Querying Semi-Structured Data

Serge Abiteboul*

INRIA-Rocquencourt
Serge.Abiteboul@inria.fr

1 Introduction

The amount of data of all kinds available electronically has increased dramatically in recent years. The data resides in different forms, ranging from unstructured data in file systems to highly structured in relational database systems. Data is accessible through a variety of interfaces including Web browsers, database query languages, application-specific interfaces, or data exchange formats. Some of this data is *raw* data, e.g., images or sound. Some of it has structure even if the structure is often implicit, and not as rigid or regular as that found in standard database systems. Sometimes the structure exists but has to be extracted from the data. Sometimes also it exists but we prefer to ignore it for certain purposes such as browsing. We call here *semi-structured data* this data that is (from a particular viewpoint) neither raw data nor strictly typed, i.e., not table-oriented as in a relational model or sorted-graph as in object databases.

As will seen later when the notion of semi-structured data is more precisely defined, the need for semi-structured data arises naturally in the context of data integration, even when the data sources are themselves well-structured. Although data integration is an old topic, the need to integrate a wider variety of data-formats (e.g., SGML or ASN.1 data) and data found on the Web has brought the topic of semi-structured data to the forefront of research.

The main purpose of the paper is to isolate the essential aspects of semi-structured data. We also survey some proposals of models and query languages for semi-structured data. In particular, we consider recent works at Stanford U. and U. Penn on semi-structured data. In both cases, the motivation is found in the integration of heterogeneous data. The "lightweight" data models they use (based on labelled graphs) are very similar.

As we shall see, the topic of semi-structured data has no precise boundary. Furthermore, a theory of semi-structured data is still missing. We will try to highlight some important issues in this context.

The paper is organized as follows. In Section 2, we discuss the particularities of semi-structured data. In Section 3, we consider the issue of the data structure and in Section 4, the issue of the query language.

* Currently visiting the Computer Science Dept., Stanford U. Work supported in part by CESDIS, NASA Goddard Space Flight Center; by the Air Force Wright Laboratory Aeronautical Systems Center under ARPA Contract F33615-93-1-1339, and by the Air Force Rome Laboratories under ARPA Contract F30602-95-C-0119.

2 Semi-Structured Data

In this section, we make more precise what we mean by semi-structured data, how such data arises, and emphasize its main aspects.

Roughly speaking, semi-structured data is data that is neither raw data, nor very strictly typed as in conventional database systems. Clearly, this definition is imprecise. For instance, would a BibTex file be considered structured or semi-structured? Indeed, the same piece of information may be viewed as unstructured at some early processing stage, but later become very structured after some analysis has been performed. In this section, we give examples of semi-structured data, make more precise this notion and describe important issues in this context.

2.1 Examples

We will often discuss in this paper BibTex files [Lam94] that present the advantage of being more familiar to researchers than other well-accepted formats such as SGML [ISO86] or ASN.1 [ISO87]. Data in BibTex files closely resembles relational data. Such a file is composed of records. But, the structure is not as regular. Some fields may be missing. (Indeed, it is customary to even find compulsory fields missing.) Other fields have some meaningful structure, e.g., author. There are complex features such as abbreviations or cross references that are not easy to describe in some database systems.

The Web also provides numerous popular examples of semi-structured data. In the Web, data consists of files in a particular format, HTML, with some structuring primitives such as tags and anchors. A typical example is a data source about restaurants in the Bay Area (from the Palo Alto Weekly newspaper), that we will call Guide. It consists of an HTML file with one entry per restaurant and provides some information on prices, addresses, styles of restaurants and reviews. Data in Guide resides in files of text with some implicit structure. One can write a parser to extract the underlying structure. However, there is a large degree of irregularity in the structure since (i) restaurants are not all treated in a uniform manner (e.g., much less information is given for fast-food joints) and (ii) information is entered as plain text by human beings that do not present the standard rigidity of your favorite data loader. Therefore, the parser will have to be tolerant and accept to fail parsing portions of text that will remain as plain text.

Also, semi-structured data arises often when integrating several (possibly structured) sources. Data integration of independent sources has been a popular topic of research since the very early days of databases. (Surveys can be found in [SL90, LMR90, Bre90], and more recent work on the integration of heterogeneous sources in e.g., [LRO96, QRS$^+$95, C$^+$95].) It has gained a new vigor with the recent popularity of the Web. Consider the integration of car retailer databases. Some retailers will represent addresses as strings and others as tuples. Retailers will probably use different conventions for representing dates, prices, invoices, etc. We should expect some information to be missing from some sources. (E.g., some retailers may not record whether non-automatic transmission is available). More generally, a wide heterogeneity in the organization of data should be expected from the car retailer data sources and not all can be resolved by the integration software.

Semi-structured data arises under a variety of forms for a wide range of applications such as genome databases, scientific databases, libraries of programs and more generally, digital libraries, on-line documentations, electronic commerce. It is thus essential to better understand the issue of querying semi-structured data.

