Scaling Heterogeneous Databases and the Design of Disco

Appears in IEEE International Conference on Distributed Computing Systems (ICDCS) 1996

Anthony Tomasic
INRIA Rocquencourt
78153 Le Chesnay, France
Anthony.Tomasic@inria.fr
http://rodin.inria.fr/tomasic

Louiqa Raschid
University of Maryland
College Park, MD, 20742, USA.
louiqa@umiacs.umd.edu
http://umiacs.umd.edu/louiqa

Patrick Valduriez
INRIA Rocquencourt
78153 Le Chesnay, France
Patrick.Valduriez@inria.fr
http://rodin.inria.fr

Abstract

Access to large numbers of data sources introduces new problems for users of heterogeneous distributed databases. End users and application programmers must deal with unavailable data sources. Database administrators must deal with incorporating new sources into the model. Database implementors must deal with the translation of queries between query languages and schemas. The Distributed Information Search COmponent `(Disco)`[1] *addresses these problems. Query processing semantics are developed to process queries over data sources which do not return answers. Data modeling techniques manage connections to data sources. The component interface to data sources flexibly handles different query languages and translates queries. This paper describes (a) the distributed mediator architecture of* `Disco`*, (b) its query processing semantics, (c) the data model and its modeling of data source connections, and (d) the interface to underlying data sources.*

1. Introduction

Every heterogeneous distributed database system has several types of users. End users focus on data. Application programmers concentrate on the presentation of data. Database administrators (DBAs) provide definitions of data. Database implementors (DBIs) concentrate on performance.

As heterogeneous database systems are scaled up in the number of data sources in the system, several fundamental issues arise which affect users. For end users and application programmers, scale makes a system harder to use. In the absence of replication, to answer a query involving n databases, all n databases must be available. If some database is unavailable, either no answer is returned, or some partial answer is returned. The availability of answers in the system declines as the number of databases rises. For database administrators, scale makes a heterogeneous system hard to maintain. To add a data source to the system, schemas must be changed, catalogs updated, and new definitions added. For database implementors, scale makes a system hard to program and tune. To add a data source, new code must be written and new cost information recorded. To more clearly explain these issues, we describe the architecture for a heterogeneous distributed database system, and then describe various features of this architecture.

[1] This research has been partially supported by the Advanced Research Project Agency under grant ARPA/ONR 92-J1929 and by the Commission of European Communities under Esprit project IDEA.

1.1 Architecture

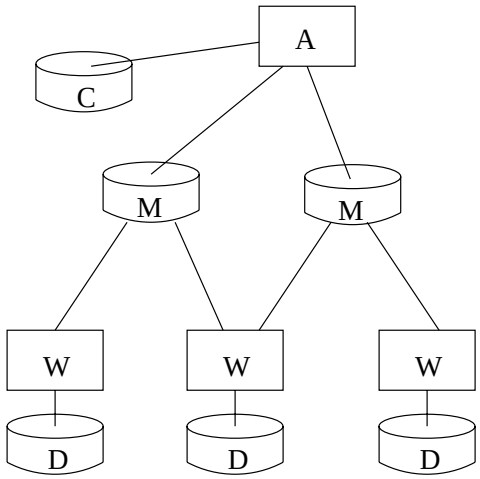

Figure 1. Disco architecture. Boxes represent stateless components, and disks components with state. A stands for application, M: mediator, C : catalog, W : wrapper, and D : database. Arcs represent exchange of queries and answers.

As shown in Figure 1, current distributed heterogeneous database systems [5, 24], scale up by adopting a distributed architecture of several specialized components. End users interact with applications (A) written by application programmers. Applications access a uniform representation of the underlying sources through a uniform query language.

Mediators (M) encapsulate a representation of multiple data sources and provide a value-added service. Mediators provide the functionality of uniform access to multiple data sources. They typically resolve conflicts involving the dissimilar representation of knowledge using different data models and database schema, and conflicts due to the mismatch in querying power of each server. This distributed architecture permits DBAs to develop mediators independently and permits mediators to be combined, providing a mechanism to deal with the complexity introduced by a large number of data sources. This architectural assumption is a good one, and we adopt it in this paper.

To permit collections of databases to be accessed in a uniform way, mediators accept queries and transform them into subqueries that are distributed to databases. DBAs provide information to mediators to accomplish query transformation. A mediator may, as in data warehousing, also keep state or summary information about its associated databases. In addition, special mediators, *catalogs*, (C), keep track of collections of databases, wrappers, and mediators in the system. Catalogs do not have total knowledge of all elements of the system; however, they provide an overview of the entire system.

