

# Scaling Access to Heterogeneous Data Sources with DISCO

DRAFT – NOT FOR DISTRIBUTION – SEE TKDE 1998 FOR FINAL VERSION

Anthony Tomasic, Louiqa Raschid and Patrick Valduriez

*Abstract*— Accessing many data sources aggravates problems for users of heterogeneous distributed databases. Database administrators must deal with *fragile mediators*, that is, mediators with schemas and views that must be significantly changed to incorporate a new data source. When implementing translators of queries from mediators to data sources, database implementors must deal with data sources that do not support all the functionality required by mediators. Application programmers must deal with *graceless failures* for unavailable data sources. Queries simply return failure and no further information when data sources are unavailable for query processing. The Distributed Information Search COmponent (DISCO) addresses these problems. Data modeling techniques manage the connections to data sources, and sources can be added transparently to the users and applications. The interface between mediators and data sources flexibly handles different query languages and different data source functionality. Query rewriting and optimization techniques rewrite queries so they are efficiently evaluated by sources. Query processing and evaluation semantics are developed to process queries over unavailable data sources. In this article we describe (a) the distributed mediator architecture of DISCO; (b) the data model and its modeling of data source connections; (c) the interface to underlying data sources and the query rewriting process; and (d) query processing semantics. We describe several advantages of our system.

*Keywords*— Heterogeneous Database, Query Reformulation, Source Capability, Heterogeneous Cost Model, Partial Answer, Partial Evaluation

## I. INTRODUCTION

DATABASE systems have 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. Distributed databases, i.e., systems that access multiple databases simultaneously, also have the same types of users. Distributed databases may be either homogeneous or heterogeneous. Homogeneous distributed databases require that every underlying database conforms to the same data model and query language. Heterogeneous distributed databases relax this restriction and permit each underlying database to have different data models, query languages, and thus, different functionality. Since some underlying databases may be very simple, i.e., a file, we use the term *data source*

This work was supported in part by the Groupement d'Intérêt Economique Dyade (a joint R&D venture between Bull and INRIA); by the Defense Advanced Research Project Agency under grants 92-J1929 and 01-5-28838; and by the National Science Foundation under Grants CDA9422138 and IRI 9630102. A. Tomasic and P. Valduriez are with the Institut National de Recherche en Informatique et en Automatique (INRIA), 78153 Le Chesnay, France. L. Raschid is with the Maryland Business School, University of Maryland, College Park, Maryland 20742, USA.

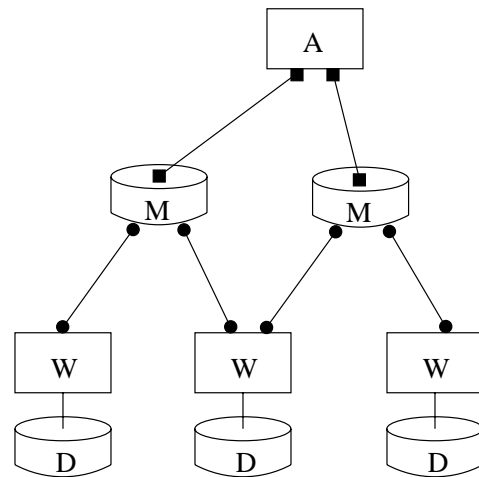


Fig. 1. DISCO architecture. Boxes represent stateless components and disks represent components with state. A stands for application, M: mediator, W: wrapper, and D: data source. Lines represent exchanges of queries and answers. Solid boxes mark the application-mediator protocol and the solid circles mark the mediator-wrapper protocol.

instead of the term database.

### A. Architecture

As shown in Figure 1, current distributed heterogeneous database systems [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11] adopt a distributed architecture that consists of several specialized components. End users interact with applications (A) written by application programmers. Applications access underlying data sources via *mediators* (M) [11]. Mediators export a mediator schema that is an integrated representation of data sources. Mediators process queries over the integrated representation. To construct this representation, DBAs provide information to mediators to accomplish query processing. This information consists of the schemas of the data sources, the mediator schema, and views which transformation queries between data source schema and the mediator schema. To deal with the different query languages of each data source, *wrappers* (W) transform sub-queries sent to data sources and transform answers returned from data sources. Wrappers map from a subset of the functionality of a general query language, used by mediators, to the particular query language of the source. The *wrapper implementor*, a new specialty of DBI, writes wrappers for each type of data source.

This architecture has several advantages. First, the specialized components of the architecture allow the various

concerns of different users to be handled independently. Second, mediators typically specialize in a related set of data sources with “similar” data, and thus export schemas and semantics related to a particular domain. The specialization of the components of this architecture leads to a flexible and extensible system.

### B. Research Issues for Mediators

To create a specific mediator, a DBA must define a mediator schema for the mediator; a collection of data sources; local schema for the data sources; and a mapping between the mediator schema and the local schemas. Generally this mapping is done via database *views*. A view is a named query. The domain of a view is the data source schemas and the range of a view is the mediator schema. (See Section II for competing approaches for mapping from the mediator to local schemas.) The DBA, via view definitions, must resolve conflicts among the different data models, different schemas, and different semantics of data sources to construct an uniform semantics for the mediator schema. Thus, the view definitions and various schema are tightly integrated. This tight integration creates a problem for the DBA, when a new data source is to be added. For each new local schema, view definitions must be changed, and perhaps the mediator schema itself must be modified. We call this the *fragile mediator* problem.

For query execution, an application sends a query to a mediator and waits for an answer. The mediator accepts the query and transforms it into sub-queries and a composition query. Each sub-query corresponds to a data source and it uses only the functionality of the data source. The composition query performs all the remaining work required to compute the answer to the query. The mediator then coordinates the execution of the query. It sends each sub-query to the wrapper that controls a data source. The wrapper translates the sub-query into a corresponding query in the local query language, and then sends this translated query to the data source. The data source computes an answer and transmits the answer to the wrapper, which translates the answer to the format required by the mediator. The mediator coordinates the arrival of the answers with the composition query. The composition query combines the answers into the final answer. To support query execution, the mediator must cope with the different functionality of each wrapper (corresponding to each data source). For instance, one data source might support only projection operations and another data source might support only selection operations. This problem is called the *source capability* problem.

After the mediators have been defined by the DBAs, the appropriate wrappers written by DBIs, and the applications programs have been developed, the system is deployed to end users. End users interact with applications that access the underlying data sources through the mediators. If an application issues a query to an underlying data source that is unavailable, in the absence of replication, generally the mediator fails to process the query, and the application informs the user that nothing can be done until the data

source is again available. We call this problem the *graceless failure* for unavailable data sources problem.

### C. The DISCO Architecture

As heterogeneous distributed database systems scale up to incorporate many data sources into the system, the three problems defined previously are aggravated. For DBAs, scaling up data sources makes a heterogeneous system hard to maintain due to the fragile mediator problem. To add a data source to the system, the DBA must change schemas and add new definitions. For DBIs, scaling up makes a system hard to program and tune due to the source capability problem. To add a data source, implementors must write new code and add new cost information. For end users and application programmers, scaling up makes a system harder to use due to the graceless failure for unavailable data sources problem.

These three problems are mitigated in some respects by the specialized components of the mediator architecture that was presented. The architecture permits DBAs to develop mediators independently and thus limits the impact of the addition of a data source on mediators. DBIs can develop wrappers independently and thus the introduction of a new data source impacts only some mediators that handle these data sources. And since applications issue queries to mediators, graceless failures only occur in mediators that access these unavailable data sources.

The design of the Distributed Information Search COmponent (DISCO) provides novel features, beyond those discussed, for all users to deal with the problems of scaling up the number of data sources. DISCO aids the DBA in adding data sources if the new data sources are similar (in the sense of similar types) to existing data sources. DISCO also supports simple type transformations when there is a mismatch between types. Solutions to the *fragile mediator problem* when there are schematic conflicts among the data sources, or semantic conflicts in the contents of the data sources, is a more complex issue. To cope with the source capability, DISCO aids the DBI by providing a wrapper language and a wrapper interface to ease the construction of wrappers. The DBI specifies the functionality of the wrapper for each data source, and the mediator guarantees that any sub-query issued to this data source will be specified based on this functionality. Thus, the wrapper need only implement the functionality of the data source. For the application programmer and end user, DISCO provides a new semantics for query processing to provide graceful instead of *ungraceful failures* for unavailable data sources. If a data source is unavailable during query evaluation, a partially evaluated query is returned. This partially evaluated query may be used, in some circumstances, by the application, to provide a partial answer to the user query.

We use a (simple) example to briefly describe each novel feature of DISCO, namely, the data model, the wrapper interface description of the data sources, and the semantics of query processing with unavailable sources.

