Modeling and Querying Semi-Structured Data

Sophie Cluet

INRIA, Domaine de Voluceau, 78153 Le Chesnay, France

1 Introduction

Since its creation in 1989, the Web has known a tremendous success due to its capacity to deliver very inexpensively information world wide.

Introduced as a mean to link the academic research community, the Web now evolves rapidly in three directions:

- Internet applications in which we find electronic commerce, marketing or technical support; this kind of applications enables commercial companies to deliver their products, information about them or services twenty-four hours a day, seven days a week.
- Intranet applications such as CIS (Corporate Information Systems) uses the same technology as Internet applications but isolated behind a corporate network, data is kept away from inquisitive eyes.
- Extranet applications use the internet infrastructure to build cross-domain secure applications. These applications are good candidates to use emerging technologies such as CORBA, DCOM, Internet tunneling in conjunction with HTML or JAVA applets.

The Web is thus no longer used as a support for hyperlinked individual pages but as an infra-structure for running world wide or corporate applications.

Common requirements can be pointed out for these applications.

- Web applications involve a very large amount of inter-related data that require some level of consistency.
- They are highly interactive. This implies strong security and concurrency mechanisms.
- Another obvious requirement is to allow users to find the information they are looking for. This enforces the need for structuring the information and providing declarative query languages.

Building applications with standard Web tools is a laborious task. The information is stored in poorly structured HTML pages, consistency and concurrency are managed by *ad hoc* programs, navigation is the only mean to access information and it can be a very tedious process. Furthermore, programs have to access data through complex protocols such as CGI or proprietary Web servers API.

Given the requirements stated above, database technology should be a good candidate to support Web applications. As a matter of fact, the database vendors and research community have been very active in this field in the last few years. The typical scenario for Web applications using the database technology is the following: data are collected and stored in a database system, applications written on top of this system and output data formatted in HTML to be displayed on the Web. This scenario raises mainly three issues that concern (i) gathering information, (ii) importing and managing new kinds of data, often poorly structured (commonly called semi-structured [1]) and (iii) exporting data on the Web.

Gathering Information

Few database vendors have studied the problem of collecting Web data. Most probably, the reason for this lack of interest is that the applications targeted by database vendors are those of their current customers who already possess the information they plan to broadcast on the Internet. However, there exists a market of data providers whose job is to provide information relevant to some topic, information that has to be collected over various sources including the Web.

The database research community has been very active in that domain (see for example [10, 19, 26, 27, 29, 37]). The postulate at the basis of most of these research projects is that search engines as found on the Web (e.g., Altavista [18], Yahoo [40]) are only partial answers to the problem of gathering information. Although they are fast and cover most of the available data, (i) they cannot be adapted easily to the requirements of a specific user, (ii) their query language is poor, and (iii) they often return too many answers, badly ordered and mainly irrelevant. The advocated solutions usually rely on database-like Web query languages and customized structured views.

Importing and Managing Semi-Structured Information

The industrial answer to the problem of Web data import (either virtual or real) mainly relies on Corba [31] or ODBC (e.g., O_2[30], Versant[38]). The programmers are thus in charge of specifying the interfaces allowing the import of the various (potentially heterogeneous) data. Once the data is stored in the database system, the programmers are supposed to used standard database tools, sometimes enhanced (notably with full text index capacities [30]), to write their applications.

Whereas the database vendors have, somehow, skirted the problem of data heterogeneity (it is up to the programmer to solve it), the research community has mainly tackled this specific problem. A few experiments have shown that database models are not always suitable when one has to integrate heterogeneous and poorly structured data. The reason for this is their lack of flexibility. New models have thus been proposed, integration models in which the notion of type/schema disappears or is loosen (see for example [9, 13, 33, 34, 35, 36, 37]). Each of these models supports a query language. Some migration tools (e.g., [2, 8, 12, 13]) and optimization techniques are also proposed (e.g., [8, 14]).

Exporting Data on the Web

Most database vendors now provide tools allowing automatic display on the Web of database stored information (see for example O_2 [30] and Oracle [32]). These tools generate HTML pages in a rather simple way. For instance, an object = an HTML page, a complex value = an HTML list of items, an atomic value = a string, a reference to some other object = a URL, a URL = a query on the database. However, this default mapping is sometimes inadequate and appropriate customization is then either difficult or impossible. The outward simplicity of the Web exportation problem may explain the disinterest of the database research community. To our knowledge, there exist few studies on this topic ([2, 37]).

The readers should now be convinced that Web applications raise many interesting problems for database researchers. This paper does not develop all of them but concentrates on the problems of modeling and querying (efficiently) semi-structured data. It is by no means exhaustive. It compares various works with some experiments led by the VERSO project at INRIA in the context of three European projects: AQUARELLE, OPAL and WIRE.

2 Modeling Data

A Web application usually deals with data found in various systems using different formats, e.g., HTML[21], SGML[23], or LaTex [28] for text, ASN.1 [22] or STEP [5] for technical data, relational data [16], JPEG[39] or MPEG[24] for images, RealAudio[7] for sound.

Most of these formats can easily be integrated in a commercial database system either because the system supports the format or because one can define a mapping to the database format. In the best case (the database model is more general than the imported data format), this mapping has no loss of information (e.g., JPEG data can easily be mapped to the "Bits" type found in most commercial systems, STEP data have a structure similar to those found in ODMG systems). The data is then well structured using the database constructors (relations, lists, tuples, etc.). In other cases (some characteristics of the imported model cannot be captured by the target model), information is lost in the process or the database model is used in a degenerated way.

The ODMG-like schema below illustrates this using the SGML format. SGML, among other things, features a choice connector, that can be seen as a union type and that complicates its import in a database system. For instance, one may specify that a paragraph is a list of elements that are either text or figures or references to other documents. This either-or construction is common to many formats, notably the simple and widely used BNF format. Instead of capturing the structure relevant to a class of documents (as is usually the case in database applications), the following schema captures the (pseudo-)syntactical tree of any SGML document. In other words, it models the imported data representation model.

```
Class Element type tuple (name: string,
                          attributes: list(Attribute),
                          content: list(Element))
Class Attribute type tuple (name: string,
```

```
value: string)
```

Figure 1 shows an instance of this schema.