2.2 Main aspects

The structure is irregular:
This must be clear from the previous discussion. In many of these applications, the large collections that are maintained often consist of heterogeneous elements. Some elements may be incomplete. On the other hand, other elements may record extra information, e.g., annotations. Different types may be used for the same kind of information, e.g., prices may be in dollars in portions of the database and in francs in others. The same piece of information. e.g., an address, may be structured in some places as a string and in others as a tuple.

Modelling and querying such irregular structures are essential issues.

The structure is implicit:
In many applications, although a precise structuring exists, it is given implicitly. For instance, electronic documents consist often of a text and a grammar (e.g., a DTD in SGML). The parsing of the document then allows one to isolate pieces of information and detect relationships between them. However, the interpretation of these relationships (e.g., SGML exceptions) may be beyond the capabilities of standard database models and are left to the particular applications and specific tools. We view this structure as implicit (although specified explicitly by tags) since (i) some computation is required to obtain it (e.g., parsing) and (ii) the correspondence between the parse-tree and the logical representation of the data is not always immediate.

It is also sometimes the case, in particular for the Web, that the documents come as plain text. Some ad-hoc analysis is then needed to extract the structure. For instance, in the Guide data source, the description of restaurant is in plain text. Now, clearly, it is possible to develop some analysis tools to recognize prices, addresses, etc. and then extract the structure of the file. The issue of extracting the structure of some text (e.g., HTML) is a challenging issue.

The structure is partial:
To completely structure the data often remains an elusive goal. Parts of the data may lack structure (e.g., bitmaps); other parts may only unveil some very sketchy structure (e.g., unstructured text). Information retrieval tools may provide a limited form of structure, e.g., by computing occurrences of particular words or group of words and by classifying documents based on their content.

An application may also decide to leave large quantities of data outside the database. This data then remains unstructured from a database viewpoint. The loading of this external data, its analysis, and its integration to the database have to be performed efficiently. We may want to also use optimization techniques to only load selective portions of this data, in the style of [ACM93]. In general, the management and access of this *external data* and its interoperability with the data from the database is an important issue.

Indicative structure vs. constraining structure:
In standard database applications, a strict typing policy is enforced to protect data. We are concerned here with applications where such strict policy is often viewed as too constraining. Consider for instance the Web. A person developing a personal Web site would be reluctant to accept strict typing restrictions.

In the context of the Lore Project at Stanford, the term *data guide* was adopted to emphasize non-conventional approaches to typing found in most semi-structured data applications. A *schema* (as in conventional databases) describes a strict type that is adhered to by all data managed by the system. An update not conforming is simply rejected. On the other hand, a *data guide* provides some information about the current type of the data. It does not have to be the most accurate. (Accuracy may be traded in for simplicity.) All new data is accepted, eventually at the cost of modifying the data guide.

A-priori schema vs. a-posteriori data guide:
Traditional database systems are based on the hypothesis of a fixed schema that has to be defined prior to introducing any data. This is not the case for semi-structured data where the notion of schema is often posterior to the existence of data.

Continuing with the Web example, when all the members of an organization have a Web page, there is usually some pressure to unify the style of these home-pages, or at least agree on some minimal structure to facilitate the design of global entry-points. Indeed, it is a general pattern for large Web sources to start with a very loose structure and then acquire some structure when the need for it is felt.

Further on, we will briefly mention issues concerning data guides.

The schema is very large:
Often as a consequence of heterogeneity, the schema would typically be quite large. This is in contrast with relational databases where the schema was expected to be orders of magnitude smaller than the data. For instance, suppose that we are interested in Californian Impressionist Painters. We may find some data about these painters in many heterogeneous information sources on the Web, so the schema is probably quite large. But the data itself is not so large.

Note that as a consequence, the user is not expected to know all the details of the schema. Thus, queries over the schema are as important as standard queries over the data. Indeed, one cannot separate anymore these two aspects of queries.

The schema is ignored:
Typically, it is useful to ignore the schema for some queries that have more of a discovery nature. Such queries may consist in simply browsing through the data or searching for some string or pattern without any precise indication on where it may occur. Such searching or browsing are typically not possible with SQL-like languages. They pose new challenges: (i) the extension of the query languages; and (ii) the integration of new optimization techniques such as full-text indexing [ACC+96] or evaluation of generalized path expressions [CCM96].

The schema is rapidly evolving:
In standard database systems, the schema is viewed as almost immutable, schema updates as rare, and it is well-accepted that schema updates are very expensive.

Now, in contrast, consider the case of genome data [DOB95]. The schema is expected to change quite rapidly, at the same speed as experimental techniques are improved or novel techniques introduced. As a consequence, expressive formats such as ASN.1 or ACeDB [TMD92] were preferred to a relational or object database system approach. Indeed, the fact that schema evolves very rapidly is often given as the reason for not using database systems in applications that are managing large quantities of data. (Other reasons include the cost of database systems and the interoperability with other systems, e.g., Fortran libraries.)

In the context of semi-structured data, we have to assume that the schema is very flexible and can be updated as easily as data which poses serious challenges to database technology.

