Journal of Digital Libraries Manuscript-Nr.
(will be inserted by hand later)

The Lorel Query Language for Semistructured Data*

Serge Abiteboul**, **Dallan Quass, Jason McHugh, Jennifer Widom, Janet L. Wiener**

Department of Computer Science
Stanford University
Stanford, CA 94402

`{abitebou,quass,mchughj,widom,wiener}@db.stanford.edu`
`http://www-db.stanford.edu`

Stanford University

Abstract. We present the *Lorel* language, designed for querying semistructured data. Semistructured data is becoming more and more prevalent, e.g., in structured documents such as HTML and when performing simple integration of data from multiple sources. Traditional data models and query languages are inappropriate, since semistructured data often is irregular: some data is missing, similar concepts are represented using different types, heterogeneous sets are present, or object structure is not fully known. Lorel is a user-friendly language in the SQL/OQL style for querying such data effectively. For wide applicability, the simple object model underlying Lorel can be viewed as an extension of the ODMG data model and the Lorel language as an extension of OQL.

The main novelties of the Lorel language are: (i) the extensive use of coercion to relieve the user from the strict typing of OQL, which is inappropriate for semistructured data; and (ii) powerful path expressions, which permit a flexible form of declarative navigational access and are particularly suitable when the details of the structure are not known to the user. Lorel also includes a declarative update language.

Lorel is implemented as the query language of the *Lore* prototype database management system at Stanford. Information about Lore can be found at
`http://www-db.stanford.edu/lore`.
In addition to presenting the Lorel language in full, this paper briefly describes the Lore system and query processor. We also briefly discuss a second implementation of Lorel on top of a conventional object-oriented database management system, the O_2 system.

* This work was supported by the Air Force Wright Laboratory Aeronautical Systems Center under ARPA Contract F33615-93-1-1339, by the Air Force Rome Laboratories under ARPA Contract F30602-95-C-0119, and by equipment grants from Digital Equipment and IBM Corporations.

** This author's permanent address is INRIA-Rocquencourt, 78153 Le Chesnay, France. His work was also supported by CESDIS-NASA.

1 Introduction

As the amount of data available on-line grows rapidly, we find that more and more of the data is *semistructured*. By semistructured, we mean that although the data may have some structure, the structure is not as rigid, regular, or complete as the structure required by traditional database management systems. (See [Abi97] for a survey on semistructured data.) Furthermore, even if the data is fairly well structured, the structure may evolve rapidly. Traditional relational database management systems require strict table-oriented data, and they are based on the notion that a schema is defined in advance and adhered to by all data managed by the system. While object-oriented database management systems permit much richer structure than relational systems, they still require that all data conform to a predefined schema.

Management of semistructured data requires typical database features such as a language for forming ad-hoc queries and updates, concurrency control, secondary storage management, etc. However, because semistructured data cannot conform to a standard database framework, trying to use a conventional DBMS to manage semistructured data becomes a difficult or impossible task. At Stanford, the goal of the *Lore* project (for *Lightweight Object Repository*[1]) is to provide convenient and efficient storage, querying, and updating of semistructured data. This paper presents Lore's query language *Lorel* (for *Lore language*). Although we have implemented Lorel in a "home grown" DBMS designed specifically for semistructured data, the data model underlying Lorel can be defined as an extension to the *ODMG* model and the language as an extension to *OQL*. (See [Cat94] for a specification of the ODMG model and OQL.) Thus, Lorel can be implemented on top of a conventional object-oriented DBMS, yielding a flexible system suitable for managing both structured and semistructured data.

[1] The Lore system is "lightweight" in two senses: the object model supported by Lore is lightweight, and the system itself is lightweight in that currently it does not support multiuser updates or other "heavyweight" DBMS features.

Semistructured data arises in a number of common situations. Some data sources are designed with non-rigid structures for convenience. A concrete example is the *ACeDB* genome database [TMD92], while a somewhat less concrete but certainly well-known example is the World-Wide Web. The Web imposes no constraints on the internal structure of HTML pages, although structural primitives such as enumerations may be used. Another frequent scenario for semistructured data is when data is integrated in a simple fashion from several heterogeneous sources and there are discrepancies among the various data representations: some information may be missing in some sources, an attribute may be single-valued in one source and multi-valued in another, or the same entity may be represented by different types in different sources.

When querying semistructured data, one cannot expect the user to be fully aware of the complete structure, especially if the structure evolves dynamically. Thus, it is important not to require full knowledge of the structure to express meaningful queries. At the same time, we do want to be able to exploit regular structure during query processing when it happens to exist and the user happens to know it. For this reason we use OEM, a self-describing data model where schematic information in the form of labels is intermixed with the data and can be queried.

In the remainder of this introductory section we first present some examples of semistructured data and queries over that data in English and in Lorel. We then further explain the relationship of Lorel and its underlying data model to OQL and ODMG. We finally discuss related work and preview the remainder of the paper before delving into the details.

1.1 Examples

We give two example queries to demonstrate the simplicity and power of Lorel on semistructured data. Intuitively, an OEM database can be thought of as a graph (see Figure 1) and a Lorel query can be viewed as specifying paths in the graph. Details of OEM and Lorel are given in later sections of the paper. For these examples, we assume a *Guide* database that collects information on local restaurants from a variety of sources (newspaper reviews, regional guidebooks, personal web pages, etc.). The first example shows how Lorel handles type coercion, which is important when the underlying data is untyped, irregularly typed, or may have missing fields. The second example shows the use of "wildcards" and regular expressions in Lorel, which are important when the structure of the data is irregular or unknown.

Example 1: Find the addresses of all restaurants in the 92310 zipcode. The Lorel query directly follows from the English statement:

```
select Guide.restaurant.address
where Guide.restaurant.address.zipcode = 92310
```

It is not necessary to know if the zipcode is represented as an integer or a string value because Lorel will coerce it accordingly, and if some zipcodes are strings and others are integers the expected result will still be retrieved. Furthermore, an address that does not contain a zipcode will not cause an error, but will simply fail the `where` condition. In most query languages, such as SQL and OQL, a type error will ensue if the types do not match or if a field is missing. In addition, in Lorel it is not necessary to worry about the cardinality (set versus singleton) of components in the path expressions, unlike in OQL. If a restaurant has several addresses, or several zipcodes for the same address, the expected result still is returned; i.e., we get any address with any 92310 zipcode.

Example 2: Find the names and zipcodes of all "cheap" restaurants. This time, we do not assume that the zipcode is a part of the address, but it may instead be a direct subobject of the restaurant. Also, we do not know if the string "cheap" will be part of a category, price, description, or other subobject of restaurant. We are still able to ask the query in Lorel as follows:

```
select Guide.restaurant.name,
       Guide.restaurant(.address)?.zipcode
where Guide.restaurant.% grep "cheap"
```

The "?" after *.address* means that the address is optional in the path expression. The wildcard "%" will match any subobject of restaurant, and the comparison operator `grep` will return true if the string "cheap" appears anywhere in that subobject's value. There is no equivalent query in SQL or OQL, since neither allow regular expressions or wildcards.

1.2 Lorel and OQL

The data model underlying Lorel is called *OEM* (for *Object Exchange Model*). OEM is a simple and flexible object model, introduced initially in the *Tsimmis* project at Stanford [PGMW95]. Roughly speaking, a database conforming to OEM can be thought of as a graph with complex values at internal nodes, atomic values at leaf nodes, and labeled edges.[2] Although the Lorel language could be presented "from scratch" based on OEM, as we have done with a previous version of Lorel [QRS+95a], for clarity and wider applicability we have chosen instead to define Lorel formally as an extension to OQL based on an OEM extension to the ODMG data model. For users familiar with OQL, the additional features introduced by Lorel for handling semistructured data are simple to learn. On the other hand, knowledge of OQL is not at all necessary to use Lorel, since the most common Lorel queries are expressed easily in a compact and intuitive form reminiscent of simple SQL.

[2] Some minor changes to the original model have been introduced to facilitate Lorel, e.g., labels were on vertices instead of edges in the original model, and we have added distinguished *names* as entry points into the database.

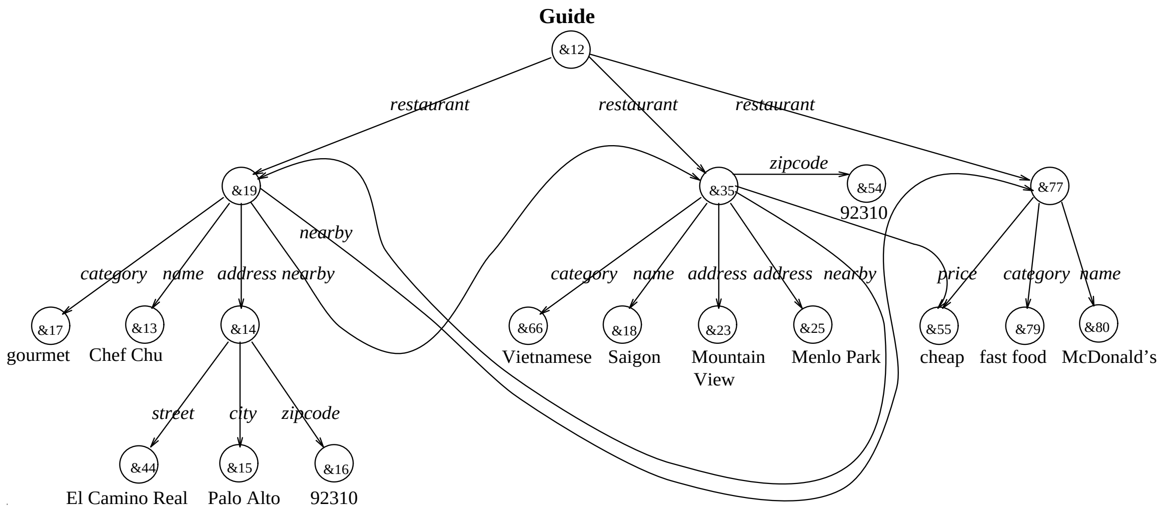

Fig. 1. An OEM graph

To define the semantics of Lorel over an OEM database in terms of OQL and ODMG, we add a new type to the ODMG model to represent OEM objects. Then, a core part of the formal Lorel language definition is to extend equality (and other base predicates and functions) in OQL to handle OEM objects. The extension relies heavily on coercion at a number of levels to relax the strong typing of OQL. At the same time, Lorel extends OQL with powerful and flexible path expressions, which allow querying without precise knowledge of the structure. Path expressions are built from labels and *wildcards* (place-holders) using regular expressions, allowing the user to specify rich patterns that are matched to actual paths in the database graph.