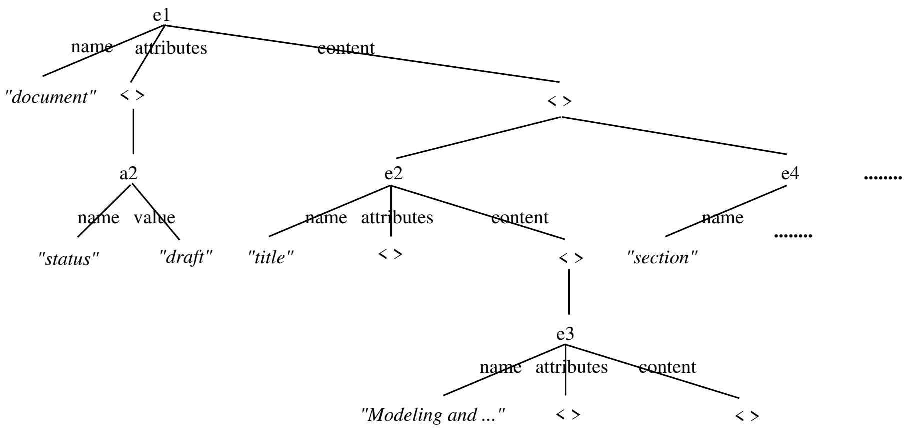

Fig. 1. Database View of a Syntactical Tree

As can be noticed, the structure of the document (i.e., the fact that a document is composed of a title, a status, some sections with titles, etc.) is part of the data itself. A consequence of this degenerated use of a database schema is that database tools are no longer adequate. As an illustration, one may try to formulate the following query using a database query language.

What are the titles contained in my document?"

To palliate this, two solutions have been considered. Both rely on the assumption that the difficulty in representing semi-structured data in a database system is due to the lack of flexibility of database models. The first solution advocates the effective support of union types in database systems. The second solution is more radical. The idea is to build new (lightweight) systems (from scratch or on top of existing ones) without types or schema. We now illustrate these two approaches.

2.1 The Union Type Approach

To capture data heterogeneity, one can use union types. This approach adds union types to an existing object-oriented database system in order to give it the flexibility required by semi-structured data. It has been followed by the Verso group at INRIA [12, 13]. In a first phase, it was used in the context of the AQUARELLE project whose goal is to manage, using Web technologies, the cultural inheritance of the European countries. Ongoing work on that model is done in the context of two Esprit projects: WIRE (Web Information Repository for the Enterprise) and OPAL (Integrated Information and Process Management In Manufacturing Engineering).

Despite of their rare presence in database systems, union types exist in various programming languages (among others, C and ML) as well as in theoretical database models (e.g., [3]). The idea is to introduce flexibility in the type system while keeping some measure of type safety and efficiency in terms of space allocation. Type safety is usually obtained through marked unions which require explicit access to the instantiated component of a union typed data and thus allow static typing of the programs. Memory is saved by allocating the space required by the largest member of the union and not the sum of the various members size.

When asked about the absence of union types in their systems, most database vendors answer that they can be either (i) captured by inheritance with appropriate methods or (ii) implemented using the tuple constructor. These *ad hoc* solutions are hardly satisfactory. They present the following drawbacks.

1. One way to implement union types using inheritance is the following: a union type between two classes C_1 and C_2 is represented by a class C super-class of both C_1 and C_2. Then, to access the C_1 component of a union typed object, one cast the object to C_1. This solution is inadequate.
   Inheritance is defined on classes (as opposed to types) and requires some structural compatibility. In that context, defining data as being either real or string requires some rather inelegant manipulation.
   In most systems, classes cannot be generated dynamically[1]. Complex types can. For example, an OQL query usually creates one or more new types and we would like new union types to be created as well.
2. OODB systems do not allow a redefinition of the operations associated to tuple types. Thus, using tuples to implement union types forces the programmer to check (by hand) the correctness of his/her programs given the semantics he/she has in mind and that may not be shared by his/her fellow programmers.
   Furthermore, the solution is greedy in terms of allocated space and there is no way to distinguish system generated tuples (e.g., as an answer to a query) from system generated union types.

Given this and the need for more flexibility, [13] proposes a model with marked union types that is an extension of IQL[3]. We assume here that the readers are familiar with object models without union types (e.g., the ODMG[11] model, but any model supported by an OO system would do) and simply (i) define marked union types, (ii) their domain and (iii) the main sub-typing rules involving marked union.

1. if $\tau_1, \ldots, \tau_n$ are distinct types and $a_1, \ldots, a_n$ are distinct attribute names, $(a_1 : \tau_1 + \ldots + a_n : \tau_n)$ is a (union) type.

[1] Classes have to be defined using a data description language preliminary to object creation

2. $dom(a_1 : \tau_1 + \ldots + a_k : \tau_k) = \{[a_i : v_i] \mid 1 \leq i \leq k \wedge \ v_i \in \ dom(\tau_i)\}$.
3. – if $\forall i \in [1..n], \exists j \in [1..m] : a_i = b_j \wedge \tau_i \leq \tau_j^{'}$
   $(a_1 : \tau_1 + \ldots + a_n : \tau_n] \leq (b_1 : \tau_1^{'} + \ldots + a_m : \tau_m^{'})$.
   – if $\exists j \in [1..m] : a_i = b_j \wedge \tau_i \leq \tau_j^{'}$
   $[a_i : \tau_i] \leq (b_1 : \tau_1^{'} + \ldots + b_m : \tau_m^{'})$.

The following schema with a union type captures the SGML document of Example 1 in a more natural way.

```
Name Documents: set(Document)
Class Document
  tuple (title: string, status: string, sections: list(Section))
Class Section
  tuple (title: string, paragraphs: list(Paragraph))
Class Paragraph
  list (text: string + figure: Figure + reference: Document)
```

Figure 2 represents a partial instance of that schema.