The type of data elements is eclectic:

Another aspect of semi-structured data is that the structure of a data element may depend on a point of view or on a particular phase in the data acquisition process. So, the type of a piece of information has to be more eclectic as, say in standard database systems where the structure of a record or that of an object is very precise. For instance, an object can be first a file. It may become a BibTex file after classification using a tool in the style of [TPL95]. It may then obtain *owner*, *creation-date*, and other fields after some information extraction phase. Finally, it could become a collection of reference objects (with complex structures) once it has been parsed. In that respect also, the notion of type is much more flexible.

This is an issue of objects with multiple roles, e.g., [ABGO93] and objects in views, e.g., [dSAD94].

The distinction between schema and data is blurred:
In standard database applications, a basic principle is the distinction between the schema (that describes the structure of the database) and data (the database instance). We already saw that many differences between schema and data disappear in the context of semi-structured data: schema updates are frequent, schema laws can be violated, the schema may be very large, the same queries/updates may address both the data and schema. Furthermore, in the context of semi-structured data, this distinction may even logically make little sense. For instance, the same classification information, e.g., the sex of a person, may be kept as data in one source (a boolean with *true* for male and *false* for female) and as type in the other (the object is of class *Male* or *Female*). We are touching here issues that dramatically complicate database design and data restructuring.

2.3 Some issues

To conclude this section, we consider a little more precisely important issues in the context of semi-structured data.

Model and languages for semi-structured data:
Which model should be used to describe semi-structured data and to manipulate this data? By languages, we mean here languages to query semi-structured data but also languages to restructure such data since restructuring is essential for instance to integrate data coming from several sources. There are two main difficulties (i) we have only a partial knowledge of the structure; and (ii) the structure is potentially "deeply nested" or even cyclic. This second point in particular defeats calculi and algebras developed in the standard database context (e.g., relational, complex value algebra) by requiring recursion. It seems that languages such as Datalog (see [Ull89, AHV94]) although they provide some form of recursion, are not completely satisfactory.

These issues will be dealt with in more details in the next two sections.

Extracting and using structure:
The general idea is, starting with data with little explicit structure, to extract structuring information and organize the data to improve performance. To continue with the bibliography example, suppose we have a number of files containing bibliography references in BibTex and other formats. We may want to extract (in a data warehousing style) the titles of the papers, lists of authors and keywords, i.e., the most frequently accessed data that can be found in every format for references, and store them in a relational database. Note that this extraction phase may be difficult if some files are structured according to formats ignored by our system. Also, issues such as duplicate elimination have to

be faced. In general, the issue of recognizing an object in a particular state or within a sequence of states (for temporal data) is a challenging issue.

The relational database then contains links to pieces of information in the files, so that all data remains accessible. Such a structured layer on top of a irregular and less controlled layer of files, can provide important gains in answering the most common queries.

In general, we need tools to extract information from files including classifiers, parsers, but also software to extract cross references (e.g., within a set of HTML documents), information retrieval packages to obtain statistics on words (or groups of words) occurrences and statistics for relevance ranking and relevance feedback. More generally, one could envision the use of general purpose data mining tools to extract structuring information.

One can then use the information extracted from the files to build a structured layer above the layer of more unformed data. This structured layer references the lower data layer and yields a flexible and efficient access to the information in the lower layer to provide the benefits of standard database access methods. A similar concept is called *structured map* in [DMRA96].

More ways to use structure: the data guide

We saw that many differences with standard databases come from a very different approach to typing. We used the term *data guide* to stress the differences. A similar notion is considered in [BDFS97]. Now, since there is no schema to view as a constraint on the data, one may question the need for any kind of typing information, and for a data guide in particular. A data guide provides a computed loose description of the structure of data. For instance, in a particular application, the data guide may say that *persons* possibly have ougoing edges labelled *name, address, hobby* and *friend*, that an *address* is either a `string`, but that it may have outgoing edges labelled *street*, and *zipcode*. This should be viewed as more or less accurate indications on the kind of data that is in the database at the moment.

It turns out that there are many reasons for using a data guide:

1. *graphical query language*: Graphical interfaces use the schema in very essential ways. For instance, QBE [Zlo77] would present a query frame that consists of the names of relations and their attributes. In the context of semi-structured data, one can view the data guide as an "encompassing type" that would serve the role of a type in helping the user graphically express queries or browse through the data.
2. *cooperative answer*: Consider for instance the mistyping of a label. This will probably result in a type error in a traditional database system, but not here since strict type enforcement is abandoned. Using a data guide, the system may still explain why the answer is empty (because such label is absent from the database.
3. *query optimization*: Typing information is very useful for query optimization. Even when the structure is not rigid, some knowledge about the type (e.g., presence/absence of some attributes) can prove to be essential. For instance, if the query asks for the Latex sources of some documents and the data guides indicate that some sources do not provide Latex sources, then a call to these sources can be avoided. This is also a place where the system has to show some flexibility. One of the sources may be a very structured database (e.g., relational), and the system should take advantage of that structure.

The notion of the *data guide associated to* some particular data with various degrees of accuracy, its use for expressing and evaluating queries, and its maintenance, are important directions of research.

