Object Exchange Across Heterogeneous Information Sources*

Yannis Papakonstantinou
Hector Garcia-Molina
Jennifer Widom

Department of Computer Science
Stanford University
Stanford, CA 94305-2140

{yannis,hector,widom}@cs.stanford.edu

Abstract

We address the problem of providing integrated access to diverse and dynamic information sources. We explain how this problem differs from the traditional database integration problem and we focus on one aspect of the information integration problem, namely *information exchange*. We define an object-based information exchange model and a corresponding query language that we believe are well suited for integration of diverse information sources. We describe how the model and language have been used to integrate heterogeneous bibliographic information sources. We also describe two general-purpose libraries we have implemented for object exchange between clients and servers.

1 Introduction

A significant challenge facing the database field in recent years has been the integration of heterogeneous databases. Enterprises tend to represent their data using a variety of conflicting data models and schemas, while users want to access all data in an integrated and consistent fashion. There has been substantial progress on database integration techniques [1,9,13,19]; in addition, emerging standards such as SQL3 are aimed at eliminating many of the problems.

At the same time, however, the problem of integration has become much more challenging because users want integrated access to *information*—data stored not just in standardized SQL databases, but also in, e.g., object repositories, knowledge bases, file systems, and document retrieval systems. In addition, users want to integrate this information with "legacy" data, and even with data that is not stored but rather arrives on-line, e.g. over a news wire. As an example, consider a stock broker tracking a company, say IBM. The broker's information sources may include IBM product announcements, the stock market ticker tape, IBM profit/loss statements, news articles, structured databases containing historical information (dividends per year), personnel information (the 100 top-paid executives), general information (the Fortune 500), and so on. Queries may range

*Research sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Material Command, USAF, under Grant Number F33615-93-1-1339. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of Wright Laboratory or the US Government. This work was also supported by the Reid and Polly Anderson Faculty Scholar Fund, the Center for Integrated Systems at Stanford University, and by Equipment Grants from Digital Equipment Corporation and IBM Corporation.

from simple ones over a single source (e.g., *What were IBM sales in 1990?*), to simple ones involving multiple sources (e.g., *Get all recent news items where an IBM executive is mentioned*), to complex analyses (e.g., *Is IBM stock a good buy today?*).

Although there are many similarities, integrating a disparate set of information sources differs from the integration of conventional databases in the following ways:

- Many of the sources contain data that is unstructured or semi-structured, having no regular schema to describe the data. For example, a source may consist of free-form text; even if the text does have some structure, the "fields" (e.g., author, title, etc.) may vary in unpredictable ways.

- The environment is dynamic. The number of sources, their contents, and the meaning of their contents may change frequently. For example, the stock broker's company may add or drop an information source depending on its cost and usefulness; a source that predicts a company's earnings may periodically redefine how it computes the earnings.

- Information access and integration are intertwined. In a traditional environment, there are two phases: an integration phase where data models and schemas are combined, and an access phase where data is fetched. In our environment, it may not be clear how information is combined until samples are viewed, and the integration strategy may change if certain unexpected data is encountered.

- Integration in our environment requires more human participation. In the extreme case, integration is performed manually by the end user. For example, the stock broker may read a report saying that IBM has named a new CEO, then retrieve recent IBM stock prices from a database to deduce that stock prices will rise. In other cases, integration may be automated, but only after a human studies samples of the data and determines the procedure to follow. For example, a human may write a program that extracts yearly sales figures from IBM letters to stockholders and then "joins" this data (by year) with a table of dividends.

In light of these differences and difficulties, we believe that the goal is *not* to perform fully automated information integration that hides all diversity from the user, but rather to provide a framework and tools to assist humans (end users and/or humans programming integration software) in their information processing and integration activities. So, what should the framework and tools look like? There are at least three categories:

1. **Information exchange.** The various components of an information system need to exchange data objects (units of information), either for examination by an end user or for integration with other data objects. For this, there needs to be an agreement as to how objects will be requested, how they will be represented, what the semantic meaning of each object (and its components) is, and how objects are actually transported over a network. Once an exchange format is agreed upon, there need to be tools for translating between an information source and the exchange format.

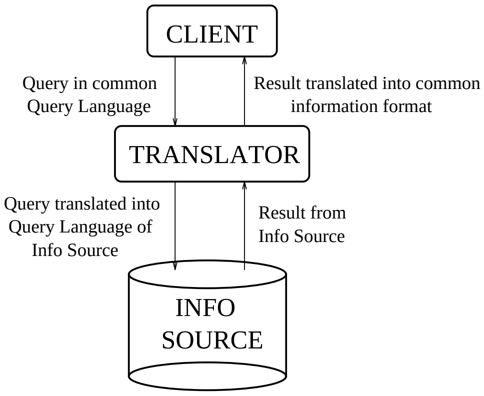

Figure 1: Communication through a translator

2. **Information discovery and browsing.** Users will want to explore the available information, discovering sources, browsing objects, and learning the semantics of objects and their components. Tools for information discovery and browsing allow humans (and ultimately software) to query for sources of interest, to request objects from sources, to navigate through objects (exploring their components), and to ask questions about the meaning of objects and their components.

3. **Mediators.** A mediator is a program that collects information from one or more sources, processes and combines it, and exports the resulting information [21]. For example, a mediator could be a program that collects IBM yearly stockholder reports, extracts key figures, and exports a table of yearly results. A second mediator might take this table and combine it with a stock price report to produce a trends analysis for IBM. We envision a variety of tools to assist the mediator writer, some resembling a programming environment, others presenting a menu of common ways of combining information.

In this paper we focus particularly on the information exchange problem discussed in point 1, since we believe this problem needs to be solved before browsing tools or mediators can be constructed. To motivate the information exchange problem further, consider an information source IS that contains bibliographic entries such as those found in many libraries. Some client C (human or otherwise) wishes to locate all books by author "J.D. Ullman" on the topic of "databases." Since IS and C are likely to be different, we need a common language and information format for communication. Client C uses the common language to express a query that requests the desired object. A front end to IS, which we call a *translator*, converts the query to a form that IS can process. When IS responds (with a set of bibliographic entries in some format), the translator converts the response into an object in the common format and transmits it to C. Finally, C may choose to translate the object (or the components it wants) into its own internal model. This form of communication is illustrated in Figure 1.

In Section 2 we present an "object exchange model" (OEM) that we believe is well suited for information exchange in heterogeneous, dynamic environments. OEM is flexible enough to encompass all types of information, yet it is simple enough to facilitate integration; OEM also includes semantic information about objects. In Section 3 we describe the query language we have designed for requesting objects in OEM. In Section 4 we describe how we have used OEM to integrate several heterogeneous bibliographic information sources. In Section 5 we present a pair of general-purpose libraries we have implemented that support OEM object exchange between any client and server processes. The procedures in these libraries provide communication services, session handling, object memory management, and partial object fetches. Calls to these procedures are embedded in client programs. In Section 6 we conclude and discuss our ongoing work in information integration using OEM.

2 Object Exchange Model

The first question to be addressed is: with so many data models around, why do we need another one? In fact we do *not* need another new model. Rather, we adopt a model that has been in use for many years. The basic idea is very simple: each value we wish to exchange is given a *label* (or *tag*) that describes its meaning. For example, if we wish to exchange the temperature value 80 degrees Fahrenheit, we may describe it as:

⟨temperature-in-Fahrenheit, integer, 80⟩

where the string "temperature-in-Fahrenheit" is a human-readable label, "integer" indicates the type of the value, and "80" is the value itself. If we wish to exchange a complex object, then each component of the object has its own label. For example, an object representing a set of two temperatures may look like:

⟨set-of-temperatures, set, {$cmpnt_1$, $cmpnt_2$}⟩
$cmpnt_1$ is ⟨temperature-in-Fahrenheit, integer, 80⟩
$cmpnt_2$ is ⟨temperature-in-Celsius, integer, 20⟩

A main feature of OEM is that it is *self-describing.* We need not define in advance the structure of an object, and there is no notion of a fixed schema or object class. In a sense, each object contains its own schema. For example, "temperature-in-Fahrenheit" above plays the role of a column name, were this object to be stored in a relation, and "integer" would be the domain for that column.[1]

Note that, unlike in a database schema, a label here can play two roles: identifying an object (component), and identifying the meaning of an object (component). To illustrate, consider the following object:

⟨person-record, set, {$cmpnt_1$, $cmpnt_2$, $cmpnt_3$}⟩
$cmpnt_1$ is ⟨person-name, string, "Fred"⟩

[1]Of course, if we are exchanging a set of objects where each object has the same structure and labels, then it would be redundant to transmit labels with every member of the set. We view this as a data compression issue and do not discuss it further here. From a logical point of view, we assume that each object in our model carries its own label.