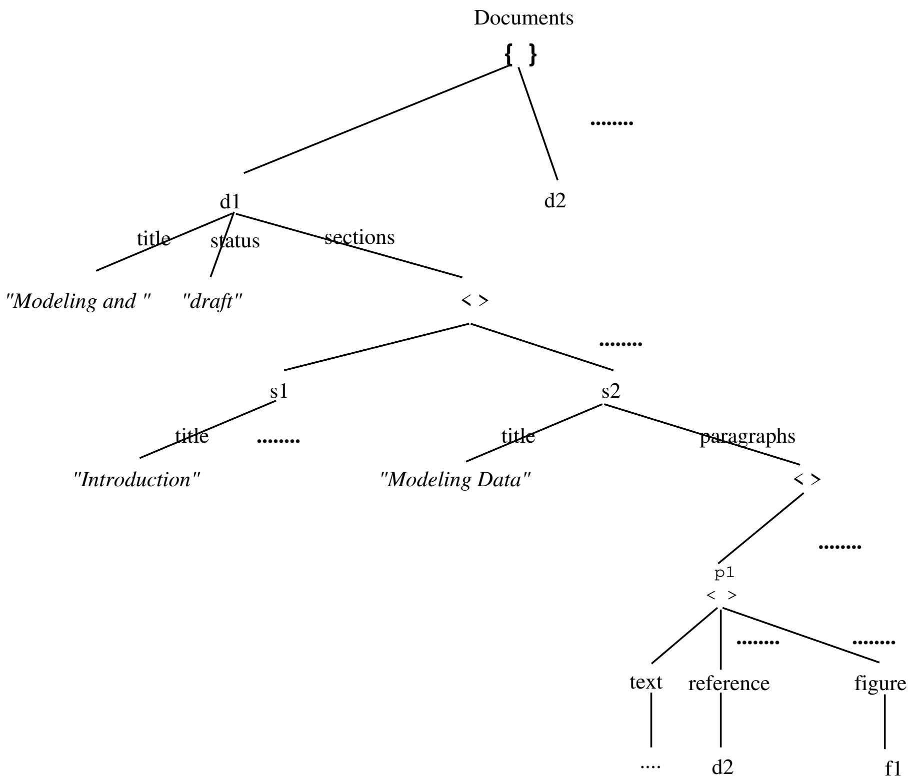

Fig. 2. Database View of a Set of Documents

We are back to a more traditional use of a database schema which captures the structure of the stored data. We will see in the next sections that this representation allows easy interrogation and that standard administration tools (e.g., clusters, indexes) can also be used (even though they might be insufficient in some cases). Thus, the main advantage of this approach is that its implementation requires few modifications of a database system.

However, it also has some drawbacks.

1. It is not clear whether union types are the panacea to all problems raised by the integration of heterogeneous data. It is reasonable to think that we will find data on the Web that will require some (too many?) manipulations in order to be modeled. Only an experimental study can validate/invalidate this approach.
2. Before accessing data of a union type, one has to check which component is instantiated. Thus, if too frequently used, union types may seriously downgrade the performance of a database application.

2.2 The No-Schema Approach

The No-Schema approach is the most popular [4, 8, 9, 37] for modeling semi-structured data. We illustrate it using two representative projects: the Lore system[4] of Stanford University and the system built by the database team at U. Pennsylvania[8, 9].

The approach consists in building new systems that do not support the notion of schema/type. Data is represented as a labeled graph/tree and new data can be added in a very flexible way by simply adding a branch to the existing graph/tree (i.e., no type restriction). Let us now study the two models.

The OEM (Object Exchange Model) Model of Lore manipulates objects. Each object has a unique identifier. Some objects are atomic, others are complex: their value is a set of object references, denoted by a set of $(label, edge)$ pairs. The labels are string. Figure 3 illustrates this. $d1$ is the identifier of an object whose complex value is a set containing, among others, the pair ("title", $a1$). $a1$ is the identifier of an atomic object whose value is "Modeling and ...".

More formally, an OEM instance of R, R being a set of names, 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 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 some object $name(N)$, N in R.

The tree model of [9] is defined recursively in the following way.

- {} is the empty tree

- $\{l \Longrightarrow t\}$ is the tree whose root has an edge labeled l towards the sub-tree t.
- $t_1 \cup t_2$ is the tree constructed by merging the roots of both t_1 and t_2 trees.

It is further extended to support cycles (i.e., graphs) but we will study this simple form here. Figure 4 illustrates the definition.
$\{title \Longrightarrow \{Modeling\ and.... \Longrightarrow \{\}\}\}$ is a linear tree.
The tree representing the whole document is the union of this linear tree with other trees.
We introduced "key" components to represent the references between document.

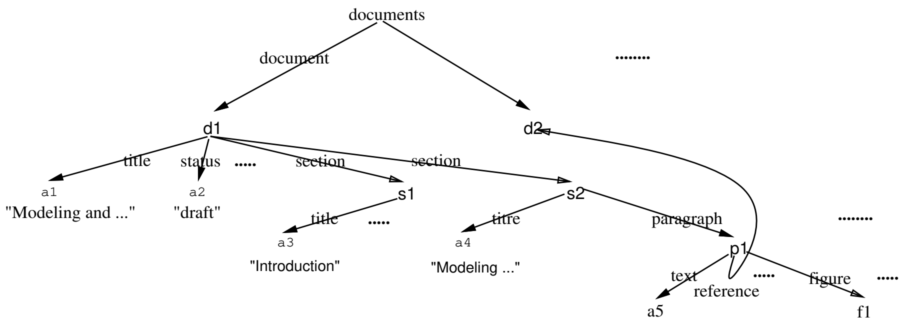

Fig. 3. An OEM Tree ([4])

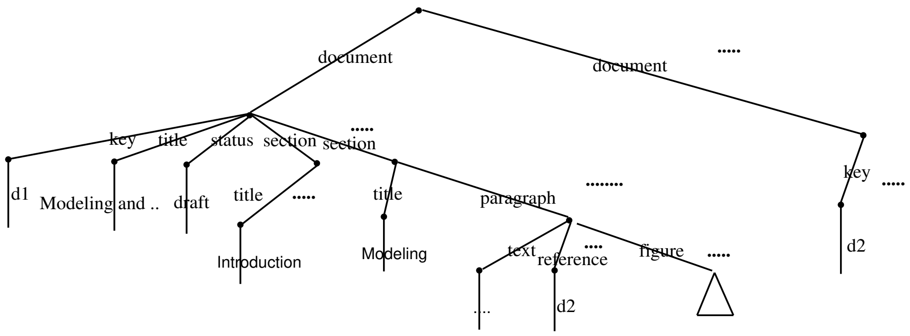

Fig. 4. A Data Tree According to [9]

As can be noted, the OEM model, as opposed to that of [9], features node identifiers. They can be used as an alternative to model cyclic references. Another difference is that only the edges contain information in [9] whereas [4] favors both edges (for the structural information) and nodes (for atomic information).