System issues:

Although this is not the main focus of the paper, we would like to briefly list some system issues. We already mentioned the need for new query optimization techniques, and for the integration of optimization techniques from various fields (e.g., database indexes and full text indexes). Some standard database system issues such as transaction management, concurrency control or error recovery have to be reconsidered, in particular, because the notion of "data item" becomes less clear: the same piece of data may have several representations in various parts of the system, some atomic, some complex. Physical design (in particular clustering) is seriously altered in this context. Finally, it should be observed that, by nature, a lot of the data will reside outside the database. The optimization of external data access (in particular, the efficient and selective loading of file data) and the interoperability with other systems are therefore key issues.

3 Modeling Semi-Structured Data

A first fundamental issue is the choice of a model: should it be very rich and complex, or on the contrary, simple and lightweight? We will argue here that it should be *both*.

Why a lightweight model? Consider accessing data over the Internet. If we obtain new data using the Web protocol, the data will be rather unstructured at first. (Some protocols such as CORBA [OMG92] may provide a-priori more structured data.) Furthermore, if the data originates from a new source that we just discovered, it is very likely that it is structured in ways that are still unknown to our particular systems. This is because (i) the number of semantic constructs developers and researchers may possibly invent is extremely large and (ii) the standardization of a complex data model that will encompass the needs of all applications seems beyond reach.

For such novel structures discovered over the network, a *lightweight* data model is preferable. Any data can be mapped to this *exchange* model, and becomes therefore accessible without the use of specific pieces of software.

Why also a heavyweight data model? Using a lightweight model does not preclude the use of a compatible, richer model that allows the system to take advantage of particular structuring information. For instance, traditional relations with indexes will be often imported. When using such an indexed relation, ignoring the fact that this particular data is a relation and that it is indexed would be suicidal for performance.

As we mentioned in the previous section, the types of objects evolve based on our current knowledge possibly from totally unstructured to very structured, and a piece of information will often move from a very rich structure (in the system where it is maintained); to a lightweight structure when exchanged over the network; to a (possibly different) very rich structure when it has been analyzed and integrated to other pieces of information. It is thus important to dispose of a flexible model allowing both a very light and a very rich structuring of data.

In this section, we first briefly consider some components of a rich model for semi-structured data. This should be viewed as an indicative, non-exhaustive list of candidate features. In our opinion, specific models for specific application domains (e.g., Web databases or genome databases) are probably more feasible than an all-purpose model for semi-structured data. Then, we present in more details the Object Exchange Model that is pursuing a minimalist approach.

3.1 A maximalist approach

We next describe primitives that seem to be required from a semantic model to allow the description of semi-structured data. Our presentation is rather sketchy and assumes knowledge of the ODMG model. The following primitives should be considered:

1. The ODMG model: the notions of objects, classes and class hierarchy; and structuring constructs such as set, list, bag, array seem all needed in our context.
2. Null values: these are given lip service in the relational and the ODMG models and more is needed here.
3. Heterogeneous collections: collections need often to be heterogeneous in the semi-structured setting. So, there is the need for some union types as found for instance in [AH87] or [AK89].
4. Text with references: text is an important component for semi-structured information. Two important issues are (i) references to portions of a text (references and citations in LaTex), and (ii) references from the text (HTML anchors).
5. Eclectic types: the same piece of information may be viewed with various alternative structures.
6. Version and time: it is clear that we are often more concerned by querying the recent changes in some data source that in examining the entire source.

No matter how rich a model we choose, it is likely that some weird features of a given application or a particular data exchange format will not be covered (e.g., SGML exceptions). This motivates the use of an underlying minimalist data format.

3.2 A minimalist approach

In this section, we present the Object Exchange Model (OEM) [AQM$^+$96], a data model particularly useful for representing semi-structured data.

The model consists of graph with labels on the edges. (In an early version of the model [PGMW95], labels were attached to vertices which leads to minor differences in the description of information and in the corresponding query languages.) A very similar model was independently proposed in [BDHS96]. This seems to indicate that this model indeed achieves the goals to be simple enough, and yet flexible and powerful enough to allow describing semi-structured data found in common data sources over the net. A subtle difference is that OEM is based on the notion of objects with object identity whereas [BDHS96] uses tree markers and *bisimulation*. We will ignore this distinction here.

Data represented in OEM can be thought of as a graph, with objects as the vertices and labels on the edges. Entities are represented by *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`. Figure 1 provides an example of an OEM graph.

OEM can easily model relational data, and, as in the ODMG model, hierarchical and graph data. (Although the structure in Figure 1 is *almost* a tree, there is a cycle via objects &19 and &35.) To model semi-structured information sources, we do not insist that data is as strongly structured as in standard database models. Observe 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; (iv) the zipcode occurs in the address for some and directly under restaurant for others; and (v) price information is sometimes given and sometimes missing.