#### D. Mediator data model

Suppose a mediator defines the type `Person` in the mediator schema as the union of person types in the underlying data sources. In many current heterogeneous distributed databases, this definition is accomplished by introducing a view for each underlying data source. When a new data source is added, a new view must be added. In DISCO, this definition is accomplished by defining an *extent* `person`, for the type `Person`, for all data sources that have persons. An *extent* represents a collection of objects (cf. Section III).

Consider two data sources handled by wrappers `w0` and `w1`. Data source `w0` contains a person relation<sup>1</sup> with a person Mary whose salary is 200, and `w1` contains a person relation with a person Sam whose salary is 50. The DBA models `w0` and `w1` as extents `person0` and `person1`, of type `Person`. The DBA then defines the extent `person` as the union of all extents associated with type `Person`, in this case `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`, i.e., the union of `person0` and `person1`, 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 modifying the extent `person` to include this new source, 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, nor does the definition of the `Person` type. In addition, DISCO provides support for incorporating new data sources with similar structure with respect to existing sources. This property simplifies the maintenance of the mediator and thus addresses, in part, the fragile mediator problem.

#### E. Mediator interface to wrappers

Each heterogeneous database system specifies the interface between the mediators and wrappers using a different level of functionality. Some systems define the interface to include complex query processing capabilities. Other systems define the interface to be a low level object interface and thus wrappers support only simple access to objects. DISCO provides a flexible wrapper interface. Mediators interface to wrappers at the level of an abstract algebraic machine of logical operators. All mediators and wrappers share the same abstract algebraic machine. When the DBI implements a new wrapper, she chooses a *subset* of the algebra to support [12]. The DBI implements the subset, and also implements a `register` call in the wrapper. This call

<sup>1</sup>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 information retrieval systems, file systems, HTTP sources, etc.

returns, among other things, a specification of the capabilities of the wrapper, i.e., the subset of the algebra supported by the wrapper.

The mediator interacts with the wrapper in two phases. In the `register` phase, the wrapper communicates to the mediator its local schema, its specification, and a (optional) description of the cost of operations in its algebra [13]. During query processing, a mediator transforms the query on the mediator schema into sub-queries on the local schema. Each sub-query is translated into a logical expression for the wrapper. The mediator uses the grammar returned by the wrapper interface to check that the logical expression only uses the subset implemented by the wrapper. If the check succeeds, the logical expression is used. If the check fails, the logical expression submitted to the wrapper is simplified (by moving processing to the mediator) until a simplified logical expression is found that can be executed by the wrapper.

Suppose that a mediator generates a logical expression to project the `name` attribute from `person0`.

```
project([name], scan(person0))
```

The mediator will pass this logical expression to a wrapper, thereby, pushing the scan and projection operations onto the wrapper, only if the wrapper interface supports the `project` and `scan` logical operators, and only if the wrapper supports composition of these logical operators. If the wrapper does not support `project`, the projection will be performed in the mediator and only the `scan` logical expression will be passed to the wrapper.

#### F. Mediator query processing

DISCO supports a new query processing semantics to deal with the graceless failure for unavailable data sources problem. 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 during query processing [14].

Consider again the query in Section I-D. Suppose that the `w0` data source does not respond. Then, the *answer* to the query would be the following *query* and not, as in current heterogeneous database systems, simply an error indicating failure. The query returned as an answer represents 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 `w0` becomes available, this partial answer could be *resubmitted* 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.

This paper is organized as follows. Section II describes related work. Section III presents the data model through a description of the extensions to an existing standard. Section IV describes mediator query processing and the wrapper interface. Section V presents a new semantics of query processing with unavailable sources. We conclude with a summary, a description of the internal architecture of the current mediator prototype, and a discussion of future plans. Earlier results of this research have been published in [15], [16].

## II. RELATED WORK

We first present a comparison of the features of the DISCO data model with other approaches developed for dealing with heterogeneous databases. Next, we make a comparison of query processing, optimization, and plan generation, in heterogeneous environments, where wrappers may have limited capability, and we examine research on cost-based optimization. Finally, we review partial evaluation from the programming language literature.

### A. Data Model

Pegasus [17], UniSQL/M [18], [19], SIMS [20], IRO-DB [3], and other projects, support mediator capabilities through a unified global schema [21], which integrates each remote database and resolves conflicts among these remote databases. Although these projects made substantial contributions in resolving conflicts among different schemas and data models, the global schema approach suffers from the fragile mediator problem; the unified global schema must be substantially modified as new sources are integrated.

For example, UniSQL/M [18], [19] is a commercial multi-database 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 composition conflicts.

Alternately, the capability of a mediator to resolve conflicts is supported by the use of higher-order query languages or meta-models [22], [23], [24], [25], [26], [27]. 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 me-

diation and inter-operation among relational and object databases [28], [29], [30], [31], [32]. Here, too, the emphasis is on resolving conflicts among schema and data models, to support interoperability of the queries. All of these approaches face the fragile mediator problem.

In DISCO, we provide underlying support for various solutions to the fragile mediator problem with a two-pronged approach. First, we adopt the standard ODMG object data model [33] to represent both mediator and data source schemas. We extend this model by introducing mediator and data source types. The *extent* of the mediator type, corresponding to objects of that type, is automatically defined over the extents of the data sources. We support incorporating new data sources, with no type mismatch, or simple type mismatch, by modifying the extent of the mediator type, and by mapping types to resolve simple type conflicts. In related research [17], [3], [24], [19], [18], [25], the mismatch of the data types, formats, values, etc., with respect to data source types and mediator types was resolved by obtaining a single unified type. In DISCO, our first objective is to support scale up by easing the introduction of new data sources. Each addition of a type and resolution of a type conflict, between data source and mediator, is independent of any other type conflict, and is handled by the data model.

The second aspect of dealing with the fragile mediator problem demands dealing with *semantic* conflicts of values (contents) in difference sources. Users may need to define *reconciliation functions* [17] which determine how data values from different sources are combined; they may need to deal with missing data [34]; and they may need to utilize techniques to resolve object identity across multiple sources [3]. In DISCO, we do not explicitly support such functionality in the data model. However, we do recognize that there are many advantages, in particular, from a performance viewpoint, if such features are directly incorporated into the data model, by a set of built-in functions, as in [17], [3]. If these functions are included in the data model, then, during query processing, the mediator may use special heuristics to optimize the processing of these functions.

The focus of research in the TSIMMIS project [4], [35] 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. 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. They address the issue of mismatch in the querying capability of different data sources, and propose techniques for query reformulation that resolve this mismatch. These techniques are not incorporated into the data model.

The Garlic system [36], [37], [38], research described in [39], [40], [30] and the Information Manifold project [41],

[42], [43], all assume a mediator environment based on a common data model. In the Information Manifold project [41], [42], [43] and in [30], the model is an extension to Datalog, and in [39], [40], the model is an extension to ODMG. Information Manifold uses a technique based on view definitions to map from the local schemas to the mediator schema. At the same time, views also express the limited capability of the remote sources, i.e., only the views may be computed on the remote sources. For query execution, mediator queries are rewritten using views. This introduces the well-known *containment* problem and various heuristics are proposed for efficient algorithms [40], [42], [30].

### B. Source Capabilities

There are several approaches to handling the source capability problem. One approach is based on standardization – all underlying data sources are required either to have the same functionality or conform to the same communication standard. Another approach uses abstract data types. The fourth approach uses view definitions. DISCO permits varying levels of functionality between data sources and the mediator. Each wrapper implements a subset of the functionality of logical algebraic machine.

In the standardization case (Pegasus [17], UniSQL/M [19], [18], SIMS [20], IRO-DB [3]) and research reported in [31], [32], some syntactic translation is required if all wrappers support the same functionality (with different syntax). However, since the same functionality is available in all data sources, the wrappers are not very complicated and the required syntactic translation is straightforward. The standardization approach considerably simplifies the construction of mediators because, in general, every sub-query can be executed at a data source. However, this solution excludes data sources with limited capability.

The Open Database Connectivity (ODBC) 3.0 standard [44] from Microsoft uses a variation on the standardization approach by using various conformance levels (similar to the conformance levels of SQL standards). The lowest level conformance standard defines the variation of SQL which must be accepted by the wrapper. Higher conformance levels add additional features. Thus, at a minimum, the wrapper and the data source must support a variation of SQL. In DISCO, a wrapper can support as little functionality as a simple file scan.

In the abstract data type approach, e.g., [1], [2], the functionality of underlying sources is encoded as a black-box function. This approach also simplifies the construction of mediators because, in general, the operations that can be pushed to a data source are fixed by the abstract data type interface.

