Querying Semistructured Heterogeneous Information[⋆]

Dallan Quass[1] Anand Rajaraman[1] Yehoshua Sagiv[2]
Jeffrey Ullman[1] Jennifer Widom[1]

[1] Stanford University
{quass,anand,ullman,widom}@cs.stanford.edu
[2] Hebrew University
sagiv@cs.huji.ac.il

Abstract. Semistructured data has no absolute schema fixed in advance and its structure may be irregular or incomplete. Such data commonly arises in sources that do not impose a rigid structure (such as the World-Wide Web) and when data is combined from several heterogeneous sources. Data models and query languages designed for well structured data are inappropriate in such environments. Starting with a "lightweight" object model adopted for the TSIMMIS project at Stanford, in this paper we describe a query language and object repository designed specifically for semistructured data. Our language provides meaningful query results in cases where conventional models and languages do not: when some data is absent, when data does not have regular structure, when similar concepts are represented using different types, when heterogeneous sets are present, and when object structure is not fully known. This paper motivates the key concepts behind our approach, describes the language through a series of examples (a complete semantics is available in an accompanying technical report [QRS+94]), and describes the basic architecture and query processing strategy of the "lightweight" object repository we have developed.

1 Introduction

An increasing amount of data is becoming available electronically to the casual user, and the data is managed under an increasing diversity of data models and access mechanisms. Much of this data is *semistructured.* By semistructured data we mean data that has no absolute schema fixed in advance, and whose structure may be irregular or incomplete. Two common examples in which semistructured data arise are when data is stored in sources that do not impose a rigid structure (such as the World-Wide Web) and when data is combined from several heterogeneous data sources (especially when new sources are frequently added).

This paper describes a query language and data repository we have developed specifically for semistructured data. An important feature of our language is that it exploits structure when it is present, but it does not require uniform structure for meaningful answers. Our language supports objects and object

[⋆] This work was supported by ARPA Contract F33615-93-1-1339, by the Anderson Faculty Scholar Fund, and by equipment grants from Digital Equipment Corporation and IBM Corporation.

relationships. However, in contrast to most object-oriented query languages, we use a very simple "lightweight" object model with only a few concepts, resulting in a "lightweight" query language that we believe is comfortable for the casual user. The following are highlights of our language.

- Queries return meaningful results even when some data is absent (Section 2.1).
- Queries operate uniformly over single- and set-valued attributes (Section 2.2).
- Queries operate uniformly over data having different types (Section 2.3).
- Queries can return heterogeneous sets, i.e., where objects in the query result have different types and structures (Section 2.4).
- Meaningful queries are possible even when the object structure is not fully known (Section 2.5).
- The query language syntax is similar in spirit to SQL. For example, our treatment of range variables generalizes SQL's approach (Section 2.6).

Our language has been developed as the unifying query language for the TSIMMIS[3] project at Stanford [CGMH$^+$94, PGMW95, PGMU95]. The goal of the TSIMMIS project is to provide a framework and tools for integrating and accessing data from multiple, heterogeneous data sources. We describe the TSIMMIS data model briefly, only to the extent it is necessary to understand the query language. A complete description of the data model and its benefits is given in [PGMW95]. The architecture of TSIMMIS and how it relates to the work presented here is further discussed in Section 6.

In addition to our query language, this paper describes an object repository, LORE (Lightweight Object REpository),[4] that supports our data model and query language. We have developed LORE not only as a proof-of-concept, but also because there are some interesting aspects to the implementation of a repository for semistructured data. In addition, the repository is a useful tool: LORE will be used in several ways within the TSIMMIS framework (see Section 6). Because LORE implements our query language, we have named our language LOREL, for LORE Language.

1.1 Outline of Paper

Section 2 highlights our reasons for developing a new query language, and specifically compares LOREL to three conventional query languages: OQL [Cat94], XSQL [KKS92], and SQL [MS93]. Other related work appears in Section 3. Section 4 describes the data model upon which LOREL is based. An exposition of the novel features of LOREL using a series of examples appears in Section 5. Section 5 also includes an informal description of the semantics of LOREL. Section 6 describes the LORE object repository and provides an overview of how queries are executed. Conclusions and future work are given in Section 7. We

[3] As an acronym, TSIMMIS stands for "The Stanford-IBM Manager of Multiple Information Sources." In addition, Tsimmis is a Yiddish word for a stew with "heterogeneous" fruits and vegetables integrated into a surprisingly tasty whole.

[4] Also Data's sinister elder brother, to Star Trek fans.

have written a complete denotational semantics [Sto77] for LOREL (which, incidentally, was very helpful as it helped uncover anomalies that might otherwise have gone unnoticed). Due to space limitations, we have not included the denotational semantics or the syntax in this paper; they are included in the full version, which is available by anonymous ftp [QRS+94].

2 Motivation and Comparison

In this section we motivate the need for a new query language by presenting several issues that must be addressed when querying semistructured data, and by showing how existing query languages are inadequate to address these issues. We describe our solutions to these issues briefly here, with further details given in Section 5.

We realize that (too) many query languages already exist. However, rather than choose an existing language for our task, we have chosen to develop a new one. The requirements for querying semistructured data are sufficiently different from traditional requirements that we feel a new language is justified. Recall that by semistructured, we mean that there is no schema fixed in advance, and the structure may be irregular or incomplete. Hence, query languages over semistructured data must uniformly handle data that is absent, data that does not conform to a regular structure, and data where the same concepts are represented using different types. Conventional query languages are designed primarily to access well structured data whose schema is known. Furthermore, object-oriented query languages focus especially on facilitating programmer access, supporting different kinds of built-in and extensible object structures and methods. We term such languages (and their underlying models) *heavyweight*, in that they expect data to conform to a regular structure, they enforce strong typing in queries, they provide different ways of dealing with sets, arrays, and record object structures, and they include other features important for queries embedded in programs but perhaps too strong for querying semistructured data.

In contrast, LOREL is a *lightweight* object query language aimed specifically at querying semistructured data. We compare LOREL with OQL, XSQL, and SQL (SQL2 specifically), which we feel are representative of the types of heavyweight query languages in existence. Several issues we use for comparison are summarized in Table 1. Although our data model is described in more detail in Section 6, we note here that all data, even scalar values, are represented as objects. Each object has a unique *identifier*, a (textual) *label*, and a *value*. The value is either an element of a scalar type, or a set of subobjects.

2.1 Coping with the Absence of Data

When querying semistructured data, it is important to produce intuitive results even when some of the data is absent. The reader may be aware of a bug/feature in SQL regarding the way disjunction (`OR`) is handled in `WHERE` clauses. Suppose we have three unary relations R, S, and T, and we wish to compute $R \cap (S \cup T)$. If each of these relations has a single attribute A, we might expect the following SQL query to do the trick.

	LOREL	OQL	XSQL	SQL
tuple/ object assignment	partial	total	total	total (outerjoins and null-valued attributes allowed)
treatment of single- versus set-valued attributes	uniform treatment through implicit existential quantification	different syntax (`y.x` versus `v in y.x`)	single-valued and set-valued path expressions treated differently	N/A
type checking	none	strong	several options discussed	explicit casts required in several cases
type of value returned	heterogeneous set of objects	an object or literal, which may be a homogeneous set	homogeneous set of objects, or a relation	relation
wildcards	over attribute labels	none	over attribute labels	over attributes of a single relation in a select clause
range variables	implicit	explicit	explicit	implicit

Table 1. Differences between LOREL and other query languages

```
SELECT R.A
FROM   R, S, T
WHERE  R.A = S.A or R.A = T.A
```

Unfortunately, if T is empty, the result is empty, even if there are elements in $R \cap S$. The reason is that SQL semantics is defined in terms of a cross product of all the relation names and tuple variables that appear in the `FROM` clause, which is equivalent to requiring a *total assignment* of tuples to the three relations R, S, and T mentioned in the `FROM` clause. If T (or S) is empty, we cannot find a total assignment (equivalently, the cross product is empty), and thus there is no way to produce an answer. The problem of absent data is addressed in SQL through the introduction of outerjoins and nulls. It is well accepted that outerjoins and nulls are difficult for the casual user to use correctly [MS93]: outerjoins are not always associative, and nulls require a three-valued logic.