The main advantages of the No-Schema approach are its simplicity and its flexibility. The integration of a new format is straightforward. We will see in the sequel, that these new models also support declarative languages.

However, and this will be illustrated in Section 4, the No-Schema approach has an important drawback. It entails a dynamic evaluation of queries. In a traditional database system, query evaluation is performed in two phases. First, the query is statically analyzed, using schema information and language semantics, in order to generate an execution plan. Second, the plan is evaluated without unnecessary access to schema information. Let us consider for instance a query that returns the main title of the documents contained in a database. A static analysis of the corresponding database schema will allow to retrieve the address of all documents as well as the offset to the attribute *title* in these documents. In a model without schema this is no longer feasible and the information has to be found dynamically and, in the absence of appropriate structure, may require a scan over the whole database.

This leads to an open and interesting issue: what new index structures will allow query optimization in the No-Schema context?

3 Querying Data

Once the data is stored in some system, one would like to access/query it. Let us first consider the union type approach. Accessing a member of a marked union type is similar to accessing an attribute of a tuple type. Thus, there is no *a priori* need for query features other than those found in a standard database system.

However, a Web application potentially contains highly heterogeneous information whose structure will probably be incompletely known by the programmers. On the other hand, standard database query languages require a complete knowledge of the schema in order to formulate a query. Consequently, there is a need to extend database languages in order to offer a more flexible access to the stored data.

It is then interesting to note that the POQL language [12, 13] that was designed to palliate this lack of flexibility, offers many similarities with the Lorel language [4] that was designed for the OEM model. Both languages rely on generalized path expressions (GPE) to query data with no or partial knowledge of the schema[2].

Other query languages for tree/graph models are those of [9, 8]. The first relies on structural recursion and differs a lot syntactically and semantically from the two previous languages. The second is more similar in syntax to that of [13, 4] and will just be briefly commented here.

In this section, we do not present the Lorel language. The reason is that it is very similar in syntax to POQL and that its semantics is defined in Section 4.

[2] GPE's are not new. They were introduced in different contexts [6, 25]

We first present POQL (with some references to Lorel), then the language from [9] will be presented .

3.1 Languages with GPE's

POQL[13] is an extension of OQL, the ODMG standard query language[11]. This design choice seems natural since the language is intended to support an object-oriented model with union types. OQL is a functional language. It is defined by a set of basic functions/queries and a way of building new ones through composition and iterators. The language iterators have an SQL-like syntax. The functional nature of OQL makes it easily expendable. One simply has to add new basic functions.

POQL extends OQL by adding (i) new textual boolean operators and (ii) generalized path expressions. We do not elaborate on the first extension: the textual operators are those found in most information retrieval systems (e.g., *contains* - does a text contain a word or a group of words?, *near* - is a word near another in a document?). The second extension is more significant since GPE's are primitives to query data with no or little knowledge of the schema, an essential feature in the semi-structured context.

A standard path expression is a combination of elementary access to object/value components (e.g., $.sections[0].title$). POQL GPE's extends these by adding to the model, on top of standard data variables, two new kinds of variables: attribute and path variables. For example, $.\#A@P(V)$ is a GPE containing an attribute variable ($\#A$, the symbol # prefixes attribute variables), a path variable ($@P$, the symbol @ prefixes path variables) and a data variable (V). If we apply this GPE to the object $d1$ of Example 2, one possible instantiation of the variables could be $\#A = sections$, $@P = [0].title$ and $V =$ "$Modeling...$". Adding these new variables implies adding the corresponding data types in the model along with their corresponding domains. The domain of type *Attribute* is the set of all attribute names. The domain of type *Path* is given below along with the syntactical definition of a GPE.

Domain of type Path

1. ϵ (the empty path) is in $dom(Path)$;
2. if a is in $dom(Attribute)$, then $\cdot a$ (access to an attribute) is in $dom(Path)$;
3. if i is in $dom(integer)$, then $[i]$ (access to the element of a list) is in $dom(Path)$;
4. if p is in $dom(Path)$ and d is an object/value, then $p(d)$ (access to a value/object) is in $dom(Path)$.
5. if p is in $dom(Path)$ and d is an object/value, then $p\{d\}$ (access to the element of a collection) is in $dom(Path)$.
6. if p is in $dom(Path)$, then pq (path combination) is in $dom(Path)$.

Some example of paths are given below:

```
[0].title
.sections[0].title(``Modeling'')
.sections{s2}.paragraphs{p1}
```

Syntax of a GPE

Let $\mathcal{A}$ be the set of attribute variables taking their values in $dom(Attribute)$, and $\mathcal{P}$ the set of path variables taking their values in $dom(Path)$. The set E_G of GPE basic elements is constructed as follows:

$E_G = \{.a \mid a \in dom(Attribute)\} \cup \{.A \mid A \in \mathcal{A}\} \cup \{P \mid P \in \mathcal{P}\}$
$\cup\{(V) \mid V \in \mathcal{V}\} \cup \{\{V\} \mid V \in \mathcal{V}\} \cup \{\{\}\ \}$

Then, $\mathcal{G}$, the set of generalized path expressions is defined as follows:

- $\epsilon \in \mathcal{G}$ (the empty GPE);
- $\forall e \in E_G, e \in \mathcal{G}$ (the elementary GPE);
- $\forall g, g' \in \mathcal{G}, gg' \in \mathcal{G}$ (composition).