In [40], [41], [45], [30] the source capability problem is addressed in part by using views. The Information Manifold extends this system by restricting queries on views to contain constants for some attributes, etc.

In Garlic [37], a dynamic source capability solution is implemented. The query processor decomposes the query into various sub-queries and passes the sub-queries to the

wrapper. The wrapper returns plans which implement part or all of the sub-queries. Properties (e.g., output columns, predicates, etc.) are used to describe which parts of the sub-query are executed by the wrapper. Garlic then adds operators in the mediator to complete the plan. The advantage of this approach is that the wrapper implement can write a program to decide on exactly what parts of the sub-query are executed by the wrapper. Thus, the wrapper can check for unusual conditions of the sub-query that are required by the underlying data source. The disadvantage of this approach is performance since the wrapper is called multiple times during query optimization. Garlic solves this problem by placing the wrapper code with the mediator.

In the approach that is followed in DISCO, the functionality of each wrapper is defined, either in the mediator or in the wrapper, as a subset of an algebraic machine. Query processing in the mediator is complicated; various transformations of the mediator query may result in sub-queries that cannot be evaluated in each data source, given the possibly limited functionality of the underlying data source. DISCO offers a solutions to the source capability problem. DISCO can specify the composition of operators in the algebraic machine, and can attempt to generate a query whose sub-queries can be evaluated in the data sources. For instance, in DISCO, one can specify that `select` can be composed with `scan`, but that it cannot be composed with `project`. Reference [12] describes this feature in detail.

### C. Mediator-Wrapper Cost Communication

There have been few projects on cost-based query optimization for heterogeneous systems [1], [46], [47]. In [46] and [47], the mediator calibrates its cost model by sending various “experiment” sub-queries to data sources and measuring the response time and size of the answer. In DISCO, wrappers (optionally) export cost equations to override the default equations in the mediator. Reference [13] describes this feature in detail.

The HERMES system [1] integrates heterogeneous sources by modeling each source as a collection of domains, defining some abstract data types. Thus, a source can be viewed as a parameterized call, which returns a set of answers. The bindings permitted to the call may be specified as magic-set style adornments. For cost computations, HERMES tracks statistics on the costs of previous calls, the pattern of arguments to the call, and the time and date of the call. To compute the cost of a new call, a cost estimate is generated from the statistics. Finally, the system caches previous calls and includes a system for specifying additional semantics of sources to perform a type of semantic query optimization. DISCO permits the wrapper implementor to define the cost of operations directly so that the mediator can obtain the cost of the particular sub-query evaluated at the wrapper. This solution is complementary to the above solutions.

There are several simple solutions to the problem of handling missing data sources. First, a system can simply generate failure during querying processing for unavailable data sources. A second solution assigns a meaning to the unavailable *data* at the data source. For instance, the unavailable data source can be considered to have no tuples or have null values. Another solution changes the meaning to the *query* in the presence of unavailable data. For instance, suppose a query requests the union of answers from three data sources,  $a$ ,  $b$ , and  $c$ , but data source  $c$  is unavailable. Then the system implicitly changes the meaning of the query to be the union of answers from the available data sources, and returns the union of  $a$  and  $b$  as the answer to the query.

In APPROXIMATE [48] unavailable data sources are tackled by sandwiching the actual answer between subsets and supersets of the answer. The sandwiching sets are computed using semantic information from the data sources and a modification to the relational algebra.

DISCO chooses a more flexible alternative: *The answer to a query is another query*. The answer is a partial evaluation of the original query. The partial evaluation represents the work done with the available data sources and the work remaining to be done with the unavailable data sources. Reference [14] describes this feature in detail.

*Partial Evaluation* has been studied in programming language research; it is defined as “a source-to-source program transformation technique for specializing programs with respect to parts of their input” [49]. As described in Section V, we use a technique inspired by partial evaluation of programs to handle queries evaluated against data sources that are unavailable at run-time. In this subsection, we compare our technique to a taxonomy of techniques in partial evaluation in programming languages.

In the programming language literature, typically partial evaluators improve the performance of programs by partial evaluation of a program given some input. This performance improvement is accomplished by generating, for a program  $p$ , a partially evaluated program  $p_s$ . Program  $p_s$  is based on some prespecified (static) input  $s$ , and some unknown (dynamic) input  $d$ . The answer to  $p_s$  given  $d$  is the same as the answer to  $p$  given  $s$  and  $d$ . Applying this analysis to a query evaluated against data sources, the static input would include the constants that appear in the query, and the extents corresponding to data source types, for the available data sources. The dynamic input would be the extents corresponding to the data source types, for the unavailable data sources.

Partial evaluators are classified as *on-line*, monolithic or *off-line*, staged. An online partial evaluator is a non-standard interpreter; it determines the treatment of each expression on-the-fly and can be very expensive. Staging involves a compiler and a run-time system. It first divides the inputs of the program into prespecified and unknown parts, and then determines, for each expression, if it should be evaluated at compile-time or run-time. This is accompanied by a specialization that constructs a specialized pro-

gram, with respect to the specified and unspecified parts.

Our technique roughly corresponds to a sequence of multiple on-line partial evaluations. The availability of data sources is determined on-the-fly, according to observed behavior of the data sources. We plan to investigate off-line evaluation also, where we first determine the availability of the data sources before evaluation starts. Another major difference between our work and existing work is the interpretation of  $p_s$ . In partial evaluation,  $p_s$  is opaque. In our work, we are interested in the contents of  $p_s$  since it contains partially evaluated answers. Thus, we plan to apply functions to  $p_s$  to extract information.

### III. MEDIATOR DATA MODEL

In this section we describe the DISCO data model as an extension to the ODMG 2.0 data model specification [33]. The ODMG standard consists of an object model, object definition language (ODL), a query language (OQL), and a language binding. Note however that the prototype implementation, described in more detail in Section VI-A, implements a relational subset of this model.

The ODMG object model is based on a type system. Types can be atomic or structured. The atomic types are predefined, such as integer, boolean, string, and the particular set of object types of the application. The structured types are the set, bag, list and tuple. Type expressions are constructed through the recursive application of structured type constructors to atomic types and type expressions. Object types are described in the data model through an object interface using ODL. An object interface specifies the properties (attributes and relationships) and operations or methods that are characteristic of the instances of this object type. A relationship is a reference-valued attribute of an object type. An object interface allows the declaration of a key constraint, and the declaration of inverse links between object types. The object types are organized along a subtype hierarchy. All the attributes, relationships and methods defined on a super-type are inherited by the subtype. An object type *extent* is a set of instances of a given object type (and its subtypes); it can be explicitly named in the object type interface, in which case they are automatically maintained.

The set of operators includes built-in operators, user-defined functions and user-defined methods. The built-in operators are comparison and arithmetic operators; aggregation operators; set, list and set membership operators; type conversion operators. Special built-in operators are value constructors, (e.g., set, bag, list and tuple constructors), attribute selection, quantifiers and select.

An object database is accessed through the set of persistent *named variables*. Particular named variables are associated with the *extent* of an object type that is automatically maintained. A *database interface* consists of a set of object type interfaces, and a set of named variables (with their types).

OQL *queries* corresponding to an interface are well-typed expressions constructed in this interface. Given an interface, OQL *expressions* are syntactically constructed by a

recursive application of user-defined and built-in functions, starting with constants and variables. Each OQL expression has an associated type.

The OQL *select-expression* is a built-in n-ary operator of particular import. The expressions corresponding to each input collection, the predicate, and the projection of a select-from-where expression, may all be general OQL expressions. As a consequence, OQL allows navigation (following object identifiers), nested selects, dependent joins, quantified predicates and user-defined functions or methods to appear in all clauses of the select operator. A *variable* is defined in the from clause of a select-expression or in a quantified expression (for all or exists). The collection-valued expression associated with a variable is its domain. The value of a variable is restricted to range over its domain.

The language binding consists of a mapping of ODL and OQL into the bound language. In addition, the binding specifies how updates to objects are accomplished.

DISCO extends ODMG ODL in two ways, to simplify the addition of data sources to a mediator:

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

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

#### A. Access to Data Sources

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 describe how a database administrator (DBA) defines access to a data source in DISCO.