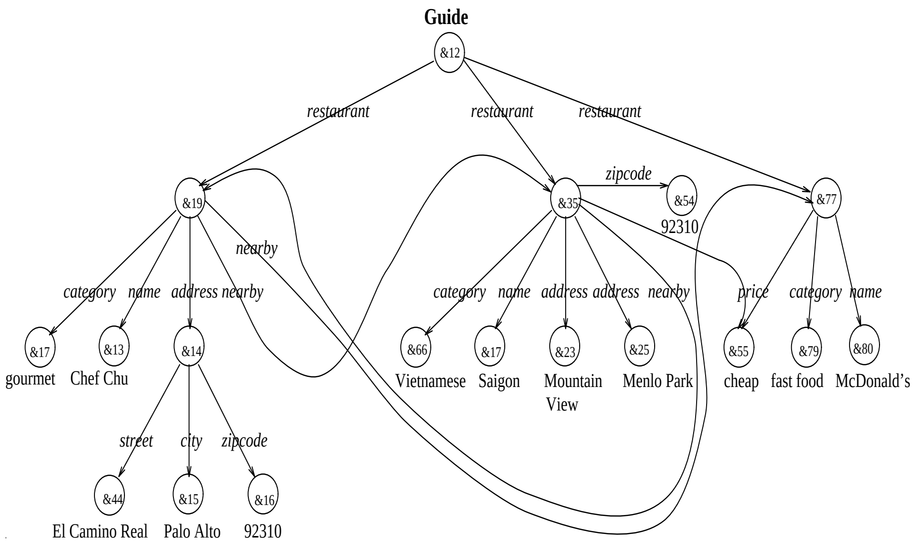

Fig. 1. An OEM graph

We conclude this section with two observations relating OEM to the relational and ODMG models:

OEM vs. relational: One can view an OEM database as a relational structure with a binary relation *VAL(oid, atomic_value)* for specifying the values of atomic objects and a ternary relation *MEMBER(oid, label, oid)* to specify the values of complex objects. This simple viewpoint seems to defeat a large part of the research on semi-structured data. However, (i) such a representation is possible *only* because of the presence of object identifiers, so we are already out of the relational model; (ii) we have to add integrity constraints to the relational structure (e.g., to prohibit dangling references); and (iii) it is often the case that we want to recover an object together with its subcomponents and this recursively, which is certainly a feature that is out of relational calculus.

OEM vs. ODMG: In the object exchange model, all objects have the same type, namely OEM. Intuitively, this type is a tuple with one field per possible label containing a set of OEM's. Based on this, it is rather straightforward to have a type system that would incorporate the ODMG types and the

OEM type (see [AQM+96]). This is a first step towards a model that would integrate the minimalist and maximalist approaches.

4 Querying and Restructuring

In the context of semi-structured data, the query language has to be more flexible than in conventional database systems. Typing should be more liberal since by nature data is less regular. What should we expect from a query language?

1. standard database-style query primitives;
2. navigation in the style of hypertext or Web-style browsing;
3. searching for pattern in an information-retrieval-style [Rie79];
4. temporal queries, including querying versions or querying changes (an issue that we will ignore further on);
5. querying both the data and the type/schema in the same query as in [KL89].

Also, the language should have sound theoretical foundations, possibly a logic in the style of relational calculus. So, there is a need for more works on calculi for semi-structured data and algebraizations of these calculi.

All this requires not only revisiting the languages but also database optimization techniques, and in particular, integrating these techniques with optimization techniques from information retrieval (e.g., full text indexing) and new techniques for dealing with path expressions and more general hypertext features.

There has been a very important body of literature on query languages from various perspectives, calculus, algebra, functional, and deductive (see [Ull89, AHV94]), concerning very structured data. A number of more recent proposals concern directly semi-structured data. These are most notably Lorel [AQM+96] for the OEM model and UnQL [BDHS96] for a very similar model. Although developed with different motivations, languages to query documents satisfy some of the needs of querying semi-structured data. For instance, query languages for structured documents such as OQL-doc [CACS94] and integration with information retrieval tools [ACC+96, CM95] share many goals with the issues that we are considering. The work on query languages for hypertext structures, e.g., [MW95, BK90, CM89b, MW93] and query languages for the Web are relevant. In particular, query languages for the Web have attracted a lot of attention recently, e.g., W3QL [KS95] that focuses on extensibility, WebSQL [MMM96] that provides a formal semantics and introduce a notion of locality, or WebLog [LSS96] that is based on a Datalog-like syntax. A theory of queries of the Web is proposed in [AV97].

W3QL is typical from this line of works. It notably allows the use of Perl regular expressions and calls to Unix programs from the `where` clause of an SQL-like query, and even calls to Web browsers. This is the basis of a system that provides bridges between the database and the Web technology.

We do not provide here an extensive survey of that literature. We more modestly focus on some concepts that we believe are essential to query semi-structured data. This is considered next. Finally, we mention the issue of data restructuring.

4.1 Primitives for querying semi-structured data

In this section, we mention some recent proposals for querying semi-structured data.

Using an object approach: The notion of objects and the flexibility brought by an object approach turn out to be essential. Objects allow to focus on the portion of the structure that is relevant to the query and ignore portions of it that we (want to) ignore.

To see that, consider first the relational representation of OEM that was described in Section 3.2 and relational query languages. We can express simple queries such as *what is the address of Toto?* even when we ignore the exact structure of *person* objects, or even if all persons do not have the same structure:

```
select unique V'.2
from   persons P, MEMBER N, MEMBER A, VAL V, VAL V'
where  P = N.1 and P = A.1 and
       N.2 = "name" and N.3 = V.1 and V.2 = "Toto" and
       A.2 = "address" and A.3 = V'.1
```

assuming a unary relation *persons* contains the oid's of all persons. Observe that this is only assuming that persons have names and addresses.