Applied on some value/object, a GPE returns the set of all "correct" valuations of the variables it contains. Each valuation is represented by a tuple with one attribute per variable. Thus, used in the *from* clause of an OQL query, a GPE allows to iterate over the different values of these variables. The notion of "correctness" for a valuation is defined below.

Correct Valuations

A valuation ν of a GPE g is said to be correct according to some data (value or object) d, iff:

- ν maps data variable to values/objects, path variables to paths and attribute variables to attributes.
- $d\ \nu(g)$ is a concrete path (see below) and $d\ \nu(g) \neq \perp$.

A concrete path corresponds to the application of a path p to some data value d. The value associated to a concrete path is defined as follows.

1. if $p = \epsilon$ then $d\ p = d$.
2. if $p = .a$, then if $d \leq [a : d']$ (note that this takes care of union types) or d is an object whose attribute/property a is equal to d',
   - then $d\ p = d'$
   - else $d\ p = \perp$.
3. if $p = [i], i \in [0..n]$, then if $d =< d_0, \ldots, d_n >$
   - then $d\ p = d_i$
   - else $d\ p = \perp$.
4. if $p = (d')$, then if $d = d'$
   - then $d\ p = d'$
   - else $d\ p = \perp$.
5. if $p = \{d_i\}, i \in [1..n]$, then if $d = \{d_1, \ldots, d_n\}$ of $d =< d_1, \ldots, d_n >$
   - then $d\ p = d_i$
   - else $d\ p = \perp$.
6. if $p = p_1 p_2$ and $d\ p_1 = d'$ then $d\ p = d'p2$

Note that this definition does not forbid cyclic paths. Thus, potentially, the set of valuation is infinite. The implementation of the valuation function avoids that by not considering path that traverses twice the same class. This is, of course, very restrictive. A better solution avoids cycle at the data level rather than at the schema level.

Let us now illustrate the language using some examples.

Q1 : what are the titles of the documents?

Supposing that one has a perfect knowledge of the database schema, this query can be formulated in OQL in the following way:

select d.title
from d in Documents
union
select s.title
from d in Documents, s in d.sections

The two subqueries of *Q1* correspond to the two attributes named *title* in the database schema. Clearly, if we are interested in an attribute occurring more often in the schema, OQL is not a good solution. Furthermore, we have to check the database schema beforehand to be sure that we do not forget any occurrences. Now, let us consider the corresponding POQL query.

select t
from Documents @P.title (t)

The *from* clause contains a generalized path expression. The path variable $@P$ is defined as being a path starting from the set of documents and ending in some object/value with a *title* attribute/property whose value will be assigned to the data variable t.

Let us now suppose that paragraphs have an attribute named *title* whose type is not a string but an object of class $Text$. Then, variable t may be instantiated with data from two different types (string and $Text$). This implies the use of union types for GPE's data variables. This has two important consequences. (i) Since system generated union markers cannot be known by the user, the language must give implicit access to the component of a union type. For example, t *like*"$Web*$" must be interpreted as "t is instantiated with a string and, assuming a is the union marker corresponding to the string type, $t.a$ *like*"$Web*$"". (ii) There is a need to perform an analysis of the query to reduce the generated union type. Considering the same example, we can reduce the type of t to a string. These problems are more complex than it may appear and are treated in [12]. Note that they do not exist in Lorel since there is no typing process. Below is the Lorel query corresponding to *Q1*.

select Documents.document ♯.title

From the database root, one accesses all edges labeled *document*, then all possible paths are followed (denoted by the ♯ symbol) until one reaches an edge labeled *title*. The query returns a tree consisting of a new root with as many outgoing edges as *title*'s edges found following any path from a document.

Let us now consider query *Q2*.

Q2: what are the documents containing the word "Web"?

Again, supposing that one has a complete knowledge of the schema and some time ahead of him, this query can be expressed in OQL.

select distinct d
from d in Documents,
s in d.sections,
p in Paragraphs
where d.title like "*Web*" or
s.title like "*Web*" or
.... or p.text->contains("Web")

The corresponding POQL and Lorel expressions are simpler:

POQL

select d
from Documents{d} @P(e)
where e contains "*Web*"

Lorel

select Documents.document
where Documents.document ♯ contains "*Web*"

It is interesting to note that this query, although simple, may require a full traversal of the database. Standard database index cannot be used. Apart from the fact that they do not support high-level textual predicates, they are specified on fixed paths and their cost forbids a total coverage of the database.

We end this presentation with a query that cannot be formulated in OQL[3] and that is formulated in the same way in Lorel and POQL.

Q3: What structural difference is there between the current and the previous versions of my document?

We suppose that the names *MyDocument* and *MyOldDocument* represent, respectively, the current and previous versions of my document.

select @P
from MyDocument @P
minus
select @P
from MyOldDocument @P

The result of this query could be:

```
d.sections[0].paragraphs[4]
d.sections[3]
d.sections[3].title
```

3.2 Structural Recursion as a Query Language

Traditionally, construction and traversal of a tree are done recursively. Most of the time, the recursion follows the same pattern (either breadth or depth first). Some variations of that pattern allow to stop the recursion at "interesting" places or at a certain depth.

[3] The fact that it cannot be expressed in OQL is due to the absence of type *Path* in the ODMG model.

Based on that, [9] proposes a language for querying trees using a simple recursive pattern, that of the recursive definition of their model given in the previous section. The language is based on three simple functions that allow the recursive traversal of a data tree. The traversal is guided by the tree structure, thus the name of "structural recursion". The first function (ext) stops the recursion at the first level of the tree. The second function ($text$) traverses the whole tree. The last function ($text'$) allows a conditional traversal (i.e., stops in the recursion are possible).