In the first step, the DBA locates a wrapper (written by a database implementor), for the data source. Wrappers are located via Java remote method invocation (RMI) name servers; thus, their addresses are URLs. For instance, the following command defines a wrapper object `w0` associated with a specific URL:

```
wrapper w0 rmi://rodin.inria.fr/AnnuaireWrapper;
```

Section IV discusses the features of the wrapper interface. When the wrapper is registered, the schema (types) of the source are exported to the mediator. For example, the `Person` type corresponding to the objects in data sources `w0` and `w1` from Section I, is defined as follows (this extent and type will be exported):

```
extent p0 of Person;
interface Person {
    attribute String name;
    attribute Short salary; }
```

Next, the DBA defines the mediator extent corresponding to the wrapper extent as follows:

```
extent person0 of Person wrapper w0 extent p0;
```

This statement adds the extent `person0` to the `Person` interface. This statement also creates an object in the `MetaExtent` type, to be described later. Thus, *each DISCO extent represents a collection of data in one data source, of some particular type.* (A more general approach associates an implementation for a mediator type with each data source [36], [50])

The wrapper extent name `p0` is used by the wrapper to access data. For a relational database, it will be the relation name. The mediator checks that the mediator type `Person` is indeed exported by the data source. 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. The DISCO data model can also handle the case where there is a mismatch of types, as discussed in Section III-B.

At this point, data access from the data source is possible. The following query returns the answer Bag("Mary").

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

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

```
extent person1 of Person wrapper w1 extent w1;
```

adds the `person1` extent to the `Person` interface, utilizing a different wrapper `w1`. The same type relationship holds with `w1` so its objects are of type `Person`.

To access objects in both data sources, the extents are listed explicitly in the following select expression. The following query returns the answer: Bag("Mary", "Sam").

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

Explicitly referencing extents is a powerful capability, but it is difficult to express more general queries that refer to collections of extents. The DISCO data model solves this problem 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 is translated automatically into instances of this meta-data type, which is defined as follows:

```
extent metaextent of MetaExtent;
interface MetaExtent {
    attribute String name;
    attribute Type interface;
    attribute Wrapper wrapper;
    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 previous extent automatically created an instance, say `m1`, where

```
m1.name="person1"
m1.interface=Person
m1.wrapper=w0
m1.map=null
```

The `map` attribute of type `MetaExtent` is described in Section III-B.

Using this meta-data, DISCO provides a way to implicitly reference all the extents associated with a mediator type, by declaring an extent in the interface definition. Thus, the following interface definition for `Person` defines `person` as an extent that is the union of all extents of type `Person`:

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

The extent `person` is defined as the union of all extents which satisfy the following query on `MetaExtent`.

```
select x.e
from x in metaextent
where x.interface=Person
```

This definition for `person` essentially accesses the meta-data of the extents and dynamically selects all of the extents associated with the type `Person`. The extent `person` can be used anywhere a normal extent can be used.

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 Section I-D will produce the answers described.

### B. Mapping DISCO types to data source types

In the previous section, the type of the data source was identical to the type defined in the mediator for accessing the data source. Thus the name of the relation in the data source was the same as the extent name. Further, the names of the attributes of the relation in the data source were identical to the names of the attributes 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 a mapping between the two types to help the DBA to resolve such conflicts.

Suppose the DBA defines a different mediator type, `Prime`, with extent `prime0`, to access a wrapper `w2` with `Person` objects, as follows:

```
extent prime0 of Prime wrapper w2 extent p2;
interface Prime {
  attribute String n;
  attribute Short s; }
```

This statement has a type conflict because `Prime` and `Person` are different types.

The DBA resolves this kind of type conflict by specifying a *map* between the attributes of the mediator type (`Prime`) and the attributes of the data source type (`Person`). A map

is a function from the attributes of a type to the attributes of another type. The map consists of a list of pairs and it is described in the `map` attribute of the extent declaration. The DBA resolves the type conflict in this example with the following map:

```
extent prime0 of Prime wrapper w2 extent p2
  map (n=name) (s=salary);
```

Since `prime0` is of type `Prime`, the map creates a one-to-one correspondence between the `n` field and `name` and `s` and `salary`, respectively. Thus, when a sub-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. Maps are restricted to a flat structure, and they are defined as a list of strings. However, they can easily be extended to handle nested types. A further extension provides functions which map between domains and ranges, allowing the mediator to resolve mismatch of values in the data sources during query processing.

### C. 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. The ODMG data model provides sub-typing for modeling similar substructures 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.

#### C.1 Sub-typing

Sub-typing organizes collections of data sources with similar substructures. Consider two data sources of students, accessed with wrappers `w4` and `w5`. The DBA defines a `Student` interface as a subtype of `Person`, and the following extents:

```
interface Student:Person (extent students) { }
extent student0 of Student wrapper w4 extent p4;
extent student1 of Student wrapper w5 extent p5;
```

Thus, queries on `students` can reference data in `w4` and `w5`. 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 sub-type hierarchy. To accommodate this, the ODMG syntax must be extended with 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 contains four extents.

#### C.2 Views and Reconciliation

Maps and sub-typing are a very restricted form of transformation for resolving mismatch between types. In general, arbitrary transformations in the representation of a data source may be needed not only to resolve mismatched types, but *reconcile semantic differences* between data sources. For instance, one data source defines salary



as an integer and another data source defines salary as the sum of regular pay and overtime pay.

In this section we show that the ODMG data model reconciles some semantic differences through arbitrary transformations implemented with views. A view is a named query. A view can reference other views, as long as the references are not cyclic. Views are not updatable in the sense that updates can not be applied to the results of view (i.e. view updates). However, updates to data sources will be automatically reflected in the view.

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.name = y.name
```

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. 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 `w0` 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.name = z.name))
  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 wrappers `w4` and `w5` will automatically be summed in the `multiple` view definition.

In general, arbitrary transformations in the representation of a data source may be needed, and this functionality is also provided by views. Suppose a data source with wrapper `w6` of type `Two`, 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 `w6` of type `Two`. The opposite assumption is also supported in the data model, but the view definition would be more complicated.

```
interface Two {
  attribute String name;
  attribute Short regular;
  attribute Short consult; }
extent two0 of Two wrapper w6 extent p6;

define new as
  union(
    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 two0)
```

#### IV. MEDIATOR QUERY PROCESSING

The mediator includes a query processor containing a query preprocessor, a query optimizer and a run-time system. It also contains a catalog that records information on local data, local schema, global schema, etc. This catalog is updated when a source is registered with the mediator by importing the local schema, capabilities and cost information of the source. The import is accomplished via the wrapper associated with the source.

The processing of a query is accomplished in several steps.<sup>2</sup>

Step 1 performs the reformulation of the query into local schemas. The query is parsed and type checked against the global schema and then it is reformulated into the local schemas of the sources. Reformulation is accomplished through view definitions and the application of maps of Section III. In particular, each extent in the reformulated query references either an extent in a particular source or an extent local to the mediator.

Step 2 performs logical search space generation. The reformulated query is transformed into a *preliminary* tree of logical operators in the relational algebra that is equivalent to the query. The logical operators in the tree belong to a universal abstract machine (UAM) of logical operators implemented in the mediator. DISCO has the usual logical operators of `scan`, `project`, `join`, etc. In addition, transformation rules rewrite trees to equivalent trees. DISCO has the usual transformation rules, e.g. commuting and associating join operations. Thus, multiple preliminary logical operator trees are generated for each query. Finally, an additional logical operator, `submit`, expresses the mediator call to a wrapper.

<sup>2</sup>We describe a top-down transformation of the query into an execution plan. However, the prototype has an implementation of both this algorithm and a dynamic-programming bottom-up style query processing algorithm.

Step 3 performs preliminary query decomposition. Each preliminary logical operator tree is then *decomposed* into a forest of  $n$  *wrapper subtrees* and a *composition* tree that combines the results of the wrapper subtrees into the final answer. Each wrapper subtree is the largest subtree of the preliminary tree of logical operators whose leaves reference (via the extent) the same wrapper. The composition tree is the remaining subtree not in any wrapper subtree. This tree links together the forest of wrapper subtrees into a single tree. The leaves of the composition tree are either one of the wrapper subtrees or a `scan` operation on data stored in the mediator.

Step 4 compares the functionality required for each wrapper subtree with the capabilities exported by the corresponding wrapper and modifies the wrapper subtree to use only the capabilities of the wrapper. This modification is accomplished by moving operations that the wrapper cannot perform from the wrapper subtree to the composition tree. The final subtrees are marked in the composition tree via the `submit` logical operator. These modifications transform the preliminary logical operator tree into the final logical operator tree. This step is described in more detail in Section IV-A.

Step 5 transforms the final logical operator tree into an execution plan by transforming the logical operations in the mediator composition tree to physical algorithms in the mediator run-time system. The logical operations done by wrappers are not transformed because the corresponding physical algorithms are executed by the wrapper itself. The `submit` logical operators in the composition tree are translated into `exec` physical algorithms. This physical algorithm is responsible for calling the associated wrapper. The composition tree can always be evaluated by the mediator since the UAM can be executed by some physical algorithm in the run-time system. The mediator run-time system is based on the iterator model [51]. DISCO has the usual physical algorithms such as nested-loop-join, file-scan, etc.