The relationships between Lorel/OEM and OQL/ODMG are depicted in Figure 2. Lorel can be translated syntactically to an extension of OQL that includes *heterogeneous objects*, described in Section 3, and path variables and wildcards, described in Section 5. (Consequently, many of the convenience features included in Lorel are actually syntactic sugaring over such an extension of OQL.) The query processor in the Lore system performs exactly this mapping before accessing an OEM data store; the process is depicted by the solid arrows in Figure 2, and the Lore system implementation is described in somewhat more detail in Section 9. We can also encode OEM objects in the ODMG model, in which case Lorel can be mapped to pure OQL. This process is depicted by the dashed arrows in Figure 2. Section 8 discusses how this approach is used to implement Lorel on top of an ODMG-conforming database management system, the O_2 system [BDK92].

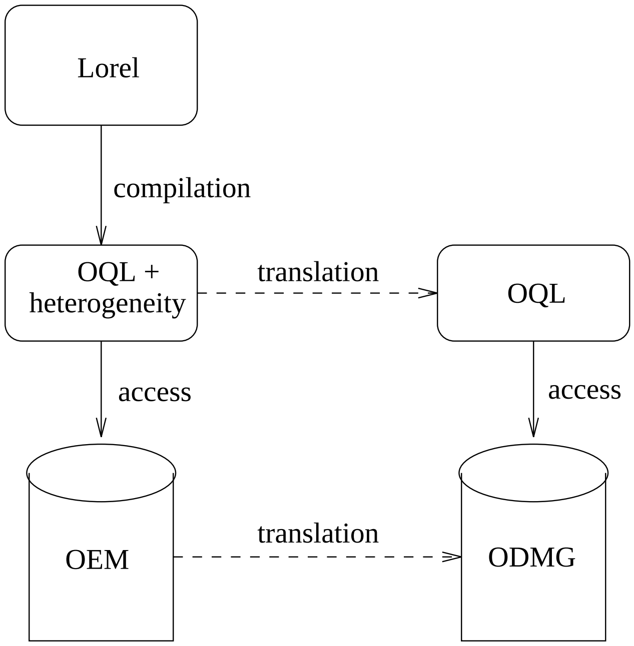

Fig. 2. Relationship between Lorel and OQL

1.3 Related work

There have been many proposals to extend query languages with path expressions. Lorel is unique in that it extends OQL (along with [CACS94]) and handles data that is semistructured. A first version of Lorel (now dubbed *Lorel1*) was introduced in [QRS+95a] and implemented in the initial version of the Lore system. Lorel1 was designed and defined from scratch, including a full denotational semantics for the language given in [QRS+95b]. As mentioned earlier, we decided to base the new version of Lorel (dubbed *Lorel96*) on an existing query language, since this approach provides a well-understood semantics and has wider applicability. The syntax of simple queries is almost identical in Lorel1 and Lorel96. However, the syntax for more complex constructs has changed, e.g., for aggregation, path variables, and construction of complex query results. In addition, because we now define Lorel in terms of OQL, coercion takes on an importance in Lorel96 that it did not have in Lorel1. A detailed comparison of Lorel1 with more conventional languages such as OQL [Cat94], XSQL [KKS92], and SQL [MS93] appears in [QRS+95a]; most comparisons carry over directly to Lorel96.

Another OEM-based language called *MSL* has been designed for mediator specification in the Stanford Tsimmis project [PGMU96, PAGM96]. MSL is a rule-based language that was designed with a different goal than Lorel, namely, to specify the integration of data drawn from multiple sources. We plan to try to characterize the relative expressiveness of MSL and Lorel.

UnQL, a language closely related to Lorel, was also designed for querying semistructured data. UnQL is based on a model similar to OEM [BDS95]. A primary feature of UnQL is a powerful construct called *traverse* that allows restructuring of trees to arbitrary depth. Such restructuring operations are not expressible in Lorel, which was designed primarily as a simple to use query language.

In [CACS94, CCM96], extensions to OQL are proposed that are somewhat similar in spirit or goals to Lorel. In [CACS94], a more rigidly typed approach is followed, but because heterogeneous collections are introduced, the model still has a strong similarity to OEM. However, the language proposed in [CACS94], called *OQL-doc*, does not use coercion the way it is used in Lorel, and the treatment of path expressions is quite different. Optimizing the evaluation of *generalized path expressions* is considered in [CCM96]. Their optimization is based on two object algebra operators, one dealing with paths at the schema level and one with paths at the data level. Since we do not have a strict schema, we cannot directly use their optimization techniques. However, we describe briefly in Section 9 the concept of a "data guide", which may serve in the role of a schema for an OEM database. We plan to consider adapting the optimization techniques of [CCM96] to OEM using the data guide.

Also related to our work are several query languages for the World-Wide Web that have emerged recently, e.g., *W3QL* [KS95], which focuses on extensibility, *WebSQL* [MMM96], which provides a formal semantics and introduces a notion of locality, and *WebLog* [LSS96], which is based on a Datalog-like syntax. Additional relevant work includes query languages for hypertext structures, e.g., [MW95, BK94, CM89, MW93], and work on integrating SGML [GR90] documents with relational databases [BCK+94] or object-oriented databases such as OpenODB [YA94] or O_2 [CACS94], since SGML documents can be viewed as semistructured.

In the area of heterogeneous database integration, which as we have suggested is a common scenario for semistructured data, most work 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.

Note that environments such as *CORBA* [OMG92] and *OLE2* [Mic94] operate at a different level from Lorel. These approaches provide a common protocol for passing messages between objects in a distributed object environment. In distributed settings, Lorel could certainly be built on top of and take advantage of environments such as CORBA and OLE2.

We believe that the powerful and user-friendly features of Lorel, together with a clean semantics inherited from OQL, a declarative update language, and a working prototype implementation, make Lorel unique among the languages cited above in the context of managing semistructured data.

1.4 Outline of paper

Section 2 specifies the Object Exchange Model (OEM) and explains how it can be viewed as an extension to the ODMG model. Sections 3–6 together specify the Lorel query language. Section 3 discusses the first important concept of Lorel, namely its extensive use of coercion. Sections 4 and 5 introduce the second important concept, path expressions. Simple path expressions are described in Section 4, while more complex expressions are introduced in Section 5. Section 6 describes how results of Lorel queries are constructed. Lorel's declarative update language is specified in Section 7. Section 8 briefly discusses a prototype implementation of Lorel on top of an object-oriented DBMS. Finally, Section 9 briefly covers the Lore system, describing its overall architecture and features as well as explaining query processing in somewhat more detail. Section 9 also covers the status of the implementation, availability of the system, and plans for future work. Appendix A contains a grammar for the full Lorel language. Note that not all constructs of Lorel are described in the body of the paper; rather, the paper focuses on those aspects of Lorel that are designed specifically for semistructured data.

2 The Object Exchange Model

In this section we present the Object Exchange Model (OEM) [PGMW95], a data model particularly useful for representing semistructured data. Data represented in OEM can be thought of as a graph, with objects as the vertices and labels on the edges. In addition, we will show how to extend the ODMG data model to capture OEM.

In the OEM data model all entities are *objects*. Each object has a unique *object identifier* (oid) from the type `oid`. Some objects are atomic and contain a value from one of the disjoint basic atomic types, e.g., `integer`, `real`, `string`, `gif`, `html`, `audio`, `java`, etc. All other objects are complex; their value is a set of *object references*, denoted as a set of (*label*, *oid*) pairs. The labels are taken from the atomic type `string`.

In Figure 3, we show an example OEM database. Each line shows the label used to reach an object and the object's oid. If the object is atomic, its value is also given on that line. If the object is complex, and has not been described earlier, subsequent indented lines describe its object references or "subobjects." For example, the object with oid &77 has three references: (*category*, &79), (*name*, &80), and (*price*, &55). The object with oid &79

```
Guide &12
  restaurant &19
    category &17 "gourmet"
    name &13 "Chef Chu"
    address &14
      street &44 "El Camino Real"
      city &15 "Palo Alto"
      zipcode &16 92310
    nearby_eating_place &35
    nearby_eating_place &77
  restaurant &35
    category &66 "Vietnamese"
    name &18 "Saigon"
    address &23 "Mountain View"
    address &25 "Menlo Park"
    nearby_eating_place &19
    zipcode &54 "92310"
    price &55 "cheap"
  restaurant &77
    category &79 "fast food"
    name &80 "McDonald's"
    price &55
```

Fig. 3. Textual representation of objects in an OEM database

is an atomic object of type `string` whose value is "fast food".

We adopt the ODMG feature of distinguished (object) *names*. There are many facets to the concept of name:

- A name can be viewed as an alias for an object in the database. For instance, *Guide* is the name of the object in Figure 3 that contains a collection of restaurants, i.e., object &12.
- As seen in the example queries, a name serves as an entry point to the database. Indeed, the only way objects can be accessed in queries is via paths originating from names.
- As in the ODMG model, we require that all objects in the database are reachable from one of the names. (The rationale is that if an object becomes unreachable, no query will ever manage to access it, so the object might as well be garbage collected.) Hence, names also serve as *roots of persistence*: an object is persistent if it is reachable from one of the names.

OEM can easily model relational data, and, as in the ODMG model, hierarchical and graph data. (Although the structure in Figure 3 is close to a tree, object &35 is "shared" by objects &12 and &19 and there is even a directed cycle via objects &19 and &35.) However, we do not insist that data is as strongly structured as in standard database models, allowing us to model, e.g., semistructured information sources, data that originates from the integration of heterogeneous sources, and documents that do not conform to a precise schema. Observe in Figure 3 that, for example: (i) restaurants have zero, one, or more addresses; (ii) an address is sometimes a string and sometimes a complex structure; (iii) a zipcode may be a string or an integer; and (iv) the zipcode occurs in the address for some restaurants and directly under restaurant for others. Lorel is designed to handle incompleteness of data, as well as structure and type heterogeneity, as exhibited in this example database.

We now give a formal definition of an OEM database, treated as a graph.

Definition: An *OEM schema* consists of a finite set of *names* $\mathbf{R}$. An *OEM instance* of $\mathbf{R}$ consists of: (i) a finite labeled graph $(V_a \cup V_c, E)$ where V_a and V_c are disjoint sets of oid's corresponding respectively to *atomic* and *complex* objects, and the edges in E are labeled by strings; (ii) a *name* function from $\mathbf{R}$ to $V_a \cup V_c$; and (iii) a value function *val* that maps the objects in V_a to atomic values. The instance must also satisfy the following two conditions:

1. Atomic vertices have no outgoing edges.
2. Each vertex is reachable from object $name(N)$ for some name N in $\mathbf{R}$. $\square$

We say that an object $o_1 \in V_a \cup V_c$ is an l *subobject of object* $o_2 \in V_a \cup V_c$ if there is an edge in E from o_2 to o_1 labeled l.[3]

Figure 1 presented earlier provides an example of an OEM database as a graph. It corresponds to the data given textually in Figure 3.