In this manner, we can query semi-structured data with almost no knowledge on the underlying structure using the standard relational model. However, the expression of the query is rather awkward. Furthermore, this representation of the data results in losing the "logical clustering" of data. The description of an object (a tuple or a collection) is split into pieces, one triplet for each component. A more natural way to express the same query is:

```
Q1   select A  from persons P, P.address A
     where  "Toto" = P.name
```

This is actually the correct OQL syntax; but OQL would require *persons* to be an homogeneous set of objects, fitting the ODMG model. On the other hand, Lorel (based on OEM) would impose no restriction on the types of objects in the *persons* set and Q1 is also a correct Lorel query. In OEM, *persons* object will be allowed to have zero, one or more names and addresses. Of course, the Lorel query Q1 will retrieve only persons with a name and an address. Lorel achieves this by an extensive use of coercion.

Using coercion: A simple example of coercion is found with atomic values. Some source may record a distance in kilometers and some in miles. The system can still perform comparison using coercion from one measure to the other. For instance, a comparison $X < Y$ where X is in kilometer and Y in miles is coerced into $X < mile\_to\_km(Y)$.

The same idea of coercion can be used for structure as well. Since we can neither assume regularity nor precise knowledge of the structure, the name or address of a person may be atomic in some source, a set in other sources, and not be recorded by a third. Lorel allows one to use Q1 even in such cases. This is done by first assuming that all properties are set-valued. The empty set (denoting the absence of this property) and the singleton set (denoting a functional property) are simply special cases. The query Q1 is then transformed by coercing the equality in *P.Name = "Toto"* into a set membership *"Toto" in P.Name*.

So, the principle is to use a data model where all objects have the same interface and allow a lot of flexibility in queries. Indeed, in Lorel, all objects have the same type, OEM.

Path expressions and Patterns: The simplest use of path expressions is to concatenate attribute names as in "Guide.restaurant.address.zipcode". If Guide is a tuple, with a restaurant field that has an address field, that has a zipcode field, this is pure field extraction. But if some properties are set-valued (or all are set-valued as for OEM), we are in fact doing much more. We are traversing collections and flattening them. This is providing a powerful form of navigation in the database graph. Note that now such a path expression can be interpreted in two ways: (i) as the set of objects at the end of the paths; and (ii) as the paths themselves. Languages such as OQL-doc [CACS94] consider paths as first class citizen and even allow the use of path variables that range over concrete paths in the data graph.

Such simple path expressions can be viewed as a form of browsing. Alternatively, they can be viewed as specifying certain line patterns that have to be found in the data graph. One could also consider non-line patterns such as `person { name , ss# }`, possibly with variables in the style of the psi-terms [AKP93].

Extended path expressions: The notion of path expression takes its full power when we start using it in conjunction with wild cards or path variables. Intuitively, a sequence of labels describes a directed path in the data graph, or a collection of paths (because of set-valued properties). If we consider a regular expression of the alphabet of labels, it describes a (possibly infinite) set of words, so again a set of paths, i.e., the union of the paths described by each word. Indeed, this provides an alternative (much more powerful way) of describing paths.

Furthermore, recall that labels are string, so they are themselves sequences of characters. So we can use also regular expressions to describe labels. This is posing some minor syntactic problems since we need to distinguish between the regular expressions for the sequence of labels and for the sequence of characters for each label. The approach taken in Lorel is based on "wild cards". We briefly discuss it next.

To take again an example from Lorel, suppose we want to find the names and zipcodes of all "cheap" restaurants. Suppose we don't know whether the zipcode occurs as part of an address or directly as subobject of restaurants. Also, we do not know if the string "cheap" will be part of a category, price, description, or other subobject. We are still able to ask the query as follows:

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

The "?" after *.address* means that the address is optional in the path expression. The wild-card "%" will match any label leading a subobject of restaurant. The comparison operator `grep` will return true if the string "cheap" appears anywhere in that subobject value. There is no equivalent query in SQL or OQL, since neither allow regular expressions or wild-cards.

This last example seems again amenable to a relational calculus translation although the use of a number of % wildcards may lead to some very intricate relational calculus equivalent, and so would the introduction of disjunction. Note that the Kleene closure in label sequences built in path expressions in [AQM+96] and OQL-doc [CACS94] takes immediately out of first order. For instance, consider the following Lorel query:

```
select t  from MyReport.#.title t
```

where "#" is a shorthand for for a sequence of arbitrary many labels. This returns the title of my report, but also the titles of the section, subsections, etc., no matter how deeply nested.

The notion of path expression is found first in [MBW80] and more recently, for instance, in [KKS92, CACS94, AQM+96]. Extended path expressions is a very powerful primitive construct that changes the languages in essential ways. The study of path expressions and their expressive power (e.g., compared to Datalog-like languages) is one of the main theoretical issues in the context of semi-structured data. The optimization of the evaluation of extended path expressions initiated in [CCM96] is also a challenging problem.