To deal with the heterogeneous nature of databases, *wrappers* (W) transform subqueries. Wrappers map from a subset of a general query language, used by mediators, to the particular query language of the source. A wrapper supports the functionality of translating queries appropriate to the particular server, and reformatting answers (data) appropriate to each mediator. The *wrapper implementor*, a new specialty of DBI, writes wrappers for each type of database.

The design of the Distributed Information Search COmponent (`Disco`) provides different special features for all users to deal with the problems of scale. For the application programmer and end user, `Disco` provides a new semantics for query processing to ease dealing with unavailable data sources during query evaluation. For the DBA, `Disco` models data sources as objects which permits powerful modeling capability. In addition, `Disco` supports type transformations to ease the incorporation of new data sources into a mediator. For the DBI, `Disco` provides a flexible wrapper interface to ease the construction of wrappers.

One of the target applications for `Disco` is an environmental application for the control of water quality. Multiple databases, distributed geographically, contain measurements of water quality at the physical site of the database. All of these measurements have the same type and the DBA must integrate a large number of data sources with similar structures. `Disco` provides special features to ease the integration of multiple data sources.

1.2 Data model

Consider a system that contains two data sources `r0` and `r1`. Suppose the `r0` data source contains a person relation with a person Mary whose salary is 200 and `r1` contains a person relation with a person Sam whose salary is 50. A mediator models `r0` and `r1` as extents `person0` and `person1`, of type `Person`. The extent `person` of the `Person` type automatically contains the two extents `person0` and `person1`.

To access the objects, the `Disco` query language is used. For example, the query

```
select x.name
from x in person
where x.salary > 10
```

constructs a bag of the names of the persons from `person` who have a salary greater than 10. The answer to this query is a bag of strings `Bag("Mary","Sam")`.

With this organization, the addition of a new data source with persons simply requires the addition of a new extent to `person`, as long as the type of the new data source is the same as the type `Person`. The same query would then access three data sources. The query itself does not change. This property greatly simplifies the maintenance of the mediator. `Disco` provides support for incorporating new data sources with similar structure with respect to existing sources. It also provides support for incorporating new data sources with dissimilar structure with respect to existing sources. This supports scalability from the viewpoint of the DBA.

1.3 Query processing

`Disco` supports a new query processing semantics to deal with the problem of unavailable data. In the absence of replication, if a data source does not respond, then a database management system is faced with two possibilities: either it waits for the data source to respond, or it returns a partial answer. `Disco` uses *partial evaluation* semantics to return a partial answer to queries, by processing as much of the query as possible, from the information that is available. Thus, the answer to a query may be another query.

Suppose that the `r0` data source does not respond. Then, answer to the previous query would be the following query, representing a partial answer:

```
union(select y.name
      from y in person0
```

```
where y.salary > 10,
Bag("Sam"))
```

(In Disco, the union of two bags is a bag). Thus, the query in the partial answer is contained in the first argument of the union and the data is contained in the second argument. When r0 becomes available, this partial answer could be submitted as a new query (since it is itself a query) and the answer Bag("Mary", "Sam") would be returned, assuming that the underlying data sources have not changed.

1.4 Wrapper interface

For the DBI, Disco provides a flexible wrapper interface. Disco interfaces to wrappers at the level of an abstract algebraic machine (AM) of logical operators. When the DBI implements a new wrapper, she chooses a (sub)set of logical operators to support. The DBI implements the logical operators, and also implements a call in the wrapper interface which returns the grammar describing the supported logical expressions. During query processing, a Disco mediator query optimizer generates a logical expression for the wrapper. The mediator calls the wrapper interface to get the grammar describing the supported logical expressions, and checks that the logical expression generated by the optimizer is legal with respect to the grammar describing the wrapper interface.

In summary, Disco attacks fundamental problems in accessing a large number of heterogeneous sources. Explicit specification of the data sources, as objects, in the Disco data model, gives the DBA the capability to express queries that range over an unspecified collection of data sources [17], or queries that refer to particular data sources. As a result, the mediator can use the full power of the OQL query language to query data in heterogeneous servers, in a transparent manner. Inclusion of the data source specification within the model also allows Disco to support a new query processing semantics. Since data sources are objects, an answer to a query can be a partial answer, and can refer to a data source object or to actual data objects, and both references will be meaningful. Such a situation can occur when a particular data source is unavailable, as is common in a networked environment. This provides alternative query evaluation schemes and is very flexible. The type hierarchy and mapping supported by the Disco model allows the mapping of multiple data sources to a single type of a mediator, and also allows the mapping of a data source to multiple types of a mediator. This aspect of the Disco model supports scaling to a large number of data sources. New data sources may also be incorporated transparently, if they map to the same mediator type.