$cmpnt_2$ is ⟨office-number-in-building-5, integer, 333⟩
$cmpnt_3$ is ⟨department, string, "toy"⟩

Like a column name in a relation, the label "person-name" identifies which component in the person's record contains the person's name. In addition, the label "person-name" identifies the meaning of the component—it is the name of a person. We would not expect to find a dog's name "Fido" or "Spot" in this component.

Thus, we suggest that labels should be as descriptive as possible. (For instance, in our example above, replacing "person-name" by "name" would not be advisable.) In addition, if an information source exports objects with a particular label, then we assume that the source can answer the question *What does this label mean?*. The answer should be a human-readable description—a type of "man page" (similar in flavor to Unix Manual pages). For example, if we ask the source that exports the above object about "person-name," it might reply with a text note explaining that this label refers to names of employees of a certain corporation, the names do not exceed 30 characters, and upper vs. lower case is not relevant.

It is particularly important to note that labels are relative to the source that exports them. That is, we do not expect labels to be drawn from an ontology shared by all information sources. For example, a client might see the label "person-name" originating from two different sources that provide personnel data for two different companies, and the label may mean something different for each source; the client is responsible for understanding the differences. If the client happens to be a mediator that exports combined personnel data for the two companies, then the mediator may choose to define a new label "generic-person-name" (along with a "man page"), to indicate that the information is not with respect to a particular company. Mediators are discussed further in Section 4.2.

We believe that a self-describing object exchange model provides the flexibility needed in a heterogeneous, dynamic environment. For example, personnel records could have fewer or more components than the ones suggested above; in our temperatures set, we could dynamically add temperatures in Kelvin, say. In spite of this flexibility, the model remains very simple.

As mentioned earlier, the idea of self-describing models is not new—such models have been used in a variety of systems (see Section 2.2 for a discussion of these models and systems). Consequently, the reader may at this point wonder why we are writing a paper about a self-describing model, if such models have been used for many years. A first reason is that we believe it is useful to formally cast a self-describing model in the context of information exchange in heterogeneous systems (something that has not been done before, to the best of our knowledge), and to extend the model to include object nesting as illustrated above. To do this, a number of issues had to be addressed, as will be seen in subsequent sections. A second reason is to provide an appropriate object request language based on the model. Our language is similar to nested-SQL languages; however, we believe that the use of labels within objects leads to a language that is more intuitive than nested-SQL (see Section 3).

2.1 Specification

Each object in OEM has the following structure:

Label	Type	Value	Object-ID

where the four fields are:

- **Label:** A variable-length character string describing what the object represents.
- **Type:** The data type of the object's value. Each type is either an *atom* (or *basic*) type (such as `integer`, `string`, `real number`, etc.), or the type `set` or `list`. The possible atom types are not fixed and may vary from information source to information source.
- **Value:** A variable-length value for the object.
- **Object-ID:** A unique variable-length identifier for the object or Λ (for `null`). The use of this field is described below.

In denoting an object on paper, we often drop the Object-ID field, i.e. we write ⟨label,type,value⟩, as in the examples above.

Object identifiers (henceforth referred to as OID's) may appear in set and list values as well as in the Object-ID field. We provide a simple example to show how sets (and similarly lists) are represented without OID's, and to motivate the kind of OID's that are used in OEM. Then we discuss OID's in set and list values.

Suppose an object representing an employee has label "employee" and a set value. The set consists of three subobjects, a "name," an "office," and a "photo." All four objects are exported by an information source IS through a translator, and they are being examined by a client C. The only way C can retrieve the employee object is by posing a query (see Section 3) that returns the object as an answer.

Assume for the moment that the employee object is fetched into C's memory along with its three subobjects. The value field of the employee object will be a set of *object references*, say $\{o_1, o_2, o_3\}$. Reference o_1 will be the memory location for the name subobject, o_2 for the office, and o_3 for the photo. Thus, on the client side, the retrieved object will look like:

⟨employee, set, $\{o_1, o_2, o_3\}$⟩
o_1 is location of ⟨name, string, "some name"⟩
o_2 is location of ⟨office, string, "some office"⟩
o_3 is location of ⟨photo, bitmap, "some bits"⟩

On the information source side, the employee object may map to a real object of the same structure, or it may be an "illusion" created by the translator from other information. Suppose IS is an object database, and the employee object is stored as four objects with OID's id_0 (employee), id_1 (name), id_2 (office), and id_3 (photo). In this case, the retrieved object on the client side would have id_0 in the Object-ID field for the employee object, id_1 in the Object-ID field for the name object, and so on. The non-null Object-ID fields tell client C that the objects it has correspond to identifiable objects at IS.

Now suppose instead that IS is a relational database, and that the employee "object" is actually a tuple. Hence, the name, office, and photo objects (attributes of the tuple) do not have OID's, so their Object-ID field at the client side will be Λ (null). The employee object may have an immutable tuple identifier, which can be used in the Object-ID field at the client. Alternatively, the employee's Object-ID field at the client might contain Λ, or it might contain an SQL statement that retrieves the employee record based on its key attribute.

So far we have assumed that the client retrieves the employee object and all of its subobjects. However, for performance reasons, the translator may prefer not to copy all subobjects. For example, if the photo subobject is a large bitmap, it may be preferable to retrieve the name and office subobjects in their entirety, but retrieve only a "placeholder" for the photo object. In this case, the value field for the employee object at the client will contain $\{o_1, o_2, id_3\}$. This indicates that the name and office subobjects can be found at memory locations o_1 and o_2, but the photo subobject must be explicitly retrieved using OID id_3.

Thus, at the client, sets and lists contain elements that may be of two forms, as follows. We assume there is an internal tag that indicates the form of each element.

- Local Object Reference: This identifies an object stored at the client. It will typically be a memory location, but if local objects are cached in an object database, then object references could be *Local OID's* in this database.

- Remote OID: This identifies an object at the information source.[2] Each Remote OID is either *lexical* or *non-lexical*. Lexical OID's are printable strings, and they may be specified directly in our query language (see Section 3). Non-lexical OID's are "black boxes," such as the tuple identifiers or SQL queries described above. Clients may pass non-lexical OID's to translators using special interfaces, but since the OID's are not printable, they cannot be used in queries. Remote OID's could be classified further by other properties [6], such as whether they are permanent or temporary [4]. (Or, OID's could include a "valid timestamp" specifying when they expire.) We do not consider these further classifications here, although we may incorporate these concepts in a future extension of our model.

Note that, regardless of the representation used in set and list values, the translator always gives the client the illusion of an object repository. Thus, we can think of our employee object as:

$\langle$employee, set, $\{cmpnt_1, cmpnt_2, cmpnt_3\}\rangle$
  $cmpnt_1$ is $\langle$name, string, "some name"$\rangle$
  $cmpnt_2$ is $\langle$office, string, "some office"$\rangle$
  $cmpnt_3$ is $\langle$photo, bits, "some bits"$\rangle$

[2]We assume that identifiers are unique for each information source. Uniqueness across information sources can be achieved by, e.g., prepending each object identifier with a unique ID for the information source.

where each $cmpnt_i$ is some mnemonic identifier for the subobject. We use this generic notation for examples throughout the paper.