An alternative approach is to use a *partial assignment* of tuples. For example, if T is empty, but R and S each contain the tuple (0), we can assign (0) to both R and S, assign nothing to T, and find that the `WHERE` condition is satisfied since $R.A = S.A$.

Total assignments are required in SQL, XSQL, and OQL. Total assignments are not generally a problem in conventional query languages because there is unlikely to be an empty relation or object set in a conventional database. However, such a situation is more likely with semistructured data. For this reason, and because outerjoins and nulls are problematic, LOREL adopts the partial assignment approach (see Section 5.4 for details).

2.2 Queries Over Single- and Set-Valued Attributes

Suppose that in a library database, the authors of each book appear as a set-valued attribute, and a name is associated with each author. The following OQL query fetches the titles of all books written by Samuel Clemens.

```
SELECT b.Title
FROM   b in Library,
       a in b.Authors
WHERE  a.Name = "Samuel Clemens"
```

This query works correctly as long as author objects in the database conform to a regular structure. But now suppose that we add some books that associate a set of names with each author, e.g., the author's pen names as well as his or her real name.[5] Accommodating these new books in an OQL environment would require changing the overall schema, and the query above would no longer work correctly.

Other conventional query languages also treat single- and set-valued attributes differently. In SQL, all attributes must be single-valued. In XSQL, path expressions resulting in set values require explicit quantifiers when used in predicates, and they cannot always appear in select clauses [KKS92].

LOREL treats single- and set-valued attributes uniformly. When attributes in path expressions are found to be set-valued, an implicit existential quantifier is assumed. For example, in LOREL the path expression in the predicate `Book.Author.Name = "Samuel Clemens"` matches any path from a `Book` object through an `Author` object to a `Name` object whose value is "Samuel Clemens" (see Section 5.1 for details).[6] If one wants to treat path expressions resulting in set values as sets, e.g., for aggregation or universal quantification, LOREL provides additional constructs for this purpose (Section 5.5). By treating single- and set-valued attributes in a compatible manner, data can have more varied structure, and the client need not have detailed knowledge about the structure in order to pose meaningful queries.

2.3 Queries Over Objects Having Different Types

Differences between single- and set-valued attributes is just one way in which structure may vary in semistructured data; another way is with regard to type. Query languages over semistructured data must have very relaxed type checking, if they perform type checking at all. Consider an OQL query to select all publishers who have published Computer Science textbooks in 1995.

[5] Samuel Clemens used the pen name Mark Twain.

[6] In our data model, `Book`, `Author`, and `Name` are object labels, and dot notation indicates subobject relationships. Details are in Section 5.2.

```
SELECT b.Publisher
FROM   b in Library
WHERE  b.Subject = "Computer Science" AND b.year-published = "1995"
```

In a semistructured environment, `b.year-published` may result in a string value for some books, while it results in a numeric value for others. This situation is not allowed in OQL, since OQL requires strong type checking. SQL does implicit casts between different data types in some situations, but requires explicit casts to convert strings to integers, and returns an error if a string does not have the correct format for conversion. XSQL proposes several possible approaches to the issue of type checking. In LOREL, we always attempt to convert the operands of a predicate to comparable types. If the operands cannot be converted to comparable types, rather than return an error, the predicate simply returns false (see Section 5.3). While this approach may allow "ill-conceived" queries, we feel that it is a reasonable approach for handling data that does not all conform to the same type. In the future we will investigate incorporating limited type checking in cases where it would be helpful.

2.4 Returning Complex Objects and Heterogeneous Sets

Another case where query languages for semistructured data must allow for objects with different types is in query results. Consider the following OQL query to find the publishers of all books written by Samuel Clemens.

```
SELECT b.Publisher
FROM   b in Library,
       a in b.Authors
WHERE  a.Name = "Samuel Clemens"
```

If for some books the publisher is represented as a string but for others it is represented as a complex object with individual attributes for name and address, then this query would return a heterogeneous set. Neither OQL, XSQL, nor SQL allow query results to be heterogeneous sets. In LOREL, all objects, including query results, are modeled as heterogeneous sets (see Section 5.8). LOREL can query over heterogeneous sets as well as return heterogeneous sets. Note that heterogeneous sets are a powerful concept, since with them it is possible to model both tuple structures and homogeneous sets.

2.5 Queries When Object Structure is Not Fully Known

With semistructured data, it is unlikely that the exact structure of the data will be known by all clients who wish to query it. SQL partially addresses this issue by allowing clients to query the system catalogs to learn about tables and attributes, but clients can only discover limited structure since the system catalogs do not contain information on how data in different tables is related. OQL does not provide a way to query object structure. XSQL addresses this issue by allowing path expressions to contain wildcards and by allowing certain query variables to range over attribute names.

LOREL is similar to XSQL and an extension to O_2 [CACS94] in that we allow path expressions to contain wildcards and we allow queries to return attribute labels. Path expressions containing wildcards are useful when part, but not all, of the structure of the data is known. For example, suppose one knows that a `Library` object contains `Book` objects, but one is unsure of the structure within book objects. In an attempt to find all books authored by "Samuel Clemens," a LOREL query could contain the predicate `Library.Book.* = "Samuel Clemens"`, which matches any sequence of objects beginning with a `Library` object, through a `Book` object, through zero or more other objects,[7] and finally ending in an object whose value is "Samuel Clemens." Wildcards can also be useful when the exact object structure is known, but it varies among the objects in the database.

The above predicate might also match books not written by Samuel Clemens, such as books whose title is "Samuel Clemens," but once the client becomes more familiar with the structure, a more specific query can be written. To facilitate exploring and posing queries about structure, LOREL provides the built-in functions `PATHOF()`, `LABELOF()`, and `TYPEOF()`. These functions can be applied to path expressions in queries to return a concatenation of the labels of all objects in the sequence matching the path expression, the label of just the last object in the sequence matching the path expression, and the type of the last object in the sequence matching the path expression, respectively (see Section 5.7).

2.6 Absence of Range Variables

LOREL does not require the introduction of range variables for specifying that different path expressions in a query should match the same element of a set, as used in OQL and XSQL. For example, in the OQL query of Section 2.3, the variable `b` had to be introduced to specify that both predicates should be satisfied by the same book object. In LOREL, path expressions that begin with the same sequence of labels by default match the same sequence of objects up to the point where the label sequences diverge. We feel that this default provides the natural behavior in most cases, and we provide an easy way for the client to override the default when desired (see Section 5.6). The absence of range variables makes LOREL similar in spirit to SQL.

3 Other Related Work

Several articles have pointed out the need for new data models and query languages to integrate heterogeneous data sources, e.g., [LMR90, Qia93]. However, most of the research in heterogeneous database integration has focused on integrating data in well structured databases. In particular, systems such as Pegasus [RAK+92] and UniSQL/M [Kim94] are designed to integrate data in object-oriented and relational databases. At the other end of the spectrum, systems such as GAIA [RJR94], Willow [Fre94], and ACL/KIF [GF94] provide uniform access to data with minimal structure.