2.1 Extending the ODMG data model

We now show how we can extend the ODMG data model to represent semistructured data by "typing" OEM objects as ODMG objects. This approach provides additional intuition to readers familiar with the ODMG model. It also allows us to use OQL as a basis for defining the Lorel language. Finally, it suggests an implementation of Lorel on top of a traditional object database system, which we discuss further in Section 8.

The difficulty in typing OEM objects is clearly the heterogeneity of the OEM data. To deal with the heterogeneity, we think of a complex OEM object as a tuple consisting of fields $a_1, a_2, \ldots, a_n$, where $a_1 \ldots a_n$ are all labels currently present in the database. (We could alternatively think of a complex object as a tuple with infinitely many fields, one for each possible string label. Such objects could still be represented finitely, since at any one time only a finite number of fields is nonempty for each object.)

An important consequence of this encoding is that all complex objects in the database are of the same type, namely *OEM*, formally specified below. In particular, all names in an OEM database are of this type. The value of the a_i field for a particular OEM object o is the (possibly empty) set of a_i subobjects of o, i.e., the set of objects referenced from o via an a_i-labeled edge. If o does not reference any objects using an a_i-labeled edge, it still has an a_i field but the value of that field is empty.

For example, in the database shown in Figure 3, complex objects are typed by representing them as tuples with `restaurant`, `category`, `name`, `address`, `nearby`,

[3] Note, however, that the subobject relationship is *not* one of containment—an object can be a subobject of many other objects.

street, **city**, and **zipcode** fields. For the object with oid &12, the **restaurant** field would contain the set {&19, &35, &77}; all other fields would be empty.

In the type definitions below, we use the symbol "+" to denote the union of types. The *OEM* type is as follows.

$$
\begin{aligned}
&type\ OEM = OEMcomplex + OEMstring + \\
&\quad OEMint + ... + OEMnil \\
&type\ OEMcomplex = \\
&\quad struct(a_1 : set(OEM), ..., a_n : set(OEM)) \\
&type\ OEMstring = string \\
&type\ OEMint = integer \\
&... \\
&type\ OEMnil = ()
\end{aligned}
$$

where $a_1, ..., a_n$ is the list of distinct labels occurring in the database. Only integer and string atomic values are shown explicitly in order to simplify the presentation. There is a single object of type *OEMnil*, namely the *oemnil* object, whose purpose will become clear in Section 4. The definition above is not quite a valid ODMG type since the ODMG model does not support the union of types. We consider the "coding" of OEM objects as pure ODMG objects for implementation in an ODMG database in Section 8.

For an object X and a label l, the expression $X.l$ denotes the set of l subobjects of X. If X is an atomic object or if l is not a label occurring in the database (the two cases where X has no l field), $X.l$ is the empty set. Observe that $X.l$ always denotes a set of objects. Having an expression always result in the same type regardless of the structure of the underlying data is a key idea in extending OQL to handle semistructured data.

3 Coercion

In this and the following three sections we describe in detail the novel aspects of the Lorel query language, namely coercion and powerful path expressions, and we explain how query results are constructed. For readability, we present these features primarily in terms of how they extend the OQL language. Note that since we are focusing only on features designed specifically for handling semistructured data, many other useful features of the Lorel language—some inherited from OQL and others not—are not covered; see Appendix A for a specification of the full Lorel language.

One of the main issues in defining Lorel as an extension to OQL is to force comparisons between objects and/or values to "do the intuitive thing" (rather than return a type error) when comparing objects and values of different types. In this section we illustrate the need for coercion by an example, define precisely the coercion we use, and introduce a new comparison operator that is very useful for semistructured data. We assume here some rudimentary knowledge of OQL syntax and semantics, although most queries are self-explanatory.

Let us consider carefully a query asking for the addresses of all restaurants with zipcode 92310, ignoring for the moment that zipcode could be nested within address. Using pure OQL syntax (although Lorel permits simpler expression of the same query), the query is:

```
select X.address
from Guide.restaurant X, X.zipcode Y
where Y = 92310
```

Strictly speaking, X is an object (e.g., object &35) and Y is a zipcode subobject of X (e.g., object &54). So, although the query corresponds to our intuition, in OQL there would be a type error in equating Y, an object, and 92310, an integer. In Lorel, this query is legal and returns the desired result.

A guiding principle for Lorel is that a query that makes sense should never result in a run-time error on any OEM data. To write a query one should not have to know the precise structure of complex objects, nor should one have to bother with the precise types of atomic objects. This flexibility is achieved: (i) by extending the base predicates (e.g., =) and base functions (e.g., +) of OQL to perform extensive coercions (Sections 3.1 and 3.2), and (ii) by defining a new value-based equality operator (Section 3.3). Readers familiar with object-oriented languages may think of the extended predicates and functions as calls to methods attached to the type *OEM* (see Section 8).

3.1 Comparing values and atomic objects

In general, certain predicates and functions expect arguments of particular atomic types. Sometimes they accept more than one type; e.g., the comparator $<$ works for integers and for reals. In the context of semistructured data, we prefer to accept conditions such as $Z = 1.0$ and $Z >$ "0.9" as *true* if Z is an object of value 1 or even of value "1". In this section, we consider coercion when comparing atomic objects and values. Coercion used to compare complex objects or collections of objects is considered in the next section.

We focus first on the basic comparison operators (e.g., $=,<,\neq$). When comparing atomic objects and values, we want to coerce the two operands to values that are comparable whenever possible. Let us assume that X is an integer OEM object. To compare X to an integer, say 555, we must first coerce the object X to its value by dereferencing it. To compare X to a real, we must first dereference X, then coerce its integer value to a real. The process is guided by the type of the operands. For an integer object X, the comparison to an arbitrary atomic value Y proceeds as follows.

let X' be the value of X;
case Y of
    integer: compare X' and Y;
    real: compare int-to-real(X') and Y
    string: if Y cannot be coerced to real then *false*
        else compare int-to-real(X') and
            string-to-real(Y)

If there are additional coercible atomic types they are included in the case statement.

arg2 / *arg1*	*string*	*real*	*int*
string	–	*string* → *real*	*both* → *real*
real		–	*int* → *real*
int			–

Table 1. Coercion for basic comparison operators

In general, coercion rules should be provided for the basic atomic types and the corresponding predicates and functions. They also could be provided for application-specific atomic types, e.g., coercion of dollars to francs, months to days, gifs to jpegs, etc. Table 1 shows (omitting dereferencing) the coercion that takes place for atomic types `string`, `integer`, and `real`, for the basic comparison operators ($=, <, \neq$). Note that the symmetric cases are omitted. Coercion for the basic comparison operators is not trivial because of the need to coerce both values to comparable atomic types.[4] For example, in the comparison "4.3" $<$ 5, both the string "4.3" and the integer 5 must be coerced to real in order to perform the comparison. Also observe that equality is no longer transitive; for example, "05" = 5 and 5 = "5," but "05" $\neq$ "5".

Coercion for other (non-arithmetic) comparison operators can be much simpler. For instance, Lorel also includes the string-based comparators `like`, `grep`, and `soundex`, which expect operands of a precise atomic type (string). The rule in this case is simply to coerce both operands to the expected atomic type, if possible.

Not all atomic types are comparable, e.g., we cannot compare `gif` images and `audio` clips. In the case of comparing values of incomparable atomic types, the comparison does not return an error—it simply returns false. Furthermore, even when the atomic types are comparable the coercion may fail, e.g., the string "apple" cannot be compared to an integer. In these cases the comparison also returns false.

Besides comparators such as those mentioned above, we also need to use coercion for the functions of the language, such as the arithmetic functions (addition, multiplication, etc.). Coercion for functions is handled similarly.

3.2 Comparing objects and sets of objects

In this section, we consider the use of coercion in comparing atomic objects, complex objects, and sets of objects. In Lorel, a variable X can be assigned to either an atomic value, an atomic object, a complex object, or a set of objects. Table 2 presents the coercion rules for equality. The coercion rules for inequality are similar. Again, the symmetric cases are not shown.

Note that some of the cases in Table 2 were covered in Section 3.1. For instance, to compare a value and an atomic object, we first dereference the object. This leads to comparing two atomic values, which is handled by the coercion rules of Table 1. Let us consider the new cases.

[4] This particular table was the outcome of a lively Lore meeting at Stanford. An interesting issue (not addressed in this paper) is the development of access techniques, e.g., indexing [Raj96] or hashing, to support such comparisons.

arg2 / *arg1*	*value*	*atomic object*	*set of objects*	*complex object*
value	*coerce*	*dereference*	*existential with* =	*false*
atomic object		*object* =	*existential with* =	*false*
set of objects			*set equality*	*false*
complex object				*object* =

Table 2. Coercion for equality =

Object against object. In this case, equality is exactly as in OQL: by oid comparison. Note that different atomic objects may contain identical atomic values, and users often want to compare objects using value equality. For instance, in querying "what are the restaurants that have a nearby restaurant with the same zipcode," the intension in comparing zipcodes is more likely by value than by object equality over zipcode. We consider another equality operator, ==, in the next section that forces value equality when comparing objects. The issue of value versus object equality does not arise for inequality operators (such as $<$), since inequality operators are not defined on objects. So, for instance, if $<$ is used between two objects, a coercion of the objects to values is performed before the comparion occurs.

Value, atomic object, or set of objects against a complex object. We do not know which subobject of the complex object should be used in the comparison. Thus, the comparison fails and returns false.

Set of objects against set of objects. In this case standard set equality is used: for each element of one set there must be an equal element in the other set.

Value or atomic object against set of objects. This is the most interesting case. Consider the following example query, again expressed in OQL syntax:

```
select X.address
from Guide.restaurant X
where X.name = "Chef Chu"
```

The condition `X.name = "Chef Chu"` seems to be another example of a type error since technically $X.name$ is a set (all *name* subobjects of X). However, the user may believe that name is a single-valued attribute or may not care whether this is the case or not. In Lorel, we interpret this `where` clause as:

```
where exists Z in X : Z = "Chef Chu"
```

arg1 \ *arg2*	*value*	*atomic object*	*set of objects*	*complex object*
value	*coerce*	*dereference*	*existential with ==*	*false*
atomic object		*value =*	*existential with ==*	*false*
set of objects			*existential with == on both sides*	*false*
complex object				*value =*

Table 3. Coercion for equality ==

The comparison of Z to a string now follows the coercion rules of Table 1. This approach captures the intension of users who expect a single name field, while gracefully handling restaurants with multiple names. Introducing the existential quantification can be viewed as a form of coercion from a set to an element. The coercion involved when comparing an object to a set of objects is similar.