A final issue regarding OEM is that of duplicate objects at the client. Suppose, for example, that set object A at the information source has B and C as subobjects. Both B and C are of set type, and both have as subobjects the same object D. A query at a client retrieves A and all of its subobjects. Will the client have a single copy of object D, or will objects B and C point to different copies of D? Our model does *not* require a single copy of D at the client, since this would place a heavy burden on translators that are not dealing with real objects at the information source. However, if both copies of D have the same (non-null) Object-ID field, then the client can discern that the two objects correspond to the same object at the source. Also note that we do not require translators to discover cyclic objects at the source. Suppose, for example, that A has B as a subobject and B has A as a subobject. If the client fetches A from a "smart" translator, the translator would return only two objects, a copy of A and a copy of B. Each object's set value would be a reference for the other object. However, a "dumb" translator is free to return, say, four objects, A_1, B_1, A_2, B_2, where A_1 references B_1, B_1 references A_2, A_2 references B_2, and B_2 contains the empty set to indicate that for performance reasons the chain was not followed.

2.2 Related Models and Systems

In this section we contrast OEM with other similar models and systems. We focus particularly on the differences between OEM and more conventional object-oriented models, and we discuss the motivation behind our design of OEM.

Labeled fields are used as the basis of several data models or data formatting conventions. For example, a *tagged file system* [20] uses labels instead of positions to identify fields; this is useful when records may have a large number of possible fields, but most fields are empty. Electronic mail messages consist of label-value pairs (e.g. label "From" and value "yannis@cs.stanford.edu"). More recently, Lotus Notes [15] has used a label-value model to represent office documents, and Teknekron Software Systems [16] has used a self-describing object model for exchange of information in their stock trading systems. In [13] and [14] self-describing databases are proposed as a solution to obtaining the increased flexibility required by heterogeneous systems.

Recent projects on heterogeneous database systems (e.g., [1,3,11]) have applied object-oriented (OO) data models to the problem of database integration. OEM differs from these and other OO data models in several ways. First, OEM is an information *exchange* model. OEM does not specify how objects are stored at the source. OEM does specify how objects are received at a client, but after objects are received they can be stored in any way the client likes. OEM explicitly handles cross-system OID's (e.g., in Section 2.1 an employee object at the client points to a photo object at the source). In a conventional OO system there may also be client copies of server objects, but there the client copy is logically identical to the server copy and an application program at the client is not aware of the difference.

A very important difference between OEM and conventional OO models is that OEM is much simpler. OEM supports only *object nesting* and *object identity*; other features such as classes, methods, and inheritance are omitted. (Incidentally, [4] claims that the only two essential features of an OO data model are nesting and object identity.) Our primary reason for choosing a very simple model is to facilitate integration. As pointed out in [2], simple data models have an advantage over complex models when used for integration, since the operations to transform and merge data will be correspondingly simpler. Meanwhile, a simple model can still be very powerful: advanced features can be "emulated" when they are necessary. For example, if we wish to model an employee class with subclasses "active" and "retired," we can add a subobject to each employee object with label "subclass" and value "active" or "retired." Of course this is not identical to having classes and subclasses, since OEM does not force objects to conform to the rules for a class. While some may view this as a weakness of OEM, we view it as an advantage, since it lets us cope with the heterogeneity we expect to find in real-world information sources.[3]

The flexible nature of OEM can allow us to model complex features of a source in a simple way. For example, consider a deductive database that contains a `parent` relation and supports the recursive `ancestor` relation through derivation rules. If we wish to provide an OEM model of this data in which it is easy to locate a person's ancestors, we can make the object that corresponds to each person contain as subobjects the objects that correspond to his/her parents. It is then simple to pose a query in our OEM query language (see Section 3) that retrieves all of a person's ancestors. In addition, a user can browse through a person's "family tree" using the browsing facility described in Section 4.1.

A final distinct difference between OEM and conventional OO models is the use of labels in place of a schema. Clearly, it would be trivial to add labels to a conventional OO model (e.g., all objects could have an attribute called "label"). The only difference then is that in OEM labels are first-class citizens. We believe this small change makes interpretation and manipulation of objects more straightforward, as discussed in the next section. Note that the schema-less nature of OEM is particularly useful when a client does not know in advance the labels or structure of OEM objects. In traditional data models, a client must be aware of the schema in order to pose a query. In our model, a client can discover the structure of the information as queries are posed.

3 Query Language

To request OEM objects from an information source, a client issues queries in a language we refer to as *OEM-QL*. OEM-QL adapts existing SQL-like languages for object-oriented models to OEM.

The basic construct in OEM-QL is an SQL-like `SELECT-FROM-WHERE` expression. The syntax is:

[3] Note that some proposed interchange standards, e.g. CORBA's Object Request Broker [8], tend to be significantly more complex than OEM. We expect that if such standards are adopted, OEM could be used to provide a simpler, more "client-friendly" front end. Other proposed standards, such as ODMG's Object Database Standard [5], are directed towards interoperability and portability of object-oriented database systems, rather than towards facilitating object exchange in highly heterogeneous environments.

```
SELECT Fetch-Expression
FROM Object
WHERE Condition
```

The result of this query is itself an object, with the special label "answer":

$\langle$answer, set, $\{obj_1, obj_2, \ldots, obj_n\}\rangle$

Each returned subobject obj_i is a component of the object specified in the `FROM` clause of the query, where the component is located by the *Fetch-Expression* and satisfies the *Condition*. Details are given below. We assume that the *Object* in the `FROM` clause is specified using a lexical object-identifier, and that for every information source there is a distinguished object with lexical identifier "root." (Sources may or may not support additional lexical identifiers.) Certainly the query language may be extended with a call interface that allows non-lexical object identifiers in `FROM` clauses.

The *Fetch-Expression* in the `SELECT` clause and the *Condition* in the `WHERE` clause both use the notion of a *path*, which describes a traversal through an object using subobject structure and labels. For example, the path "`bibliography.document.author`" describes components that have label "`author`," and that are subobjects of an object with label "`document`" that is in turn a subobject of an object with label "`bibliography`." Paths are used in the *Fetch-Expression* to specify which components are returned in the answer object; paths are used in the *Condition* to qualify the fetched objects or other (related) components in the same object structure. A path specified in a *Fetch-Expression* may be terminated by the special symbol "`OID`," in which case only object identifiers are returned in the answer object, rather than the objects themselves.[4] A syntax and semantics for the basic constructs of OEM-QL is given in the Appendix. In the remainder of this section we provide a number of examples that serve to illustrate its capabilities.

For the examples, suppose that we are accessing a bibliographic information source with the object structure shown in Figure 2. (Note that we are using mnemonic object references; recall Section 2.) Let the entire object (i.e., the top-level object with label "bibliography") be the distinguished object with lexical object identifier "root". Note that although much of this object structure is regular—components have the same labels and types—there are some irregularities. For example, the call number format is different for each document shown, and the third document uses a different structure for author information.

Example 3.1 Our first example retrieves the topic of each document for which "Ullman" is one of the authors:

```
SELECT bibliography.document.topic
FROM root
WHERE bibliography.document.author-set.author-last-name = "Ullman"
```

[4]If there are qualifying objects without OID's, these objects are not returned in the answer object.