[7] To handle cyclic data, the length of object sequences matching a "*" would in practice be limited to a constant.

The goal of the TSIMMIS project is to uniformly handle unstructured, semi-structured, and well structured data [PGMW95]. In this goal our effort is similar to the work on integrating SGML [ISO86] documents with relational databases [BCK$^+$94] or integrating SGML documents with object-oriented databases such as OpenODB [YA94] or O_2 [CACS94]. These approaches tend to extend existing data models and languages [BCD92, F$^+$89]. The ideas behind LOREL could instead have been used to extend an existing language. Our choice to design a new language has its advantages and disadvantages, of course. A disadvantage is that we are unable to manage our objects using an existing DBMS. An advantage is that we do not have to work around the limitations of a data model and language designed originally for querying well structured data with a fixed schema. Another language designed for the TSIMMIS project, described in [PGMU95], is used for mediator specification. In contrast, LOREL is intended for inter-component communication in TSIMMIS and for the end user.

Environments such as CORBA [OMG92] and OLE2 [Mic94] operate at a different level from TSIMMIS and LOREL. These approaches provide a common protocol for passing messages between objects in a distributed object environment. In contrast, TSIMMIS and LOREL provide a common data model and query language. Our approach could easily be built on top of and take advantage of environments such as CORBA and OLE2.

We have already shown how LOREL compares to OQL, XSQL, and SQL. LOREL relates in similar ways to a number of other query languages for object-oriented [BCD92, CDV88, Har94] and nested relational [DKA$^+$86] systems. A final important difference between LOREL and these query languages is that the simplicity of our object model yields many fewer concepts in the query language, resulting in a language that we believe is more appropriate for the casual user.

4 Data Model

In the TSIMMIS project we have developed a simple data model called OEM (for Object Exchange Model) [PGMW95], based essentially on tagged values. Every object in our model has an *identifier*, a *label*, and a *value*. The *identifier* uniquely identifies the object among all objects in the domain of interest. The *label* is a string (the tag) presumably denoting the "meaning" of the object. Labels may be used to group objects by assigning the same label to related objects. The *value* can be of a scalar type, such as integer or string, or it can be a set of (sub)objects. We define *atomic objects* as objects with scalar values, and *complex objects* as objects whose values are sets of subobjects. Note that due to the simplicity of our model, even immutable values such as numbers are represented as values of distinct objects.

An object is thus a 3-tuple:

$\langle$*identifier*, *label*, *value*$\rangle$

A *database* $D = \langle O, N \rangle$ is a set O of objects, a subset N of which are *named* (or *top-level*) objects. The intuition is that named objects provide "entry points" into the database from which subobjects can be requested and explored. To ensure that named objects can be specified uniquely when writing queries, we require that the labels of named objects be unique within a given database. We shall use *label*(o), *value*(o), and *identifier*(o) to denote the label, value, and identifier, respectively, of an object o.

Figure 1 shows a segment of an entertainment database. This structure is typical of the semistructured data that is available on, e.g., the World-Wide Web.[8] In the figure, indentation is used to represent subobject relationships. Each object appears on a separate line, with its identifier inside brackets at the far left, followed by its label, followed by its value if the value is a scalar. Complex values are represented by indenting the subobject labels underneath the parent object. Hence, this database contains a single top-level object labeled `Frodos`. `Frodos` is a complex object with three subobjects, one having label `Restaurant`, and two having label `Group`. Although a real-world entertainment database would of course be much, much larger, this example concisely captures the sort of structure (or lack thereof) needed to illustrate the features of our language. For example, the performance dates and ticket prices for the Palo Alto Savoyards are absent, the Savoyards perform only a single work per performance as opposed to (possibly) multiple works performed by the Peninsula Philharmonic, prices of restaurant entrees are of strings while prices of performing group tickets are of integers, and the work listed for the second performance of the Peninsula Philharmonic is a string rather than a complex object with title and composer subobjects.

5 The Language

In this section we describe our language (LOREL), primarily through a series of examples. In Section 5.1, we present a simple LOREL query and explain intuitively what it does. Section 5.2 introduces the basic concepts needed to understand the semantics of LOREL queries. Section 5.3 presents some further LOREL examples. Section 5.4 explains the use of boolean connectives (`AND` and `OR`) in queries. Sections 5.5 through 5.8 then discuss more advanced features of LOREL, including subqueries and correlation, schema browsing, and complex query results. The complete LOREL syntax and denotational semantics are given in the extended version of this paper [QRS$^+$94]. All of the example queries in this section refer to the database (fragment) in Figure 1.

5.1 An Introductory Query

Suppose we wish to find the names of all opera groups. We issue the following query:

```
SELECT Frodos.Group.Name
FROM   Frodos
WHERE  Frodos.Group.Category = "Opera"                 (1)
```

Recall that `Frodos` is the label of a unique named object in the database of Figure 1. This query finds all `Group` subobjects of the the `Frodos` object that

[8] For example, the URL `http://gsb.stanford.edu/goodlife` presents a database of semistructured restaurant information.

```
[1]   Frodos
[2]       Restaurant
[3]           Name "Blues on the Bay"
[4]           Category "Vegetarian"
[5]           Entree
[6]               Name "Black bean soup"
[7]               Price "10.00"
[8]           Entree
[9]               Name "Asparagus Timbale"
[10]              Price "22.50"
[11]          Location
[12]              Street "1890 Wharf Ave."
[13]              City "San Francisco"
[14]      Group
[15]          Name "Peninsula Philharmonic"
[16]          Category "Symphony"
[17]          Performance
[18]              Date "3/12/95"
[19]              Date "3/19/95"
[20]              Date "3/26/95"
[21]              Work
[22]                  Title "Eine Kleine Nachtmusik"
[23]                  Composer "Mozart"
[24]              Work
[25]                  Title "Toccata and Fugue in D minor"
[26]                  Composer "Bach"
[27]          Performance
[28]              Date "12/20/95"
[29]              Work "Seasonal selections to be announced"
[30]          TicketPrice
[31]              AgeGroup "Adults"
[32]              Price 15
[33]          TicketPrice
[34]              AgeGroup "Students"
[35]              Price 8
[36]          Location
[37]              Street "100 Middlefield Ave."
[38]              City "Palo Alto"
[39]              Phone "415-777-5678"
[40]      Group
[41]          Name "Palo Alto Savoyards"
[42]          Category "Opera"
[43]          Performance
[44]              Work
[45]                  Title "The Yeoman of the Guard"
[46]                  Composer "Gilbert"
[47]                  Composer "Sullivan"
[48]          Location
[49]              Street "101 University Ave."
[50]              City "Palo Alto"
[51]              Phone "415-666-9876"
```

Fig. 1. Frodo's Guide to Good Living in the Bay Area

contain a `Category` subobject whose value is `"Opera"`. The query returns a set that contains copies of the `Name` subobjects of all such `Group` objects. The result of Query (1) looks like this:

```
[60]  Answer
[61]        Name "Palo Alto Savoyards"
```

The result set is "packaged" inside a single complex object with the default label `Answer`. (This default label can be overridden; see Section 5.8.) In this case, the result set is a singleton set, but in general it can contain more than one object. The `Answer` object becomes a new named object of the database. Packaging the result set in a new object has the advantage that the result of a query can be treated as new data, i.e., it can be browsed or queried using the same mechanisms that are used on the database.

5.2 Semantics of Simple Queries

This section provides an informal overview of the semantic concepts underlying LOREL, with just enough detail (we hope) for the reader to understand the remainder of the paper. For a complete formal treatment of this material the reader is referred to [QRS+94].