Step 6 assigns a cost to the execution plan by considering the cost (in terms of total time and statistical information) of the physical algorithms in the mediator and the costs of the logical operations on the wrappers. If a wrapper has exported cost equations, those are also considered. This step is described in more detail in Section IV-B.

Steps 2 through 6 are repeated until the execution plan with lowest cost is generated.

Step 7 executes the lowest cost plan. During query execution, the mediator calls the wrappers and passes the final wrapper subtree. The wrappers evaluate their subtree and send their results back to the mediator. The mediator run-time system combines the results using execution plan which represents the final composition query. If a wrapper is unavailable, a partial answer is produced, as described in Section V.

#### A. Defining source capability specifications

In response to a `registration` call from the mediator, a wrapper exports its source capabilities. The source capabil-

ities are sentences of a language which describe the logical operations supported by the wrapper and the attributes and predicates that can appear in a logical operation [12]. Thus, the sentences describe a subset of the expressions of the UAM.

In the simplest case, the wrapper returns a set of operators (e.g., `{scan,project}`) which means that the wrapper supports the entire definition of those operations in the UAM. Thus, projections are permitted across all attributes, etc. In addition, more detail is permitted describing which attributes can appear as arguments to an operation. For example, the following source capability specification

```
scan [person0]
select [person0 1 {bind name (=)
                bind salary (<)} ]
```

means that the `scan` operation applies only to the `person0` extent. For the `select` operation, only 1 attribute can be selected. If the `select` operation applies to the `name` attribute of the `person0` extent, it must use the equality (=) predicate. Otherwise, it must select the `salary` attribute with the less-than (<) predicate. Thus, the following expressions, for some constant `C` are permitted:

```
scan(person0)
select(scan(person0), name=C)
select(scan(person0), salary<C)
```

The use of these expressions in query processing is straight-forward. Each wrapper subtree is searched bottom-up, comparing each logical operator with the specification. When a logical operator  $o$  is found which does not match the specification, the subtree is truncated to the sub-tree below  $o$ . This subtree can be executed by the wrapper. The subtree from the root to  $o$  is moved to the preliminary composition tree.

The mediator provides additional functionality to the wrapper implementor by accepting a context free grammar from the wrapper. If the context free grammar can parse the string representation, then the entire subtree can be executed by the wrapper. Otherwise the expression is rejected. This functionality permits the wrapper implementor to check the expression for properties that are not expressible in the specification language. Complete details are given in [12].

The principal advantages of the capability specification is its simplicity for the wrapper implementor and the efficient use of the capabilities during query processing. The algorithm for capabilities based reasoning is  $O(n)$  where  $n$  is the number of nodes the preliminary tree. Using algorithms of a higher complexity, e.g. reference [52], may produce place which push more operations onto a wrapper. In this latter work, instead of computing source capabilities over trees, conjunctive queries are used instead. Unfortunately the complexity of this latter work is in the worst case exponential in the size of the capability expression. An open research issue is the definition of a capability system which balances the quality of the plans produced with the complexity of the capability algorithm.

## B. Defining cost information

In response to a `registration` call from the mediator, a wrapper optionally exports some cost information. Cost information is exported in the form of logical rules. This logical rules encode classical cost model equations. If the wrapper does not export some particular cost equation, a default equation in the mediator is used. Cost equations describe the total time required for an operation and statistical information of the total number of tuples produced by an operation, the total number of bytes, and the number of distinct values in each attribute.

For example, the following rules, expressed as facts, describe the cost information for a `scan` operation

```
totaltime(scan(person0), 10000)
countobject(scan(person0), 100)
countbyte(scan(person0), 10000)
cardinality(scan(person0), [100, 90])
```

The first rule gives the total time to scan the `person0` extent as 10,000 milliseconds. (This time does *not* include the overhead of the network connection between the mediator and wrapper, since the wrapper may connect to multiple mediators.) The extent has 100 objects and is represented with 10,000 bytes. There are 100 distinct values of the `name` attribute and 90 distinct values of the salary attribute. In a more interesting example is the following rules

```
countobject(select(X,P), 1) :-
  contains(P,name = C)
totaltime(select(scan(X),P), T) :-
  totaltime(scan(X), T)
```

These rules declare that the `select` logical operator returns a single object for equality comparisons on the `name` attribute and that the total time to perform a `scan` operation followed by a `select` operation is the same as the time needed to perform the `scan` operation. Note that the wrapper does *not* export the total time needed for the `scan` operation and the default rule (and implicitly, the corresponding equation) in the mediator is used. Thus, the language gives very fine control over exactly which rules use the default and which rules use a more specific wrapper information. In addition, default rules for a whole class of wrappers is possible. Complete details are given in [13]. These rules are used during query processing to determine the cost of execution of a subtree on a wrapper. In fact, the same mechanism is used to determine the cost of physical algorithms used at run-time in the mediator.

Note that in the case the wrapper always wants the maximum acceptable subtree to be pushed to the wrapper, a trick can be used: simply return rules that represent cost equations of zero cost. That is, the cost equations do not represent the actual cost of execution. The mediator will simply choose the plan which has the largest number of operations performed at the mediator because every operation must be executed either at the data source or in the mediator. The zero cost of the operation at the data source will always be cheaper than the cost of the same operation in the mediator.

## C. Example

In steps 1 through 4 the query optimizer translates a query into a final logical operator tree. For example, the query optimizer translates the following query:

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

where `person` has extents `person0` and `person1`, associated with wrappers `w0` and `w1`, respectively, into the following query plan:

```
union(project([name],
              submit(w0, scan(person0))),
       project([name],
              submit(w1, scan(person1))))
```

Reading this query plan, in the order of application, from right to left, the query retrieves tuples with the `scan` operation from the `person0` collection. The location of the tuples is specified in the `w0` wrapper. The `submit` operator accesses the tuples in the data source, and the `name` attribute is projected out of each tuple in the collection. This projection is done by the run-time system of the mediator. A similar operation is done with `w1` and the results are combined into a bag, by the `union` operator in the mediator.

Alternately, the query optimizer may generate the following final logical operator tree,

```
union(submit(w0, project([name],
                        scan(person0))),
       project([name],
              submit(w1, scan(person1))))
```

In this latter case, the projection of attribute `name` from instances of the collection `person0` is performed in the wrapper (or in the data source), whereas the sub-query to the `w1` wrapper is unchanged from the previous case. The `union` operator is executed in the mediator, as before. The `w0` wrapper must export the capability to perform `project` on the `name` attribute. Note that in this latter case, *both* of the above plans are considered, since the execution of the `projection` operation may be less expensive in the mediator than in the wrapper, depending on the cost equation used to determine the cost of the `project` operation in the wrapper and the cost equations for the corresponding physical algorithm in the mediator.

To re-iterate, determining which plan to use is based on two factors. First, the arguments to all `submit` calls must pass the corresponding capability description. Second, the plan must have the lowest cost. To determine the cost of an entire plan, the mediator determines the cost of the part of the plan executed in the mediator in the usual way, by using the cost of the operators performed and the costs of the arguments to the operations performed. For instance, the cost of the `union` logical operator, implemented by, say, a `dropduplicate` physical operator, depends in part on the size of the arguments, i.e., the size of the results of the `project` operations. The result sizes of the `project`

operation, when performed on the wrapper, take into consideration any relevant equations exported by the wrapper. The plan with the lowest cost is sent to the run-time system for execution.

#### D. Query Execution

The execution plan consists of a tree of nodes. Each node is a physical algorithm that implements the logical operators. There is not a direct correspondence between physical algorithms and logical operators. All physical algorithms support the Graefe iterator model [51] consisting of three methods, `open`, `get-next` and `close`. The `open` method prepares the node for query execution. The `get-next` method retrieves the next tuple of the answer. This method is called multiple times to retrieve the entire answer. The `close` method performs clean-up operations. Execution proceeds by calling `open` on the root node, which recursively calls `open` on its children. Then, `get-next` is repeatedly called on the root node to retrieve the answer. The root node calls its children as needed to construct each tuple of the answer. Finally, `close` is recursively called to finish query processing.

A leaf node of the execution plan is either a `exec` physical algorithm which is responsible for accessing wrappers or a physical algorithm which access data in the mediator. Each `exec` node contains a wrapper subtree that is executed by a wrapper. Upon receiving an `open` method call, the node issues the wrapper subtree to the wrapper. Available data sources return answers which are materialized in the mediator. Unavailable data sources either explicitly return a negative acknowledgment or timeout with an error. The node returns *available* to its parent as the return value for the `open` call if the entire answer was materialized in the mediator. Otherwise the node returns *unavailable*. Other physical algorithms have an `open` call that returns available if all the `open` calls to the children nodes returned available and otherwise returns unavailable. In the next section we use this extension to help manage available and unavailable data sources.