1. Given a function $f : label \times tree \rightarrow tree$, ext applies f in the following way:
   - $ext(\{\}) = \{\}$
   - $ext(\{l \Longrightarrow t\}) = f(l,t)$ /* end of the recursion */
   - $ext(t_1 \cup t_2) = ext(t_1) \cup ext(t_2)$
2. Given a function $f : label \times tree \rightarrow tree$, $text$ applies f in the following way:
   - $text(\{\}) = \{\}$
   - $text(\{l \Longrightarrow t\}) = f(l, text(t))$ /* recursive call */
   - $text(t_1 \cup t_2) = text(t_1) \cup text(t_2)$
3. Given a function $f : label \times tree \times tree \rightarrow tree$ $text'$ applies f in the following way:
   - $text'(\{\}) = \{\}$
   - $text'(\{l \Longrightarrow t\}) = f(l, t, text'(t))$ /* treatment of t before recursive call */
   - $text'(t_1 \cup t_2) = text'(t_1) \cup text'(t_2)$

We now reformulate Queries *Q1* and *Q2* to illustrate this language. Given the language description[9], we did not find a way to formulate Query *Q3*.

Q1 : What are the titles of the documents ?

$text'(\lambda(l,t,r).$ **if** (l='title) **then** t **else** r)(Documents)

Q2 : What are the documents containing the word "Web" ?

$text'(\lambda(l,t,r).$ **if** (l = 'document)
    then (**if** [$text(\lambda(l2, r2).$ **if** ($l2$ contains 'Web)
            then $l2 \Longrightarrow \{\}$
            else $r2$) (t) = {}]
        then {}
        else t)
    else r) (Documents)

Obviously, this language lacks "user-friendliness". It was originally designed to support optimization of the tree traversals. In [8], the authors propose a new SQL-like language. Its semantics is based on a calculus called UnCal which can itself be captured using structural recursion.

4 Optimizing Queries

We now consider the problem of query evaluation. As mentioned in Section 2, the Union-Type approach is more advantageous in terms of optimization opportunities since it is possible to work on schema and language before considering data. As a matter of fact, it allows all the optimization techniques that have been developed in the OO context. However, standard OO optimization does not consider GPE's and we will see in this survey that GPE's do require new techniques.

Although difficult, we will then show that optimization in the No-Schema context is possible.

4.1 Optimizing GPE's with Schema

To our knowledge, GPE's were first introduced in [6] along with a (naive) evaluation technique that was also adopted in [13]. The technique implements somehow an intuitive understanding of GPE's:

1. look for all possible instantiations of attribute and path variables,
2. replace the attribute and path variables by their instantiations,
3. eliminate the not well-typed alternatives,
4. union the remaining instantiated queries,
5. perform standard optimization and evaluate the resulting query.

In other words, applied on the POQL formulation of Query *Q1*, Step 4 of this technique results in the corresponding OQL query. This solution has the following drawbacks:

- The number of subqueries explodes exponentially in the number of attribute or path variables, but also in the number of attribute of each tuple type involved.
- If a distinct is required within the original query, expensive duplicate elimination becomes necessary.
- No rewriting previous to, or intermixed with, the GPE's instantiation is possible.

To palliate these inconvenients, [14] proposes an algebraic treatment that we review now.

In the naive approach, GPE's are processed from a schema perspective before being evaluated on the object base. The approach of [14] relies on the fact that these two instantiations are somehow unavoidable but offers some optimization possibilities that should be exploited. The main idea is thus to integrate schema lookup and object base lookup in an algebra, and thereby be able to apply optimization techniques in a homogeneous fashion to both lookups. For this, [14] extends the algebra of [15] with two new operators, S_inst and D_inst, that instantiate GPE's from a schema and a data perspective. We do not define the algebra here, simply gives an intuition, using an example, of the optimization process and of its potential. Readers are referred to [14, 12] for more details.

The example is the following:

Q4 : What are the couples of documents referencing each other and such that the first document contains the word "Web"?

select d1, d2
from Documents{d1} @P (x),
Documents{d2} @Q (y),
where d2=x **and** d1=y **and** d.title contains "Web"

The algebraic translation of Query *Q4* is illustrated on Figure 5.

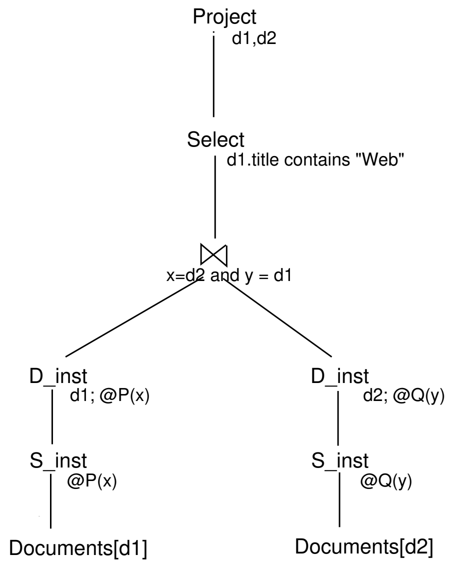

Fig. 5. Algebraic Translation of Query Q4

The left sub-tree (resp. right) represents the schema and data instantiations of the first (resp. second) GPE. The two branches are followed by a join, a selection and, finally, a projection operation.

Let us now study how this expression could be optimized.

1. Left and right sub-trees are similar; a factorization should be possible.
2. The selection operation could be pushed down the query tree. This would present several advantages. First, we may reasonably assume that the (expensive) *D_inst* operation will be applied to a smaller set. Second, this will allow the use of an index to evaluate the selection.
3. By analyzing the join predicate, we could reduce the schema lookup phase.

Figure 6 illustrates this. It shows an algebraic rewriting of the expression of Figure 5.

Note that the second *S_inst* operation has disappeared and be replaced by a renaming (*Map*) of the result of the first. The join operation has been integrated to the second *D_inst* operation, which follows $@Q$ from the $d2$ data elements that can be found at the end of $@P$. This somehow takes care of the first join

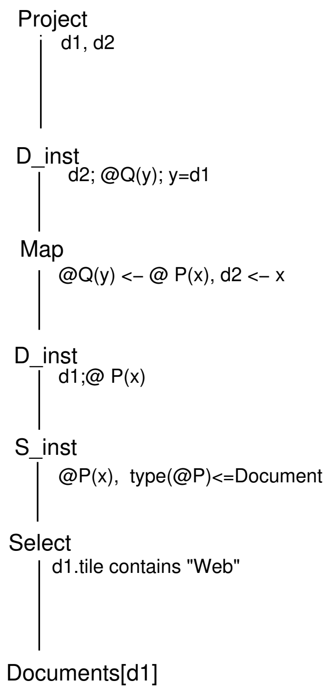

Fig. 6. Algebraic Optimization of Query Q4

condition ($x = d2$). The second join condition ($y = d1$) is given to the *D_inst* operation that will consider only elements satisfying it.