3.3 More on equality

As mentioned earlier, in semistructured environments, users are interested primarily in the *values* of objects. Thus, value equality is often more appropriate than oid equality in Lorel. In Lorel we have chosen to retain oid equality for the comparison of objects with objects since it may sometimes be useful to test whether the same OEM object occurs in two "locations" (i.e., detect the sharing of a subobject). To handle value equality we introduce a new operator, denoted "==". This operator is not a substantial increase in complexity—a naive user of Lorel could use only this form of equality, ignore =, and almost certainly get the desired result.

Let us illustrate the use of == by an example. Consider the following query:

```
select X.name
from John.name JN, John.child X, X.name XN
where JN == XN
```

The intended meaning is "retrieve the children of John bearing his name." We will see a simpler way of expressing this query below. Note that JN and XN are the names of John and a child of John, respectively. The operator == expects atomic values on both sides of $JN == XN$, so coercion is performed to obtain the object values, which results in comparing the two strings and not the oids. Note that had we used = instead of == in the query, we would not get the desired answer (assuming names are stored as separate atomic objects and not shared).

A better way to express the previous query is:

```
select X.name
from John.child X
where John.name == X.name
```

This is a case of the comparison of two sets. Since the == predicate expects atomic values, the sets are coerced into atomic values using existential quantification as follows:

```
select X.name
from John.child X
where exists JN in John.name :
      exists XN in X.name : JN == XN
```

where $JN == XN$ itself involves coercion to string values.

The coercion rules for operator == are summarized in Table 3.[5]

4 Simple Path Expressions

When querying semistructured data, especially when the exact structure is not known, it is convenient to use a form of "navigational" querying based on path expressions. The idea is to specify paths in the OEM graph based on the sequence of labels on edges. In this section, we describe *simple path expressions*, which allow users to obtain the set of objects reachable by following a sequence of labels starting from a named object in the OEM graph. A more powerful form of path expressions based on wildcards and regular expressions is described in Section 5.

A *simple path expression* is a sequence $Z.l_1 \ldots .l_n$, where $l_1, \ldots, l_n$ are labels and Z is an object name or a variable denoting an object. A *data path* is a sequence $o_0, l_1, o_1, \ldots, l_n, o_n$, where the o_i's are objects and, for each i, there is an edge labeled l_i between o_{i-1} and o_i. Starting from an object $Z = o_0$ there may be several data paths that "match" the simple path expression $Z.l_1 \ldots .l_n$. Path expressions are an extremely convenient and user-friendly feature of Lorel. However, as we will see, simple path expressions are a syntactic convenience. Indeed, we explain the semantics of simple path expressions in this section by describing how they can be reduced in a query to one or more OQL-style object-component references.

We first illustrate this reduction with an example. Consider the object named *Guide* and the simple path expression *Guide.A.B.C*. This path can be interpreted navigationally as: start from object *Guide*, follow an A edge, then a B, and finally a C edge. Since there are possibly many A, B, and C labeled edges, the path expression can be matched to a number of data paths in the OEM graph. Alternatively, we can interpret this path expression using OQL-style object-component referencing: *Guide.A* denotes the set of objects R with an A edge from *Guide* to R, *Guide.A.B* denotes the objects Z such that for some R in *Guide.A*, there is a B edge from R to Z, and similarly for *Guide.A.B.C*. The following concrete example illustrates the notion. The Lorel query, shown on top, is equivalent to the OQL query, shown on the bottom, with the path expression reduced.

[5] Note that some languages assign different meanings than Lorel to = and ==. We chose these assignments in order to keep the meaning of = as it is in OQL.

```
select Z
from Guide.restaurant.zipcode Z
---
select Z
from Guide.restaurant R, R.zipcode Z
```

The precise reduction of a simple path expression depends upon whether the path expression appears in the `from`, `select`, or `where` clauses. We consider each case in turn.

4.1 From clause

The case of a path expression appearing in the `from` clause was illustrated by the previous example. Indeed, as the example suggests, the general intuition for reducing path expressions in the `from` clause is to insert a variable after each label. The actual algorithm is somewhat more complex since Lorel gives a particular semantics to common prefixes of multiple path expressions.

Consider the following `from` clause:

```
from Guide.restaurant.address.zipcode Z,
     Guide.restaurant.name N
```

Intuitively, we want to view *Guide.Restaurant* as a set. In SQL, the name of a relation is used as a variable that ranges over the relation. In the same spirit, we want to think of *Guide.restaurant* as a variable that ranges over the restaurants, so that two occurrences of this path expression are then bound to the same variable. The previous `from` clause is thus translated to:

```
from Guide.restaurant R,
     R.address A,
     A.zipcode Z,
     R.name N
```

The general case follows directly from this example.

4.2 Select clause

We now consider simple path expressions in the `select` clause. Two cases arise: either the entire path expression also occurs in the `from` clause or it does not.

If a path expression in the `select` clause also occurs in the `from` clause (possibly as a prefix of a longer path expression), then after translating the `from` clause we already have a variable that denotes the meaning of the path expression. It therefore suffices to replace the path expression by the corresponding variable. More precisely, the largest prefix of a path expression in the `select` clause that also occurs in the `from` clause is replaced by the variable introduced in the `from` clause for that prefix. For example, the query:

```
select Guide.restaurant
from Guide.restaurant.address.zipcode Z
where Z = 92310
```

is translated to:

```
select R
from Guide.restaurant R,
     R.address A,
     A.zipcode Z
where Z = 92310
```

Now suppose that path expression $p = X.l_1 \ldots l_n$ in the `select` clause shares a common prefix with a path expression in the `from` clause only up to label l_i, $1 \leq i < n$. Then for each assignment to the variables in the `from` clause, p returns the set of objects resulting from the path expression $v_i.l_{i+1} \ldots l_n$, where v_i is the variable assigned in the `from` clause to $v_{i-1}.l_i$, and v_{i-1} is defined similarly (by recursion). This set can be expressed in OQL by translating the remainder of p after label i to a nested `select` clause returning the result of $v_i.l_{i+1} \ldots l_n$. For example, the query:

```
select Guide.restaurant.address.zipcode
from Guide.restaurant
```

is translated to:

```
select (select Z
        from R.address A,
             A.zipcode Z)
from Guide.restaurant R
```

This query returns the set of zipcodes associated with each restaurant. Observe that for a given restaurant, the zipcodes or even the addresses of the restaurant may be empty sets, but the query does not return an error.

4.3 Where clause

Finally, we consider path expressions occurring in the `where` clause, which is the most challenging case.

As in the `select` clause, if a path expression in the `where` clause is a prefix (not necessarily strict) of some path expression in the `from` clause, we replace the path expression by the corresponding variable from the `from` clause. Now suppose the path expression is not such a prefix, and consider a simple example:

```
select Guide.restaurant
from Guide.restaurant
where Guide.restaurant.address.zipcode = 92310
```

This query compares a set of zipcodes to an integer. Thus, by the coercion rules introduced in Section 3 we get:

```
select R
from Guide.restaurant R
where exists A in R.address :
      exists Z in A.zipcode : Z = 92310
```

The query will return the restaurants that have at least one address with at least one zipcode matching 92310.

When generalizing this treatment of simple path expressions, a difficulty arises from the fact that the same simple path expression may occur more than once in the `where` clause without occurring in the `from` clause. Following our general philosophy that identical path expression prefixes should match the same data paths, we would like to have all occurrences relate to the same existentially quantified variable. For instance, consider the query:

```
select Guide.restaurant.name
from Guide.restaurant
where Guide.restaurant.address.zipcode = 92310
  or
  (Guide.restaurant.address.street = "Palm"
  and
  Guide.restaurant.address.city = "Palo Alto")
```

that returns the names of all restaurants having an address with a zipcode of 93210, or that are located on Palm in Palo Alto. One possibility is to place all existential quantifiers at the beginning of the `where` clause, as in the following query:

```
select R
from Guide.restaurant R
where exists A in R.address :
      exists Z in A.zipcode :
      exists S in A.street :
      exists C in A.city :
      (Z = 92310 or
      (S = "Palm" and C = "Palo Alto"))
```

But this solution is not satisfactory for semistructured data, since it would discard a restaurant R that has an address with zipcode 92310 in cases where the address has no street. In the above query, R would not be selected since "exists S in A.street" would fail.[6]

To overcome this difficulty, the newly introduced variables are also allowed to take the value *oemnil*. The presence of this value in any condition makes the condition false: *oemnil* = *oemnil* is false and so is *not*(*oemnil* = *oemnil*). This approach guarantees that existential quantification will not "block" the evaluation of the condition, nor will it make the condition true by "mistake" because of the nil objects. Now the (correct) translation of the previous query is:

```
select R
from Guide.restaurant R
where
  exists A in (X.address union set(oemnil)) :
  exists Z in (A.zipcode union set(oemnil)) :
  exists S in (A.street union set(oemnil)) :
  exists C in (A.city union set(oemnil)) :
  (Z = 92310 or
  (S = "Palm" and C = "Palo Alto"))
```

We conclude this section with three final topics: the implementation of simple path expressions; the sharing of path expressions between the `select` and `where` clauses; and allowing queries without a `from` clause.

[6] This problem explains why the notion of *partial object assignments* was introduced to define the semantics of Lorel1 [QRS+95a]. The remainder of this subsection essentially shows how to achieve the effect of partial object assignments in OQL.

4.4 Implementing simple path expressions

Although *oemnil* is needed for the general case, in many cases we can avoid using it. It usually suffices to "push" each existential quantifier to the innermost point in the `where` clause such that it encompasses all occurrences of its corresponding variable. For example, the query in the previous section can be translated instead to:

```
select R
from Guide.restaurant R
where exists A in R.address :
     ((exists Z in A.zipcode :
          Z = 92310) or
     ((exists S in A.street :
          S = "Palm") and
     (exists C in A.city :
          C = "Palo Alto") ) )
```

The existential quantifier for *address* needs to be placed surrounding all three conditions involving *address*, but each of the other existential quantifiers need surround only one condition. In this case, a restaurant R having an address A with a zipcode Z of 92310 would succeed in the `where` clause even if $A.street$ or $A.city$ were missing.