Path Expressions and Object Assignments *Path expressions* form the basis of LOREL queries. A path expression is a sequence of labels separated by dots. Query (1) above contains two path expressions: one (`Frodos.Group.Name`) in the `SELECT` clause, and one (`Frodos.Group.Category`) in the `WHERE` clause. Path expressions describe paths through the object structure (called *database paths*, or simply *paths*), by specifying the labels of the objects along the paths. For example, the path expression `Frodos.Group.Name` "matches" every database path consisting of a sequence of three objects, $\langle o_1, o_2, o_3\rangle$, such that

- $label(o_1) = \texttt{Frodos}$, $label(o_2) = \texttt{Group}$, and $label(o_3) = \texttt{Name}$; and
- o_1 and o_2 are complex objects such that $o_2 \in value(o_1)$ and $o_3 \in value(o_2)$; o_3 can be either atomic or complex.

There are two paths in the database of Figure 1 that match `Frodos.Group.Name`: $\langle[\texttt{1}], [\texttt{14}], [\texttt{15}]\rangle$ and $\langle[\texttt{1}], [\texttt{40}], [\texttt{41}]\rangle$.

The result of a query is based on matching its path expressions with database paths. When matching the two path expressions in Query (1), both database paths in a match must contain the same `Frodos` and `Group` objects. (Intuitively, common prefixes of path expressions must match the same database paths, as discussed in Section 2.6.) For example, one of the two possible matches for Query 1 is:

$$\begin{array}{ll} \texttt{Frodos.Group.Name} & \rightarrow \langle[1], [14], [15]\rangle \\ \texttt{Frodos.Group.Category} & \rightarrow \langle[1], [14], [16]\rangle \end{array}$$

The pair of matching paths above also corresponds to a mapping from all the prefixes of path expressions appearing in Query (1) to database objects:

```
Frodos                 → [1]
Frodos.Group           → [14]
Frodos.Group.Name      → [15]
Frodos.Group.Category  → [16]
```

We call such a mapping from path expression prefixes to objects an *object assignment.*

The `FROM` Clause The `FROM` clause contains a list of labels of named objects, specifying that only database paths that begin with these objects should be considered. In the absence of wildcards (Section 5.7), the `FROM` clause is optional and redundant, because path expressions must each begin with one of the objects mentioned in the `FROM` clause. We omit `FROM` in most of our example queries.

The `WHERE` Clause Given an object assignment that maps some path expression in the `WHERE` clause of a query to an object o,[9] the *value* of the path expression is either

– the value of o if o is an atomic object, or
– the identifier of o if o is a complex object.

Hence the language treats path expressions differently depending on whether an object is atomic or complex. This approach is needed because, in our semistructured environment, data may contain both atomic and complex objects with the same label.

Now, suppose we have an object assignment for some or all of the path expressions that appear in the `WHERE` clause of a query. We evaluate the `WHERE` condition in the conventional manner: replace each path expression by its value and then evaluate the expression following the `WHERE`. It is important to note that there are times when we do not need a total object assignment in order to evaluate the `WHERE` clause. In particular, when the `WHERE` clause is the `OR` of two expressions, it is not necessary to assign objects to path expressions on both sides of the `OR`. As discussed in Section 2.1, this point distinguishes LOREL from other languages, and is essential for querying in a semistructured environment. We shall have more to say about partial object assignments in Section 5.4.

The `SELECT` Clause A partial object assignment for a query is *successful* if it satisfies the `WHERE` clause as explained above. The result set of the query contains copies of all the objects that are matched with the path expression in the `SELECT` clause by a successful object assignment. All objects in the result set are made subobjects of a new named object with the label `Answer`.

Notice that the result set can in general be a heterogeneous set, since neither our data model nor our language requires that a path expression map to objects of a single type. Heterogeneous result sets also arise when the `SELECT` clause contains more than one path expression (Section 5.8).

[9] Note that each path expression is also a path expression prefix.

Relationship to SQL Semantics Although the semantics of SQL is usually defined in terms of a cross product of the relations mentioned in the `FROM` clause, it can easily (and equivalently) be defined in terms of mappings from the relation names and tuple variables that appear in the `FROM` clause to actual database tuples. When SQL semantics is defined in this way, there is a clear correspondence between the LOREL concepts we have seen so far and SQL concepts, as shown in Table 2.

SQL	LOREL
Relation name or tuple variable	Path expression prefix
Database tuple	Database object
(Total) tuple assignment	(Partial) object assignment

Table 2. Relationship between SQL and LOREL concepts

5.3 Additional Simple Queries

The result of Query (1) is a set of atomic objects. The path expression in a `SELECT` clause can also match complex objects, as in the following variant of Query (1):

```
SELECT Frodos.Group
WHERE  Frodos.Group.Category = "Opera"                          (2)
```

The result of this query on our example database is:

```
[62] Answer
[63]     Group
[64]         Name “Palo Alto Savoyards”
[65]         Category “Opera”
[66]         Performance
[67]             Work
[68]                 Title “The Yeoman of the Guard”
[69]                 Composer “Gilbert”
[70]                 Composer “Sullivan”
[71]         Location
[72]             Street “101 University Ave.”
[73]             City “Palo Alto”
[74]             Phone “415-666-9876”
```

Operators that can be used in the `WHERE` clause include the familiar =, <, >, <=, >=, and ! =. Path expressions can be compared with other path expressions, rather than constants, as the following example demonstrates.

```
SELECT Frodos.Group.Performance.Work.Title
WHERE  Frodos.Group.Performance.Work.Title =
       Frodos.Group.Performance.Work.Composer                   (3)
```

This query returns the titles of all performances where the title is the same as one of the composers. The result set of Query (3) will contain titles more than once if there are pieces that are performed several times. As in SQL, `SELECT DISTINCT` eliminates duplicates.

Query (3) appears rather cumbersome, since the same path expression prefix is repeated three times. LOREL permits an abbreviation so that common prefixes can be written only once. Query (3) is abbreviated to:

```
SELECT Frodos.Group.Performance.Work:W.Title
WHERE  W.Title = W.Composer                                    (4)
```

Every occurrence of `W` after the first expands to the path expression prefix with which `W` is associated.

It is not a type error in LOREL to compare objects of different types, or to use a comparison operator that is not defined for a given type; such comparisons merely return *false*. Thus, if in the future we had computers authoring music, some `Work.Composer` values might contain numbers (the Internet address of the computer) while others contain strings (for human composers). There could also be pieces without any composers. In all of these cases, Query (3) would still be legal. This absence of typing in queries is a powerful and, we feel, a necessary feature for querying semistructured data.

Path expressions can be used without any comparison operators to produce "existential" queries. For example, suppose we are interested only in works performed by groups whose ticket price is known in advance. We use the query:

```
SELECT Frodos.Group.Performance.Work
WHERE  Frodos.Group.TicketPrice                                (5)
```

The result of Query (5) is a heterogeneous set, since it contains complex `Work` objects (with `Title` and `Composer` attributes), as well as a `Work` object of type string. The result of the query is:

```
[75]  Answer
[76]      Work
[77]          Title "Eine Kleine Nachtmusik"
[78]          Composer "Mozart"
[79]      Work
[80]          Title "Toccata and Fugue in D Minor"
[81]          Composer "Bach"
[82]      Work "Seasonal selections to be announced"
```

Path expressions can be arguments to *external predicates* as well. Suppose we have a predicate `isInCounty` that accepts two strings—a city and a county—and returns *true* if the city is in the given county. Then the query:

```
SELECT Frodos.Group.Name
WHERE  isInCounty(Frodos.Group.Location.City, "Santa Clara")      (6)
```

returns the names of all groups in Santa Clara County. LOREL supports external functions as well as external predicates. External functions and predicates are most useful when using LOREL in the TSIMMIS context, where the functions and predicates would be supported by an underlying information source; see Section 6.

