Scaling Heterogeneous Databases and the Design of Disco

Anthony Tomasic, Louiqa Raschid, Patrick Valduriez

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Scaling Heterogeneous Databases and the Design of Disco

Anthony Tomasic, Louiqa Raschid et Patrick Valduriez

N° 2704

Novembre 1995

PROGRAMME 1

ISSN 0249-6399

Scaling Heterogeneous Databases and the Design of Disco

Anthony Tomasic*, Louiqa Raschid** et Patrick Valduriez***

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet Rodin

Rapport de recherche n°2704 — Novembre 1995 — 25 pages

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 each new data source into the system. Database implementors must deal with the transformation of queries between query languages and schemas. The Distributed Information Search COmponent (DISCO) addresses these problems. Query processing semantics give meaning to queries that reference unavailable data sources. Data modeling techniques manage connections to data sources. The component interface to data sources flexibly handles different query languages and different interface functionalities. This paper describes in detail (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. We describe several advantages of our system and describe the internal architecture of our planned prototype.

Key-words: Heterogeneous, Distributed, Database, Autonomous, Mediator, Wrapper, Partial Evaluation, Unavailable Data, Database Implementation, Query Optimization

(Résumé : tsvp)

*Address: INRIA Rocquencourt, 78153 Le Chesnay, France. e-mail: Anthony.Tomasic@inria.fr, http://rodin.inria.fr/person/tomasic

**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. Address: University of Maryland, College Park, MD, 20742, USA. e-mail: louiqa@umiacs.umd.edu, http://www.cs.umd.edu/users/louiqa/

***Address: INRIA Rocquencourt, 78153 Le Chesnay, France. e-mail: Patrick.Valduriez@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Bases de données hétérogène en grand et le conception de Disco

Résumé :

L'accès à un grand nombre de sources de données introduit de nouveaux problèmes pour les utilisateurs de sources de données hétérogènes distribuées. Les utilisateurs et programmeurs d'applications doivent prendre en compte les sources de données non disponibles. Les adminstrateurs de sources de données s'occupent d'ajouter de nouvelles sources de données dans le système. Les développeurs de sources de données s'occupent de la transformation des requêtes. Le *Distributed Information Search COmponent* (DISCO) s'occupe de ces problemes. La sémantique de traitement des requêtes gère les situations où les sources de données ne sont pas disponibles. Les techniques de modélisation de données gèrent la communication avec les sources de données. L'interface de médiateur avec les sources des données gère les différents langages et les fonctionnalités des différents interfaces. Ce article décrit en détail (a) l'architecture de mediateurs distribuées de DISCO, (b) la sémantique des requêtes, (c) le modele de donné et le modele de communication avec les sources de données, et (d) l'interface avec les sources de données. Nous décrivons aussi plusieurs avantages de notre système et ainsi que l'architecture interne de notre prototype.

Mots-clé : Base de Donnée Hétérogène Distribué, Autonome, Médiateur, Wrapper, Évaluation partielle, Donnée non disponible

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 hard to query. 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.

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. Since the data source accessed by an object is defined by an object, several objects can access the same data source as attributes of different object signatures. The DBA is faced with the problem of integrating a large number of data sources which are very similar in structure. DISCO provides special features to ease the integration of multiple data sources having the same type. To more clearly explain these issues, we describe the architecture for a heterogeneous distributed database system, and then describe various features of this architecture.

1.1 Architecture

As shown in Figure 1, current distributed heterogeneous database systems [5, 9, 21] deal with scale 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 data 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 of 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.

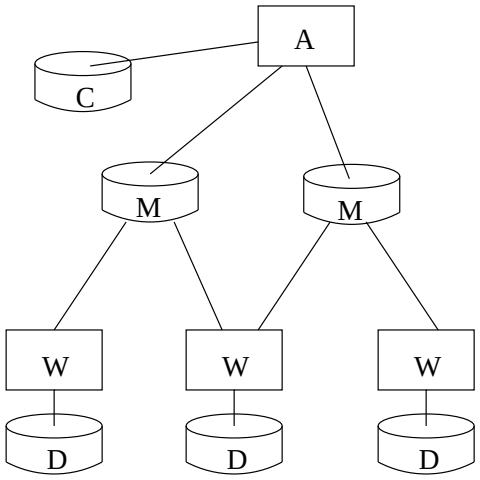

Figure 1: DISCO architecture. Boxes represent stateless components. Disks represent components with state. A stands for application, M for mediator, C for catalog, W for wrapper, and D for database. Lines represent exchange of queries and answers between components.