This paper is organized as follows: Section 2 presents the data model through a description of the extensions to an existing standard. Section 3 describes mediator query processing and the wrapper interface. Section 4 presents a new semantics of query processing. Section 5 describes related work. We conclude with a summary and discussion of future plans.

2 Mediator data model

2.1 Extensions to the ODMG standard

Disco is based on the ODMG standard. The ODMG standard consists of an object data model, an object definition language (ODL), a query language (OQL), and a language binding. In the data model, an interface defines a type signature for accessing an object. An extent, associated with an interface, instructs a system to automatically maintain the collection of objects of the interface. An extent is a named variable whose value is the collection of all objects of the associated interface. When objects are created or destroyed, the extent is updated automatically. Extents are the primary entry point for access to data.

The Disco data model is based on the ODMG-93 data model specification [6]. We extend ODMG ODL in two ways, to simplify the addition of data sources to a mediator.

extents This extension associates multiple extents with each interface type defined for the mediators.

type mapping This extension associates type mapping information between a mediator type and the type associated with a data source.

In addition to these extensions, we define two (standard) ODMG interfaces; Wrapper models wrappers and Repository models repositories. A repository is essentially the address of a database or some other type of repository. Repositories typically contain several data sources. Each data source in a repository is associated with an extent, and this provides the entry point to the data source.

Disco extends the concept of an extent for an interface, to include a bag of extents for the interface, for any type defined for the mediator. Each extent in the bag mirrors the extent of objects in a particular data source, associated with this mediator type. Since this extension is fully integrated into the ODMG model, the full modeling capabilities of the ODMG model are available for organizing data sources. Disco evaluates queries on extents and thereby on the data sources. To describe the data model, we proceed with the steps a database administrator (DBA) uses to define access to a data source in Disco.

The first step is to model the data source. The DBA creates an instance of the Repository type, which defines the repository and the data source that it contains. For example,

```
r0 := Repository(host="rodin.inria.fr",
      name="db", address="123.45.6.7")
```

creates a object of type Repository with the information necessary to access the data source in the repository, and assigns the object id to the variable r0. The definition of the Repository type is not completely specified; our example shows some necessary fields. Other attributes which describe the maintainer of the data source, the cost of accessing the data source, etc., can be added.

In the second step, the DBA locates a wrapper (written by a database implementor), for the data source. Section 3 discusses the features of Disco to aid the database implementor. A wrapper is an object with an interface that, when supplied with information to access a repository and a query, returns objects to a mediator which answer the query. For instance, the following wrapper object w0 might access a relational database; details of the wrapper are not specified in this paper:

```
w0 := WrapperPostgres();
```

In the third step, the DBA defines the type in the mediator which corresponds to the type of the objects in the data source. For example, the Person type corresponding to the objects in data sources r0 and r1, is defined as follows, where the interface is a standard ODL interface:

```
interface Person {
  attribute String name;
  attribute Short salary; }
```

Finally, the DBA specifies the extent of this mediator type, which accesses the r0 repository utilizing the w0 wrapper. Our specification of the extent is a modification of the meta-data information. Disco provides the following special syntax, for the addition of an extent:

```
extent person0 of Person wrapper w0
          repository r0;
```

This specification adds the extent person0 to the Person interface. The type of the objects of extent person0 are of the same type as the interface Person. This specification states that access to objects in the data source are through the wrapper w0, and objects are located in the repository r0. The extent name person0 is determined by the name of the data source in the repository. The type of objects in the data source are assumed to be the same as the type of the objects in the extent. Thus, the type of the objects in the data source associated with person0 is Person. At run-time, the wrapper checks that these types are indeed the same. We note that the Disco data model can also handle the case where there is a mismatch of types, and this is discussed in Section 2.2.2. Thus, *each Disco extent represents a collection of data in one data source*. This intuition is the key to the Disco data model. (A more general approach associates an implementation with each data source [5, 29])

At this point, data access from the data source is possible. The following query:

```
select x.name
from x in person0
where x.salary > 10
```

returns the answer Bag("Mary") with respect to the data source defined in the introduction. Several conditions must hold for this answer to be returned. The name of the data source in repository r0 is person0, the same as the extent name. The type of every object in the data source must be of type Person. We modify these restrictions in Section 2.2.2.