### V. QUERY PROCESSING WITH UNAVAILABLE DATA

As mentioned in the introduction, scaling the number of heterogeneous data sources aggravates 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 queries that range 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.

#### A. Partial evaluation of queries

Partial evaluation of queries is performed in two phases. The first phase executes a query and determines which data sources are available. If all the data sources are available, query processing finishes normally and produces an answer. If some data sources are unavailable, a new *query* is generated in the second phase. This query represents both the work that has been accomplished and the work that

remains to be done once the unavailable data sources are again available. This transformation is possible because each physical algorithm has a corresponding logical operator tree, and each logical operator tree has a corresponding part of a declarative query. The resulting declarative query is the partial evaluation of the query. It is also the answer to the query.

The first phase of partial answer evaluation is coded in the `open` call of each physical algorithm. Each node calls `open` on *all* its children nodes. If all children nodes return *available*, then the node returns *available* to its parent. Otherwise the node returns *unavailable*. In effect, the leaf `exec` nodes are opened and the wrappers are accessed. Then all nodes in the tree label themselves as available depending on the status of the subtree rooted at the node.

The second phase of partial answer evaluation is as follows. If the root node is available then all nodes in the plan must be available and query processing proceeds normally. If the root node is unavailable, each subtree containing only available nodes materializes its (sub) result. Then, in a recursive bottom-up fashion, each node returns a declarative description of the subtree rooted at the node. For the materialized subtrees, this description is simply the identifier of the materialized result. For other nodes, the declarative description depends on the semantics of the physical algorithm. For example, a node which performs selection will attach the selection predicate to the declarative description of the subtree rooted at the node. The description associated with the root is the answer to the query. Complete details are given in [14].

Consider the following query which retrieves persons of the same name from three sources:

```
select x.name
from x in person0, y in person1, z in person2
where x.name = y.name and y.name = z.name
```

The following is a query execution plan for this query, where `slice` is the physical algorithm for projection, `join-nl` is the nested loop join algorithm.

```
slice([name],
      join-nl(name,
              name,
              join-nl(name,
                      name,
                      exec(w0, scan(person0)),
                      exec(w1, scan(person1))),
              exec(w2, scan(person2))))
```

Suppose wrappers `w0` and `w1` answer and `w2` does not answer during query execution. The plan will now be as follows:

```
slice([name],
      join-nl(name,
              name,
              t0,
              exec(w2, scan(person2))))
```

where `t0` is the result of executing the `join-nl` operation, in the mediator, over the results returned from the

exec calls to `w0` and `w1`. A partial evaluation is returned, by mapping from the currently existing partially evaluated query execution plan to a new query. The result is as follows:

```
select w.name
from w in t0
      z in person2
where w.name = z.name
```

where the contents of `t0` have been directly materialized for the partially evaluated query. This result is returned to the user. Detailed algorithms for the construction of this result are given in [14].

### B. Research issues in partial evaluation

Currently, the result of partial evaluation are returned to the user. We have defined some functions which return *ad hoc* information about the partial evaluation, e.g., a list of all materialized results, a list of wrappers that returned answers, a list of wrappers that did not return answers, etc. This information is used by the user interface to manage the interaction between the user and the partial answer. However, in general the problem of extracting information from a partial answer is an open research issue. For the moment the principal functionality provided is the re-submission of the partially evaluated query to the mediator for completion of evaluation. The completion of evaluation is possible if the previously unavailable sources are operating at the time that the query is re-submitted. A query may be partially evaluated multiple times, thus requiring the user to re-submit partially evaluated results multiple times, before a final answer is obtained.

A second problem with partial evaluation involves the underlying semantics of executing queries within transaction boundaries. Suppose two data sources `w0` and `w1` time stamp each data value when it is added to the data source. Suppose data source `w0` is available, and a user evaluates a query over both sources. The (partially evaluated) answer will contain data from `w0` but no data from `w1`. Suppose `w0` and `w1` are both updated, and the (partially evaluated) answer is re-submitted as a new query. This final answer, assuming that `w1` is available, will contain the updated tuples from `w1` but none of the updated tuples from `w0` since the partial answer will access materialized data in the mediator. The detection of updates to sources, and the semantics of partial evaluation with updates to sources, is an open research issue.

Thirdly, materializing the partial answer, `t0`, as indicated above, has an impact on the implementation of the query processor. In this case, the state of the partial evaluation, `t0`, is explicitly maintained by the mediator, to be re-used if the query is re-submitted. The mediator must maintain this state until the user re-submits the partially evaluated query. This introduces a problem of garbage collecting states of partial evaluation, corresponding to queries that are never re-submitted. This garbage collection problem can be partially solved by periodically replacing the materialized result with the sub-query used to generate it.

Access to the materialized result triggers the mediator to execute the associated sub-query. In addition, maintenance of materialized results is an optimization issue. For example, it may be cheaper to re-submit the original query, rather than maintain the materialized result and re-submit the (partially evaluated) query. For very small materialized results, it may also be cheaper to embed as constants the data of the materialized result directly in the query.

Finally, partial evaluation semantics interact with pipelined evaluation run-time system architectures. The problem stems from the need to construct all the results of an operator at any point in time. In pipelined evaluation, this corresponds to the materialization of a partially executed pipeline. For the moment, we simply materialize answers from wrappers. This restriction may impose large storage requirements on the mediator. Relaxing this restriction is an open area of research.

## VI. CONCLUSION

### A. Description of the current prototype

Figure 2 shows a diagram of the architecture of prototype 0 [53]. The prototype consists of a collection of mediator and wrappers, both running as servers. Thus mediators and wrappers can be arbitrarily paired, so wrappers can serve multiple mediators. In addition, every mediator is a wrapper, so mediators can call other mediators as if they were wrappers. The entire system, except for the data sources, is coded in about 36,000 lines of Java. Java remote method invocation is used for communication between applications, mediators, and wrappers.

The mediator has several interfaces. The administration interface is used to declare global schemas, local data elements, and register wrappers. The application interface is used to accept queries and return answers and partial answers. The wrapper interface is used to contact wrappers (at registration time) to load schema, source capability, and cost information. This interface is also used to submit sub-queries to the wrappers and obtain answers to sub-queries. Currently, the mediator accepts a subset of the ODMG standard corresponding to the relational algebra. Query processing follows the outline described in Section IV and partial evaluation follows the outline described in Section V. However, the query processing described in Section III has not yet been implemented. The functionality of the mediator includes a local storage system and a pipelined run-time system.

The wrapper has several interfaces also, for responding to registration requests, answering sub-queries, and communicating with the data sources. The functionality of the wrapper includes a pipelined run-time system. Thus, if the data source is pipelined, pipeline processing is possible from the data source to the application.

### B. Summary

Scaling the number of data sources in heterogeneous distributed databases aggravates problems for end users, application programmers, database administrators and