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 the mediators, to the particular query language of the data source. A wrapper supports the functionality of transforming 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 novel 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.

To be more concrete, we describe each feature of DISCO namely the data model, the semantics of query processing and the wrapper interface description of the data sources, using examples.

1.2 Mediator 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 Mediator 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 a return partial answer to queries, by processing as much of the query as is possible, from the information that is available. Thus, the answer to a query may be another query.

Consider the above query and suppose that the `r0` data source does not respond. The above query would be answered with 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. Note that 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. We are able to support this query processing semantics since the data model represents each data source as an object.

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 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 set of supported logical operators. During query processing, a DISCO mediator query optimizer generates a logical expression for the wrapper. The mediator calls the wrapper interface to get the supported set of logical operators, and checks that the logical expression is legal with respect to the supported set of operators.

For example, a mediator may generate a logical expression to project the `name` attribute from a relation `r`.

```
project(name, get(r))
```

The mediator will pass this logical expression to a wrapper, thereby, pushing the project operation onto the wrapper, only if the wrapper interface supports the `project` and `get` logical operators, and only if the wrapper supports composition of these logical operators.

1.5 Summary

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 [15], or queries that refer to particular data sources. As a result, the mediator can use the full power of the 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. Further, the full power of the OQL query language may be applied to support view interfaces that range over the mediator types. 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

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 data model of DISCO is based on the ODMG-93 data model specification [6]. We extend the 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.

2.1 Extensions to the ODMG Standard

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", 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, for the mediator. Thus, DISCO provides a special syntax for the addition of an extent, as follows:

```
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, 27])

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
```

This query will return 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 sections 2.2.3 and 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 transformed 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 and deleted directly by adding and deleting objects of type `MetaExtent`. For example, the extent created by the expression

```
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
```

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.

This modeling feature distinguishes DISCO from other systems and permits the DBA to more easily manage scaling to a larger number of data sources. Two other approaches are used. One approach explicitly reference databases in the rules and thus require direct modification of rules when new data sources are introduced. Another approach uses higher-order logics which carry an added complexity. DISCO chooses a compromise between these two approaches.

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 substructure, 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, 11, 13, 14, 15], 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. Suppose there are two data sources of students in repositories `r2` and `r3`. The DBA simply 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 repository r3;
```

The `person` extent still contains only the two extents, `person0` and `person1`. Thus, the extent of a type does not automatically reference the extents of the sub-types of that type, 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 mediator applies the map to queries before passing them to wrappers.

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, 11, 13], 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.

2.2.3 Views

Maps and subtyping are a very restricted form of transformation to resolve mismatch between types. In general, arbitrary transformations in the representation of a data source may

be needed. In addition, DBAs may need to define *reconciliation functions* [1] which determine how data values from different sources are combined. 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. The following view:

```
define double as
    select struct(name: x.name, salary: x.salary + y.salary)
    from x in person0 and y in person1
    where x.id = y.id
```

specifies a mapping from the query name `double` to the corresponding query. This query uses a `select` expression. The query is evaluated over the extents `person0` and `person1`. Thus, the query definition specifies a mapping to underlying data sources. The variable `double` or the query name, is a bag of `structs`. Access to `double` computes all people who reside in both data sources and returns a bag containing, for all people in both data sources, the name of the person from the `r0` data source and the sum of the salaries of the person from both data sources. Thus, reconciling the salaries of two data sources has been done by simply using the addition function. Reconciliation functions are indistinguishable from other functions. Since the full power of OQL is available in the view definition language, aggregate functions are also possible.

To aggregate over an arbitrary number of data sources, we simply use select in the aggregate function, as follows:

