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

Querying Documents in Object Databases*

Serge Abiteboul[1], Sophie Cluet[1], Vassilis Christophides[1], Tova Milo[2], Guido Moerkotte[3], and Jérôme Siméon[1]**

[1] INRIA-Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France
[2] Tel Aviv University, Ramat Aviv, Tel Aviv 69978, Israel
[3] Lehrstuhl für Praktische Informatik III, Seminargebäude A5, Universität Mannheim, 68131 Mannheim, Germany

December 5, 1996

Abstract. We consider the problem of storing and accessing documents (SGML and HTML, in particular) using database technology.

To specify the database image of documents, we use *structuring schemas* that consist in grammars annotated with database programs. To query documents, we introduce an extension of OQL, the ODMG standard query language for object databases. Our extension (named OQL-doc) allows to query documents without a precise knowledge of their structure using in particular *generalized path expressions* and *pattern matching*. This allows us to introduce in a declarative language (in the style of SQL or OQL), navigational and information retrieval styles of accessing data.

Query processing in the context of documents and path expressions leads to challenging implementation issues. We extend an object algebra with new operators to deal with generalized path expressions. We then consider two essential complementary optimization techniques:

1. we show that almost standard database optimization techniques can be used to answer queries without having to load the entire document into the database.
2. we also consider the interaction of full-text indexes (e.g., inverted files) with standard database collection indexes (e.g., B-trees) that provide important speedup.

The paper is an overview of a research project at INRIA-Rocquencourt. Some particular aspects are detailed in [ACM93, CACS94, ACM95, CCM96].

1 Introduction

Electronic documents are usually stored in file systems. If editing tools are quite sophisticated, other document management tools are in general very primitive. In this paper, we argue that (object) database technology, with appropriate extensions, can bring major improvements to document management. Many of these improvements come from the very nature of database systems: efficient management of secondary storage, version and change control, etc. In this work, we focus more on object database technology and show how it can be used to facilitate data access.

There are three typical ways of accessing and searching data. The first, which is often used with textual data, is the search for a string, or more generally a pattern, in the text. This type of search is well-managed by information retrieval systems and involves automata techniques and full-text indexing capabilities. Another typical access are searches that involve the structure of data (i.e., attributes) and associative access (i.e., joins). Database technology provides the best solutions for this type of search, using index structures based on hashing or B-trees and query optimization. Finally, a third common way of accessing data is by browsing and navigation. This last approach is at the origin of hypertext systems and, in particular, of the WWW.

We believe that all three kinds of access/search should be supported by a system providing storage and access to documents. Therefore, it is essential to provide (1) query languages that would accommodate all, (2) engines to support them, and (3) appropriate optimization techniques. This is the topic of the present paper.

We focus in this work on the management of documents with an implicit inner structure that obeys some grammar. Examples of such documents are SGML, HTML, LaTeX or BibTeX files. Although the text of such documents has some predefined structure, this structure is not explicit enough to enable querying with a high level (logical) query language in the style of SQL or OQL. The logical structure of the information and the context of the various data pieces has to be specified. For that we propose to view the textual information as a "logical structure", and more precisely, as an object-oriented database. The mapping of a file into an object structure is obtained using a specification called *structuring schema*. A structuring schema consists of the grammar

* This work was in part developed within the EC projects GoodStep, Opal and Wire.
** Part of this author's work performed while visiting Stanford University, USA

of the file, annotated with database operations. This provides a mapping between the various portions of the file and their corresponding database images. This mapping is the first issue discussed in this paper and is studied in Section 3.

Once the textual information is viewed as an object database, it can be queried using a query language. Our starting point is the ODMG[1] [Cat94] standard query language, namely, the Object Query Language (OQL). Although ODMG is rather well-adapted to represent documents, and OQL includes many primitives that are useful for document querying, the language lacks two features essential for querying documents: sophisticate pattern matching facilities, and querying data without exact knowledge of their structure. It thus needs to be extended. This extension is the second issue discussed in this paper.