Unfortunately, this approach fails in certain unusual cases, as shown by the following two queries that are not equivalent:

```
select A.H
from someroot.somelabel A
where ( A.B.C = 5
        or A.D.E = 6 ) and
      ( A.B.F = 7
        or A.D.G = 8 )
---
select A.H
from someroot.somelabel A
where exists b in A.B : exists d in A.D :
      ( (exists c in b.C : c = 5) or
        (exists e in d.E : e = 6) ) and
      ( (exists f in b.F : f = 7) or
        (exists g in d.G : g = 8) )
```

The absence of a D edge always makes the bottom query false, whereas the top one is true if there are appropriate $A.B.C$ and $A.B.F$ paths. Note that adding a union with *oemnil* for each `exists` clause in the bottom query would yield the correct answer. It is possible to use a simple test to: (i) verify whether an expression is free of the pathological behavior of the last example; and (ii) if it is, push existential quantification as shown above and avoid the use of *oemnil*.

4.5 Select and where clauses

A path expression common to the `select` and the `where` clauses will use the same variable only if this path expression also occurs in the `from` clause. Consider for instance:

```
select Guide.restaurant.price
from Guide.restaurant
where Guide.restaurant.price > 25
```

This query is translated to OQL as follows:

```
select (select P from R.price P)
from Guide.restaurant R
where exists Q in R.price : Q > 25
```

A subtlety is that there is no connection between the prices in the `select` and `where` clause. All prices for a restaurant that has at least one price over 25 are retrieved, even those prices that are less than 25. To keep only those prices above 25, one must write:

```
select (select P from R.price P where P > 25)
from Guide.restaurant R
where exists Q in R.price : Q > 25
```

Observe the different roles of the two clauses: the `where` clause filters restaurants, whereas the embedded query in the `select` clause filters prices.

4.6 Omitting the from clause

Queries in Lorel need not have a `from` clause. If a `from` clause is not provided in the query, it is generated from the `select` clause by introducing a path expression in the `from` clause for each path expression in the `select` clause.[7] If the `from` clause is omitted, the `select` clause can only consist of paths originating from database names. For example:

```
select Guide.restaurant.name
where Guide.restaurant.category = "gourmet"
```

becomes

```
select Guide.restaurant.name
from Guide.restaurant
where Guide.restaurant.category = "gourmet"
```

which brings us back to familiar ground. By using simple path expressions and omitting the `from` clause, we find that straightforward queries are extremely easy to express in Lorel, and we shall express them in this manner in the remainder of the paper when it is appropriate to do so.

5 General Path Expressions

In this section, we extend the notion of simple path expressions to a more powerful syntax for path expressions, called *general path expressions*. (Note that our general path expressions are not the same as the *generalized path expressions* of [CCM96].) General path expressions allow both regular expressions and label completion to be used in paths, which provides a powerful mechanism for finding objects in the database. Disregarding the details of the syntax for the moment, examples of general path expressions are:

```
Guide.restaurant(.address)?.zipcode
Guide.restaurant.#@P.comp%.name
Guide.restaurant(.nearby)*{R}.name
```

The first expression specifies the paths starting from *Guide*, following a *restaurant* edge, then a *zipcode*, with an optional *address* in between.

Ignoring the term $@P$, the second expression specifies paths starting from *Guide* with a *restaurant* edge, followed by an arbitrary number of edges with unspecified labels (symbol #), followed by an edge having a label beginning with "comp" (comp%), and finally terminating with an edge labeled *name*. The *path variable* P is bound by $@P$ to each data path that matches "#" in this path expression.

Ignoring the term $\{R\}$, the last expression specifies all paths going through a *restaurant* edge, then an arbitrary number (symbol $*$) of *nearby* edges, and finally a *name* edge. For each data path matching this path expression, the (object) variable R is bound by $\{R\}$ to the object immediately before the name label. Note that $\{R\}$ is just a useful syntactic way to attach variables to objects in the middle of long paths.

Using general path expressions we can, e.g., obtain the name of restaurants with zipcode 92310 either in the address or directly as a field of the restaurant. Note that in this query we also employ several of the syntactic conveniences introduced in Section 4.

```
select Guide.restaurant.name
where
  Guide.restaurant(.address)?.zipcode = 92310
```

We first consider the exact syntax for specifying general path expressions, then we turn to wildcards. The last two subsections deal with the use of path and object variables within general path expressions.

It is important to note that while simple path expressions can always be translated to OQL, general path expressions cannot.

5.1 Regular expressions for paths

A *general path expression* (*gpe*), like a simple path expression, starts with an object name or a variable. General path expressions extend simple path expressions by allowing the object name or variable to be followed by one or more *gpe_components*, rather than just a sequence of labels as in simple path expressions. The syntax of a gpe_component is given by:

1. If l is a label, then $.l$ is a gpe_component.
2. If s_1 and s_2 are gpe_components, then the following are also gpe_components:

$$s_1 s_2 \quad s_1|s_2 \quad (s_1) \quad (s_1)? \quad (s_1)+ \quad (s_1)*$$

[7] We could also use path expressions in the `where` clause to generate the `from` clause, but in practice we have found that doing so is unnecessary. Note also that instead of generating the `from` clause only in cases where it is missing entirely, we could take a more general approach where we add missing components to the `from` clause based on path expressions appearing elsewhere in the query. For simplicity, we have decided against this more general approach.

3. If X is a string object, then $.unquote(X)$ is a gpe_component.

In Case 2, the symbol | is used for disjunction, ? means 0 or 1 occurrences, + means 1 or more, and $*$ means 0 or more. The $unquote(X)$ function in Case 3 takes the value of an object X (which must be coercible to a string) and uses it as a label in the path expression. So, for instance, if X contains the string "restaurant", then *Guide.unquote(X)* can be used instead of *Guide.restaurant*.

The only difficulty with our use of regular expressions here is that because of the Kleene closure ($*$), a general path expression may match an infinite number of data paths if the data is cyclic. Now, if we only care about the objects at the extremities of the paths, there are a finite number of them. However, if we also care about the paths themselves (e.g., because of the path variables considered further on), the infinite number of paths becomes an issue. In Lorel, we choose to avoid dealing with infinite sets of paths by deciding that a data path is not allowed to cross the same object twice when matching a gpe_component terminating with a $*$ or + in a general path expression. This acyclicity condition may appear artificial but it seems general enough for the applications we have considered so far, and it is easy to implement using a cycle detection mechanism. An alternative method would be to compute a finite representation of the infinite set of data paths matching a given path expression, which is possible because of the regularity of this set [Cou83]. However, this approach would seriously complicate the implementation.

5.2 Wildcards

The regular expressions specified above already allow some flexibility in querying. However, when querying semistructured data, one often does not know all of the labels of the objects or their precise relative orderings. It is therefore also useful to have a concept of "wildcards."

The first wildcard is "%," which matches 0 or more characters in a label. One can use any combination of letters, digits, or % in place of a label (i.e., in $.l$) in the definition of a gpe_component. For example, suppose we know that restaurants have a label "zip" or "zipcode," and some other label that contains a description pertaining to price. We can express the query "Find the names of cheap restaurants with zip(code) 92310" as follows:[8]

```
select Guide.restaurant.name
where Guide.restaurant.zip% = 92310 and
      Guide.restaurant.% = "cheap"
```

The second wildcard is "#". The # symbol in a path expression, .#, is shorthand for the expression "(.%)*", which is useful since it matches any data path of length 0 or more. In practice we find that # is used very, very frequently in queries.

[8] For even more flexibility, this query could use one of Lorel's built-in string matching predicates such as `like` or `grep` in place of =.

5.3 Path variables

Another important feature of general path expressions is the ability to attach variables to data paths using *path variables*. The value of a path variable is a data path in the OEM graph, i.e., a list of objects and labels. As such, the value of a path variable cannot be output in the query result. However, using path variables one can test data paths for equality, and a function, namely *path-of*, turns a data path into a single string containing the labels in the data path separated by dots.

The *path-of* function allows one to ask queries to "discover" the structure of the data. For instance, one could ask:

```
select distinct path-of(P)
from Guide.#@P.zipcode
```

(where P is a path variable) to obtain the set of paths in the database that lead to zipcode. One would obtain:

```
restaurant
restaurant.address
restaurant.nearby
restaurant.nearby.address
```

Perhaps the most practical use of path variables is to obtain the names of labels. Suppose that we want to obtain all labels leading to objects containing the string "cheap." We use the query:

```
select distinct path-of(L)
from Guide.#.%@L X
where X = "cheap"
```

This query will return label L if there is a path from *Guide* to some object X with value "cheap" and final label L. Here, L is a path variable for a path of length one, so it can be thought of as a *label variable*. Note that "`Guide.#@L X`" would instead bind L to the entire path of labels originating from *Guide*.

One last use of path variables that we will consider is as "path distinguishers," to force identical path expressions to match distinct paths in the OEM graph, as in the following query:

```
select R
from Guide.restaurant R
where R(.#.nearby)@P = R(.#.nearby)@Q
      and  P <> Q
```

This query returns all restaurants R that have two distinct paths to the same nearby restaurant.

5.4 More on object variables

Except in the sample general path expressions at the beginning of this section, so far object variables have always appeared at the end of a path expression. We now show how object variables can be introduced in the middle of a path expression, a simple feature provided primarily for syntactic convenience. The query:

```
select N
from Guide.restaurant{R}.name N
where R.category = "gourmet"
```

is equivalent to:

```
select N
from Guide.restaurant R, R.name N
where R.category = "gourmet"
```

Using object variables within path expressions is an alternative way of distinguishing between two path expressions that would otherwise be syntactically identical and thus be assigned the same variable. For example, the following query finds all restaurants with addresses in both Palo Alto and Menlo Park:

```
select N
from Guide.restaurant{R}.name N
where R.address{A1}.city = "Palo Alto" and
      R.address{A2}.city = "Menlo Park"
```

Without $A1$ and $A2$, a single existential variable would be used for address, which would always result in an empty query result (assuming a single address always has a single city).

In summary, then, a *general path expression* is a sequence $Z.q_1 \ldots q_n$, where $q_1, \ldots, q_n$ are *qualified_gpe_components* and Z is an object name or a variable denoting an object. A qualified_gpe_component is an expression of the form:

$$gpe\_component\ [\ @P\ ]\ [\{Y\}]$$

where P is an optional path variable and Y is an optional object variable. We restrict the use of path and object variables so that path variables are not allowed to appear in path expressions in the `select` clause, and the same path or object variable may not be defined in more than one path expression in a query.

6 Constructing Results

A `select-from-where` query in Lorel has the same semantics as a `select-from-where` query in SQL or OQL: it results in a bag (multiset), or in a set if the keyword `distinct` is used. In Lorel, the result is always a collection of OEM objects, and duplicate elimination is by oid. For a "top-level" query (i.e., a query that is not a nested subquery), or a query used at the top level in an assignment (see Section 7), the final collection is packaged into a single OEM object. We now explain how results are constructed in more detail.