⟨bibliography, set, $\{doc_1,\ doc_2,\ \ldots,\ doc_n\}$⟩
  doc_1 is ⟨document, set, $\{authors_1,\ topic_1,\ call\text{-}number_1\}$⟩
    $authors_1$ is ⟨author-set, set, $\{author_1^1\}$⟩
      $author_1^1$ is ⟨author-last-name, string, "Ullman"⟩
    $topic_1$ is ⟨topic, string, "Databases"⟩
    $call\text{-}number_1$ is ⟨internal-call-no, integer, 25⟩
  doc_2 is ⟨document, set, $\{authors_2,\ topic_2,\ call\text{-}number_2\}$⟩
    $authors_2$ is ⟨author-set, set, $\{author_2^1,\ author_2^2, author_2^3\}$⟩
      $author_2^1$ is ⟨author-last-name, string, "Aho"⟩
      $author_2^2$ is ⟨author-last-name, string, "Hopcroft"⟩
      $author_2^3$ is ⟨author-last-name, string, "Ullman"⟩
    $topic_2$ is ⟨topic, string, "Algorithms"⟩
    $call\text{-}number_2$ is ⟨dewey-decimal, string, "BR273"⟩
  ⋮
  doc_n is ⟨document, set, $\{authors_n,\ topic_n,\ call\text{-}number_n\}$⟩
    $authors_n$ is ⟨single-author-full-name, string, "Michael Crichton⟩
    $topic_n$ is ⟨topic, string, "Dinosaurs"⟩
    $call\text{-}number_n$ is ⟨fiction-call-no, integer, 95⟩

Figure 2: Object structure for example queries

Intuitively, the query's `WHERE` clause finds all paths through the subobject structure with the sequence of labels [`bibliography`, `document`, `author-set`, `author-last-name`] such that the object at the end of the path has value "Ullman." For each such path, the `FROM` clause specifies that one component of the answer object is the object obtained by traversing the same path, except ending with label `topic` instead of labels [`author-set`, `author-last-name`]. Hence, for the portion of the object structure shown in Figure 2 the query returns:

⟨answer, set, $\{obj_1,\ obj_2\}$⟩
  obj_1 is ⟨topic, string, "Databases"⟩
  obj_2 is ⟨topic, string, "Algorithms"⟩ □

Example 3.2 Our second example illustrates the use of "wild-cards" and an existential `WHERE` clause. This query retrieves the topics of all documents with internal call numbers.

```
SELECT bibliography.?.topic
FROM root
WHERE bibliography.?.internal-call-no
```

The "?" label matches any label. Therefore, for this query, the `document` labels in Figure 2 could be replaced by any other strings and the query would produce the same result. By convention, two occurrences of ? in the same query must match the same label unless variables are used (see below). Note that there is no comparison operator in the `WHERE` clause of this query, just a path. This means we only check that the object with the specified path exists; its value is irrelevant. Hence, for the portion of the object structure shown in Figure 2 the query returns:

⟨answer, set, {obj_1}⟩
    obj_1 is ⟨topic, string, "Databases"⟩ □

Example 3.3 In Example 3.2, the wild-card symbol ? was used to match any label. We also allow "wild-paths," specified by the symbol "$*$". Symbol $*$ matches any path of length one or more.[5] Using $*$, the query in the previous example would be expressed as:

```
SELECT *.topic
FROM root
WHERE *.internal-call-no
```

The use of $*$ followed by a single label is a convenient and common way to locate objects with a certain label in a complex structure. Similar to ?, two occurrences of $*$ in the same query must match the same sequence of labels, unless variables are used. □

Example 3.4 Our next example illustrates how variables are used to specify different paths with the same label sequence. This query retrieves each document for which both "Aho" and "Hopcroft" are authors:

```
SELECT bibliography.document
FROM root
WHERE bibliography.document.author-set.author-last-name(a1) = "Aho"
  AND bibliography.document.author-set.author-last-name(a2) = "Hopcroft"
```

Here, the query's `WHERE` clause finds all paths through the subobject structure with the sequence of labels [`bibliography`, `document`, `author-set`], and with two distinct path completions with label `author` and with values "Aho" and "Hopcroft" respectively. The answer object contains one "`document`" component for each such path. Hence, for the portion of the object structure shown in Figure 2 the query returns:

⟨answer, set, {*obj*}⟩
    obj is ⟨document, set, {$authors_2$, $topic_2$, $call\text{-}number_2$}⟩
        $authors_2$ is ⟨author-set, set, {$author_2^1$, $author_2^2$,$author_2^3$}⟩
            $author_2^1$ is ⟨author-last-name, string, "Aho"⟩
            $author_2^2$ is ⟨author-last-name, string, "Hopcroft"⟩
            $author_2^3$ is ⟨author-last-name, string, "Ullman"⟩
        $topic_2$ is ⟨topic, string, "Algorithms"⟩
        $call\text{-}no_2$ is ⟨dewey-decimal, string, "BR273"⟩ □

[5]Note that our use of wild-card symbols is similar to, e.g., Unix, X-windows, etc.

Example 3.5 Our next example illustrates how object identifiers may be retrieved instead of objects.[6] This query retrieves the OID's for all documents with a Dewey Decimal call number:

```
SELECT *.OID
FROM root
WHERE *.dewey-decimal
```

In this query, since the path in the FROM clause ends with "OID," only object identifiers are returned. Hence, for the portion of the object structure shown in Figure 2 the query returns:

$\langle$answer, set, $\{id_1\}\rangle$

where id_1 is the OID for the object referred to as doc_2 in Figure 2. □

Example 3.6 Although we have used only equality predicates so far, OEM-QL permits any predicate to be used in the *Condition* of a WHERE clause. The predicates that can be evaluated for a given information source depend on the translator and the source. Suppose, for example, that a bibliographic information source supports a predicate called author that takes as parameters a document and the last name of an author; the predicate returns *true* iff the document has at least one author with the given last name. Then the query in Example 3.4 might be written as:

```
SELECT bibliography.document
FROM root
WHERE author(bibliography.document, "Aho")
  and author(bibliography.document, "Hopcroft")
```

One of the translators we have built (see Section 4) is for a bibliographic information source called *Folio* that does in fact support a rich set of predicates. All of the predicates supported by Folio are available to the client through OEM-QL. □

In the Appendix we provide a grammar for the basic OEM-QL syntax and a semantics specified as the answer object returned for an arbitrary query. The basic OEM-QL described in this paper is certainly amenable to extensions. For example, here we have allowed only one object in the FROM clause, so "joins" between objects cannot be described at the top level of a query. The language can easily be extended to allow multiple objects in the FROM clause. Similarly, the SELECT clause allows only one path to be specified; "constructors" can be added so that new object structures can be created as the result of a query. While these extensions are clearly useful, and we plan to incorporate them in the near future, we also expect that many translators (especially translators for unstructured and semi-structured information sources) will support only the basic OEM-QL (some may even support just a subset), since supporting the full extended language may result in unreasonable increase of the translator's complexity. One useful extension we plan for OEM-QL,

[6]Here the client is explicitly requesting OID's instead of objects. In other cases OID's may be retrieved instead of objects for efficiency; recall Section 2.1.

and we expect will be supported by most translators, is the ability to express queries about labels and object structure: we expect that clients will frequently need to "learn" about the objects exported by an information source before meaningful queries can be posed.

3.1 Related Languages

Many query languages for object-oriented and nested relational data models are based on an extension of SQL with path expressions, e.g. [10,11,12,17]. As stated earlier, OEM-QL can be viewed as an adaptation of these languages to the specifics of OEM.

In OEM-QL, path expressions range only over objects, while in most other languages they range over the schema and the objects. For example, consider the `WHERE` condition `document.author = "Smith"`. In OEM-QL, we simply find all objects with label `document` that have a subobject with label `author` and value "Smith." In a conventional OO language, we would have to identify a class `document` with an attribute named `author`. Then we would range over all objects of class `document` looking for the matching name. We believe that the simplicity of ranging over objects only leads to a more intuitive language and a more compact language definition.

A significant feature of OEM-QL is that it lets us query information sources where there is no regular schema. A conventional language breaks down in such a case, unless one defines an object class for every possible type of irregular object. (Note that such a schema would have to be modified each time a different object appeared.) Of course, if a particular information source does have a schema and a regular structure, the translator for that source should take advantage of the schema. For example, suppose all objects are stored in a relational database, and the translator receives the `WHERE` condition `document.author = "Smith"`. The translator could first check that there is a relation `document` with attribute `author` and, if so, could use an index to fetch the matching objects. Thus, the fact that the model and language do not require a schema does not mean that a schema cannot be used for query processing.

