ray-pictures
       FROM   Image P, Temp2 T
       WHERE  P.insur-no = T.insur-no;
```

We can now send the subqueries SubQ1, SubQ2, SubQ3 to the corresponding wrappers for subquery translation and execution. Each wrapper will first convert the subquery in IDL into a query expression that is executable at the data source to which the wrapper serves. When the repository is a RDBMS, say Oracle, the wrapper will map an IQL expression into an Oracle/SQL query statement. It is also the wrapper's responsibility to collect and package the result in terms of the DIOM objects before sending the result of a subquery back to the mediator. If the data source is an OODBMS (say ObjectStore), the wrapper will map each IQL expression into an ObjectStore query that is bounded to an ObjectStore class dictionary. When the data source is stored and managed solely by a file server, say in the form of HTML or SGML, the wrapper will map the IQL expression into, for example, a C module or a Perl module that scans the source data and returns the matching records.

We are currently investigating the generic wrapper architecture and the generic conversion functions for transforming DIOM-IQL queries to different types of data repositories. Our first implementation is done through a Netscape based multi-source information browser, which currently runs through two types of wrappers: an Oracle wrapper for structured data sources and a HTML wrapper for accessing unstructured or semi-structured web data sources (e.g., WWW pages, NetNews articles).

4 Related Work

Several systems (e.g., TSIMMIS [5], Garlic [2], CARNOT [4], SIMS [1], DISCO [3]), Information Manifold [7]) for integrating multiple information sources are being built based on the mediator architecture [12]. The key aspect distinguishing DIOM from the other systems is its emphasis on scalability and extensibility of the query services by promoting source-independent query processing at the mediator level and by describing the content and capability description of each source independently of description of other sources and of the way how queries are posed. Thus, users may pose queries on the fly and the new sources can be incorporated into the query scheduling process dynamically and seamlessnessly. For example, SIMS and Information Manifold are the two systems that also describe information sources independently of the queries that are subsequently being asked on them. However, neither SIMS nor Information Manifold provide distributed query scheduling services that find a relatively optimal execution schedule to submit the set of subquery plans to the corresponding sources (via wrappers). TSIMMIS and DISCO both use a set of sophisticated query rewriting patterns to mediate the queries and the multiple information sources. TSIMMIS provides a rich set of pre-defined query patterns and map each query into one of the pre-defined patterns and then apply the rewriting rules for each pre-defined pattern to generate an executable plan. It is not clear whether the set of pre-defined patterns is extensible and the new patterns can be added into their rewriting system seamlessly. DISCO uses multiple F-logic object schema to interoperate among multiple sources via the KIF knowledge interchange logic, but it is unclear how their logic-based transformation system scales when changes occur in the number of the sources or the content of individual sources.

In addition, a number of proposals have competed as the basic enabling technologies for implementing interoperable objects in distributed and dynamic object computing environments. Examples include Microsoft's Object Linking and Embedding (OLE), IBM's System Object Model (SOM) and its distributed version (DSOM), OMG's Common Object Request Broker Architecture (CORBA), and CI labs OpenDoc. Many of these are available as deployed software packages. Although these proposals are clearly practical and important, they focus primarily on the software interface problem, not the uniform query interface for end-users and the scalable query scheduling services for accessing multiple heterogeneous information sources. DIOM can be seen as a glue that spans and integrates these interface models at a higher level.

5 Conclusion

We have described the DIOM distributed query scheduling framework for accessing heterogeneous multimedia information sources. We demonstrated by examples how the distributed query scheduling framework is implemented through relating information sources to user queries based on user query profiles and source capability profiles. This proposed framework has a number of important features: (1) modularity and composability in terms of representing user queries in terms of aggregation and generation abstraction, (2) scalability in terms of seamless incorporation of newly added information sources into the process of consumers' query requests, (3) transparency in terms of posing queries without relying on any global and integrated schema of all the participating information sources, (4) autonomy and robustness in terms of evolution of information sources and the requirement changes of information consumers, and (5) efficiency in terms of the strategies and techniques developed for finding an optimal query schedule for each consumer query, which has, relatively speaking, the minimal overall response time.

Our ongoing research continues in several directions. We are currently working on the formal development of a distributed object query algebra, including inter-site joins, and algorithms for query decomposition and for generation of optimal query schedule, which also takes into account the efficient processing of aggregate functions such as SUM, COUNT, MAX and MIN. We are also interested in caching/replication issues and their effect on performance and availability, and in introducing learning capabilities to further raise the quality of the query scheduling result. Last but not the least, systematic construction of wrappers for various types of multimedia information sources is a non-trivial task. We are currently investigating on the generic architecture for wrapper construction by building wrappers to UNIX files and ObjectStore databases, in addition to the wrappers to HTML files and ORACLE databases.

Acknowledgement We would like to thank the DIOM implementation team at University of Alberta, especially Yooshin Lee, Kirill Richine, and David Buttler.

References

[1] Y. Arens and et al. Retrieving and integrating data from multiple information sources. *International Journal of Intelligent and Cooperative Information Systems*, 2(2):127–158, 1993.

[2] M. Carey, L. Haas, et al. Towards heterogeneous multimedia information systems: the garlic approach. In *Technical Report, IBM Almaden Research Center*, 1994.

[3] Y. Chang, L. Raschid, and B. Dorr. Transforming queries from a relational schema to an equivalent object schema: a prototype based on f-logic. In *Proceedings of the International Symposium on Methodologies in Information Systems (ISMIS)*, 1994.

[4] C. Collet, M. Huhns, and W. Shen. Obtaining complete answers from incomplete databases. In *IEEE Computer*, 1991.

[5] H. Garcia-Molina and et al. The tsimmis approach to mediation: data models and languages (extended abstract). In *Technical Report, Stanford University*, 1994.

[6] R. Hull and R. King. Reference architecture for the intelligent integration of information (version 1.0.1). http://isse.gmu.edu/I3_Arch/index.html, May 1995.

[7] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, 1996.

[8] L. Liu. A recursive object algebra based on aggregation abstraction for complex objects. *Journal of Data and Knowledge Engineering*, 11(1):21–60, 1993.

[9] L. Liu and C. Pu. Customizable information gathering across heterogeneous information sources. Technical report, Department of Computer Science, University of Alberta, Dec. 1995.

[10] L. Liu and C. Pu. The distributed interoperable object model and its application to large-scale interoperable database systems. In *ACM International Conference on Information and Knowledge Management (CIKM'95)*, Baltimore, Maryland, USA, November 1995.

[11] L. Liu, C. Pu, and Y. Lee. An adaptive approach to query mediation across heterogeneous databases. In *Proceedings of the International Conference on Cooperative Information Systems*, Brussels, June 19-21 1996.

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