The addition of a new Person data source now only requires adding an extent to type Person, assuming that the appropriate wrapper is available. For example, the following extent expression:

```
extent person1 of Person wrapper w0
          repository r1;
```

adds the person1 extent to the Person interface, utilizing the same wrapper, but referencing a different repository object r1. We assume that the objects in person1, which are from the r1 repository are of type Person. To access objects in both data sources, the extents are listed explicitly in the following select expression:

```
select x.name
from x in union(person0,person1)
where x.salary > 10
```

which returns the answer: Bag("Mary", "Sam").

The Disco data model allows us to explicitly refer to the extents for mediator type, in the queries. Although, this is a powerful capability, which is exploited in examples in Section 2.3, it also makes it difficult to express queries, when the extents are not explicitly specified. The Disco data model solves this by using a special meta-data type MetaExtent, which records the extents of all the mediator types. The special extent syntax used previously to add or delete extents can be translated to automatically create instances of this meta-data type, MetaExtent, which is defined as follows:

```
interface MetaExtent (extent metaextent)
  {
  attribute String name;
  attribute Extent e;
  attribute Type interface;
  attribute Wrapper wrapper;
  attribute Repository repository;
  attribute Map map; }
```

Thus, extents for the mediator types can be added or deleted by adding or deleting objects of type MetaExtent. For example, the following extent:

```
extent person1 of Person wrapper w0
          repository r1;
```

will create an instance, say m1, of type MetaExtent, where m1.e=person1, m1.interface=Person, etc. Note that the map attribute of type MetaExtent provides a type conversion facility between the mediator type and the data source type, and is described in Section 2.2.2. It is possible to generalize the association of extents to type into a full hierarchy of extents. However, it is not clear that this generality brings any real modeling benefits to the DBA.

Using this meta-data, Disco can now provide an implicit reference to all the extents associated with a mediator type, by declaring an extent in the interface definition. Thus, the following interface definition for Person implicitly assumes a query definition expression for the corresponding extent person:

```
interface Person (extent person){
  attribute String name;
  attribute Short salary; }

define person as
  flatten(select x.e from x in metaextent
            where x.interface=Person)
```

This query definition expression for person accesses the meta-data of the extents, to dynamically select all of the extents associated with the type Person. Thus, the following query dynamically accesses all the extents defined for the type Person:

```
select x.name
from x in person
where x.salary > 10
```

This modeling feature distinguishes Disco from other systems and permits the DBA to more easily manage scaling to a larger number of data sources. With the above ODL definitions, the query in the introduction will produce the answers described. Note that if the wrapper cannot match (or convert) the type in the mediator to the type in the data source, a run-time error will occur.

2.2 Matching similar and dissimilar structures

In general, when a DBA defines the aggregation of data from data sources, the need to access multiple data sources of similar structure or subsstructure, or sources of dissimilar structure, may arise. Disco provides subtyping for modeling similar substructures, *maps* for modeling similar structures, and *views* for modeling dissimilar structures. All these features can be applied while incorporating new data sources, and associating types of objects in the data sources to the types defined in the mediators. In related research [1, 14, 16, 15, 17], the main objective when integrating multiple data sources was obtaining a single unified type. In contrast, in Disco, we apply these features to the task of providing support for incorporating new data sources, by specifying the mapping among types in the mediator and the data source. We note that in this paper, we use an example of relational data sources. However, the Disco model can be applied to a variety of information servers, such as WAIS servers, file systems, specialized image servers, etc.

2.2.1 Subtyping

Subtyping is a method to organize collections of data sources with similar substructures. The subtype concept described here is directly obtained from the ODMG data model. Consider two data sources of students, in repositories r2 and r3. The DBA defines a Student interface as a subtype of Person, and the following extents:

```
interface Student:Person { }
extent student0 of Student wrapper w0
          repository r2;
extent student1 of Student wrapper w0
          epository r3;
```

The person extent still contains the two extents, person0 and person1. Thus, the extent of a type does not automatically reference the extents of its subtypes, in the subtype hierarchy. Disco therefore provides a special syntax, e.g., person*, for type Person, which recursively refers to the extents of all the subtypes of this type. Thus, the person* extent now contains four extents.