5.4 Boolean Connectives

Conditions in the `WHERE` clause of a query can be combined using the connectives `AND` and `OR`. Conjunctions (conditions involving `AND`) are handled in the usual manner. Disjunctions (conditions involving `OR`) are more subtle. We might be tempted to say that an object assignment succeeds for a condition with an `OR` if at least one of the disjuncts is satisfied. But consider the following query:

```
SELECT Frodos.Group.Name
WHERE  Frodos.Group.Category = "Opera" OR
       Frodos.Group.Performance.Date = "3/19/95"                (7)
```

Presumably, the query is intended to find the names of all groups such that either the group is an opera group or it performs on 3/19/95. Looking at Figure 1, we would intuitively expect that since Palo Alto Savoyards is an opera group, their name should be in the result of the query. However, no date is specified for any performance by the Savoyards. Thus, there is no total object assignment that would put the Savoyards in the result set. As motivated earlier, LOREL semantics is defined in terms of partial object assignments. When evaluating the `WHERE` condition with partial object assignments, if some path expression involved in an atomic condition (such as a comparison) is not mapped, then the condition evaluates to *false*. As usual, a condition involving an `OR` evaluates to *true* if at least one of the conditions connected by the `OR` evaluates to *true*. Hence, the result of Query (7) will include the Palo Alto Savoyards.

5.5 Subqueries and Correlation

So far, conditions in the `WHERE` clause involving path expressions have used implicit existential quantification over sets. For example, in Query (5) the `WHERE` clause is satisfied if *there exists* a path with successive objects labeled `Frodos`, `Group` and `TicketPrice`. Subqueries enable universal quantification over all objects in a set. For example, the following query finds the names of restaurants whose entrees all cost less than $10.

```
SELECT Frodos.Restaurant.Name
WHERE  Frodos.Restaurant SATISFIES
       10 > ALL (SELECT Frodos.Restaurant.Entree.Price)         (8)
```

We extend the semantics of simple queries given in Section 5.2 as follows. For every (partial) object assignment to the the top-level query (but not the subquery), evaluate the subquery with the restriction that the path expression `Frodos.Restaurant` (the path expression preceding the keyword `SATISFIES`) already has its mappings fixed by the object assignment for the enclosing query. The subquery returns a set of objects, whose values form the set for evaluating the `WHERE` clause.

In Query (8) the subquery is evaluated for every restaurant in the database. The subquery produces the set of entree prices for the restaurant. Only restaurants all of whose entrees cost less than $10 will satisfy the condition in the `WHERE` clause and will therefore have their names in the result. Query (8) contains a subquery with *correlation*: the path expression `Frodos.Restaurant` preceding the keyword `SATISFIES` links together each evaluation of the subquery with the rest of the path expressions in the enclosing query. Note that for efficiency, the subquery could be evaluated just once with the result set then grouped by the object assignment for `Frodos.Restaurant`.

Any binary operator can be converted into an operator for comparing a single value and a set by appending one of the modifiers `ALL` or `ANY`, for example, $<$ `ANY` or $\neq$ `ALL`. Two other mixed set/value operators are `IN` and `NOT IN`, which are used to test for set membership.[10] Two sets can be compared using the `CONTAINS` and `SETEQUAL` operators. More than one path expression can precede the keyword `SATISFIES` (for more than one correlation with the subquery), and the condition following `SATISFIES` can be arbitrarily complex. The full version of this paper [QRS+94] describes how the semantics described above generalizes naturally in these cases.

Subqueries can also be used as operands to the *aggregation operators* `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX`. The following query finds the names of restaurants that offer more than seven entrees priced $10 or less:

```
SELECT Frodos.Restaurant.Name
WHERE  Frodos.Restaurant SATISFIES
       7 < COUNT (SELECT Frodos.Restaurant.Entree
                  WHERE Frodos.Restaurant.Entree.Price <= 10)     (9)
```

Aggregation operators can also appear in the `SELECT` clause; see [QRS+94].

5.6 Label Distinguishers

Sometimes it is necessary to distinguish among prefixes in path expressions that otherwise would be forced to match the same database paths. We do so by appending to a label a colon and a *label distinguisher*. Label distinguishers make it possible to express queries that could not otherwise be expressed in LOREL. For example, the query

```
SELECT Frodos.Group.Performance.Work.Title
WHERE  Frodos.Group.Performance.Work.Composer:A = "Gilbert" AND
       Frodos.Group.Performance.Work.Composer:B = "Sullivan"    (10)
```

selects the titles of all performances of the works of Gilbert and Sullivan.[11] Label distinguishers were actually introduced in Section 5.3, where they were used to avoid repeating path expression prefixes in a query; that abbreviation is an additional function of label distinguishers.

5.7 Wildcards and Schema Browsing

One of the most important requisites for querying in a semistructured environment is adequate capabilities for browsing and discovering object structure. Our data model does not require that data be structured according to a schema fixed in advance; however, in most cases we do expect some common structure

[10] `IN` and `NOT IN` have exactly the same functionality as = `ANY` and $\neq$ `ALL`, respectively. All these constructs have a direct analogy in SQL.

[11] We realize that Gilbert was a librettist, but we refer to him as a composer for simplicity.

to the data (which we shall hereafter call "schema" for convenience). LOREL provides mechanisms for schema discovery, as well as the ability to pose queries with incomplete information, by the use of the wildcards "$*$" and "?" in path expressions, and by providing convenient operators to summarize the results of such queries.

The wildcards $*$ and ? may be used anywhere in a path expression that a label can appear. The $*$ stands for any sequence of zero or more labels, while the ? stands for any single label. Occurrences of $*$ (respectively, ?) in different path expressions where the $*$'s (?'s) are preceded by the same path expression prefix are assumed to stand for the same sequence of labels (single label), unless one or both occurrences are modified by a label distinguisher.

As an example, suppose we are interested in restaurants that are located in the city of Palo Alto. We might reasonably assume that most `Restaurant` objects will contain the city in which they are located, but we might not know at what level of the object hierarchy the city would appear for different restaurants. The following query solves our problem:

```
SELECT Frodos.Restaurant.Name
WHERE  Frodos.Restaurant.*.City = "Palo Alto"                    (11)
```

The wildcard feature is very useful for forming queries when one has incomplete knowledge about the structure of the underlying data, as well as for succinctly expressing queries when the structure of the underlying data is known but is highly heterogeneous. Our absence of type checking allows queries containing wildcards that would not be considered legal in strongly typed languages. Wildcards can occur in the `SELECT` clause as well as in the `WHERE` clause. Wildcards in the `SELECT` clause may result in heterogeneous result sets, but as we have already seen, queries returning heterogeneous sets are legal in LOREL.

For querying object structure, the built-in operator `PATHOF` takes a path expression (which may contain wildcards) as its argument and produces a string that describes a matching path in terms of its label structure (e.g., `"Frodos.Restaurant.Name"`). The operator `LABELOF` is similar, but produces a string that corresponds only to the label of the last object in the path (e.g., `"Name"`). `LABELOF` and `PATHOF` can be used with the `DISTINCT` operator for schema browsing and discovery. For example, the query:

```
SELECT DISTINCT PATHOF(Frodos.*)                                 (12)
```

returns all possible sequences of labels in the `Frodos` database and can be used to get a feel for the overall structure of the database. The `TYPEOF` operator returns the type of the last object in a path (e.g., `"Integer"`, `"String"`, or `"Complex"`). In general, there could be more than one type that is associated with a path expression, and in such cases the `TYPEOF` operator returns all of them. Finally, the built-in predicates `ISATOMIC` and `ISCOMPLEX` test whether the last object in a path is atomic or complex, respectively.