4 Implementation of Translators, Browsers, and Mediators

We have argued that OEM and its query language are designed to facilitate integrated access to heterogeneous data sources. To support this claim, in this section we describe how we have applied OEM to a particular scenario. The scenario consists of a variety of bibliographic information sources, including a conventional library retrieval system, a relational database holding structured bibliographic records, and a file system with unstructured bibliographic entries. Using our OEM-based system, these sources are accessible through a general-purpose user interface that allows evaluation of queries and object exploration.

Our first operational translator accesses the Stanford University *Folio* System. Folio provides access to over 40 repositories, including a catalog of the holdings of Stanford's libraries, and several commercial sources such as INSPEC that contain entries for Computer Science and other published articles. Folio is the most difficult of our information sources, partly because the translator must emulate an interactive terminal. The translator initially must establish a connection with Folio, giving the necessary account and access information. When the translator receives an OEM-QL query to evaluate, it converts the query into Folio's Boolean retrieval language. Then it extracts the relevant information from the incoming screens and exports the information as an OEM answer object. The Folio translator is written in C and runs as a server process on Unix BSD4.3 systems. We have also implemented several simple mediators that refine the objects exported by the translator (see Section 4.2). Translators for the other bibliographic sources are nearly complete—they have involved substantially less coding because the underlying sources (e.g., a relational database) are much easier to use. Our translators and mediators are discussed further in Section 4.2.

We have also implemented *OEM Support Libraries* to facilitate the creation of future translators, mediators, and end-user interfaces. These libraries contain procedures that implement the exchange of OEM objects between a server (either a translator or a mediator) and a client (either a mediator, an application, or an interactive end-user). The Support Libraries handle all TCP/IP communications, transmission of large objects, timeouts, and many other practical issues. A Unix BSD4.3 and a Windows version of the package have been implemented and demonstrated. The Support Libraries are described in Section 5.

Finally, we have implemented a *Heterogeneous Information Browser* that lets a user submit queries and explore resulting objects.[7] The Browser is implemented in Visual C++ and runs under Windows. The next subsection describes the Browser in more detail. We believe the Browser illustrates the desirability of a simple model and language from the point of view of a user who may not be familiar with the underlying information.

4.1 The Heterogeneous Information Browser

The Heterogeneous Information Browser (HIB) provides a graphical user interface for submitting queries and exploring results. We illustrate its operation by walking through a particular interaction. Refer to Figure 3.

When the HIB is opened, it displays a menu of known translators and/or mediators (hereafter referred to as TM's). Each entry of the menu specifies the name of a TM, the site where it can be found, the communication protocol it uses, and other information that may be needed for locating the TM and connecting to it. The user may select any of the TM's on the menu, or the user may enter a new TM not listed.

After a connection is established, an information exchange *session* starts. The user can either type a query directly into the *Active Query* window, or he may select one of the *Frequently-Asked-Queries* shown in the *Queries* window. If a Frequently-Asked-Query is selected, it is copied to the Active Query window. (Typically, these are *fill-in-the-form* queries, so the user must complete the missing parts.) Frequently-Asked-Queries may come from two places: (1) the user may cache previously formulated queries; or (2) the source may provide a list of common queries (we have

[7] In [18] it is argued that user interfaces and browsers will play an important role in exploring heterogeneous information sources.

Figure 3: Querying and Object Browsing

not implemented this feature yet). For example, a translator for Folio may provide templates for finding documents by author, title, and subject, by far the most common queries. The ability to suggest common queries is especially important for "low end" TM's that do not implement the full OEM-QL. In such a case, the user needs guidance as to what queries the source will be able to process.

If a submitted query is valid and successfully executed by the TM, the answer object is returned to the HIB. The user can then navigate through the object structure of the answer. This is better understood if we think of the answer as a tree (or a graph, in the most general case), where the atom objects are the leaves, and the set objects are the internal nodes. Initially, the root and its immediate subobjects are displayed in the object viewer, as illustrated in the left window of Figure 3. Here, the root (label `answer`) is a set of six documents (label `doc`). The user can move from the current node to another node by clicking on any of the highlighted direction buttons at the bottom of the window. If a button is not activated, there is no object in that direction. For example, in the left window of Figure 3 one cannot move `UP` because there are no objects "above" the root. However, the user can move `DOWN` to the first child of the answer object; the result is shown in the right window of Figure 3. During navigation, the object viewer always shows two levels of the structure (which can be generalized to k levels). Thus, when the current object is a document (label `doc`) one can see its components, i.e., the `TITLE`, `AUTHOR`, and so on (right window). If an atom value is too large to be seen in the viewer (e.g., the abstract of a document), the user can click on it to open a full window that displays the value.

At any time, the user can click on the `HELP` button to display the "man page" for the label of the current object. As discussed in Section 2, each TM answers the question *What does label* `X` *mean?* by returning a manual entry. This entry describes in English the meaning of the label and how the value of the object should be interpreted. For example, the entry for the `author` label under Folio would explain that names consist of a last name followed by a first name or initials, it would specify the maximum length allowed, it would explain how multiple authors are displayed, and so on. We feel this is a very useful feature of our approach: any time one sees a data value, it is accompanied by a label, and one can immediately find the meaning of the label. This is not only useful to the end-user, but also to the mediator implementor who needs to understand the data that is being integrated or processed.[8]

Notice that the self-describing nature of OEM makes it easy for a user to navigate through unknown objects. If a user knows nothing about a particular source, he can simply pose the query:

```
SELECT ?
FROM root
```

and then browse. As he examines the retrieved labels and their "man pages," he can learn the meaning of each component. Then he can pose more refined queries.

4.2 Translators and Mediators

In this section we illustrate how OEM is used for translation and mediation in the context of our heterogeneous bibliographic information source scenario. The general architecture is shown in Figure 4. Translators are built for all participating bibliographic sources. On top of the translators we use mediators [22] to support objects and queries that are more refined than the objects and queries supported by lower-level translators or mediators. In particular, the mediators directly above the translators reconcile discrepancies between sources (e.g., differences in the structure of objects, the naming of labels, the format of values, etc.), simplifying the task of the mediator that combines information from multiple sources.

To illustrate the operation of the translators and mediators, consider the *Folio* information source and its translator. The Folio translator T receives OEM-QL queries and issues Folio queries. The set of queries $q(T)$ that T is able to translate and execute should have two properties:

1. The translation of any $q(T)$ query into a corresponding Folio query should be as simple as possible, to minimize the translation implementation effort.

2. The set $q(T)$ should preserve as much as possible the power of the underlying query language. Ideally, there should be no Folio query that does not have a corresponding query in $q(T)$.

We have satisfied both properties in the case of Folio by supporting predicates in OEM-QL that correspond directly to the access methods that Folio provides. As an example, Figure 4 shows a typical query entering Folio, asking for the bibliographic entries where the last name of one of the authors is "Ullman" and the first name starts with "J." The corresponding query in OEM-QL is:

[8] The requirement of providing a "man page" for each label could be viewed as a burden, but if the meaning of information is not documented, there is no hope for heterogeneous information access!