2.2.2 Mapping Disco types to data source types

In the previous section, we assumed that the type of the data source, and the type defined for the mediator accessing the data source, were identical. Recall that we assumed a relational data source. Then, the name of the data source relation is the name of the extent of the mediator type. Further, the names of the fields of the relation in the data source are identical to the names of the fields of the mediator type. In many existing systems, the burden of resolving the conflict between the two types is in the hands of the wrapper implementor. Disco provides some functionality to the DBA to resolve such conflicts. Here we consider the simple case where the type of the mediator and the type of the data source are different. A similar technique can be used to map multiple data sources to the same mediator type, or to map several mediator types to a single data source.

Suppose the DBA defines a different type, PersonPrime, with extent personprime0, to access

the data source named person, which has objects of type Person, as follows:

```
interface PersonPrime {
  attribute String n;
  attribute Short s; }
extent personprime0 of PersonPrime
          wrapper w0 repository r0;
```

Since Disco binds objects in data sources to types at run-time, these ODL statements are legal. Since objects returned from r0 are of type Person, the extent personprime0 has a type conflict with objects returned, and Disco will simply generate a run-time error. Disco allows the DBA to resolve this type conflict.

The DBA resolves type conflicts by specifying a mapping between a mediator type and a data source type. A mapping is a function from type to type. The mapping is called the *local transformation map*.

The local transformation map consists of a list of strings and is recorded in the map field of the extent. This corresponds to the field map of the meta-data type MetaExtent. Each string is either (1) an equivalence between the name of the data source (relation) and the name of the extent of the mediator type, or (2) an equivalence between the name of a field of the data source (relation) and the name of a field of the mediator type. The DBA resolves the type conflict in this example with the following map:

```
extent personprime0 of PersonPrime
          wrapper w0 repository r0
   map ((person0=personprime0),(name=n),
                (salary=s));
```

This map associates the name of the data source relation person0 with the name of the extent personprime0. Further, since personprime0 is of type PersonPrime, the map creates a one-to-one correspondence between the name field and n and salary and s, respectively. Thus, when a query is generated for this data source, by the mediator, it will refer to the attributes in the map to obtain the correct type for the data source. At present, maps are restricted to a flat structure, and they are defined as a list of strings. We plan to extend maps to handle nested types. A further extension is functions which map between domains and ranges, and will allow the mediator to resolve mismatch of values in the data sources during query processing.

In prior research [1, 14, 16], there has been much discussion about the mismatch of the data types, formats, values, etc., with respect to data sources and mediator types. In these previous approaches, the DBA resolves all conflict to obtain a single unifying type. Disco has no such objective. Our objective is to provide distinct types and appropriate techniques to resolve type mismatch. Our approach makes all types explicit in the mediators. Each addition of a type and resolution of a type conflict should be independent of any other type conflict.

2.3 Reconciling structures and data

The previous sections introduced features such as maps and subtyping to resolve mismatch between types. In general, some arbitrary transformations in the representation of a data source may be needed. This functionality is provided by query definition expressions, or *views* in Disco. The define ... as ... OQL syntax specifies a view consisting of a query name and a query. Views do not have explicit objects associated with them. The objects are referenced through the query name and are generated through executing the query.

Suppose a data source r5 of type PersonTwo, does not have a single salary field, but has two fields, regular for regular pay and consult for consulting pay. We may still wish to aggregate over the data sources. To do so, the different structures are included in a view definition. In this example, we assume that the people in the data sources of type Person are distinct from the people in r5 of type PersonTwo. The opposite assumption is also supported in Disco, but the view definition would be more complicated.

```
interface PersonTwo {
  attribute String name;
  attribute Short regular;
  attribute Short consult; }
extent persontwo0 of PersonTwo
          wrapper w0 repository r5;

define personnew as
  bag(select struct(name: x.name,
                    salary: x.salary)
     from x in person,
     select struct(name: x.name,
            salary: x.regular+x.consult)
        from x in persontwo0)
```

A view can reference other views, as long as the references are not cyclic. These views are not updatable.

3 Mediator query processing

The Disco mediator contains an internal database. The internal database records information on data sources, types, interfaces, views, etc. The mediator also contains a query optimizer and run-time system. The query optimizer searches for the best way to execute a query on the run-time system. The search is accomplished by transforming the query into several alternative logical expressions over an abstract machine (AM). Each logical expression can be executed by the mediator or by a wrapper, as appropriate. Each

expression has an associated estimated cost. The expression with the lowest estimated cost is then executed by the run time system [11].