5.8 Creating Complex Object Structures

Until now, the `SELECT` clause in our queries has contained just one path expression. In general, a `SELECT` clause can contain a list of path expressions. For example, in the query:

```
SELECT AS LocalGroups
       Frodos.Group.Name, Frodos.Group.*.Phone
WHERE  Frodos.Group.Location.City = "Palo Alto"                (13)
```

the result set contains both `Name` and `Phone` objects for every group in Palo Alto. Query (13) introduces another LOREL feature: the label of the query result can be changed using the optional `AS` clause (recall that the default label was `Answer`). In Query (13), the result is labeled `LocalGroups` instead of `Answer`.

The result of Query (13) actually may not be very useful if the result set contains several names and phone numbers, because there is no way of telling which phone number goes with which name. We can solve this problem by having the result set contain, for every group, a complex object whose subobjects have the name and the phone number(s) for that group. This result is achieved by using subqueries with correlation in the `SELECT` clause. The following query is issued:

```
SELECT AS LocalGroups
       FOREACH Frodos.Group {
           (SELECT Frodos.Group.Name),
           (SELECT Frodos.Group.*.Phone)
       }
WHERE  Frodos.Group.Location.City = "Palo Alto"                (14)
```

The result of this query applied to our Frodo's database is:

```
[83]  LocalGroups
[84]       Group
[85]            Name “Peninsula Philharmonic”
[86]            Phone “415-777-5678”
[87]       Group
[88]            Name “Palo Alto Savoyards”
[89]            Phone “415-666-9876”
```

The semantics of Query (14) can be understood as follows. Take all successful object assignments and group them according to the object to which they map the path expression `Frodos.Group` (the path expression after the keyword `FOREACH`). Select one such group of object assignments. For each object assignment in the group, evaluate the two subqueries with the restriction that path expression `Frodos.Group` has already been mapped by the object assignment that was fixed. Collect together all the resulting objects into a set, and package this set in a new complex object labeled `Group` (the last label in the `FOREACH` path expression). Repeat the above process for each group of path expressions, and collect together all the `Group` objects that result to form the result set of the query.

Note that performing groups with no phone numbers will have no `Phone` subobject in the result, and groups with more than one phone number will have more than one `Phone` subobject. LOREL handles such cases more gracefully than other query languages.

As a convenient abbreviation, a path expression by itself in a `FOREACH` block stands for a subquery that selects that path expression. Thus, a shorter version of Query (14) is:

```
[201] BBB
[202]     Restaurant
[203]         Name "Blues on the Bay"
[204]         Rating 4
[205]     Restaurant
[206]         Name "The Greasy Spoon"
[207]         Rating 1
```

Fig. 2. The BBB restaurant ratings

```
SELECT AS LocalGroups
       FOREACH Frodos.Group {
           Frodos.Group.Name,
           Frodos.Group.*.Phone
       }
WHERE  Frodos.Group.Location.City = "Palo Alto"                (15)
```

The labels of all objects in the result set, not only the top-level object, can be changed from their defaults by using **AS**. In addition, the **FROM** clause can contain more than one named object. Query (16) illustrates both relabeling objects in the result and a **FROM** clause with multiple named objects. Assume for this query that we have a BBB restaurant guide database, which provides ratings for restaurants (see Figure 2). We have generally omitted the **FROM** clause since, in the absence of wildcards, it can be deduced from the rest of the query, but we include it in the following example for clarity.

```
SELECT FOREACH Frodos.Restaurant AS RatedRestaurant {
           Frodos.Restaurant.Name,
           Frodos.Restaurant.Category AS Type,
           BBB.Restaurant.Rating AS BBB-Rating
       }
FROM   Frodos, BBB
WHERE  BBB.Restaurant.Name = Frodos.Restaurant.Name            (16)
```

The result of Query (16) is:

```
[90]  Answer
[91]      RatedRestaurant
[92]          Name "Blues on the Bay"
[93]          Type "Vegetarian"
[94]          BBB-Rating 4
```

When subqueries with correlation appear in the **SELECT** clause, they are a powerful tool for materializing complex result structures from a database. We have not illustrated their full power here. For example, it is possible for a **FOREACH** clause to contain more than one path expression, and **FOREACH** clauses may be nested to any depth. Subqueries with correlation are an extension and generalization of the *OID functions* available in XSQL [KKS92]; they are more powerful than the *construction expressions* provided by OQL [Cat94] that apply to sets and structures. For more details, the reader is referred to [QRS$^+$94].

6 The Repository

We are currently building LORE, a "Lightweight Object REpository" based upon our data model and query language. Recall that the goal of the TSIMMIS project is to integrate data from heterogeneous information sources. Figure 3 illustrates the portion of the TSIMMIS framework relevant to query processing. Queries are posed by the client using LOREL and are sent to *mediators*, whose purpose is to provide a uniform view of data from one or more information sources. (TSIMMIS mediator specification is described in [PGMU95].) A mediator splits the incoming query into one or more single-source LOREL queries and sends them to *translators*. (Queries may also be sent directly from the client to translators, but this is not shown in the figure.) A translator converts incoming queries from LOREL to a source-specific query language and sends the source-specific queries to the information source. When data is returned from the source, the translator converts the data from the source-specific data format to our data model. The mediator then processes and combines data from the translators to construct an answer for the client. The entire TSIMMIS framework is explained in [CGMH$^+$94].

Even though the purpose of the TSIMMIS framework is to integrate data from existing information sources, an object repository is useful in several places within the framework. In the figure we highlight four places where LORE is useful:

- **Storing query results at a client.** When a client wants to find information, the search may involve issuing several queries, examining results from a query before issuing the next, and perhaps combining query results. Storing query results in LORE facilitates browsing large results and permits saving results for later review and use. In addition, a client can use LORE to create a "personal information workspace," enabling personal data to easily be integrated with the rest of the TSIMMIS framework.
- **Executing multisource queries.** Clients pose queries to mediators, which merge data from multiple sources. In some cases the mediator may itself need to do a significant amount of processing over a large amount of data. LORE can be used by a mediator to manage intermediate results during query execution.
- **Translating local queries.** Not every LOREL query can be translated to a single query for every information source, especially if the source provides only primitive query mechanisms. Like mediators, translators can use LORE during query execution to manage temporary query results.
- **Importing data.** Some data formats (such as structured files) are not well suited for querying. For these formats, it may be best to import the data into a database, especially if the data changes infrequently [SLS$^+$93]. Using LORE is an easy way to make the data available for querying, and the data can easily be integrated with the rest of TSIMMIS since there is no need for a translator.