```
define multiple as
    select struct(name: x.name,
                  salary: sum(select z.salary
                              from z in person
                              where x.id = z.id))
    from x in person*
```

In this case, suppose a new student data source `r4` is added, and an extent is added to the type `Student`, which is a subtype of `Person`. Since `person*` references the extents of `Student`, through the type hierarchy, the salaries of students in the new data source `r4` will automatically be summed in the `multiple` view definition.

Reconciliation functions deal with *semantic* conflicts between data at each source. Some systems [1] provide a built-in set of reconciliation functions. During query processing, the mediator uses special heuristics to optimize the processing of the built-in functions. For instance, some data sources process reconciliation functions directly, and mediators pass the function call to the data source. We plan to include these techniques into later prototypes of DISCO.

2.3 Reconciling Structures and Data

The transformational language of the previous section introduced features to permit two data sources to appear alike, when the structures of the types of the collections in the local data sources were similar. However, we may also want to aggregate over data sources with dissimilar structures. To accomplish this, we introduce multiple levels of views.

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, and the different structures must be included in the 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 is 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.

3 Mediator Query Processing

The DISCO mediator contains an internal database. The internal database records information on data sources, types, interfaces, and view, 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 expressions which can be executed by the run-time system. Each expression has an associated estimated cost. The expression with the lowest estimated cost is then executed by the run time system. The run time system makes calls to the wrapper interface to access external data sources. The next section briefly describes the model of the query optimizer. We then describe an extensions to this model to incorporate wrappers.

3.1 The query optimizer model

The query optimizer manipulates several abstractions [10] when searching for an optimal plan. The optimizer first accepts queries written in the declarative OQL and transforms the

query into an expression on an algebraic machine. *Logical operators* compose the expression. DISCO has the usual logical operators of `project`, `join`, etc. *Transformation rules* rewrite logical expressions to equivalent logical expressions. DISCO has the usual transformation rules such as commuting and associating join. The optimizer also has *physical algorithms* which implement operations. DISCO has the usual physical algorithms such as for merge-join, file-scan, etc. Logical operations are transforms into physical expressions using *implementation rules*. DISCO has the usual transformations rules that implement `join` with `merge-join`. Thus, the query optimizer transforms logical expressions into physical expressions. Finally, *cost functions* estimate the cost of a physical algorithm's execution in a tree. The optimizer searches the space of logical and physical trees for the physical tree with the lowest cost. The run-time system executes the physical expression with the lowest cost.

3.2 The `submit` logical operator

DISCO models calls to a wrapper with the `submit(source, expression)` logical operator. This operator means that the meaning of `expression` is located at `source`. The query optimizer generates a `submit` operator for each access to a data source. When the query optimizer transforms an OQL query into a logical expression, references to extents are transforms into the `submit` operator.

For example, the query optimizer transforms the query

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

when `person` has extents `person0` and `person1` into the 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 access 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. Note that the arguments of `submit` are in the name space of the mediator. The arguments do not refer to names in the local data source. Finally, a similar operation is done with `r1` and the results are combined into a bag.

Logical expressions containing the `submit` logical operator can be written 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. When applying a transformation rule to the `submit` operator, the transformation rule consults the wrapper interface with a call to the `submit-functionality` method.

The method returns a definition of the functionality of the wrapper. The definition includes the set of logical operators supported and support for (or lack of) composition of logical operators. For instance, the call may return {`get, project, compose`} for `r0` but only {`get`} for `r1`. This functionally produces the following logical expression.

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

More generally, multiple features of the composition of operators, the support for certain comparison operators, etc., can be defined by returning a grammar, e.g. for the case that a wrapper understands `get` and `project` of sources, but not the composition of this operations, the grammer

```
a :- b
a :- c
b :- get OPEN SOURCE CLOSE
c :- project OPEN ATTRIBUTE COMMA SOURCE CLOSE
```

would be returned where `get`, `project`, `ATTRIBUTE`, `COMMA`, and `SOURCE` are obvious and predefined terminal symbols in the grammer. `OPEN` and `CLOSE` mean “(” and “)”. A wrapper that understands these two operations and composition of them would return

```
a :- b
a :- c
b :- get OPEN s CLOSE
c :- project OPEN ATTRIBUTE COMMA s CLOSE
s :- b
s :- c
s :- SOURCE
```

Another example of transformation rules permits join operations to be pushed to the wrapper. The logical expression

```
join(submit(get(r0), get(employee0)),
     submit(get(r0), get(manager0)),
     dept)
```

can be rewritten with a transformation rule to