As in SQL and OQL, for each assignment of the variables in the `from` clause that passes the condition of the `where` clause, a value is generated according to the expressions in the `select` clause. Each of these values is then coerced into an OEM object. The coercion is explained in detail below. The coercion may result in the creation of new objects and edges in the OEM graph. Thus, the query result may refer to original database objects as well as to new objects created by the coercion.

As mentioned above, the result of a top-level query is a single OEM object that is generated to hold the query result. The default name `answer` identifies this object, and edges link it to elements of the answer. For instance, the query:

```
select X
from Guide.restaurant X
```

would generate the following answer object:

```
answer &155
    restaurant &19
    restaurant &35
    restaurant &77
```

Observe that only &155 is a new object. The result of this query can be reused in later queries, although renaming is necessary (see Section 7.1) so that *answer* is not overwritten. We discuss below how the label *restaurant* is chosen for the edges leading to the elements of the answer.

Each value in the result of the `select` clause is coerced to an OEM object according to the *to_oem* coercion function specified in Table 4. Let us examine each line in Table 4. As in the previous example, the function *to_oem* does nothing to an OEM object. From an atomic value, it creates a new OEM object of the appropriate type and value using the function *new_oem*. We will return to this function when dealing with updates in Section 7. The interesting cases are: (iii) when each value returned by the `select` clause is a collection, in which case the function creates a new complex object that holds the collection with default labels (discussed below) leading to the subobjects, and (iv) when the `select` clause returns `struct` values, in which case the function creates a complex object and uses the attributes of the `struct` as labels leading to the subobjects. Because a `select` clause in Lorel containing more than one expression is interpreted as an implicit `struct` construction, case (iv) arises frequently.

Note that the coercion also applies to the result of nested subqueries. For example, if a query Q contains a nested subquery Q' in its `select` clause, the result of Q' is coerced to a set of OEM objects before coercing the result of Q. Observe also that this coercion can be extended to transform an arbitrary (portion of an) ODMG database to our Object Exchange Model. An OEM object is created for each ODMG object, and the values of the objects are coerced according to the rules in Table 4.

As an example, the following query illustrates the use of multiple expressions in a `select` clause. The query returns the names and addresses for each restaurant in *Guide*.

```
select X.name, X.address
from   Guide.restaurant X
```

The result of the query on our sample database is:

```
answer &100
   restaurant &101
      name &13 "Chef Chu"
      address &14 ...
```

Case	Result of Select expression	Coercion Function
i	OEM object o	no coercion needed
ii	atomic value v	new_oem(type,v) where type is the type of v
iii	collection V	new_oem(complex,struct(default: {to_oem(v) | v $\in$ V}))
iv	struct(a_1 : v_1,..., a_n : v_n)	new_oem(complex,struct(a_1 : w_1,..., a_n : w_n)) where w_i = { to_oem(v) | v $\in$ v_i } if v_i is a collection w_i = { to_oem(v_i) } otherwise

Table 4. Function *to_oem* coerces values to a single OEM object

```
restaurant &102
   name &17 "Saigon"
   address &23 "Mountain View"
   address &25 "Menlo Park"
restaurant &103
   name &80 "McDonald's"
```

Lorel determines the appropriate label for each element of the result at run-time. There are three cases: (1) If the object already exists in the database, then the last label on the data path that was matched by the query (causing the object to be selected by the query) is chosen. (2) If a new object is based on an existing object, e.g., by projecting some of its subobjects (as in the new object &101 above), then the label leading to the existing object is chosen. (3) If neither (1) nor (2) holds, then we use the label "default" for the new object.

7 Updates

We have now seen the novel features of Lorel for querying semistructured data. This section introduces Lorel's declarative update language. Using the update language, it is possible to create and delete database names, create a new atomic or complex object, modify the value of an existing atomic or complex object, and bulk load an OEM database. As mentioned earlier, object deletion occurs implicitly when an object becomes unreachable, so there is no explicit deletion operation.

7.1 Assigning names to objects

Names are entry points into the database and are created using the *name* statement:

```
name <name> := <expression>
```

Names also may be created while bulk loading the database, as discussed in Section 7.4 below. The expression returns a single object that is assigned to the name. Coercion is performed to coerce the expression into a single OEM object if necessary, using the function *to_oem* specified earlier in Table 4. Expressions may be queries, new object creations, or the `null` keyword. If the name does not yet exist, this statement creates a new name called `<name>`. If the name already exists then it is reassigned to the returned object.

For example, the following statement creates an entry point to the Saigon restaurant.[9]

```
name myFavorite := element
  (Select Guide.Restaurant
   where Guide.Restaurant.name = "Saigon")
```

The same name may later be reassigned as follows:

```
name myFavorite := element
  (select Guide.Restaurant
   where Guide.Restaurant.name = "Chef Chu")
```

Names are deleted by assigning them to null:

```
name myFavorite := null
```

We note again that object deletion is by unreachability (garbage collection), so a name's assignment to null may result in the deletion of some objects.

7.2 Object creation

For object creation, we use the function *new_oem*:

$$new\_oem(val\text{-}type, value) \rightarrow object$$

This function creates a single object with the specified type and value. (Objects also may be created during bulk loading, of course.) The possible value types for an object are the atomic types, e.g., `integer`, `real`, `string`, `gif`, etc., and the complex object type `complex`. Complex object values are specified as `struct`'s, where each field describes a label and a set of OEM objects for that label. Lorel also includes a second function, *load_oem*, which is used for creating "binary large objects" such as `gif` images and audio. *Load_oem* is identical to *new_oem*, except that the name of a file containing the value is given in place of the value itself.

Here are two examples of *new_oem*:

```
new_oem(int,5)
new_oem(complex ,
        struct(a:{new_oem(int,5)}, b:{X,Y}))
```

The first example constructs an integer OEM object with value 5. The second example creates a new complex object, say *o*, puts an *a* edge between *o* and a new object of value 5, and puts *b* edges between *o* and the objects named *X* and *Y*.

We allow shorthand notation in the creation of objects when the omitted information is redundant:

[9] `Element` is an OQL keyword that extracts and returns the single member of a singleton set.

1. When the value type can be deduced from the value it may be omitted. For example, 5 is inferred to be an integer.
2. Values are coerced to objects using the function *to_oem* in Table 4 if needed.
3. The `struct` constructor may be omitted.

Thus, the two examples above may be written more compactly as:

```
new_oem(5)
new_oem(a:5, b:{X,Y})
```

Note in particular that the operator *new_oem* itself may be omitted and "5" understood as "new_oem(int, 5)" after coercion.

7.3 Updates to objects

The values of objects may be modified using the `update` statement. We first consider updating single named (complex or atomic) objects and then look at updating many objects simultaneously with one construct.

Suppose that *Price* is a named atomic (integer) object. Its value may be modified using the statement:

```
update Price := 7
```

This statement changes the value inside the object identified by *Price*. After the statement, *Price* continues to identify the same object. By contrast,

```
name Price := 7
```

would create a new object containing the value 7 and then assign it to *Price*. After the statement, *Price* would identify the new object.

Updates also may increment (add to) or decrement (delete from) the value. The following example adds 1 to the *Price* value:

```
update Price += 1
```

Similarly, to decrement a value we use `-=`.

Complex objects also may be modified by changing, adding to, or deleting from the subobjects with a given label. For instance, the following update indicates that a new branch of my favorite restaurant has opened in Sunnyvale.

```
update MyFavorite.address += "Sunnyvale"
```

The general form of the `update` statement for complex objects is:

```
update <object-selector>.<label>
       (+|-|:)= <expression>
```

and the semantics for updating complex objects is defined as follows. The `<object-selector>` determines an object o to be updated. It is usually a database name, but could also be the unique object result of a query (e.g., `element(...)`). The `<expression>` identifies a set O of objects. If the operator is `+=`, then new edges are created from o to each object in O and given the label `<label>`. If the operator is `-=`, then existing edges with the label `<label>` from o to objects in O are removed. If the operator is `:=`, all edges from o with label `<label>` are removed and new edges with label `<label>` are introduced between o and each object in O.

Observe that we can change the type of an object simply by assigning it a value of a different type, an important convenience feature for semistructured data.

Now let us consider a way of modifying many objects simultaneously. We can do so using a statement of the form:

```
update P := <expression>
from <from-clause>
where <where-clause>
```

where P is a variable bound in the `from` clause. The `from` and `where` clauses are the same as in the Lorel `select` statement. The binding of the variables in the `from` and `where` clauses is done before evaluating the update, and the variables may be used in a query in the `<expression>`. Logically, the update

```
P := <expression>
```

is performed for each binding in the `from` clause that satisfies the `where` clause. We can also modify the values of multiple objects using `+=` and `-=` with this construct.

For example, the following query adds the restaurant's city as a direct subobject of the restaurant object whenever the city is Palo Alto or Menlo Park:

```
update X.city += Z
from Guide.restaurant{X}.address.city Z
where Z = "Palo Alto" or Z = "Menlo Park"
```

Finally, we observe that it takes two operations to update a label. For example, the following two statements transform all the *restaurant* labels to *eatery* labels.

```
update Guide.eatery := select Guide.restaurant
update Guide.restaurant := {}
```

7.4 Bulk loading

Lorel provides a "`load <filename>`" statement, which reads the load file `<filename>` and creates the objects described in it. In the load file, objects may be of any type. If the object is atomic, then both its type and value are given together. If the object is complex, then it is described by its subobjects, which may include other new objects created by the load file and named objects that existed prior to the load. Cyclic data is supported. A (persistent) name may be assigned to any new object as part of the load. Lorel's `load` statement can also add additional subobjects to existing named objects. The load file syntax and further details are given in [HW96].

8 Implementation on Top of an OODB

In this section, we briefly consider a prototype implementation of Lorel on top of a standard ODMG database. We first reconsider the type *OEM* defined in Section 2.

Since we have already discussed in Sections 3–6 the primary aspects of translating Lorel to OQL extended with heterogeneous objects, we only touch on a few additional issues here.

OEM objects can be implemented using the following ODMG Object Definition Language (ODL) type definition:

```
interface OEM;
interface OEMcomplex: OEM
  { attribute set
     (struct(label:string,
             values:set(OEM))) complex-value; };
interface OEMstring: OEM
  { attribute string atomic-value; }
interface OEMint: OEM
  { attribute int atomic-value; }
...
interface OEMnil: OEM;
```

Changes from the type definition in Section 2 are due to minor restrictions of ODL: (i) the internal structure of an object is a tuple and cannot simply be an atomic value, which forces us to introduce the attributes "atomic-value," and (ii) we need to represent a complex OEM object as a set of pairs (*label*, *set of values*).