This algebraic treatment offers already much more than the naive technique. However, it is still insufficient. This can be shown using query *Q2* of the previous section. The algebraic expression corresponding to this query is presented in Figure 7.

Since the selection operation carries over elements found at the end of path $@P$, it cannot be pushed. This is rather bad. It means that, in order to evaluate the query, most of the database must be scanned. This clearly implies that some alternative solutions must be found. Unsurprisingly, we propose to use full text indexes. This is still ungoing work and will not be presented here.

4.2 Optimization in the No-Schema Context

As far as we know, there have been only two proposals to process queries in the No-Schema context and little or no optimization *per se*.

The first proposal concerns the UnQL language[8]. Queries are translated into a calculus and are then processed using structural recursion. The proposed optimization consists mainly in pushing selections and factorizing common sub-expressions.

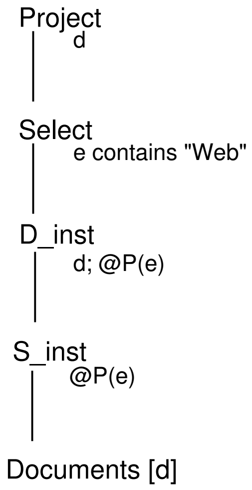

Fig. 7. Algebraic Optimization of Query Q2

We do not comment this work more. The reason is that we are not familiar with the formalism and don't really see what could be done to improve the optimization process. On the other hand, we are more familiar with the formalism used in the second proposal which is that of the Stanford group[4]. We study it here.

The first step taken in order to process a query is to rewrite it into an OQL-like query. For example, consider the following query:

Q5 : What are the titles of the sections of the documents whose title contains the word "Web"?

select Documents.document.section.title
where Documents.document.title contains "Web"

It is translated in OQL in the following way:

select t
from d in Documents.document, s in d.section, t1 in s.title
where exists t2 in d.title: t2 contains "Web"

Note that all common prefixes of paths are factorized and that for each of them a new variable is introduced in the added *from* clause. Additionally, since all attribute values are thought of as set-valued (leaving a node, there may be several edges with the same label), additional variables are introduced for paths occurring in the *select* clause (t in the example) so as to obtain an unnested result. Last — again since attribute are set-valued — comparisons between simple values and sets are removed by introducing explicit existential quantifiers. In the above Lorel query, the simple value *Web* is compared with the set of titles of the database documents. Hence, an explicit existential quantifier is introduced in the corresponding OQL query.

After this step, the logical query plan given in Fig. 8 is constructed. It consists of several *scan* operators, three *d-joins* [15], some *select* and *project* operators

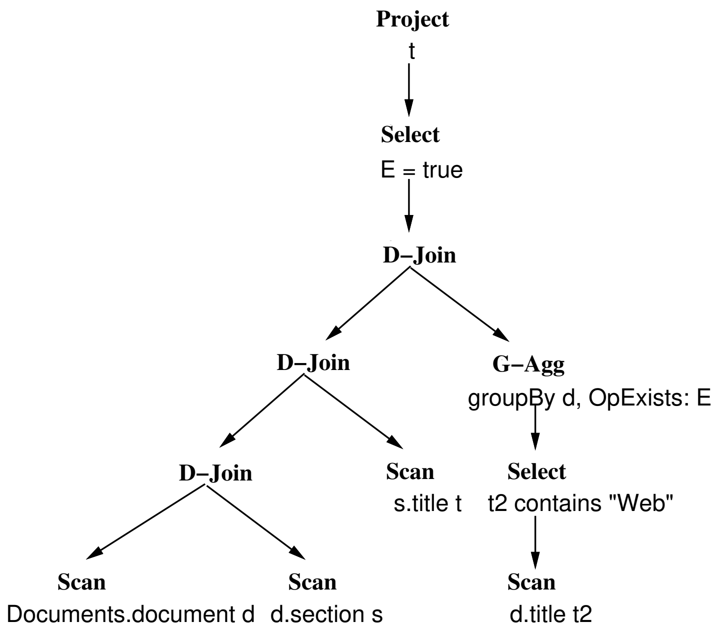

Fig. 8. Lorel Plan for Query Q5

as well as a *generalized aggregate* operator [17]. The *d-join* is a dependent join operator, sometimes called functional or implicit join. Its right input contains a free variable that is bound in the left input. Hence, the evaluation of the right input of a *d-join* depends on the left input. The *generalized aggregate* operator performs a group-by operation and then evaluates an aggregate function on each group. In the example plan, the aggregate is an existential quantifier.

After the query plan construction, query optimization, translation into a physical query plan, the plan execution follows. However, the authors of [4] note that the plan shown in Fig. 8 is — apart from pushing selections — *essentially evaluated directly*. The following disadvantages become obvious:

1. The *scan* operator of [4] returns , given an object identifier and a generalized path expression, the set of object identifiers at the end of the path. Since the scan operator is not allowed to branch, it is not able to extract information that is possibly stored together. For every single piece of information (e.g., corresponding to an attribute value), an extra scan is necessary. Clearly, this is a waste.
2. Since the information flows from one scan to another scan (object identifiers which are the output of one scan become the input of another scan by a transfer via the d-join), the access to the information is no longer sequential in nature. Instead, direct access via the object identifiers has to be performed. This is known to be far slower than a sequential scan.