Disco models calls to a wrapper with the submit(source, expression) logical operator. This operator means that the meaning of the logical expression expression is located at source. When the query optimizer translates an OQL query into a logical expression, references to extents are translated into the submit operator. The query optimizer generates a submit operator for each reference to an extent.

For example, the query optimizer translates the query

```
select x.name
from x in person
```

where person has extents person0 and person1, into the following logical expression:

```
union(project(name,submit(r0,
                          get(person0))),
      project(name,submit(r1,
                          get(person1))))
```

Reading in the order of application, from right to left, this logical expression means that the query retrieves tuples with the get operation from the person0 collection. The location of the tuples is specified in the r0 object. The submit operator accesses the tuples in the data source, and the name attribute is projected out of each tuple in the collection. The projection is done by the run-time system of the mediator. A similar operation is done with r1 and the results are combined into a bag.

Logical expressions containing the submit logical operator can be rewritten using transformation rules. For instance, one rule is to push a project into the argument of the submit, and therefore model the execution of the project directly on the data source. There are restrictions on the transformation rules. Some of these restrictions are based on the algebra and are well known. Additional restrictions are imposed by the functionality of the wrapper.

To determine the transformation rules applicable to the submit operator, Disco consults the wrapper interface(s) with a call to the submit-functionality method. The method returns a grammar. The grammar specifies the subset of the abstract machine, corresponding to the capability of the wrapper. For this example, the call may return a grammar for r0, indicating that it supports composition of the operators get and project, and it may return a grammar for r1, indicating that it only supports the operator get. More generally, multiple features of the composition of operators, the support for certain comparison operators, etc., can be defined. Transformation rules would operate on the logical expression output by the query optimizer, and produce the following logical expression, for this example:

```
union(submit(get(r0),
             project(name,get(person0))),
      project(name,submit(r1,
             get(person1))))
```

Note that the arguments of submit are in the name space of the mediator and do not yet refer to names in the local data source. That translation is performed by the wrapper.

Each submit call has a cost function which estimates the cost of execution for the particular submit call, during run-time. In the case of heterogeneous databases, estimating the cost function is difficult, since the data source may not export enough information to determine the run-time cost of each submit call. Disco solves this problem by recording previous submit calls to a data source and the actual cost of the call. When the submit call finishes, the arguments of the call, the time taken and the amount of data generated is recorded. A new call is compared to the previous calls. In the case that an submit call exactly matches a sequence of previous submit calls to a data source, a smoothing function may be used to combine the associated data to generate a new estimate. Only a fixed number of exactly matching calls are recorded.

In the case that the submit call does not exactly match, Disco searches for close matches. A close match is, e.g., a selection logical operator whose comparison operators match, but where the constants do not match. We believe that a variant of predicate-based caching [13] will accomplish close matching. While the associated statistics may be somewhat inaccurate, particularly if there is high data skew, we believe that these estimates are still useful. We plan to conduct experimental analysis of this problem.

4 Query processing with unavailable data

As mentioned in the introduction, scaling the number of heterogeneous data sources introduces the problem of access to unavailable data sources in a query. Since the Disco data model models data sources as objects, and the query language permits quantification over data sources, it is straightforward to write a query which accesses many data sources. It is likely that some of the data sources will be unavailable.

One approach to this problem assigns the meaning that the unavailable data source is considered to have no matching tuples. Another approach assigns a different meaning that the unavailable sources do not exist. Disco choses a third alternative. *The answer to a query is another query.* If all sources are available, then the query (answer) will contain only data. If data sources are unavailable, then, the answer is a partial evaluation of the original query. The partial evaluation corresponds to the available data sources. The unevaluated part of the answer corresponds to the unavailable data sources. This definition of an answer as a

query is included in OQL, since both queries and answers are treated as expressions. That is, OQL is closed with respect to queries and data.

Query processing proceeds normally until a designed time has elapsed. At this point, data sources are classified as unavailable or available. The query is rewritten into two parts, one which contains a query to the unavailable data sources, and the other containing the query that is currently being processed on the available sources. Query processing proceeds until the latter query consists only of data. Query processing then terminates and a two part answer which is a query in a special form is returned. The first part contains a query on the unavailable data sources and the second part contains data. It is also possible that sources that are initially available become unavailable before query processing has terminated; we have not considered this possibility here.