Hence, in TSIMMIS LORE manages primarily data that is either temporary or is (relatively) easily recreated. In general there is no need for sharing data, except for read-only data imported from external data sources. For these uses,

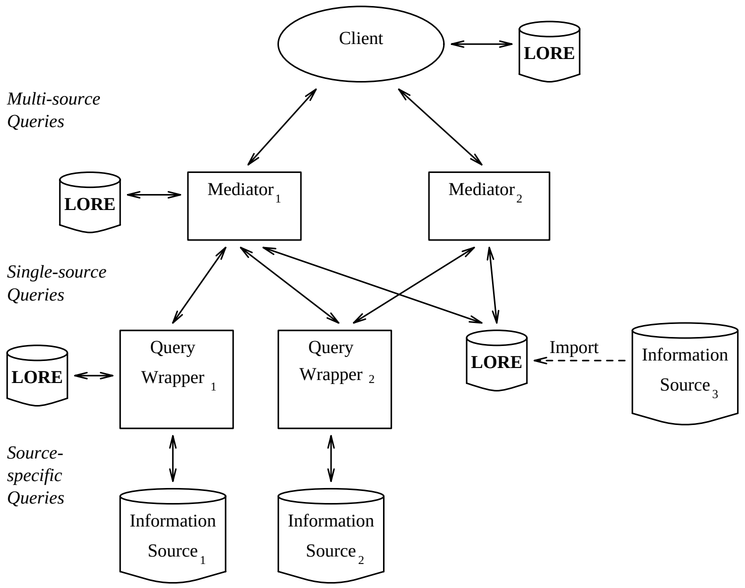

Fig. 3. TSIMMIS framework. LORE is used in several places.

LORE need not be a full-feature DBMS. Therefore, LORE is not only a repository for lightweight objects, but also a lightweight repository for objects! In particular, currently LORE does not support locking, logging, or transactions, making the implementation effort much less complex. If the need for multiuser access arises, we will add these features later.

6.1 Query Processing

Figure 4 illustrates the approach we are using for executing queries in LORE. Note that the architecture is quite similar to that of a typical relational DBMS, with parsing, query rewrite, query optimization, and query execution phases. Also similar to relational implementations, we form query execution plans as trees with operators at each node (outlined below). Objects are stored by an object manager based on our data model. In addition to the *identifier*, *label*, and *value* properties, each object contains type and length information when stored on disk.

Our query plan operators are similar to those used in relational and nested-relational languages, except that ours typically act on sets of object assignments (recall Section 5.2) rather than sets of tuples. Table 3 lists some of the operators. The operators for grouping and for returning complex objects in a query result are more complicated and are not shown. In the table, $\mathcal{S}$ is the domain of database states, $\mathcal{AS}$ is the domain of sets of object assignments, $\mathcal{O}$ is the domain of database objects, $\mathcal{PS}$ is the domain of sets of path expression prefixes, ⟨PATH EXPR⟩ is the domain of path expressions, ⟨PREDICATE⟩ is the domain of predicates, and ⟨LABEL⟩ is the domain of labels.

Operator	*Signature*	*Short Description*
Match	$\langle\text{PATH EXPR}\rangle \to (\mathcal{S} \to \mathcal{AS})$	Given a path expression, returns a function that maps a database state into a set of object assignments, where each object assignment is a mapping from the prefixes of the path expression to matching objects in the database state.
Select	$\langle\text{PREDICATE}\rangle \to (\mathcal{AS} \to \mathcal{AS})$	Similar to relational select. Given a predicate, returns a function that restricts the object assignments in an assignment set to those that satisfy the predicate.
Join	$\mathcal{AS} \times \mathcal{AS} \to \mathcal{AS}$	Similar to relational join. Takes the union of related object assignments, where two object assignments are related iff every path expression prefix common to both is mapped to the same database object (or database path, if "*" wildcards are involved).
Semijoin	$\mathcal{AS} \times \mathcal{AS} \to \mathcal{AS}$	Similar to relational semijoin. Restricts the set of object assignments in the first argument to those that are related to some object assignment in the second argument.
Project	$\mathcal{PS} \to (\mathcal{AS} \to \mathcal{AS})$	Similar to relational project. Given a set of path expression prefixes *ps*, returns a function such that for each object assignment in the assignment set, the mappings between path expression prefixes and database objects (or paths) are restricted to those for path expression prefixes appearing in *ps*. Duplicates are removed.
Union	$\mathcal{AS} \times \mathcal{AS} \to \mathcal{AS}$	Similar to set union, except duplicates are not removed.
Difference	$\mathcal{AS} \times \mathcal{AS} \to \mathcal{AS}$	Similar to set difference. Restricts the object assignments in the first argument to those that are not equivalent to some object assignment in the second argument, where two object assignments are equal iff they map the same set of path expression prefixes to the same database objects.
CreateResult	$(\langle\text{LABEL}_1\rangle \times \langle\text{LABEL}_2\rangle \times \langle\text{PATH EXPR}\rangle) \to (\mathcal{AS} \to \mathcal{O})$	Given a label l_1, a label l_2, and a path expression p, returns a function that creates an object labeled l_1, with a subobject for each object assignment in the assignment set. Each subobject is labeled l_2 but is otherwise the same as the object to which the corresponding object assignment maps p (or, if p ends in a "*" wildcard, the last object in the path). Here, "the same as" indicates structure and values, not necessarily object identifiers [PGMW95].

Table 3. Query plan operators

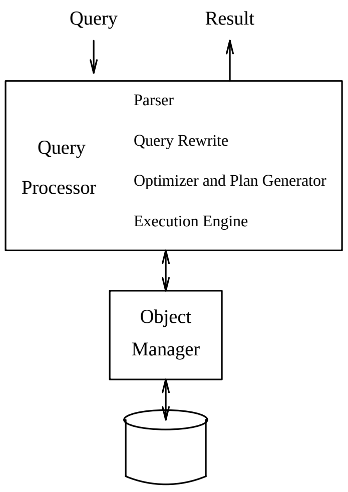

Fig. 4. LORE architecture

We briefly describe how queries are processed in LORE. First, the query is parsed and an initial query plan is generated. This plan is then optimized in the query rewrite and plan optimization modules, and finally executed by the query execution module. Figure 5 shows one plan for Query (1) of Section 5.1 using the operators in Table 3. The arrows between operators are annotated with the sets of object assignments resulting from each operator, with an object assignment represented as a set of "*path expression prefix* : [*identifier*]" pairs. The path expression `Frodos.Group.Category` is matched in the database, resulting in a set of object assignments, one for each matching database path. The set is then restricted to those object assignments where the value of `Frodos.Group.Category` is `"Opera"`. Next, the path expression `Frodos.Group.Name` is matched in the database and the result is semijoined with the first result, effectively restricting the object assignments for `Frodos.Group.Name` to those where the group's category is opera. Finally, this restricted assignment set is passed to *CreateResult*, which generates an object labeled `Answer` with subobjects labeled `Name`. Each `Name` subobject corresponds to an object mapped to by the path expression `Frodos.Group.Name` under an object assignment in the set.

As in relational systems, there may be many plans for a given query. Currently we generate naive query plans for most queries. Obviously there is a great deal of room for optimization, including both query rewrite strategies, more efficient physical operators, and cost-based selection of a query plan using statistics; we are beginning to investigate this area. We also are examining techniques to efficiently answer queries that involve the wildcards $*$ and ?, together with the operators `PATHOF`, `TYPEOF`, and `LABELOF`. Finally, we are designing appropriate indexing mechanisms for LORE, along with techniques that can determine automatically which indexes to build and destroy as the structure of the data evolves.