The type extent for the type *OEM* is empty. Certain methods apply to all OEM objects and are therefore defined in type *OEM*, although they only have subtype instances. These are methods to obtain the value(s) of an OEM object, to compare OEM objects, to update them, etc. For example, the following method can be used to extract subobjects from complex OEM objects:

```
set(OEM) field(in string label);
```

If X is a complex OEM object, the expression $X.field$("address") returns the set of *address* subobjects of X. If X is not complex, or if it has no *address* subobjects, then the empty set is returned.

The comparators also are defined as methods. For instance, the following methods in the class OEM are used for comparing OEM objects:

```
boolean value-equal(in OEM val)
boolean equal-to-int(in int val)
boolean equal-to-string(in string val)
   ...
```

Note for instance that method `equal-to-int` is defined as false in class *OEM* but redefined in class *OEMint* as

```
self.atomic-value = val
```

and in class *OEMreal* as

```
self.atomic-value = int-to-real(val).
```

Updates also are implemented using methods. We do not consider updates that modify the type of objects since such updates are not permitted in the ODMG model. Object creation simply uses the *new* function with the type (*OEMreal*, *OEMint*, etc.) as the first argument. For atomic objects, the *new* function also takes the initial value as an argument. Complex OEM objects (which are initialized to empty) take no additional arguments.

Other update methods are the following:

```
boolean assign-real(in real new-value)
boolean assign-int(in int new-value)
...
boolean add-edges(label:string,
                  added-set:set(OEM))
boolean method remove-edges(
                  label:string,
                  removed-set:set(OEM))
```

All of these methods are defined in the class *OEM*. When one of the last two methods is applied to an object that is not complex, it has no effect on the database and simply returns false. Since we are not considering updates to an object's type, we say that an improper update (e.g., assigning a real to a complex object) also has no effect on the database and returns false. So, in particular, *assign-real* is redefined only in class `real` (with the obvious meaning).

Our current implementation on top of the O_2 system does not yet include query optimization. We conclude this section by noting that the performance of such an implementation depends heavily on two features: clustering and indexes. For clustering, the system should at least be capable of clustering an object and its subobjects together, recursively. Indexes are important for managing complex objects with many subobjects. For example, an index can be used for speeding up the evaluation of the method *field* described above.

9 The Lore System

We have implemented Lorel as the query language for our prototype database management system *Lore*. Because we are interested in exploring the many facets of managing semistructured data, Lore has been built entirely from scratch. As we have shown in the previous section, Lorel could instead be implemented on top of a conventional object-oriented DBMS. Here we discuss the architecture and query engine that comprise the Lore system. A comprehensive discussion of the Lore system is beyond the scope of this paper.

The basic architecture of Lore is depicted in Figure 4. While much of this section will focus on the query processor, we also briefly describe the textual interface, the HTML Graphical User Interface, and the object manager.

The current Lore system has two user interfaces. There is a simple textual interface, primarily used by the developers for debugging. The graphical interface, the primary interface for end users, provides powerful tools for browsing query results; a *data guide* feature for examining the structure of the data and formulating simple queries "by example;" a way of saving frequently asked queries; and mechanisms for viewing the more exotic atomic types such as `video`, `audio`, and `java`.

The object manager component, which appears just above the persistent storage component in the Lore architecture, functions as the interface between the query

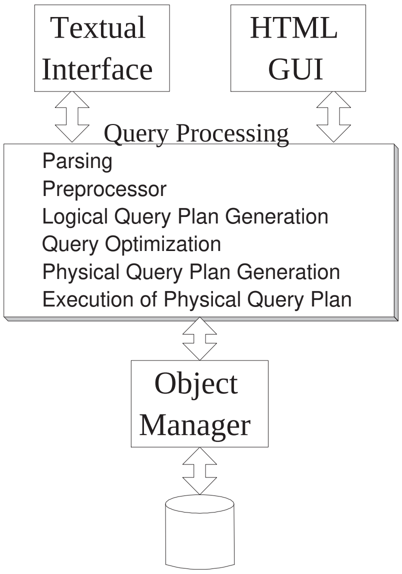

Fig. 4. Lore architecture

processor and the low-level file constructs. It supports basic primitives such as fetching an object, comparing two objects, performing simple coercion, and iterating over the subobjects of a complex object. In addition, some performance features, such as a cache of frequently accessed objects, are implemented in this component.

The query processor, which resides between the user interface and the object manager, follows these basic steps when answering a query:

1. the query is parsed,
2. the parse tree is preprocessed to translate it into an OQL-like query,
3. a logical query plan is constructed,
4. query optimization occurs,
5. the optimized logical plan is translated into a physical query plan, and
6. the physical plan is executed.

As an example, consider the following simple Lorel query:

```
select Guide.restaurant.address
where Guide.restaurant.category = "gourmet"
```

The query is parsed, then translated into an OQL-like query using the techniques described throughout this paper. The OQL-like query is:

```
select Y
from Guide.restaurant X, X.address Y
where exists Z in X.category : Z = "gourmet"
```

Then, a logical query plan is generated. A plan for our example query is shown in Figure 5. Although Lorel is based on an object-oriented data model, our query execution strategy is based primarily on familiar relational operators. The relational "tuples" we operate on are *Object Assignments*, or *OAs*. We use a recursive iterator approach in query processing, as described in, e.g., [Gra93]. We now explain how OAs are constructed and operated upon by the nodes in our logical plan.

An OA is a simple data structure containing slots corresponding to range variables in the query, along with some additional slots depending on the form of the query. For example, an OA structure for the example query is:

OA_0 Guide	OA_1 OA_0.restaurant	OA_2 OA_1.address
OA_3 OA_1.category	OA_4 Aggr	

Intuitively, each slot within an OA holds the oid of a node on a data path currently being considered by the query engine. For example, if OA_1 has the oid for a restaurant "Saigon," then OA_2 and OA_3 can hold the oids for one of Saigon's *address* subobjects and one of its *category* subobjects, respectively. Note that at a given point during query processing, it is not necessarily the case that all slots of the current OA contain a valid oid. Indeed, the function of query execution is to build complete OAs.

We now briefly explain each of the operators in Figure 5. The *Scan* operator, which is used in several leaf nodes, is similar in functionality to a relational scan. Here, however, instead of scanning over all tuples based on the name of a relation, our scan returns all oids that are subobjects of a given oid with respect to a given *gpe_component*. The *Scan* operator is defined as:

```
Scan (StartingOASlot, gpe_component,
      TargetOASlot)
```

Scan starts the search from the oid stored in the slot `StartingOASlot`, and at each iteration places into the `TargetOASlot` the oid of the next subobject that satisfies the `gpe_component`. The `gpe_component` is a string describing which labels *Scan* should match, and is similar to the syntax for *gpe_components* described in Section 5. *Scan* is called repeatedly for a given `StartingOASlot` until the `TargetOASlot` no longer holds a valid oid. When dealing with the currently supported regular expressions, the *Scan* operator uses a stack of objects that satisfy the `gpe_component`. For regular expressions, the path taken through the database to an object can be reconstructed using the *Scan* stack. In order to handle the full range of regular expressions, a finite state automaton would be needed within the *Scan* operator. As a simple example, consider the following *Scan* that appears in our example plan:

```
Scan (OA1, "address", OA2)
```

This scan iterator will place into slot OA_2, one at a time, all *address* subobjects of the oid in slot OA_1. Note the special form for the lower left *Scan*:

```
Scan (Root, "Guide", OA0).
```

Instead of using an OA slot as the first argument, the value *Root*, which is a system-known oid from which all names can be reached, is used.

Each child of a *Join* node fills in information in the current OA. Like a relational nested-loop join operator, one function of the *Join* node is to coordinate its left and right children. For each partially completed OA that the left child returns, the right child is called exhaustively until no more new OAs are possible. Then the left child is

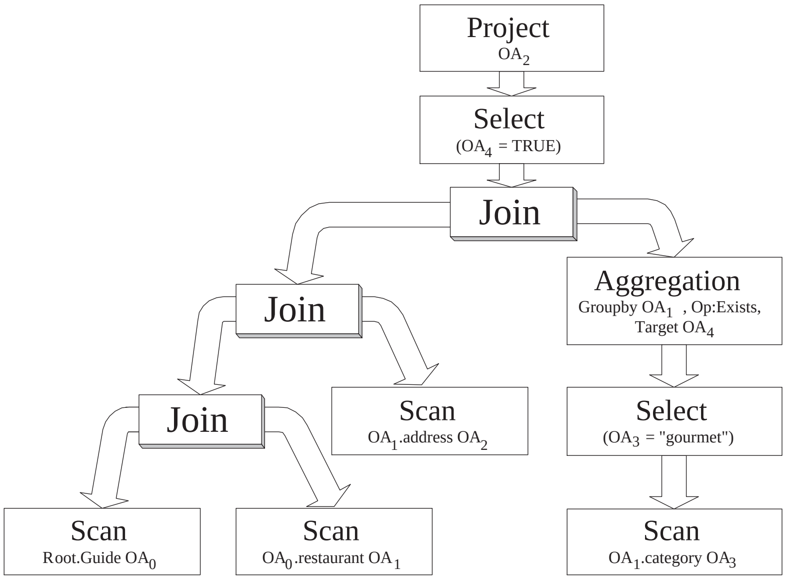

Fig. 5. Sample Lore query plan

instructed to retrieve its next (partial) OA. The iteration continues until the left side produces no more OAs.

The *Select* and *Project* nodes are nearly identical to the corresponding relational operators. The one difference is that while relational *select* and *project* deal with relation and attribute names, in Lore query plans these operators implicitly operate upon the objects identified by the oids within the current OA. Thus, the *Project* operator is used to limit which subobjects should be returned by specifying a set of OA slots, while the *Select* operator applies predicates to the objects identified in the OA slots.

The *Aggregation* node (shown in Figure 5 as the right child of the first *Join* node) is used in a somewhat novel way. Besides functioning as the standard grouping and aggregation operation, it also serves as an evaluation mechanism for quantified variables. The aggregation node groups the OAs received from its child based on the specified slot (OA_1 in the example), then applies the aggregation operator, in this case *exists*. It adds to the specified slot in the current OA (OA_4 in the example) the result of the aggregation, which here is the value *true* if the existential quantification is satisfied and *false* otherwise. Filtering of OAs whose quantification is *true* occurs in the final *Select* node. Note that the *exists* operator uses lazy evaluation to quit when it finds the first satisfying OA, while other aggregation operators need to look at all OAs in each group.