```
submit(get(r0), join(get(employee0), get(manager0), dept))
```

if the functionality of the wrapper accepts `join`.

The `submit` logical operator has a disadvantage. The operator accepts a logical expression as an argument and has function call (or remote procedure call) semantics. The operator cannot accept data from another data source. This restriction implies that the full generality of distributed and parallel database algorithms can not be expressed. For

example, semijoin cannot be expressed with the `submit` operator because it requires the transmission of results directly between data sources. Future work for the DISCO project will extend the logical model to include more logical operators [10].

In addition, wrappers do not support references to mediator objects and values. That is, object oids are not pass through the wrapper interface. We assume that all references to objects are based on values. Furthermore, path expressions also cannot cross from parts of the data model in the mediator to parts of the data model in data sources. Finally, for the moment, function calls defined in the mediator cannot be passed to the data source. Transformation rules insure that wrapper functionality is not violated.

The passing of operations onto data sources through wrappers introduces a subtle problem of semantics. The definition of the semantics of the operator must be exactly the same for the mediator and the underlying data source. This exact match of semantics is rarely achieved in practice. Unfortunately, if the semantics do not match exactly, but a transformation rule permits the rewriting of two (non) equivalent expressions, the meaning of the answer of the query changes depending on the query optimization plan chosen.

3.3 The `exec` physical algorithm

The logical expression is transforms into a physical expression using implementation rules. The `submit` logical operator is implemented by the `exec` physical algorithm. Thus, the above logical expression is transformed into the physical expression

```
mkunion(exec(field(r0), project(name,get(person0))),
      mkproj(name,exec(field(r1),get(person1)))),
```

where `field` is the physical algorithm corresponding to `get` when the argument is a single object. Notice that the second argument of `exec` is still a logical expression, because the wrapper interface accepts a logical expression. `exec` transforms the second argument logical expression a logical expression in the name space of the data source using the map. In addition, it is responsible for calling the wrapper retrieving the results from the wrapper.

Each physical algorithm has a cost function which estimates the cost of the execution of a physical algorithm during run-time. In the case of heterogeneous databases, this cost function introduces a problem, since the data source may not export enough information to determine the run-time cost of a physical algorithm. DISCO solves this problem by recording previous `exec` calls to a data source and the actual cost of the call. When the `exec` 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 the an `exec` call exactly matches a sequence of previous `exec` calls to a data source, a smoothing function is 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 `exec` call does not exactly match, DISCO searches for close matches and uses the close matches as input to a smoothing function. A close match is, e.g., a `selection` logical operator whose comparisons operators match but whose constants do

not match. We believe that a variant of predicate-based caching [12] will accomplish close matching. While the associated statistics may be somewhat inaccurate, particularly in this case if there is high data skew, we believe that the statistics are still useful. We plan to conduct experimental analysis of this problem.

In the case that there are no close matches to the exec call, a default time cost of 0 and a data cost of 1 is used. This default implies that in the case that no cost information is available for a collection of data sources, the optimizer will choose plans of a special form. It will chose plans where the maximum amount of computation is done at the data source, since every logical operation done at the data source has a 0 time cost. In addition, once all possible computations are pushed to the data source, the optimizer will choose the plan with the lowest cost with respect to the mediator, since the cost of computation at all data sources is equal.

Finally, if query optimization plans are cached, the mediator must monitor updates to extents, and modify or recompute plans that are affected by updates to the extents understood by the mediator.

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.

On approach to this problem assigns a meaning to an unavailable data source. For instance, the data source can be considered to have no tuples. Another approach assigns a new meaning to queries in the presence of unavailable data. For instance, a query can be evaluated as if the data source objects which reference unavailable sources do not exist. DISCO chooses a third alternative. *The answer to a query is another query.* 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 simply in the OQL, since both queries and answers are simply 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 are unavailable. The query is rewritten into two parts, one which contains a query to the unavailable data, and the other contain the remaining of the query to be processed. Query processing proceeds until the remainder part consists only of data. Query processing then terminates and a two part answer is returned. The answer is a query in a special form. The first part contains a query on the unavailable data sources and the second part contains data.

The partial evaluation proceeds as follows. The query is transformed into a physical expression and submitted to the run-time system. The physical expression contains calls to the `exec` 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 physical expression is *transformed back into a high level query*. This transformation is possible because each physical operation has a corresponding logical operation, and each logical operation 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 transform the unavailable data into a high level query and combine it with

```
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. Note that the user may always simply issue the original query again.