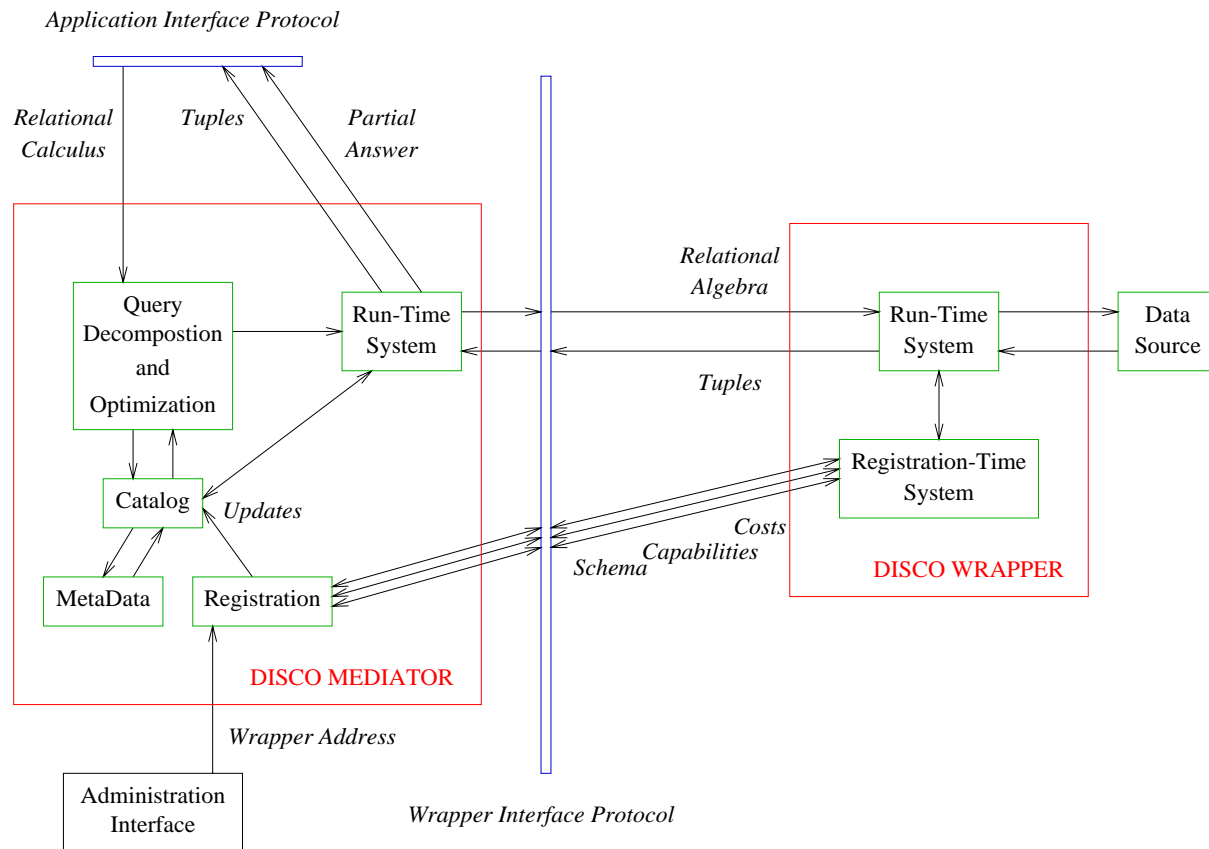


Fig. 2. The architecture of the DISCO Prototype 0.

database implementors (wrapper implementors). The design of DISCO provides novel solutions to some of the problems encountered by these users.

1. The model of data sources, the use of schema mapping, and views, aid the database administrator in modeling the system.
2. A flexible wrapper interface aids the wrapper implementor in dealing with the problem of describing the functionality of the underlying data source.
3. The query optimizer in the mediator handles the task of solving the mismatch between the expressive power of the DISCO system and the underlying data source, and generating expressions which can be evaluated in the data source.
4. A cost-based query optimizer use the cost-model for the mediator physical operators and the cost-model provided by the wrapper.
5. Partial evaluation semantics provides end users and applications programmers with the ability to deal with queries evaluated with unavailable data sources.

### C. Future work

Several aspects of DISCO present new research problems. The distributed architecture introduces several performance issues, since in the most general case network communication occurs between several components during query processing.

The `submit` logical operator works well with the autonomous nature of the data sources. The disadvantage

of this operator is that it cannot accept data from another data source, or from the mediator. This restriction implies that the full generality of algorithms for implementing distributed and parallel databases cannot be expressed. For example, a semi-join algorithm to implement a join cannot be used in the wrapper interface, since it requires the transmission of results into a data source.

For the problem of graceful failure for unavailable data sources, partial evaluation presents many open research issues, discussed in Section V-B. For example, extracting information from a partially evaluated source in the general case is difficult because the structure of the partially evaluated query may vary widely, depending on how much work was done during partial evaluation. Some information may or may not be available, depending on which data sources were unavailable. Second, resubmitting a partially evaluated result is a manual operation, left up to the user. Clearly some automatic form of re-submission is desirable. Third, partial evaluation is not clearly defined for aggregate functions.

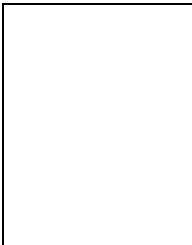
*Acknowledgments* Thanks to Daniela Florescu, Michael Franklin, Helena Galhardas, Georges Gardarin, Catherine Hamon, Alon Levy, Yannis Papakonstantinou, Peter Schwarz and the anonymous referees for discussions on various aspects of this article.

### REFERENCES

- [1] S. Adali, K. S. Candan, Y. Papakonstantinou, and V. S. Subrah-

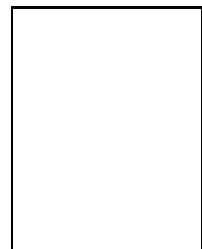
- maniam, "Query caching and optimization in distributed mediator systems," in *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1996, pp. 137-148.
- [2] Jose Blakeley, "Data access for the masses through OLE DB," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, New York, 1996, vol. 25, 2 of *ACM SIGMOD Record*, pp. 161-172, ACM Press.
- [3] G. Gardarin et al., "IRO-DB: A distributed system federating object and relational databases," in *Object-Oriented Multi-database Systems: A solution for Advanced Applications*, O.A. Bukhres and A.K. Elmagarmid, Eds. Prentice Hall, 1996.
- [4] Joachim Hammer, Hector Garcia-Molina, Kelly Ireland, Yannis Papakonstantinou, Jeffrey Ullman, and Jennifer Widom, "Information translation, mediation, and Mosaic-based browsing in the TSIMMIS system," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1995, Project Demonstration.
- [5] Richard Hull and Roger King, "Index to the reference architecture for the intelligent integration of information (R<sup>3</sup>)," US Government ARPA, May 1995, [http://is.se.gmu.edu/13\\_Arch/index.html](http://is.se.gmu.edu/13_Arch/index.html).
- [6] Won Kim, *Modern Database Systems: The Object Model, Interoperability, and Beyond*, ACM Press, New York, NY, 1995.
- [7] Ling Liu, Calton Pu, Roger Barga, and Tong Zhou, "Differential evaluation of continual queries," in *Proceedings of the 16th International Conference on Distributed Computing Systems*, Hong Kong, 1996, pp. 458-465, IEEE Computer Society Press.
- [8] Edmond Mesrobian, Richard Muntz, Eddie Shek, Silvia Nittel, Mark LaRouche, and Marc Krigger, "OASIS: an open architecture scientific information system," in *Proceedings of RIDE '96*, New Orleans, 1996, IEEE Press.
- [9] Marjorie Templeton, Herbert Henley, Edward Maros, and Darrel J. Van Buer, "InterViso: Dealing with the complexity of federated database access," *The VLDB Journal*, vol. 4, 1995.
- [10] Mary Tork Roth, Manish Arya, Laura M. Haas, Michael J. Carey, William Cody, Ron Fagin, Peter M. Schwarz, John Thomas, and Edward L. Wimmers, "The Garlic project," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, Montréal, Québec, Canada, June 1996, pp. 557-558, Project Demonstration.
- [11] G. Wiederhold, "Mediators in the architecture of future information systems," *IEEE Computer*, vol. 25, no. 3, pp. 38-49, 1992.
- [12] Olga Kapitskaia, Anthony Tomic, and Patrick Valduriez, "Dealing with discrepancies in wrapper functionality," Tech. Rep. RR-3138, INRIA, 1997.
- [13] Hubert Naacke, Georges Gardarin, and Anthony Tomic, "Leveraging mediator cost models with heterogeneous data sources," in *Proceedings of the 14th International Conference on Data Engineering (ICDE '98)*, Orlando, Florida, 1998.
- [14] Philippe Bonnet and Anthony Tomic, "Partial answers for unavailable data sources," Tech. Rep. RR-3127, INRIA, 1997.
- [15] A. Tomic, L. Raschid, and P. Valduriez, "Scaling heterogeneous databases and the design of Disco," Tech. Rep., Technical Report Number 2704, INRIA Rocquencourt, France, 1995, Extended Version.
- [16] A. Tomic, L. Raschid, and P. Valduriez, "Scaling heterogeneous databases and the design of Disco," *Proceedings of the International Conference on Distributed Computing Systems*, pp. 449-457, 1996.
- [17] Rafi Ahmed, Philippe De Smedt, Weimin Du, William Kent, Mohammad A. Ketabchi, Witold A. Litwin, Abbas Rafii, and Ming-Chien Shan, "The Pegasus Heterogeneous Multidatabase System," *Computer*, vol. 24, no. 12, pp. 19-27, Dec. 1991.
- [18] W. Kim et al., "On resolving schematic heterogeneity in multidatabase systems," *Distributed and Parallel Databases*, vol. 3, no. 1, 1993.
- [19] W. Kim and J. Seo, "Classifying schematic and data heterogeneity in multi-database systems," *IEEE Computer*, pp. 12-18, December 1991.
- [20] Y. Arens, C. Y. Chee, C.-N. Hsu, and C. A. Knoblock, "Retrieving and integrating data from multiple information sources," *International Journal of Intelligent and Cooperative Information Systems*, vol. 2, no. 2, pp. 127-158, 1993.
- [21] C. Batini, M. Lenzerini, and S. B. Navathe, "A comparative analysis of methodologies for database schema integration," *ACM Computing Surveys*, vol. 18, no. 4, pp. 323-364, December 1986.
- [22] T. Barsalou and D. Gangopadhyay, "M(dm): An open framework for interoperability of multimodel multidatabase systems," *Proceedings of the International Conference on Data Engineering*, 1992.
- [23] J. Chomicki and W. Litwin, "Declarative definition of object-oriented multidatabase mappings," in *Distributed Object Management*, M.T. Oszu, U. Dayal, and P. Valduriez, Eds. Morgan Kaufmann, 1993.
- [24] William Kent, "Solving domain mismatch and schema mismatch problems with an object-oriented database programming language," in *Proceedings of the 17th Conference on Very Large Databases*, Barcelona, Spain, Sept. 1991, Morgan Kaufmann.
- [25] R. Krishnamurthy, W. Litwin, and W. Kent, "Language features for interoperability of databases with schematic discrepancies," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, Denver, CO, May 1991.
- [26] Laks V. S. Lakshmanan, Fereidoon Sadri, and Iyer N. Subramanian, "SchemaSQL - a language for interoperability in relational multi-database systems," in *Proceedings of the 22nd VLDB Conference*, Mumbai, India, 1996, pp. 239-250.
- [27] L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian, "On the logical foundations of schema integration and evolution in heterogeneous database systems," *International Conference on Deductive and Object-Oriented Databases*, 1993.
- [28] S. Chakravarthy, W.-K. Whang, and S.B. Navathe, "A logic-based approach to query processing in federated databases," Tech. Rep., University of Florida, 1993.
- [29] A. Lefebvre, 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.
- [30] X. Qian, "Query folding," *Proceedings of the International Conference on Extended Database Technology*, 1996.
- [31] X. Qian and L. Raschid, "Translating object-oriented queries to relational queries," *Proceedings of the IEEE International Conference on Data Engineering*, 1995.
- [32] L. Raschid and Y. Chang, "Interoperable query processing from object to relational schemas based on a parameterized canonical representation," *International Journal of Intelligent and Cooperative Information Systems*, 1995.
- [33] R. G. G. Cattell, Douglas K. Barry, et al., *The Object Database Standard - ODMG 2.0*, Morgan Kaufmann, 1997.
- [34] A.L.P. Chen, J.L. Koh, T.C.T. Kuo, and C.C. Liu, "Schema integration and query processing for multiple object databases," *Integrated Computer-Aided Engineering, Special issue on Multidatabase and Interoperable Systems*, vol. 2, no. 1, 1995.
- [35] Yannis Papakonstantinou, Serge Abiteboul, and Hector Garcia-Molina, "Object fusion in mediator systems," in *Proceedings of the 22nd VLDB Conference*, Mumbai, India, 1996, pp. 413-424.
- [36] M. Carey et al., "Towards heterogeneous multimedia information systems: the Garlic approach," Tech. Rep., IBM Almaden Research, 1995.
- [37] Mary Tork Roth and Peter Schwarz, "Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources," in *Proceedings of the 23rd VLDB Conference*, Athens, Greece, 1997, pp. 266-275.
- [38] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang, "Optimizing queries across diverse data sources," in *Proceedings of the 23rd VLDB Conference*, 1997, pp. 276-285.
- [39] D. Florescu, L. Raschid, and P. Valduriez, "Using heterogeneous equivalences for query rewriting in multidatabase systems," *Proceedings of the International Conference on Cooperating Information Systems*, 1995.
- [40] D. Florescu, L. Raschid, and P. Valduriez, "Answering queries using OQL view expressions," *Workshop on Materialized Views: Techniques and Applications, in conjunction with the ACM SIGMOD International Conference*, 1996.
- [41] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille, "Querying heterogeneous information sources using source descriptions," in *Proceedings of the 22nd VLDB Conference*, Mumbai, India, 1996.
- [42] A.Y. Levy, A.O. Mendelzon, Y. Sagiv, and D. Srivastava, "Answering queries using views," *Proceedings of the ACM PODS Symposium*, 1995.
- [43] A.Y. Levy, D. Srivastava, and T. Kirk, "Data model and query evaluation in global information systems," *International Journal on Intelligent Information Systems - special issue on Networked Information Retrieval*, 1995.