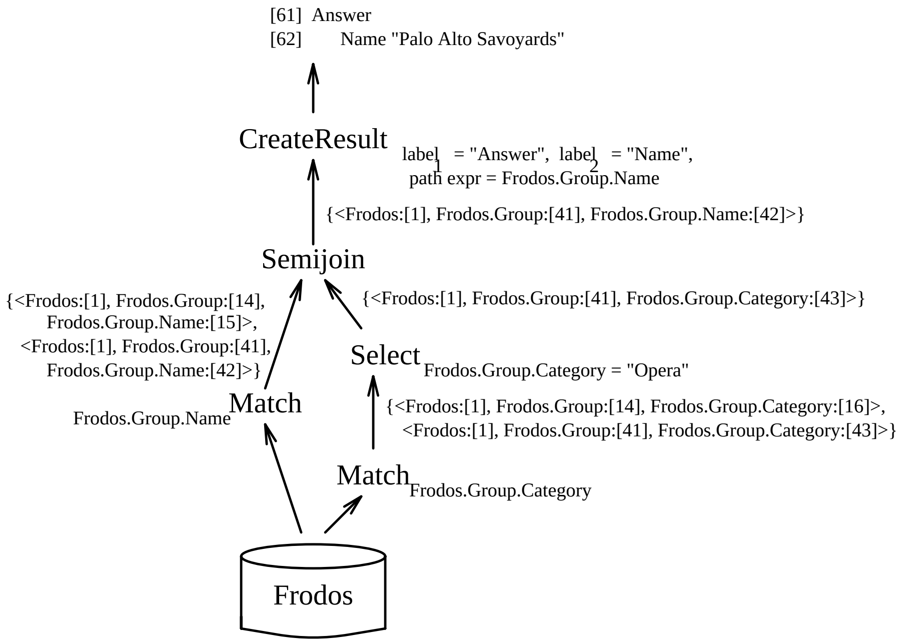
[61] Answer
[62] Name "Palo Alto Savoyards"
CreateResult
label1 = "Answer", label2 = "Name",
path expr = Frodos.Group.Name
{<Frodos:[1], Frodos.Group:[41], Frodos.Group.Name:[42]>}
Semijoin
{<Frodos:[1], Frodos.Group:[14], Frodos.Group.Name:[15]>, <Frodos:[1], Frodos.Group:[41], Frodos.Group.Name:[42]>}
{<Frodos:[1], Frodos.Group:[41], Frodos.Group.Category:[43]>}
Select
Frodos.Group.Category = "Opera"
Match
Frodos.Group.Name
{<Frodos:[1], Frodos.Group:[14], Frodos.Group.Category:[16]>, <Frodos:[1], Frodos.Group:[41], Frodos.Group.Category:[43]>}
Match
Frodos.Group.Category
Frodos

Fig. 5. Example query plan

7 Conclusion

We have presented the need for a new "lightweight" object query language for semistructured data. We pointed out several key areas where conventional query languages are inadequate for querying semistructured data, such as when data is absent, when data does not have a regular structure, when similar concepts are represented using different types, and when heterogeneous sets are present. We then introduced, through a series of examples, our LOREL query language, which addresses these issues. We explained the uses of an object repository, LORE, implementing our language, and we briefly described query processing in LORE.

We have learned a number of interesting things in our work so far. We are convinced of the importance of formally specifying language semantics. Defining a denotational semantics for LOREL helped us discover and resolve a number of discrepancies and omissions in our informal understanding. However, there are still areas in the language that can be improved. For example, although the language has powerful constructs for schema browsing, there is currently no way to query which external predicates and functions are applicable to an object. Instead, for now, in the TSIMMIS context we expect each translator or mediator to supply a "help page" describing the external functions and predicates available [PGMW95].

We are working towards the completion of the repository (Version 1.0), and we plan to use it in several places within the TSIMMIS framework, as shown in

Figure 3. We also intend to add data modification statements to the language, and develop a set of equivalent query plan transformations for use in query optimization. In the future we also intend to explore the specification and exploitation of integrity constraints within our language and system, and to add monitoring (active database) capabilities.

Acknowledgements

We are grateful to Hector Garcia-Molina, Yannis Papakonstantinou, and the entire Stanford Database Group for numerous fruitful discussions.

References

[BCD92] F. Bancilhon, S. Cluet, and C. Delobel. A query language for O_2. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an Object-Oriented Database System – The Story of O_2*, pages 234–255. Morgan Kauffmann, 1992.

[BCK+94] G. Blake, M. Consens, P. Kilpeläinen, P. Larson, T. Snider, and F. Tompa. Text / relational database management systems: Harmonizing SQL and SGML. In W. Litwin and T. Risch, editors, *Applications of Databases: First International Conference*, pages 267–280. Vadstena, Sweden, 1994.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilties. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 313–324, Minneapolis, MN, May 1994.

[Cat94] R. Cattel, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, 1994.

[CDV88] M. Carey, D. DeWitt, and S. Vandenberg. A data model and query language for Exodus. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 413–423, Chicago, IL, June 1988.

[CGMH+94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *Proceedings of the 100th IPSJ*, Tokyo, Japan, October 1994.

[DKA+86] P. Dadam, K. Kuespert, F. Andersen, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, and G. Walch. A DBMS prototype to support extended NF^2 relations: An integrated view on flat tables and hierarchies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 356–367, 1986.

[F+89] D. Fishman et al. Overview of the Iris DBMS. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Languages, and Applications*, pages 219–250. Addison-Wesley, 1989.

[Fre94] M. Freedman. WILLOW: Technical overview. Available by anonymous ftp from `ftp.cac.washington.edu` as the file `willow/Tech-Report.ps`, September 1994.

[GF94] M. Genesereth and R. Fikes. Knowledge interchange format reference manual (version 3.0). Available at the URL `http://logic.stanford.edu/sharing/papers/kif.ps`, 1994.

[Har94] C. Harrison. An adaptive query language for object-oriented databases: Automatic navigation through partially specified data structures. Available by anonymous ftp from `ftp.ccs.neu.edu` as the file `pub/people/lieber/adaptive-query-lang.ps`, 1994.

[ISO86] ISO 8879. Information processing—text and office systems—Standard Generalized Markup Language (SGML), 1986.

[Kim94] W. Kim. On object oriented database technology. UniSQL product literature, 1994.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 393–402, 1992.

[LMR90] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, 1990.

[Mic94] Microsoft Corporation. *OLE2 Programmer's Reference*. Microsoft Press, Redmond, WA, 1994.

[MS93] J. Melton and A.R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, San Mateo, California, 1993.

[OMG92] OMG ORBTF. *Common Object Request Broker Architecture*. Object Management Group, Framingham, MA, 1992.

[PGMU95] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. MedMaker: A mediation system based on declarative specifications. Available by anonymous ftp from `db.stanford.edu` as the file `pub/papakonstantinou/1995/medmaker.ps`, 1995.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, March 1995.

[Qia93] X. Qian. Semantic interoperation via intelligent mediation. In *Proceedings of the Third International Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems*, pages 228–231. IEEE Computer Society Press, April 1993.

[QRS$^+$94] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. Available by anonymous ftp from `db.stanford.edu` as the file `pub/quass/1994/querying-full.ps`, 1994.

[RAK$^+$92] A. Rafii, R. Ahmed, M. Ketabchi, P. DeSmedt, and W. Du. Integration strategies in Pegasus object oriented multidatabase system. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, Volume II*, pages 323–334, January 1992.

[RJR94] R. Rao, B. Janssen, and A. Rajaraman. GAIA technical overview. Technical Report, Xerox Palo Alto Research Center, 1994.

[SLS$^+$93] K. Shoens, A. Luniewski, P. Schwarz, J. Stamos, and J. Thomas. The RUFUS system: Information organization for semi-structured data. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, pages 97–107, Dublin, Ireland, August 1993.

[Sto77] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, Cambridge, Massachusetts, 1977.

[YA94] T. Yan and J. Annevelink. Integrating a structured-text retrieval system with an object-oriented database system. In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, Santiago, Chile, September 1994.