One problem with partial evaluation involves the underlying semantics of queries. Suppose two data sources a and b time stamp each data value when it is added to the data source. Suppose data source b is unavailable, and a user evaluates a query over both sources. The answer will contain data from a but no data from b. Suppose a and b change, and the answer is resubmitted as a new query. The answer (interpreted as a query) will contain the updated tuples from b but no updated tuples from a. It would be convenient for the user to able to check if the data from a was still valid. That is, the answer to the query contains additional predicates which check the most recent time stamps of the relevant relations for new tuples. This check is similar to checking for incremental updates to integrity constraints.

5 Related Work

Pegasus [1], UniSQL/M [13, 14] 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.

UniSQL/M [13, 14] is a commercial multidatabase product; virtual classes are created in the unified schema to resolve and "homogenize" heterogeneous entities from relational and object-oriented schema. Instances of the local schema are imported to populate the virtual classes of the integrated schema, and this involves creating new instances. The first step in integration is defining the attributes (methods) of a virtual class, and the second step is a set of queries to populate this class. They provide a vertical join operator, similar to a tuple constructor, and a horizontal join, which is equivalent to performing a union of tuples. The major focus of their research is conflicts due to generalization, for e.g., an entity in one schema can be included i.e., become a subclass of an entity in the global schema, or a class and its subclasses may be included by an entity in the global schema. Attribute inclusion conflicts between two entities can be solved by creating a subclass relationship among the entities. Other problems that are studied are aggregation and/or composition conflicts. In Pegasus [1], queries access the local schema via the *imported* Pegasus global schema. They use the HOSQL high-level language to define "imported types" (corresponding to class definitions) and functions (relationships among instances). New objects are generated for instances of each imported type. For supporting schema integration, they define equivalences among objects, reconciliation of discrepancies, and "covering" supertypes which are collections of instances of different imported types. In the SIMS system [2], information sharing from multiple relational schema is facilitated through using the LOOM knowledge representation schema to construct a global schema for each application domain. Here, the global query language is a LOOM query. Each external relation has to be mapped into a single LOOM concept, based on some notion of a primary key, and they cannot express a view over the external relations. This can be a drawback if the corresponding concepts or entities in the schemas are mismatched. Although they research many issues in query processing, this work cannot be applied in the context of heterogeneous DBMS, supporting SQL-like query languages.

Alternately, the capability of a mediator is supported by the use of higher-order query languages or meta-models [3, 8, 11, 15, 16, 21]. The language or model provide constructs to resolve conflicts among the sources. Here, too, scalability is a problem, since the higher-order queries or the model have to be significantly changed, as additional sources are incorporated.

In [15], the higher-order language features needed for interoperability based on relational schema is presented. They define a powerful language which can query schema; its variables can range over databases, relations, attributes and values. Queries against a unified schema (which is a nested relational object) are expressed using an Interoperable Definition Language (IDL). A disadvantage is that the DBMS must support the higher-order language, and queries are also expressed in this language. Thus, it does not allow the interoperation of legacy applications. SchemaLog [16] is a higher-order logic with formal semantics. It is a very expressive declarative language that can query multiple schema. A query has higher-order syntactic features but the logic is a first-order logic and has model and proof semantics. One disadvantage is that resolution (unification) for literals in the formula can be complex, compared to Prolog unification. Although the research is interesting,

it is not applicable in a DBMS environment with legacy applications. In [8], a language for declarative specification of mapping between different object-oriented multidatabases is presented. Finally, the M(DM) meta-model uses meta-level descriptions of schemas to facilitate interoperation [3]. They build an inheritance lattice to organize meta-types of the schema. Thus, there is no unified schema based on a single data model. Queries are expressed against the meta-model and transformed against local relational, object, or other schema. Second-order logic is used to reason about the meta-types. Again, this research does not support the interoperation of legacy applications.

Mediators are also implemented through the use of mapping knowledge bases that capture the knowledge required to resolve conflicts among the local schema, and mapping or transformation algorithms that support query mediation and interoperation among relational and object databases [7, 17, 24, 25, 26]. Here, too, the emphasis is on resolving conflicts among schema and data models, to support interoperability of the queries.