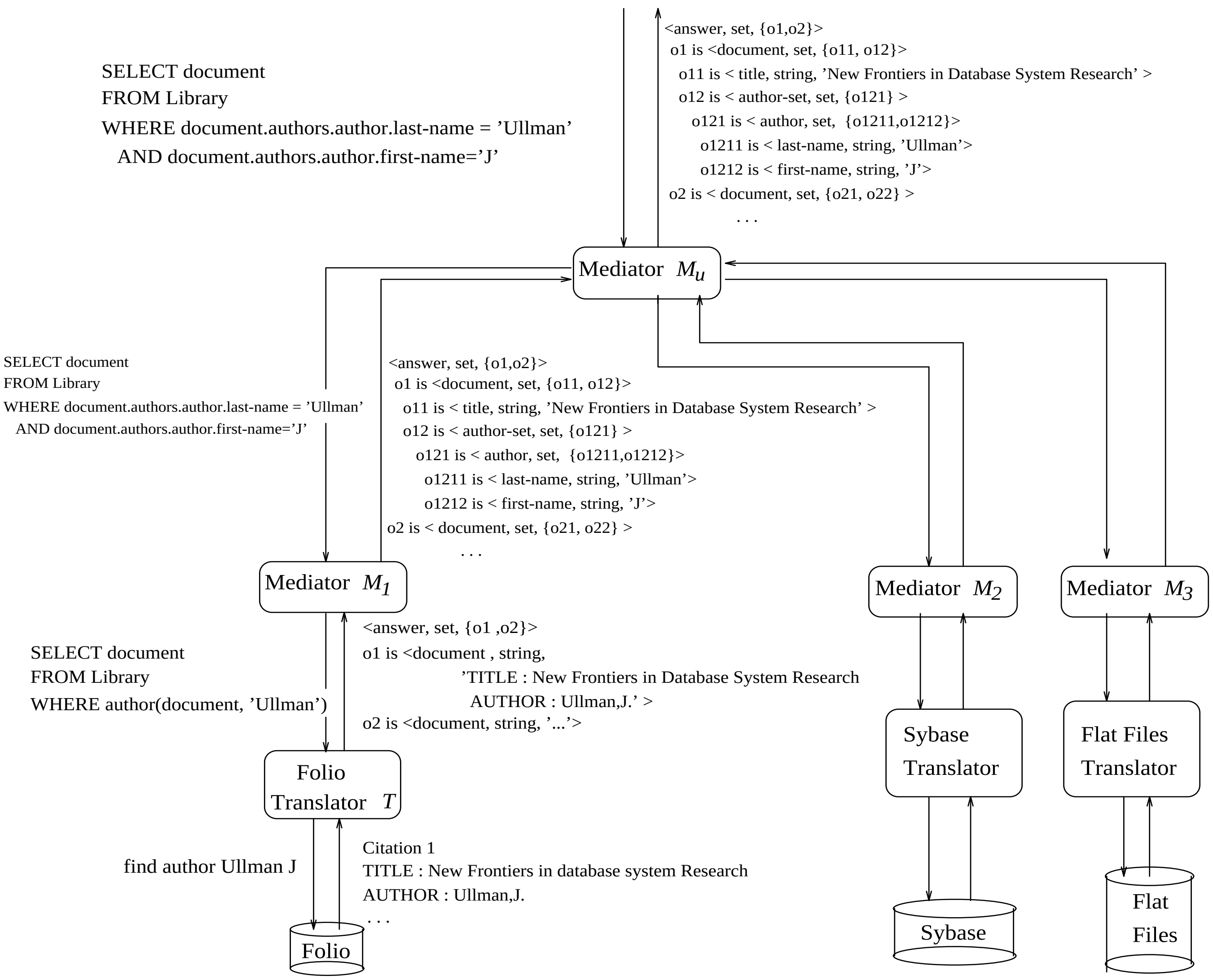

Figure 4: Translation and Mediation Architecture

```
SELECT collection.document
FROM Folio
WHERE author(collection.document, "Ullman J")
```

From this query, T only needs to translate the `author` predicate to the corresponding author search construct.

As illustrated in Figure 4, translator T uses a straightforward mapping to translate the citations returned from Folio (as a string) into an OEM object. Mediator M_1 refines the structure of the objects exported by T, by extracting the basic components of each bibliographic object (e.g., authors, title). In addition, M_1 supports a wider and more generic set of queries than T. For example, M_1 is able to translate the incoming query shown in Figure 4 to the outgoing one.

A key design criterion here is modularity. Since the translators are likely to be the most complex components (they must deal with the idiosyncrasies of the information sources), our goal is to keep the work of the translators to a minimum. Once a translator produces its object in some OEM format, additional work can be done by mediators. Note that [7] suggests an average of 6 months effort to implement a translator for a conventional DBMS. In our experience, the total effort can be reduced substantially by shifting work from translators to mediators, and by using the Support Libraries described in Section 5.

The top level mediator M_u in Figure 4 combines the information from several sources into a single document collection. The simplest implementation of this mediator performs a union of all the collections. When M_u receives a query, it effectively "broadcasts" the query to all mediators at lower levels, then merges the answers. Certainly more sophisticated mediation techniques could be useful, such as recognizing and eliminating duplicate results. In the following subsection we describe some initial ideas we have for specifying and implementing mediators.

4.2.1 Mediator Generation

Implementing mediators is a non-trivial task, so our eventual goal is to develop tools for mediator generation. (Similar tools can be used for translator generation, but we focus on mediators here.) The approach described in this section has not yet been implemented, but the ideas are presented to illustrate the type of generators we expect OEM will lead to.

The object translation work of mediator M_1 in Figure 4 (i.e. the "upward" direction) could be described by the following "rule":

```
document replaced by derive_structure(document)
```

This rule specifies that whenever M_1 receives an object O from T, M_1 replaces each (sub)object O_i of O that has label `document` by a (sub)object O_i' created by `derive_structure`(O_i). The function `derive_structure()` may be implemented in a conventional programming language. However, we are currently developing *object-pattern-matching* and *string-pattern-matching* tools that describe object transformations in a high level "label-driven" language. In this way we will often eliminate the need for conventional programming of mediators.

Mediator generators can also describe the process of query translation (the "downward" direction). One approach that easily tackles simple cases relies on templates that describe how predicates (or groups of predicates) in incoming queries are replaced by predicates (or groups of predicates) in outgoing queries. For example, M_1 might use the following query rewriting templates, where `X` and `Y` represent variables to be matched:

```
T1. document.authors.author.last-name = "X"
    AND document.authors.author.first-name = "Y"  =>  author(document, "XY")
T2. document.authors.author.last-name = "X"  =>  author(document, "X")
T3. document.authors.author.first-name = "Y"  =>  author(document, "Y")
```

Then, all queries processed by M_1 will be matched against the above templates. For example, if the query:

```
SELECT document
FROM Library
WHERE document.authors.author.last-name = "Ullman"
```

is received by M_1, template `T1` will be matched. Variable `X` will be instantiated to `Ullman` and the following query will be generated for T:

```
SELECT document
FROM Library
WHERE author(document, "Ullman")
```

As is commonly done in rule systems, our templates may be given an evaluation priority. Assume that `T1`, `T2`, and `T3` are in decreasing priority. The query received by M_1 in Figure 4 matches all three templates. Since template `T1` has highest priority, it is used for the translation shown in the Figure.

5 The OEM Support Libraries

OEM and OEM-QL are designed for a *client* to send queries and obtain corresponding answer objects from a *server*. The server may be a translator or a mediator, while the client may be a mediator or an end-user program (such as the HIB described in Section 4.1). We have implemented general-purpose OEM Support Libraries that provide the common functionality needed for object and query exchange. There are two main components: the *Client Support Library* (CSL) and the *Server Support Library* (SSL).

Figure 5 illustrates how the Support Libraries are used. The implementor of client applications links CSL with the client program in order to create programs with *embedded* CSL calls; CSL calls are used to establish connections with TM servers, to send OEM-QL queries, and to receive OEM objects.[9] CSL procedures handle all low level communications, and deposit retrieved objects in a main memory object buffer. At the server side, the SSL handles incoming connections, buffer management, and management of "slave" processes to execute queries. Note that if a server S obtains its information from another translator or mediator, then S also acts as a client, so it also uses the CSL.

We expect that our Support Libraries will expedite the implementation of mediators, translators, and end-user programs. In addition, implementing these libraries has brought to the surface a number of interesting issues regarding the exchange of objects when one or more participants are not inherently object-oriented. As far as we know, these issues do not arise in conventional, homogeneous object-oriented systems (or at least not in quite this way). Here we discuss one of the most important issues that has arisen, namely that of *partial object fetches*.