There are some fairly obvious optimizations that can be done to the logical plan in Figure 5, such as pushing the top *Select* down the right subtree and moving selection conditions into scans. In the current Lore query processor, only a few query optimization techniques are implemented and the physical query plan is very similar to the logical plan. Thus, we essentially evaluate the plan shown in Figure 5 directly. Implementation of query optimization and "real" physical plans is under design.

The Lore system includes several notable features in addition to the Lorel language. Of particular interest are the *data guide* and *external objects*:

- The data guide for a given OEM database is an OEM object that encapsulates the structure of the graph in terms of edge labels, without repeating identical paths [NUWC97]. Essentially, the data guide provides a structural summary of the current database, which in a semistructured environment can be extremely useful in understanding how the data is structured and formulating queries. In our graphical user interface, the data guide also can be used to form simple queries in a "by example" style.
- External objects allow Lore to dynamically fetch and integrate information stored in external data sources during query processing, and cache the information for later use. Any object in Lore may be a placeholder for an external object, allowing Lore to serve both as a storage repository for semistructured data and a query-driven integration engine.

9.1 System status

As of Fall, 1996, the query processor and the rest of the Lore system are functional and robust for a subset of the Lorel language. Language features whose implementation is still underway include path variables, external predicates and functions, complex select clauses, full aggregation, and the declarative update language. In addition, the complete functionality of general path expressions is not yet implemented, although a substantial and very useful subset is. While Lore currently maintains indexing structures, the query plans are not "intelligent" enough to make use of them yet. As noted above, currently little query optimization takes place, so there is a considerable amount of work to do in this area of query processing. Finally, although Lore was designed initially as a "lightweight" DBMS to be used primarily in single-user or read-only mode, as we find more and more uses for Lore we are feeling the need to add "heavyweight" features such as transactions, concurrency control, and recovery.

A Lore server with a number of sample databases is available for public use. Users can submit queries in the subset of the Lorel query language currently frozen and can experiment with features such as result browsing, data guides, and external objects. Please visit us at `http://www-db.stanford.edu/lore`.

Acknowledgements

Many thanks to all the members of the Lore research project, past and present, including Roy Goldman, Kevin Haas, Qingshan Luo, Svetlozar Nestorov, Anand Rajaraman, Hugo Rivero, Shuky Sagiv, and Jeff Ullman, and to the rest of the Stanford Database Group for many lively discussions on Lorel.

References

[Abi97] S. Abiteboul, Querying semi-structured data. In *Proceedings of the International Conference on Database Theory*, Delphy, Greece, 1997. To appear.

[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 *Proceedings of the First International Conference on Applications of Databases*, pages 267–280, Vadstena, Sweden, 1994.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann, San Francisco, California, 1992.

[BDS95] P. Buneman, S. Davidson, and D. Suciu. Programming constructs for unstructured data. In *Proceedings of the 1995 International Workshop on Database Programming Languages (DBPL)*, 1995.

[BK94] C. Beeri and Y. Kornatski. A logical query language for hypermedia systems. *Information Sciences*, 77, 1994.

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

[Cat94] R.G.G. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, San Francisco, California, 1994.

[CCM96] V. Christophides, S. Cluet, and G. Moerkotte. Evaluating queries with generalized path expressions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 413–422, Montreal, Canada, June 1996.

[CM89] M.P. Consens and A.O. Mendelzon. Expressing structural hypertext queries in graphlog. In *Proceedings of the Second ACM Conference on Hypertext*, pages 269–292, Pittsburgh, Pennsylvania, November 1989.

[Cou83] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25, 1983.

[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. 1994. Available as `http://logic.stanford.edu/sharing/papers/kif.ps`

[GR90] C. F. Goldfarb and Y. Rubinsky. *The SGML handbook*. Clarendon Press, Oxford, UK, 1990.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[HW96] K. Haas and J.L. Wiener. How to Bulk Load a Lore Database. Working Document, Stanford University Database Group, July 1996.

[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, San Diego, California, June 1992.

[KS95] D. Konopnicki and O. Shmueli. W3QS: A query system for the World Wide Web. In *Proceedings of the Twenty-First International Conference on Very Large Data Bases*, pages 54–65, Zurich, Switzerland, September 1995.

[LSS96] L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian. A declarative language for querying and restructuring the Web. In *Proceedings of the Sixth International Workshop on Research Issues in Data Engineering (RIDE '96)*, New Orleans, February 1996.

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

[MMM96] A. Mendelzohn, G. A. Mihaila, and T. Milo. Querying the world wide web, 1996. Draft.

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

[MW93] T. Minohara and R. Watanabe. Queries on structure in hypertext. In *Foundations of Data Organization and Algorithms, (FODO '93)*, pages 394–411. Springer, 1993.

[MW95] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM Journal of Computing*, 24(6), 1995.

[NUWC97] S. Nestorov, J. Ullman, J. Wiener, and S. Chawathe. Representative objects: Concise representations of semistructured hierarchical data. In *Proceedings of the Thirteenth International Conference on Data Engineering*, Cambridge, UK, April 1997. To appear.

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

[PAGM96] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proceedings of the Twenty-Second International Conference on Very Large Data Bases*, Bombay, India, 1996.

[PGMU96] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A mediation system based on declarative specifications. In *Proceedings of the Internation Conference of Data Engineering, (ICDE '96)*, pages 132–141, 1996.

[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.

[QRS+95a] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. In *Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 319–344, Singapore, December 1995.

[QRS+95b] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. Technical report, Stanford University Database Group, 1995. Document is available as `ftp://db.stanford.edu/pub/papers/querying-full.ps`.

[Raj96] A. Rajaraman. Indexing semistructured data for flexible comparisons. Working Document, Stanford University Database Group, March 1996.

[RAK+92] A. Rafii, R. Ahmed, M. Ketabchi, P. DeSmedt, and W. Du. Integration strategies in the 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.

[TMD92] J. Thierry-Mieg and R. Durbin. Syntactic definitions for the ACeDB data base manager. Technical report, MRC Laboratory for Molecular Biology, Cambridge, England, 1992.

[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*, pages 740–749, Santiago, Chile, September 1994.

A Syntax

The complete Lorel syntax appears in Figures 6 and 7. Note that not all the constructs in the language have been discussed in the body of the paper, since the paper focuses on the innovative features in Lorel.

In the grammar "{}*" means 0 or more repetitions, "{}+" means 1 or more repetitions, and "[ ]" means optional. The exception is Rule 25, where [ ] is used to delimit a character class and the following + means that a sequence of one or more characters can be drawn from the class.

Rule 19 has a higher precedence than Rule 23, meaning that a path expression consisting of multiple label expressions separated by dots is parsed as multiple qualified paths, rather than a single qualified path consisting of multiple paths.

Note that some "factoring" of the grammar has occurred to facilitate parsing, e.g., the introduction of *safe_query* and *safe_set_query*.

B Update Example

The example database can be constructed using the following updates:

```
name X := newOEM(
             category: ''fast food'',
             name: ''Mac Donald'');
name Y := newOEM(
             category: ''Vietnamese'',
             name: ''Saigon'',
             address: { ''Mountain View'',
                        ''Menlo Park''} );
name Z := newOEM(
             category: ''gourmet'',
             name: ''Coupe Chou'',
             address: newOEM(
                street: ''El Camino Real'',
                city: ''Palo Alto'',
                zipcode: 92310),
             nearby_eating_place: { X, Y});
update Y.nearby_eating_place += Z;
name Palo_Alto_Businesses :=
   newOEM(Restaurant: {X,Y,Z});
name X,Y,Z := NULL;
```

This article was processed by the author using the LaTeX style file *cljour2* from Springer-Verlag.

```
(1) query          ::=  set_query
                        | atomic_query
                        | value_query

(2) set_query      ::=  sfw_query
                        | path_expr
                        | set_query intersect set_query
                        | set_query union set_query
                        | set_query except set_query
                        | (set_query)

(3) atomic_query   ::=  var
                        | element(set_query)

(4) value_query    ::=  *atomic_query
                        | constant
                        | pathof(path_var)
                        | external function name(query_list)
                        | (query) arith_op (query)
                        | − query
                        | abs(query)
                        | aggr_function(set_query)

(5) query_list     ::=  query
                        | (query){, (query)}*

(6) sfw_query      ::=  select [ distinct ]
                        select_expr {, select_expr }*
                        [ from from_expr {, from_expr }* ]
                        [ where predicate ]

(7) select_expr    ::=  query [ as select_identifier ]
                        | select_identifier : query
                        | new_oem(select_expr {, select_expr }*)
                          [ as select_identifier ]

(8) from_expr      ::=  path_expr [ [ as ] var ]
                        | var in path_expr

(9) predicate      ::=  not predicate
                        | predicate and predicate
                        | predicate or predicate
                        | safe_query comp_op safe_query
                        | safe_set_query
                        | exists(set_query)
                        | boolean_constant
                        | exists var in
                          safe_set_query : predicate
                        | for all var in
                          safe_set_query : predicate
                        | safe_query in safe_set_query
                        | safe_query comp_op quantifier
                          safe_set_query
                        | external predicate name(query_list)
                        | (predicate)
```

Fig. 6. Lorel syntax

```
(10) safe_set_query             ::=  (set_query)
                                     | path_expr

(11) safe_query                 ::=  (query)
                                     | atomic_query
                                     | path_expr
                                     | variable
                                     | constant

(12) select_identifier          ::=  identifier
                                     | unquote(path_var)

(13) arith_op                   ::=  + | − | * | / | mod

(14) comp_op                    ::=  < | <= | = | <> | >= | >
                                     | like | grep | soundex

(15) aggr_function              ::=  min | max | count | sum | avg

(16) quantifer                  ::=  some | any | all

(17) constant                   ::=  nil
                                     | integer_literal
                                     | real_literal
                                     | quoted_string_literal
                                     | boolean_constant

(18) boolean_constant           ::=  true | false

(19) path_expr                  ::=  var {qualified_gpe_component}+

(20) qualified_gpe_component    ::=  gpe_component [ @path_var ] [ {var} ]

(21) path_var                   ::=  identifier

(22) var                        ::=  identifier

(23) gpe_component              ::=  . label_expr
                                     | gpe_component | gpe_component
                                     | gpe_component gpe_component
                                     | (gpe_component) [ regexp_op ]

(24) regexp op                  ::=  * | + | ?

(25) label_expr                 ::=  #
                                     | [A-Za-z0-9%_]+
                                     | unquote(path_var)
```

Fig. 7. Lorel syntax continued