Lefebvre *et al* (1992) In [17], F-logic, a second order logic, is used to express the mapping information among relational schemas and to express the algorithm for query transformation. In [24], a language which has minimal *representation bias* expresses mismatch in representation among heterogeneous schema. They choose a first order deductive database to represent mapping knowledge among different relational schema. Each SQL query is converted to some restricted clausal form. The relational schema and the corresponding integrity constraints are also expressed in the form of an implication of some restricted clausal form, where all variables in the body of the clause are universally quantified and all variables in the head are existentially quantified. A mediation knowledge base (of such restricted clauses) is built. An advantage of this approach is that the query, the schema, the constraints and the mediation knowledge, are all expressed in the same language. An important aspect of this research is that it uses a theorem proving approach rather than a transformation approach to transform the queries. In [25], this approach is extended to resolve mapping among object and relational schema. They consider queries with higher-order features in the XSQL language. They use a canonical deductive database to represent the object schema and the mapping knowledge. The higher-order features in the query are resolved in the first step and in the next step the query is simplified and optimized. Finally, it is transformed using a set of mapping rules to obtain a query wrt some target relational schema.

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, 18, 19], 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 [20]. 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 equivalences, in a general form, $\text{query}_i \equiv \text{query}_j$, 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, 18, 19]. 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 system described in [18] performs query reformulation using schema mapping knowledge. Their common object model is an object-oriented extension of the relational model based on a description logic. The representation language is Datalog-like, and thus, their queries are not as expressive as OQL queries. A concept in the world view (MDBMS) may be expressed as a conjunctive Datalog-like query over the local relations, and they may also express a local relation as a (conjunctive) query over the world view relations. However, they are not able to express general integrity constraints in the local interfaces. The reformulation algorithm described in [18] is limited, since they try to match each global entity in the world view, against the mapping knowledge. Thus, they are not able to match all conjunctive queries expressed over the the world view entities, even if there exists a local entity defining this world view query (or a fragment of it). They cite an extension of their algorithm [19], which is able to answer a larger class of queries, by matching a conjunctive query against a conjunctive view, to produce an equivalent query, and the algorithm is NP-complete. The intent is to obtain an equivalent query which is minimal, in that they reduce the number of literals that appear in the equivalent query. However, they note that minimality is not essential in obtaining an optimized equivalent query. This is especially true in a heterogeneous environment, where the view may be expressed over local information sources, which have dissimilar costs. In comparison to [18], the OQL query language that we use to express semantic knowledge is much more expressive. We are able to express rewrite rules which replace a view in the MDBMS interface with an OQL query over the union of the local *and* the MDBMS interface. Thus, we are directly able to describe a mapping corresponding to an object in the in the MDBMS interface, which may have a reference, (ODMG relationship), with another object. Such a mapping for object references could not be explicitly expressed in any previous work. We are also able to utilize other semantic knowledge, e.g., data replication, for query reformulation. The extended pattern matching of our reformulation algorithm allows us to identify (a subquery of) a user query which can be replaced by a rewrite rule. Since the result of query, which is essentially a view, can be used to replace a subquery in the user query, we are able to cover the same space as the the algorithm in [19], with the caveat that we are reformulating wrt a much more complex and expressive query language. We also note that the space of query reformulation is not necessarily those queries in which we minimize the number of collections, as described in [19]. However, we are able to eliminate some collections in the query, based on semantic knowledge. This simplification is more general than the minimality criterion of [19], which does not exploit semantic knowledge.

The focus of research in the TSIMMIS project [21, 22, 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 [22], they describe techniques for rapid prototyping of wrappers using query transformation 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. However, they do not explicitly model each of the data sources, and scalability and flexibility of query processing, as additional sources are incorporated, may still pose problems.

TSIMMIS has components that extract properties from unstructured objects, transform information into a common model, combine information from several sources, allow browsing of information, and manage constraints across heterogeneous sites. specification. A corresponding query language, LOREL, is also proposed. For a given query, different attributes of objects are obtained from different information sources and the results are resolved for data inconsistencies.

6 Conclusion

6.1 Summary

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.

1. Partial evaluation query semantics provides end users and applications programmers with queries over unavailable data sources.
2. Data sources are first class objects which aid the database administrator in modeling the system.
3. A collection of tools (subtyping, schema mapping, and views) aid the data administator in modeling data sources.
4. A flexible wrapper interface aids the wrapper implementor in dealing with the problem of the mismatch between the expressive power of the DISCO system and the underlying data source.