[9]Interactive (as opposed to embedded) OEM-QL queries can be posed using the browser described in Section 4.1, which is built on top of the Support Libraries.

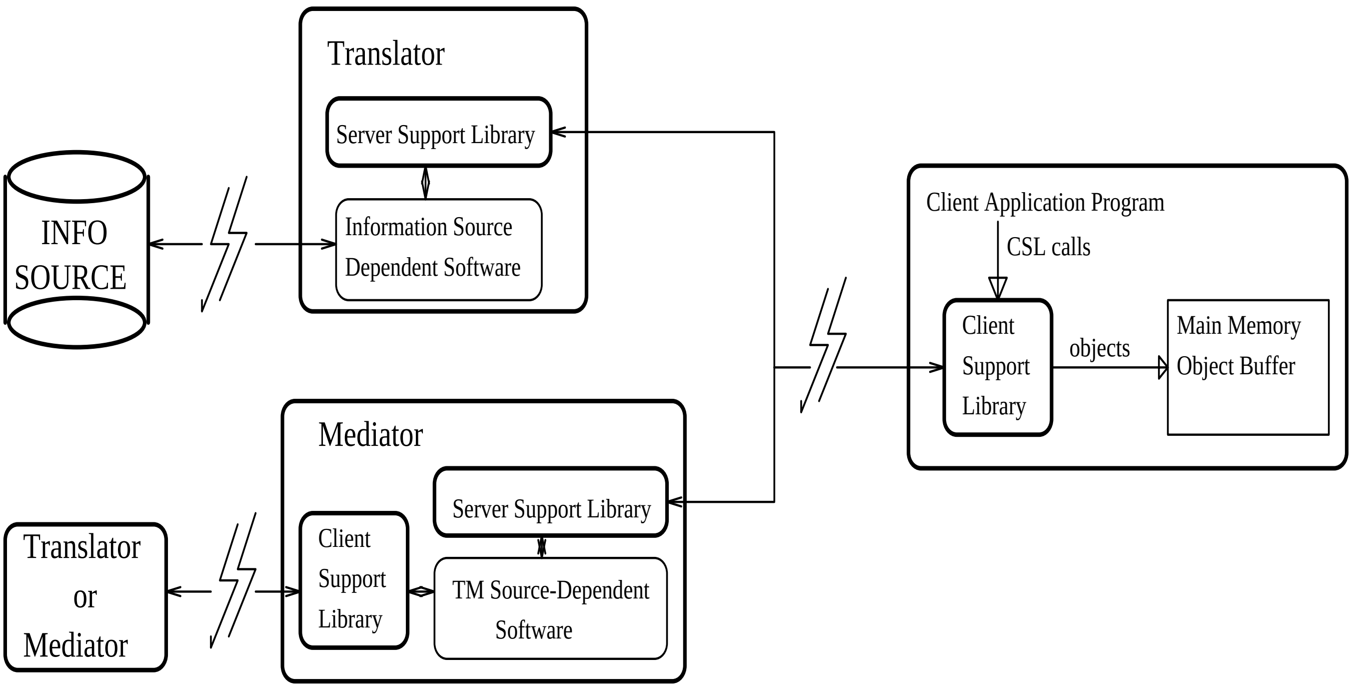

Figure 5: Use of the OEM Support Libraries

In many cases it is extremely inefficient to send the complete answer object to the client in one step. In particular:

1. The client has to wait until the full answer is retrieved from the information source before examining the object. This prevents "pipelined" operation, where the client starts processing subobjects as they arrive. The problem is exacerbated if we have a string of mediators between the source and the client: the client cannot begin processing the answer until all of the intermediate TM's have completed their work.

2. The answer object may be very large. Once a client inspects part of the answer object, the client may determine that it does not need some portions of the answer object, or perhaps does not need the object at all.

To avoid these problems, the Support Libraries provide a *partial fetch* mechanism that enables clients to retrieve only parts of the answer object. The mechanism is used as follows. When the client wishes to request an object, it calls a `query()` function, passing the OEM-QL query as a parameter. The client can then fetch either the full answer object (including subobjects) by calling the `getFullObject()` function, or the client can fetch only the root of the answer object by calling the `getRootObject()` function. In the latter case, additional `getFullObject()` and/or `getRootObject()` calls are used to fetch the subobjects.

Calls to the `getRootObject()` function lead to *incomplete* objects in the client's memory. To illustrate, consider an answer object A whose value is a set of three subobjects, B, C, and D. As discussed in Section 2.1, the copy of A placed in the client's memory can identify its subobjects in a variety of ways. For example, if subobject B has been fetched to memory, then A will contain a reference to B's memory location. If subobject C is a very large object and the server decides not to transfer it (as in, e.g., the bitmap object described in Section 2.1), then A will contain an OID for C. With partial object fetch there is a third possibility: a subobject, say D, may be "unfetched," i.e. it may be in the server's buffers, or not yet returned by the underlying source. The reference to an unfetched subobject is something that only the Support Libraries understand, and it is specific for the particular call in progress.

Consider what happens when a client wants to examine an unfetched object. One option is to support on-demand retrieval of any unfetched objects. However, this allows the client to traverse answer objects in arbitrary order, implying that the server must cache the entire answer object. Such on-demand fetching would be very difficult for translators such as the one for Folio (recall Section 4). The Folio bibliographic source returns a *stream* of documents, and the translator has no control over the order of the records. For on-demand service, all records would have to be stored by the translator. If the user poses a query that is too broad, the answer object might be enormous.

Consequently, instead of on-demand service, the Support Libraries provides a stream model for retrieving unfetched objects. A "preorder traversal" of the answer object is used, and the client must perform partial fetches in this order. To illustrate, suppose that after a first `getRootObject()` call, the client retrieves an object A whose set value contains three unfetched references, u_1, u_2, and u_3. If the client decides that the number of documents is too large, the client may choose to submit a different query. Otherwise, if the first document is desired, the client issues a `getRootObject()` call with u_1 as a parameter. The first subobject is fetched; suppose it is another set with unfetched references u_{11} and u_{12}. Next the client fetches u_{11}, which happens to be the title of the document. Based on this, the client may decide it wants to skip the rest of the u_1 object. It can do so by issuing a `getRootObject()` call with u_2; this causes the u_1 subobjects that were not fetched to be discarded. Thus, even though the client is constrained to traverse the answer object in a particular order, uninteresting parts can be skipped. At the server side, the uninteresting parts still have to be fetched, but they can be discarded without being transmitted to the client.

Due to space limitations, our description of the OEM Support Libraries and their services has been cursory. Our goal has not been a full description of the Support Libraries, but rather an illustration of the challenging practical issues that arise when there is an "impedance mismatch" between the way an information source provides objects and the way a client wishes to see them. We believe that our Support Libraries provide a general-purpose framework for handling many of these issues.

6 Conclusions and Future Work

We are developing a complete environment and set of tools for integrated access to diverse and dynamic heterogeneous information sources. Exchange of information in our environment is based on the *Object Exchange Model* (OEM) introduced in this paper. OEM retains the simplicity of relational models while allowing the flexibility of object-oriented models. Objects in OEM have a very simple structure, yet the model is powerful enough to encode complex information. For flexibility, OEM objects are *self-describing*. This approach eliminates the need for regular structure or a predefined schema. However, when structure or schema are present, they can be exploited by OEM translators and mediators.

OEM objects are requested using a declarative query language *OEM-QL*, which is based on nested-SQL query languages. We have found OEM-QL to be both expressive and easy to use. In this paper we have defined the basic constructs of OEM-QL. We are extending the query language along the lines discussed in Section 3. In addition, we plan to add language constructs and underlying support for data modification operations and for *monitors* (or *active rules*).