Gluing information and rest variables: As mentioned above, a difficulty for languages for semi-structured data is that collections are heterogeneous and that often the structure of their components is unknown. Returning to the *persons* example, we might want to say that we are concerned only with *persons* having a name, an address, and possibly other fields. MSL [PGMW95] uses the notion of *rest* variables to mention "possibly other fields" as for instance in:

```
res(name:X, address:Y; REST1) :- r(name:X, address:Y; REST1),
                                 Y = (city:"Palo Alto"; REST2)
```

Here r is an collection of heterogeneous tuples. The first literal in the body of the rule will unify with any tuple with a *name* and *address*. The $REST1$ variable will unify with the remaining part of the tuple. Observe that this allows filtering the tuples in r without having to specify precisely their internal structure.

This approach is in the spirit of some works in the functional programming community to allow dealing with heterogeneous records, e.g, [Wan89, CM89a, Rem91]. One of the main features is the use of extensible records that are the basis of inheritance for objects as records. However, the situation turns out to be much simpler in MSL since: (i) there is much less emphasis on typing; and (ii) in particular, it is not assumed that a tuple has at most one l-component for a given label l.

Object identity is also used in MSL [PAGM96] to glue information coming from possibly heterogeneous various objects. For instance, the following two rules allow to merge the data from two sources using *name* as a surrogate:

```
&person(X) ( name:X, ATT:Y ) :- r1 ( name:X, ATT:Y )
&person(X) ( name:X, ATT:Y ) :- r2 ( name:X, ATT:Y )
```

Here *&person(X)* is an object identifier and ATT is a variable. Intuitively, for each tuple in *r1* (or *r2*) with a name field X, and some ATT field Y, the object $\&person(X)$ will have an ATT field with value Y. Observe the use of object identity as a substitute for specifying too precisely the structure. Because of object identity, we do not need to use a notion such as $REST$ variable to capture in one rule instantiation all the necessary information.

We should observe again that these can be viewed as Datalog extensions that were introduced for practical motivations. Theoretical result in this area are still missing.

4.2 Views and restructuring

Database languages are traditionally used for *extracting* data from a database. They also serve to specify *views*. The notion of view is particularly important here since we often want to consider the same object from various perspectives or with various precisions in its structure (e.g., for the integration of heterogeneous data). We need to specify complex restructuring operations. The view technology developed for object databases can be considered here, e.g., [dSAD94]. But we dispose of much less structure to start with when defining the view and again, arbitrarily deep nesting and cycles pose new challenges.

Declarative specification of a view: Following [dSAD94], a view can be defined by specifying the following: (i) how the object population is modified by hiding some objects and creating virtual objects; and how the relationship between objects is modified by hiding and adding edges between objects, or modifying edge labels.

A simple approach consists of adding hide/create vertices/edges primitives to the language and using the core query language to specify the vertices/edges to hide and create. This would yield a syntax in the style of:

```
define view Salary with
       hide select P.salary from persons P
            where P.salary > 100K
       virtual add P.salary := "high" from persons P
            where P.salary > 100K
```

For vertex creation one could use a Skolem-based object naming [KKS92].

The declarative specification of data restructuring for semi-structured data is also studied in [ACM97].

A more procedural approach A different approach is followed in [BDHS96] in the languages UnQL and UnCAL. A first layer of UnQL allows one to ask queries and is in the style of other proposals such as OQL-doc or Lorel, e.g., it uses wild cards. The language is based on a comprehension syntax. Parts of UnQL are of a declarative flavor. On the other hand, we view the restructuring part as more procedural in essence. This opinion is clearly debatable.

A particular aspect of the language is that it allows some form of restructuring even for cyclic structures. A *traverse* construct allows one to transform a database graph while traversing it, e.g., by replacing all labels A by the label A'. This powerful operation combines tree rewriting techniques with some control obtained by a guided traversal of the graph. For instance, one could specify that the replacement occurs only if particular edge, say B, is encountered on the way from the root.

A lambda calculus for semi-structured data, called UnCAL, is also presented in [BDHS96] and the equivalence with UnQL is proven. This yields a framework for an (optimized) evaluation of UnQL queries. In particular, it is important to be able to restructure a graph by local transformations (e.g., if the graph is distributed as it is the case in the Web). The locality of some restructuring operations is exploited in [Suc96].

Acknowledgements This paper has been quite influenced by discussions on semi-structured data with many people and more particularly with Peter Buneman, Sophie Cluet, Susan Davidson, Tova Milo, Dallan Quass, Yannis Papakonstantinou, Victor Vianu and Jennifer Widom.

References

[ABGO93] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *VLDB*, 1993.

[ACC+96] S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, and Jerome Simeon. Querying documents in object databases. Technical report, INRIA, 1996.

[ACM93] S. Abiteboul, S. Cluet, and T. Milo. Querying and updating the file. In *Proc. VLDB*, 1993.

[ACM97] S. Abiteboul, S. Cluet, and T. Milo. Correspondence and translation for heterogeneous data. In *Proc. ICDT*, 1997.

[AH87] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Trans. on Database Systems*, 12:4:525–565, 1987.

[AHV94] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, Reading-Massachusetts, 1994.