We are currently constructing Mediator Prototype 0. The architecture of the prototype is simple so that the basic issues of the language extensions can be studied. The prototype consists of a single process and a single system which combines the functionality of wrappers and mediators. Figure 2 diagrams the architecture of Prototype 0. The prototype

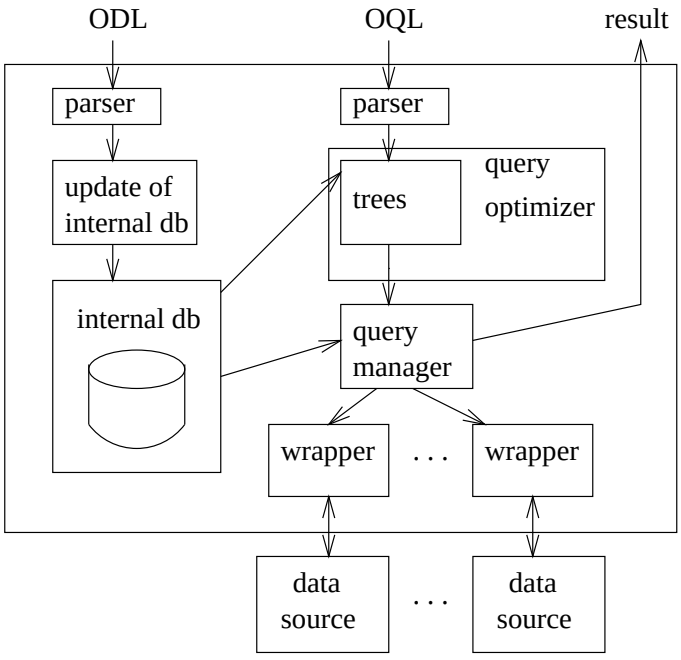

Figure 2: The architecture of Mediator Prototype 0.

accepts ODL, OQL, and the DISCO extensions to these languages. Query processing will follow the outline described in Section 3.

6.2 Future Work

Several aspects of DISCO present new research problems. The distributed architecture introduces several performance issues, since in its the most general case network communication occurs between several components to process a single query. The introduction of data sources as objects mixes the meta data level and the data level, since the schema of the objects is at the meta data level. This mix introduces several problems to the query optimizer, since any compile-time optimizations are based on the state of the data in the mediator. The data model is not orthogonal with respect to distributed computing. That is, the data model has been changed to reflect the semantics of wrappers, etc. We believe that a definition based on implementations of interfaces produces an model which is orthogonal with respect to the distributed semantics of the model. The grammar returned by a wrapper may have various levels of expressive power. The appropriate level is not clear at the moment. One generalization of this interface uses *subtree match* to matching the largest part of the expression submitted to the interface. This matching problem is similar to back end code generation of parse trees in compiler technology. Essentially, the interface searches for the lowest cost subtree which it can process. This search is analogous to searching for the largest parse subtree which can be transformed into a sequence of assembly instructions. In addition, the language interface level to the wrapper is an abstract algebra machine. If the data source underlying the wrapper understands SQL, this algebraic expression will be transformed “up” into a high level expression, optimized again, and transformed back down into a low-level expression. On the other hand, the wrapper may use the underlying database API. Partial answers introduces several problems. First, it is not clear that users want partial answers. Second, application programs must now handle expressions as answers, as opposed to simple relations or sets of objects. The nature of this interface is an open issue. In addition, it’s not clear exactly how the mapping of value from the mediator to the domain of the database (and in reverse) is accomplished. For example, the mediator models salaries as yearly values, but the data sources models salaries as weekly values. Finally, we have not really address the issue of representation of object oids. We simply assume every reference is value-based.

Acknowledgements Thanks to Eric Dujardin, Daniela Florescu, Mike Franklin, Jean-Robert Gruser, Catherine Hamon, Alexandre Lefebvre, Yannis Papakonstantinou, Peter Schwarz, and Victor Vianu for discussions on this project. Olga Kapitskaia and Nicolas Gouble constructed a trial prototype based on the Flora query optimizer [9].

References

[1] R. Ahmed *et al* ., “The Pegasus Heterogeneous Multidatabase System”. *IEEE*

Comp.,24(12), 1991.

[2] Y. Arens, C.Y. Chee, C.-N. Hsu and C.A. Knoblock. “Retrieving and Integrating Data From Multiple Information Sources.” *Int. Journal of Intelligent and Coop. Information 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 and M. Lenzerini and S.B. Navathe. “A Comparative Analysis of Methodologies for Database Schema Integration.” *ACM Computing Surveys, volume 18, number 4, pages 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.G.G. Cattell *et al* . *The Object Database Standard - ODMG 93*. Morgan Kaufmann, 1993.