- [44] *Microsoft Open Database Connectivity (ODBC) Documentation*, Microsoft Corporation, One Microsoft Way, Redmond, WA, 1997, <http://www.microsoft.com/odbc>.
- [45] Y. Papakonstantinou, A. Gupta, and L. Haas, "Capabilities-based rewriting in mediator systems," Tech. Rep., IBM Almaden Research, 1996.
- [46] W. Du, R. Krishnamurthy, and M. C. Shan, "Query optimization in a heterogeneous DBMS," in *Proceedings of the 18th Conference on Very Large Databases*, Vancouver, BC, Aug. 1992, Morgan Kaufmann.
- [47] G. Gardarin, F. Sha, and Z.-H. Tang, "Calibrating the query optimizer cost model of IRO-DB," in *Proceedings of the 22nd VLDB Conference*, Mumbai, India, 1996.
- [48] S. V. Vrbsky and J. W. S. Liu, "APPROXIMATE: A query processor that produces monotonically improving approximate answers," *Transactions on Knowledge and Data Engineering*, vol. 5, no. 6, pp. 1056-1068, Dec. 1993.
- [49] Charles Consel and Olivier Danvy, "Tutorial notes on partial evaluation," in *Proceedings of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Charleston, South Carolina, 1993.
- [50] P. Schwarz and K. Shoens, "Managing change in the Rufus system," *Proceedings of the IEEE International Conference on Data Engineering*, 1994.
- [51] Goetz Graefe, "Query evaluation techniques for large databases," *ACM Computing Surveys*, vol. 25, no. 2, 1993.
- [52] Vasilis Vassalos and Yannis Papakonstantinou, "Describing and using query capabilities of heterogeneous sources," in *Proceedings of the 23rd VDLB Conference*. Athens, Greece, 1997.
- [53] Anthony Tomic, Rémy Amouroux, Philippe Bonnet, Olga Kapitskaia, Hubert Naacke, and Louiqa Raschid, "The distributed information search component (DISCO) and the World-Wide Web," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, Tuscon, Arizona, 1997, Prototype Demonstration.

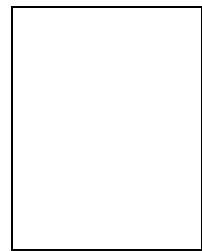


**Patrick Valduriez** received his Ph.D. degree in Computer Science from the University of Paris 6 in 1981. He is currently a Director of Research at INRIA. There he heads a group of researchers working on distributed database technology. Since 1995, he is also heading the R&D joint venture Dyade between Bull and INRIA to foster technology transfer in the areas of Internet/Intranet. From 1985 to 1989, he was a senior scientist at Microelectronics and Computer Technology Corp. (Austin, Texas).

There he participated in the design and implementation of the Bubba parallel database system. He is the author or co-author of many technical papers and several books, among which "Principles of Distributed Database Systems" published by Prentice Hall in 1991, and "Object Technology" published by Thomson Computer Press in 1997. He is a trustee of the VLDB endowment and an associate editor of several journals including ACM Transactions on Database Systems, the VLDB Journal and Distributed and Parallel Databases. He has also served as program chair of PDIS'93 and SIGMOD'97. He was the recipient of the 1993 IBM France scientific prize.



**Anthony Tomic** received his B.S. degree in 1983 from Indiana University, Bloomington, and his M.A. and Ph.D. degrees from Princeton University in 1992 and 1994, respectively. He is currently an *ingénieur expert* at the Institut National de Recherche en Informatique et en Automatique (INRIA), the national research center for computer science in France. He is responsible for the design and implementation of DISCO. His research interests include databases and information retrieval.



**Louiqa Raschid** received a Bachelor of Technology in electrical engineering from the Indian Institute of Technology, Madras, in 1980, and a Ph.D. in electrical engineering from the University of Florida, Gainesville, in 1987. Since 1987 she has been at the University of Maryland at College Park. She holds a joint appointment with the College of Business and Management, the Institute for Advanced Computer Studies and the Department of Computer Science (affiliate). She was promoted to associate professor in September 1993. She is co-director of the Laboratory for Computational Linguistics and Information Processing. Since 1994, she has been a visiting scientist at INRIA. Dr. Raschid's research interests include database accessibility over the WWW; query processing with networked information servers; semantic query optimization for object and relational databases; and rule processing in database management systems. Her research is supported by grants from the National Science Foundation and the Defense Advanced Research Projects Agency. Dr. Raschid serves on the editorial board of the INFORMS Journal of Computing. She is a member of IEEE, ACM, and the Society of Women Engineers.