The partial evaluation proceeds as follows: The query is translated into a logical expression and submitted to the run-time system. The logical expression contains calls to the `submit` operator. These calls proceed in parallel. Calls to available data sources succeed. Calls to unavailable data sources block. After a designated time period, query evaluation stops. Then, the logical expression is *translated back into a high level query*. This translation is possible because each logical expression has a corresponding OQL expression. The new high level query is the partial evaluation of the query. It is also the answer to the query.

Thus, continuing the example from the previous section, suppose that the `r0` repository does not respond, but the `r1` repository produced the bag of strings `Bag("Sam")` as the result. `Disco` would translate the query on the unavailable source into a high level query, and combine it with the data obtained from `r1` to produce the following answer:

```
union(select x.name
    from x in person0,
    Bag("Sam"))
```

This approach has two advantages. First, the semantics of an answer are clearly defined. Second, if the unavailable data sources become available, and the answer is evaluated again, the original answer to the first query will be returned, as if all data sources were available in the first place.

5 Related work

Pegasus [1], UniSQL/M [16, 15] and SIMS [2] support mediator capabilities through a unified global schema which integrates each remote database and resolves conflicts among these remote databases [4] within this unified schema. These projects made substantial contributions in resolving conflicts among different schema and data models. Scalability was not explicitly addressed, and will pose problems, since the unified schema must be substantially modified as new sources are integrated. They also do not consider data sources that do not have a fixed schema, or servers which have a less powerful query capability. Federated multidatabases include Interbase* [22], which provides transaction semantics for federations of heterogeneous servers, and IRO-DB [10]. The latter also uses ODMG as the common model and provides integrated schemas and global transaction management. `Disco` differs from these projects since we focus on dynamic features such as scalability as new sources are added, and query processing when sources are unavailable.

Alternately, the capability of a mediator is supported by the use of higher-order query languages or meta-models [3, 8, 14, 17, 18]. Mediators are also implemented by mapping knowledge bases that capture the knowledge required to resolve conflicts among the local schema, and transformation algorithms that support query mediation and interoperation among relational and object databases [7, 19, 26, 27, 28]. Here, too, scalability is a problem, since the higher-order queries or the mapping knowledge has to adapt, as additional sources are incorporated.

In contrast to the unified global schema which resolves all conflicts among the entities of the local schema, the Garlic system [5], and research described in [9, 21, 20], assume a mediator environment based on a common data model. In [9], the common data model is the ODMG standard object model [6], which extends the OMG object-oriented data model [12]. Semantic knowledge expresses the mappings among the multidatabase interface description and the local interface descriptions corresponding to each local database. Semantic knowledge is expressed as general equivalences, $\text{query}_i \equiv \text{query}_j$, where each query is expressed using the OQL query language. Semantic knowledge includes mapping knowledge in the form of queries that are views over the union of the MDBMS and the local interfaces; equivalences expressing integrity constraints in the local and MDBMS interfaces, and equivalences expressing data replication in the local interfaces. All these equivalences are used for query reformulation. They address the problem of mismatch in the querying capability of the servers, since a query is reformulated using the views [9, 21, 20]. However, they do not focus on scalability issues. Although it is not described in this paper, we assume that there is such semantic knowledge, and it is used in query reformulation.

The focus of the TSIMMIS project [24, 25, 23] is the integration of structured and unstructured (schema-less) data sources, techniques for the rapid prototyping of wrappers and techniques for implementing mediators. The common model is an object-based information exchange model (OEM), which has a very simple specification. They too address the issue of mismatch in the querying capability of different data sources, and propose techniques for query reformulation that resolves this mismatch. In [25], they de-

scribe techniques for rapid prototyping of wrappers using query translation techniques. We expect to use similar techniques, and we extend the model with the explicit representation of data source objects, the ability to express mappings among types and a flexible query processing semantics.

6 Conclusions and future work

In summary, scaling the number of data sources in heterogeneous distributed databases introduces problems for end users, application programmers, database administrators and database implementors (wrapper implementors). The design of Disco provides solutions to some of the problems encountered by these users. Partial evaluation query semantics are provided to end users and applications programmers. Data modeling tools for modeling data sources as objects, and a simple language for resolving type conflicts are provided to the database administrator. The wrapper implementor uses a flexible wrapper interface to deal with the problem of the mismatch between the expressive power of the Disco system and the underlying data source.

Acknowledgements Thanks to Eric Dujardin, Daniela Florescu, Michael Franklin, Jean-Robert Gruser, Catherine Hamon, Peter Schwarz, and Victor Vianu. Olga Kapitskaia and Nicolas Gouble constructed a prototype `Disco 0`, based on the Flora query optimizer [9].