[7] Chakravarthy, S, Whang, W-K. and Navathe, S.B. “A Logic-Based Approach to Query Processing in Federated Databases.” *Technical Report*. 1993.

[8] J. Chomicki and W. Litwin. “Declarative definition of object-oriented multidatabase mappings.” *Distributed Object Management, M.T. Oszu and U. Dayal and P. Valduriez, editors. Morgan Kauffman* 1993

[9] D. Florescu, L. Raschid and P. Valduriez. “Using Heterogeneous Equivalences for Query Rewriting in Multidatabase Systems.” *Proceedings of the Intl. Conf. on Cooperating Information Systems*, 1995.

[10] Graefe, Goetz. “Encapsulation of Parallelism in the Volcano Query Processing System.” *ACM SIGMOD Int. Conf.*, 1990.

[11] Kent W. “Solving Domain Mismatch and Schema Mismatch Problems with an Object-Oriented Database Programming Language.” *Proceedings of the International Conference on Very Large Data Bases*, *Proceedings of the VLDB*, 1991.

[12] Keller, Arthur M. and Basu, Julie, “A Predicate-based Caching Scheme for Client-Server Database Architectures.” to appear in VLDB Journal, January 1996.

[13] Kim,W. and J.Seo, “Classifying Schematic and Data Heterogeneity in Multi-Database Systems.” *IEEE Computer, 12-18,* Dec. 1991.

[14] W. Kim *et al* ., “On Resolving Schematic Heterogeneity in Multidatabase Systems.” *Distributed and Parallel Databases, 1(3)*, 1993.

[15] R. Krishnamurthy, W. Litwin and W. Kent, “Language Features for Interoperability of Databases with Schematic Discrepancies”. *ACM SIGMOD Intl. Conf.*, 1991.

[16] L.V.S. Lakshmanan, F. Sadri and I.N. Subramanian, “On the Logical Foundations of Schema Integration and Evolution in Heterogeneous Database Systems”. *Int. Conf. on DOOD, 1993.*

[17] Lefebvre,A., P. Bernus and R. Topor, “Query Transformation For Accessing Heterogeneous Databases.” *Proceedings of the Joint International Conference and Symposium on Logic Programming, Workshop on Deductive Databases*, 1992.

[18] A.Y. Levy, D. Srivastava and T. Kirk, “Data Model and Query Evaluation in Global Information Systems.” *Int. Journal on Int. Inf. Systems - special issue on Networked Information Retrieval*, to appear 1995.

[19] A.Y. Levy, A.O. Mendelzon, Y. Sagiv and D. Srivastava, “Answering Queries Using Views”. *Proceedings of the ACM PODS Symposium*, 1995.

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

[21] Y. Papakonstantinou and H. Garcia-Molina and J. Widom. “Object Exchange Across Heterogeneous Information Sources.” *Proceedings of the International Conference on Data Engineering*, 1995.

[22] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina and J. Ullman. “A Query Translation Schema for Rapid Implementation of Wrappers.” *Fourth International Conference on Deductive and Object-Oriented Databases*, to appear 1995.

[23] Y. Papakonstantinou and H. Garcia-Molina and J. Ullman. “MedMaker: A Mediation System Based on Declarative Specifications.” *Proceedings of the IEEE Conference on Data Engineering* , to appear 1996.

[24] X. Qian. “Semantic Interoperation via Intelligent Mediation.” *Proceedings of the IEEE Data Engineering Conference, Workshop on Research Issues in Data Engineering*, 1993.

[25] Qian,X. and Raschid, L., “Translating Object-Oriented queries to Relational Queries.” *Proceedings of the IEEE International Conference on Data Engineering*, 1995.

[26] Raschid, L. and Chang, Y. “Interoperable Query Processing from Object to Relational Schemas Based on a Parameterized Canonical Representation.” *To appear in the International Journal of Intelligent and Cooperative Information Systems*, 1995.

[27] Schwarz, Peter and Shoens, Kurt. “Managing Change in the Rufus System.” *Proceedings of the IEEE International Conference on Data Engineering*, 1994.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LES NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399