3. Since only one path can be followed by the *scan* operator, the information in the tree is flattened. Assume that a document has many sections. Then, due to the recursive iterator [20] organization of their *scan* operator [4] there is a new object assignment for each section. There is no possibility to derive all the sections in a single step and work with the set of all sections afterwards. This way, all the leaf nodes that could be thought of as a set-valued attribute get unfolded, resulting in (1) a waste number of recursive iterator calls and (2) large intermediate results if they have to be materialized. This clearly slows down the evaluation process.
4. The extensive use of *d-joins* has several negative implications. First, the right argument cannot be evaluated independently. This implies a nested-loop where for each element of the left input the right input is evaluated. A nested-loop evaluation is the slowest evaluation procedure known. Further, due to the dependency, there is no way to reorder *d-joins*, which implies that the most powerful optimization technique — join reordering — cannot be applied.
5. Finally, the implementation of queries featuring GPE's is still unclear.

We believe that these drawbacks can be fixed and are currently working in that direction. The idea is to use a more powerful scan operator. More specifically, this operator should be able to do whatever is possible in a single (!) traversal of the data tree. This is in analogy to the *scan* over relations in the relational context or an extent scan in the object-oriented context. This will minimize access to the data storage while leaving the control of the more sophisticated processing to the optimizer.

5 Acknowledgments

This paper summarizes some of the research themes of the VERSO group. Many thanks to Serge Abiteboul, Bernd Amann, Vassilis Christophides, Claude Delobel, Tova Milo, Guido Moerkotte, Michel Scholl and Jérôme Siméon with whom I had the pleasure to work.

References

1. S. Abiteboul. Querying semi-structured data. In *Proc. of International Conference on Database Theory*, Athènes, January 1997.
2. S. Abiteboul, S. Cluet, and T. Milo. Correspondence and translation for heterogeneous data. In *Proc. ICDT, Athènes, Grèce*, 1997.
3. S. Abiteboul and P. Kanellakis. Identity as a query language primitive. In *Proc. SIGMOD, Portland, Oregon*, 1989.
4. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The lorel query language for semistructured data, 1996. http://www-db.stanford.edu.
5. S. Arbouy, A. Bezos, A.-F. Cutting-Decelle, P. Diakonoff, P. Germain-Lacour, J.-P. Letouzey, and C. Viel. *STEP : Concepts fondamentaux*. AFNOR, 1994.

6. E. Bertino, F. Rabitti, and S. Gibbs. Query Processing in a Multimedia Document System. *ACM Transactions on Office Information Systems*, 6(1):1–41, January 1988.
7. brinkley.realaudio.com. Realaudio, 1996. http://brinkley.realaudio.com/index.html.
8. P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1996.
9. P. Buneman, S. Davidson, and D. Suciu. Programming constructs for unstructured data. In *Proc. Workshop on Database Programming Languages (DBPL)*, 1995.
10. T. Catarci, S. Chang, D. Nardi, and G. Santucci. Wag: Web-at-a-glance. Technical report, Universita di Roma La Sapienza, 1997.
11. R.G.G Cattell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, San Francisco, CA, 1994. with contributions from Tom Atwood et.al.
12. V. Christophides. *Documents Structurés et Bases de Données Objet.* PhD thesis, CNAM-Paris, Octobre 1996.
13. V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *SIGMOD'94*, pages 313–324. ACM, 1994.
14. V. Christophides, S. Cluet, and G. Moerkotte. Evaluating queries with generalized path expressions. In *Proceedings ACM SIGMOD, International Conference on Management of Data*, pages 413–422, Montreal, Quebec, Canada, June 1996.
15. S. Cluet and G. Moerkotte. Nested queries in object bases. In *Proc. Workshop on Database Programming Languages (DBPL)*, 1993.
16. E. Codd. A relational model of data for large shared data banks. *Communications ACM*, 13(6), 1970. pp 377-387.
17. U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *VLDB*, pages 197–208, 1987.
18. Digital. Altavista, 1997. http://altavista.digital.com/.
19. P. Atzeni et. al. The araneus project, 1996. http://poincare.inf.uniroma3.it:8080/Araneus/araneus.html.
20. G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), 1993.
21. I. Graham. Html- documentation and style guide, 1994. http://www.utirc.utoronto.ca/HTMLdocs/NewHTML/htmlindex.html.
22. ISO. Specification of astraction syntax notation one (asn.1), 1987. Standard 8824, Information Processing System.
23. ISO 8879. Information processing—text and office systems—Standard Generalized Markup Language (SGML), 1986.
24. C. Fogg J. L. Mitchell, D. Le Gall. *MPEG Video Compression Standard.* Chapman Hall, 1996.
25. M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 393–402, 1992.
26. 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, 1995.
27. Laks V. S. Lakshmanan, Fereidoon Sadri, and Iyer N. Subramanian. A declarative language for querying and restructuring the Web. In *Proc. of 6th. International Workshop on Research Issues in Data Engineering, RIDE '96*, New Orleans, February 1996. In press.

28. L. Lamport. *Latex: A Document Preparation System.* AW Computer and Engineering Publishing Group, 1994. second edition.
29. A. Mendelzohn, G. A. Mihaila, and T. Milo. Querying the world wide web, 1996. draft, available by ftp: milo@math.tau.ac.il.
30. O2Technology. The O_2 server. http://www.o2tech.fr/ ou http://www.o2tech.com/.
31. OMG ORBTF. *Common Object Request Broker Architecture.* Object Management Group, Framingham, MA, 1992.
32. Oracle. The oracle server. http://www.oracle.com:81/.
33. Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proc. International Conf. on Very Large Data Bases*, Bombay, 1996. to appear.
34. 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.
35. D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. Technical report, Stanford University, December 1995. Full version of [QRS+95a], available by anonymous ftp from db.stanford.edu.
36. A. Rafii, R. Ahmed, M. Ketabchi, P. DeSmedt, and W. Du. Integration strategies in Pegasus object oriented multidatabase system. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, Volume II*, pages 323–334, January 1992.
37. D. Suciu. Proceedings of the workhop on management of semi-structured data. http://www.research.att.com/ suciu/workshop-papers.html.
38. Versant. The versant server. http://www.versant.com/.
39. K. Gregory Wallace. The jpeg still picture compression standard. *Communications of the ACM*, 34(4):30–44, 1991.
40. Yahoo. The yahoo search engine, 1997. http://www.yahoo.com/.

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