We have experimented with OEM and OEM-QL by implementing OEM-based access to several quite different bibliographic information sources. Our implementation so far has served a number of purposes:

- It has helped us refine and ratify our design of the model and query language.
- We have uncovered a number of important issues and generic functionalities in the implementation of OEM-based object exchange. This led to our development of the OEM Support Libraries described in Section 5.
- We have realized a need for browsing tools, leading to the Heterogeneous Information Browser described in Section 4.1.
- We have used a layered architecture for translators and mediators (recall Figure 4), which we believe expedites the integration of heterogeneous information sources.

Implementation is currently underway to incorporate additional bibliographic information sources into our system. We are also implementing a translator for the Sybase relational database system, and a browser based on *Mosaic* and the World Wide Web system. Meanwhile, we are beginning to explore techniques for information mediation using OEM. In Section 4.2.1 we described our initial ideas for mediator generation. We plan to refine these concepts to develop a number of useful mediators that combine bibliographic information from multiple sources. We expect that our powerful but simple object exchange model and query language will provide the appropriate platform for quickly achieving this goal.

Acknowledgements

We are grateful to Ed Chang for implementing the Heterogeneous Information Browser, to Ashish Gupta, Laura Haas, and Dallan Quass for valuable comments, and to the entire Stanford Database Group for numerous fruitful discussions.

References

[1] R. Ahmed et al. The Pegasus heterogeneous multidatabase system. *IEEE Computer*, 24:19–27, 1991.

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

[3] E. Bertino. Integration of heterogeneous data repositories by using object-oriented views. In *Proceedings of the 1st International Workshop on Interoperability in Multidatabase Systems*, pages 22–29, Kyoto, Japan, April 1991.

[4] R. G. G. Cattell. *Object Data Management*. Addison-Wesley, 1991.

[5] R. G. G. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1994.

[6] F. Eliasen and R. Karlsen. Interoperability and object identity. *SIGMOD Record*, 20:25–29, 1991.

[7] A. K. Elmagarmid and A. A. Helal. Hetrogeneous database systems. Technical Report TR-86-004, Program of Computer Engineering, Pennsylvania State University, University Park, PA, 1986.

[8] Object Request Broker Task Force. The Common Object Request Broker: Architecture and Specification, December 1993. Revision 1.2, Draft 29.

[9] A. Gupta. *Integration of Information Systems: Bridging Heterogeneous Databases*. IEEE Press, 1989.

[10] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 59–68, San Diego, California, June 1992.

[11] W. Kim et al. On resolving schematic heterogeneity in multidatabase systems. *Distributed And Parallel Databases*, 1:251–279, 1993.

[12] H. F. Korth and M. A. Roth. Query languages for nested relational databases. In *Nested Relations and Complex Objects in Databases*, pages 190–204. Springer-Verlag, 1989.

[13] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22:267–293, 1990.

[14] L. Mark and N. Roussopoulos. Information interchange between self-describing databases. *IEEE Data Engineering Bulletin*, 10(3):46–52, September 1987.

[15] D. S. Marshak. Lotus Notes release 3. *Workgroup Computing Report*, 16:3–28, 1993.

[16] B. Oki et al. The information bus—an architecture for extensible distributed systems. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 58–68, Asheville, NC, December 1993.

[17] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems*, 13:389–417, 1988.

[18] A. Silberschatz, M. Stonebraker, and J. D. Ullman. Database systems: Achievements and opportunities. *Communications of the ACM*, 34:110–120, 1991.

[19] G. Thomas et al. Heterogeneous distributed database systems for production use. *ACM Computing Surveys*, 22:237–266, 1990.

[20] G. Wiederhold. *File Organization for Database Design*. McGraw Hill, New York, 1987.

[21] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25:38–49, 1992.

[22] G. Wiederhold. Intelligent integration of information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 434–437, Washington, DC, May 1993.

A Appendix

Here we provide a rigorous specification of the query language that was described informally in Section 3. The syntax of the language is given in the grammar of Figure 6. We discuss two points regarding label variables and predicates, then we define the semantics of queries in our language.

The parenthesized variable following each label in a path is optional. However, if a label L does not include a variable, then L is assigned a variable automatically as follows: L's variable is the concatenation of L's position number in its path together with the name of the nearest specified variable to the left of L; if there is no variable to the left of L then L's variable is L's position number. For example, the path "`bibliography.document(d).topic`" becomes "`bibliography(1).document(d).topic(3d)`". This scheme ensures that two labels in different paths have the same variable if and only if they should refer to the same object component. (Recall from Section 3 that two labels L_1 and L_2 refer to the same object component when they appear in paths that are specified identically from the beginning of the path through L_1 and L_2.) "Wild-card" labels (`?`) and "wild-path" labels (`*`) are assigned variables in the same manner.

The predicates that may be specified in a *Condition* are not fixed and may vary from information source to information source, as described in Section 3. Our syntax provides a general notation for arbitrary predicates over multiple arguments. We assume that most information sources support commonly used binary predicates (e.g. equality and inequality over integers), and for convenience we allow these predicates to be written using conventional infix notation, as in the examples of Section 3.

We now define the semantics of an arbitrary query Q. Let O be the object specified in Q's `FROM` clause. We define the semantics in two steps. First we define the set of components of object O that satisfy query Q. Then we define the object A that is returned as the answer to Q. In our definitions we often refer to the label-variable pairs that constitute the paths in Q's `SELECT` and `WHERE` clauses; for brevity we refer to these pairs as *LV*'s.

The first definition formalizes the notion of "path traversals" discussed in Section 3.

Definition A.1 (Valid Binding) A *binding* is a mapping from each LV appearing one or more times in query Q to an object component in O. A binding is *valid* if:

1. Each LV is bound to an object whose label matches the LV's label. If the LV's label is `?` or `*`, then any object label matches.

2. If an LV with a label that is not $*$ appears as the first element on a path, then the LV is bound to object O.

3. Let LV lv_2 follow LV lv_1 on a path. Then lv_1 is bound to an object o of type set or list. If lv_2 does not have label $*$, then lv_2 is bound to a subobject of o. If lv_2 has label $*$, then lv_2 is bound to a direct or indirect subobject of o.

4. For each predicate in Q's `WHERE` clause, if each path appearing in the predicate is replaced by the value of the object bound by the last LV on that path, then the predicate is satisfied. □

Query	::=	SELECT *Fetch-Exp* FROM *Object* WHERE *Condition*
Fetch-Exp	::=	*Path* | *Path*.OID
Path	::=	*Label* | *Label*.*Path*
Label	::=	string [(variable)] | ? [(variable)] | * [(variable)]
Object	::=	string /* lexical object identifier */
Condition	::=	true
	|	*Path*
	|	predicate($Value_1$, $Value_2$, ..., $Value_n$)
	|	$Condition_1$ and $Condition_2$
Value	::=	*Path* | constant

Figure 6: Query language syntax

With this Definition we can specify the object components of O that satisfy query Q.

Definition A.2 An object component o *satisfies* query Q if and only if there is a valid binding such that o is bound to the last LV in the path specified in Q's SELECT clause. □

Next we specify the structure of the answer object A that is returned as a result of query Q. We consider two cases separately: (1) when the path specified in Q's SELECT clause does not end with "OID"; (2) when the path specified in Q's SELECT clause does end with "OID".

Definition A.3 (Answer Object: Non-OID) The answer object for a query Q whose SELECT path does not end with "OID" is:

$\langle$answer, set, $\{obj_1, \ldots, obj_n\}\rangle$
obj_1 is $\langle\ldots\rangle$
...
obj_n is $\langle\ldots\rangle$

where obj_1, ..., obj_n are exactly those object components of O that satisfy query Q according to Definition A.2. □

Now suppose query Q's SELECT path does end with "OID". In this case the answer object includes only the identifiers for the relevant object components, and not the objects themselves.

Definition A.4 (Answer Object: OID) The answer object for a query Q whose SELECT path ends with "OID" is:

$\langle$answer, set, $\{OID_1, \ldots, OID_n\}\rangle$

where OID_1, ..., OID_n are the object identifiers for exactly those object components of O that have non-Λ OID's and satisfy query Q according to Definition A.2 □