References

[1] R. Ahmed et al. The pegasus heterogeneous multidatabase system. *IEEE Computer*, 24(12), 1991.

[2] Y. Arens, C. Chee, C.-N. Hsu, and C. Knoblock. Retrieving and integrating data from multiple information sources. *Int. Journal of Intelligent and Coop. Inf. Systems*, 2(2), 1993.

[3] T. Barsalou and D. Gangopadhay. M(DM): An open framework for interoperation of multimodel multidatabase systems. *Int. Conf. on Data Engineering*, 1992.

[4] C. Batini, M. Lenzerini, and S. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.

[5] M. Carey et al. Towards heterogeneous multimedia information systems: the garlic approach. Technical report, IBM Almaden Research, 1995.

[6] R. Cattell et al. *The Object Database Standard - ODMG 93*. Morgan Kaufmann, 1993.

[7] S. Chakravarthy, W.-K. Whang, and S. Navathe. A logic-based approach to query processing in federated databases. Tech. rep., University of Florida, 1993.

[8] J. Chomicki and W. Litwin. Declarative definition of object-oriented multidatabase mappings. In *Distributed Object Management*. Morgan Kauffman, 1993.

[9] D. Florescu, L. Raschid, and P. Valduriez. Using heterogeneous equivalences for query rewriting in multidatabase systems. *Intl. Conf. on Cooperative Inf. Systems*, 1995.

[10] G. Gardarin et al. IRO-DB: A distributed system federating object and relational databases. In *Object-Oriented Multidatabase Systems : A solution for Advanced Applications*. Prentice Hall, 1996.

[11] G. Graefe. Encapsulation of parallelism in the volcano query processing system. *ACM SIGMOD Intl. Conf.*, 1990.

[12] *The Common Object Request Broker : Architecture and Specification*. Framingham, MA, 1992.

[13] A. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. *To appear in the VLDB Journal*, January 1996.

[14] W. Kent. Solving domain mismatch and schema mismatch problems with an object-oriented database programming language. *Very Large Data Bases*, 1991.

[15] W. Kim et al. On resolving schematic heterogeneity in multidatabase systems. *Distributed and Parallel Databases*, 3(1), 1993.

[16] W. Kim and J.Seo. Classifying schematic and data heterogeneity in multi-database systems. *IEEE Computer*, pages 12–18, December 1991.

[17] R. Krishnamurthy, W. Litwin, and W. Kent. Language features for interoperability of databases with schematic discrepancies. *ACM SIGMOD International Conference*, May 1991.

[18] L. Lakshmanan, F. Sadri, and I. Subramanian. On the logical foundations of schema integration and evolution in heterogeneous database systems. *Deductive and Object-Oriented Databases*, 1993.

[19] A. Lefebvre, P. Bernus, and R. Topor. Query transformation for accessing heterogeneous databases. *Joint Intl. Conf. and Symp. on Logic Programming*, 1992.

[20] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. *ACM PODS Symp.*, 1995.

[21] A. Levy, D. Srivastava, and T. Kirk. Data model and query evaluation in global information systems. *Journal on Intelligent Inf. Systems –Networked Information Retrieval*, 1995.

[22] J. Mullen, O. Bukhres, and A. Elmagarmid. Interbase*: A multidatabase system. In *Object-Oriented Multidatabase Systems*. Prentice Hall, 1995.

[23] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A mediation system based on declarative specifications. *IEEE Conference on Data Engg.*, 1996.

[24] Y. Papakonstantinou et al. Object exchange across heterogeneous information sources. *IEEE Conf. on Data Engg.*, 1995.

[25] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation schema for rapid implementation of wrappers. *Deductive and Object-Oriented Databases*, 1995.

[26] X. Qian. Semantic interoperation via intelligent mediation. *IEEE Data Engg. Conference, RIDE Workshop*, 1993.

[27] X. Qian and L. Raschid. Translating object-oriented queries to relational queries. *IEEE Intl. Conf. on Data Engg.*, 1995.

[28] L. Raschid and Y. Chang. Interoperable query processing from object to relational schemas based on a parameterized canonical representation. *Intl. Journal of Intelligent and Cooperative Inf. Systems*, 1995.

[29] P. Schwarz and K. Shoens. Managing change in the rufus system. *IEEE Intl. Conf. on Data Engg.*, 1994.