[AK89] S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 159–173, 1989. to appear in *J. ACM*.

[AKP93] Hassan Ait-Kaci and Andreas Podelski. Towards a meaning of Life. *Journal of Logic Programming*, 16(3-4), 1993.

[AQM+96] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The lorel query language for semistructured data, 1996. `ftp://db.stanford.edu//pub/papers/lorel96.ps`.

[AV97] S. Abiteboul and V. Vianu. Querying the web. In *Proc. ICDT*, 1997.

[BDFS97] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proc. ICDT*, 1997.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *SIGMOD*, San Diego, 1996.

[BK90] C. Beeri and Y. Kornatski. A logical query language for hypertext systems. In *VLDB*, 1990.

[Bre90] Y. Breitbart. Multidatabase interoperability. *Sigmod Record*, 19(3), 1990.

[C+95] M.J. Carey et al. Towards heterogeneous multimedia information systems: The Garlic approach. In *Proc. RIDE-DOM Workshop*, 1995.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *SIGMOD'94*. ACM, 1994.

[CCM96] V. Christophides, S. Cluet, and G. Moerkotte. Evaluating queries with generalized path expressions. In *SIGMOD*, Canada, June 1996.

[CM89a] Luca Cardelli and John C. Mitchell. Operations on records. In *Proceedings of the Fifth Conference on the Mathematical Foundations of Programming Semantics*. Springer Verlag, 1989.

[CM89b] Mariano P. Consens and Alberto O. Mendelzon. Expressing structural hypertext queries in graphlog. In *Proc. 2nd. ACM Conference on Hypertext*, Pittsburgh, 1989.

[CM95] M. Consens and T. Milo. Algebras for querying text regions. In *Proc. on Principles of Database Systems*, 1995.

[DMRA96] L.M.L. Delcambre, D. Maier, R. Reddy, and L. Anderson. Structured maps: Modelling explicit semantics over a universe of information, 1996. unpublished.

[DOB95] S.B. Davidson, C. Overton, and P. Buneman. Challenges in integrating biological data sources. *J. Computational Biology 2*, 1995.

[dSAD94] C. Souza dos Santos, S. Abiteboul, and C. Delobel. Virtual schemas and bases. In *Intern. Conference on Extending Database Technology*, Cambridge, 1994.

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

[ISO87] ISO. Specification of astraction syntax notation one (asn.1), 1987. Standard 8824, Information Processing System.
[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *SIGMOD*, 1992.
[KL89] M. Kifer and G. Lausen. F-logic: A higher-order language for reasoning about objects. In *sigmod*, 1989.
[KS95] D. Konopnicki and O. Shmueli. W3QS: A query system for the World Wide Web. In *VLDB*, 1995.
[Lam94] L. Lamport. *Latex*. Addison-Wesley, 1994.
[LMR90] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *Computing Surveys*, 22(3), 1990.
[LRO96] A. Levy, A. Rajaraman, and J.J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. VLDB*, 1996.
[LSS96] Laks V. S. Lakshmanan, Fereidoon Sadri, and Iyer N. Subramanian. A declarative language for querying and restructuring the Web. In *RIDE*, New Orleans, February 1996. In press.
[MBW80] J. Mylopoulos, P. Bernstein, and H. Wong. A language facility for designing database-intensive applications. *ACM Trans. on Database Sys.*, 5(2), June 1980.
[MMM96] A. Mendelzohn, G. A. Mihaila, and T. Milo. Querying the world wide web, 1996. draft, available by ftp: milo@math.tau.ac.il.
[MW93] T. Minohara and R. Watanabe. Queries on structure in hypertext. In *Foundations of Data Organization and Algorithms, FODO '93*. Springer, 1993.
[MW95] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comp.*, 24(6), 1995.
[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 *VLDB*, Bombay, 1996.
[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Data Engineering*, Taipei, Taiwan, 1995.
[QRS+95] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. Technical report, Stanford University, December 1995. Available by anonymous ftp from `db.stanford.edu`.
[Rem91] D. Remy. Type inference for records in a natural extension of ml. Technical report, INRIA, 1991.
[Rie79] C.J. Van Riejsbergen. *Information retrieval*. Butterworths, London, 1979.
[SL90] A. Sheth and J. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *Computing Surveys*, 22(3), 1990.
[Suc96] D. Suciu. Query decomposition and view maintenance for query languages for unstructured data. In *Proc. VLDB*, 1996.
[TMD92] J. Thierry-Mieg and R. Durbin. Syntactic definitions for the acedb data base manager. Technical report, MRC Laboratory for Molecular Biology, Cambridge, CB2 2QH, UK, 1992.
[TPL95] M. Tresch, N. Palmer, and A. Luniewski. Type classification of semistructured data. In *Proc. of Intl. Conf. on Very Large Data Bases*, 1995.
[Ull89] J.D. Ullman. *Principles of Database and Knowledge Base Systems, Volume I,II*. Computer Science Press, 1989.
[Wan89] M. Wand. Complete type inference for simple objects. In *Proceedings of Symp. on Logic in Computer Science*, 1989.
[Zlo77] M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16:324–343, 1977.

This article was processed using the LaTeX macro package with LLNCS style