For pattern matching, it is easy to extend OQL and this remains mostly an optimization issue (considered below). The second problem, namely querying the data without full knowledge of its structure, is more challenging. Indeed, the ODMG framework insists on strongly typing both the structure and the query. This is a severe limitation for document systems where one usually finds a large heterogeneity and where users are not aware of the details of document structures. One possible direction to go is to *turn off* typing, as done in Lorel [QRS$^+$95, AQM$^+$96]. Another direction, that we follow in this paper, is to enrich the type system of OQL with a *union type* that allows for some heterogeneity. From a language viewpoint the difference is quite limited (if any). From a performance viewpoint, it is not yet clear what approach will win or even if some will win for all applications. More experiments are clearly needed to settle the issue.

The extension of OQL that we propose is named OQL-doc and is described in Section 4. The most interesting novelty comes from the use of *generalized path expressions* that allow to navigate through documents without precise knowledge of their structure. Standard path expressions consist of sequences of labels. We extend them in a number of ways, e.g., path variables and disjunctions. An example query using a generalized path expression is:

```
select  t
from    my_article.PATH_p.title(t)
where   t contains "digital library"
```

where "*PATH_p*" is a path variable. The query searches all titles in my document (including titles of portions of the document such as sections) containing the string "digital library".

The mapping of textual information into an object database and the extensions of OQL to handle textual data lead naturally to the question of performance. How should queries like the one above be processed to ensure efficiency? This is the third and last topic addressed in this paper. To allow good performance, we extend the standard object algebra and develop several new optimization techniques for query evaluation. In this paper, we focus on three main techniques:

- First, we consider the efficient evaluation of generalized path expressions. The standard way of dealing with generalized path expressions is to rewrite the query based on the schema to eliminate them. This entails the extensive use of unions or disjunctions (to handle alternatives in the data structure), potentially involves a combinatorial explosion, and is performance-wise prohibitive. To overcome this problem, we extend the object algebra to handle paths both at the schema and at the instance level, and propose appropriate optimization rules.
- A second issue is a better integration of standard database access structures (such as B-trees) and text specific access structures (such as inverted files). We consider a loose coupling between an object database and a full-text indexing mechanism that allows the retrieval of database paths and the use of various indexing granularities adapted to the application needs. The full-text index can be called directly by an algebra operator. We can therefore (within the same query) take advantage of both kinds of access structures.
- Finally, we consider the issue of querying documents that are stored as files. Recall that we use structuring schemas to obtain a database image of a file. The question is whether a file is systematically translated into its corresponding database image when queried, or whether we see the database image as a *virtual* representation of the file and only materialize portions of the file when needed. The latter approach leads to better performance. By adapting rewriting techniques (such as selection-pushing) known from traditional query optimization, we show how the reconstruction of the entire database view of the document can be avoided. Indeed, we show that one needs to construct only the portion of the database that is involved in the query and can take advantage of full-text indexes if present.

The extended object algebra is described in Section 5, and the various optimization techniques in Section 6.

The paper is an overview of a research project at INRIA-Rocquencourt. The goal of the project was to store and query SGML documents in an object database management system, namely the O_2 system [BDK92]. Some particular aspects are detailed in [ACM93, ACM95, CACS94, CCM96]. The paper is organized as follows. First, section 2 deals with related work. Structuring schemas are presented in Section 3 and the language OQL-doc in Section 4. Our extension of the algebra with specific path operators is described in Section 5. The topic of Section 6 is optimization. The last section is a conclusion.

[1] ODMG stands for Object Database Management Group.

2 Related Work

Generalized path expressions were originally introduced in [KKS92]. They are one of the basic extensions to OQL [Cat94] in order to yield OQL-doc. Lately, a more powerful language for specifying generalized path expressions involving regular expressions over attributes, attribute variables and path variables was proposed for the Lorel query language [AQM+96]. However, the practical usefulness of this extension remains to be seen. It would be interesting to consider OQL-doc's extension with features found in more powerful languages, e.g., rule languages such as MSL [PGMU95, PAGM96] and restructuring languages such as UnQL [BDS95]. The latter project is founded on a data model based on labeled trees.

An important component of OQL-doc is navigation. In that sense it belongs to the newly emerging family of query languages for the Web W3QL [KS95], WebSQL [MMM96], WebLog [LSS96] or hypertext-based query languages [MW95, BK90a, CM89, MW93]. We briefly discuss the most distinguished features of these query languages. The main motivation of W3QL and its implementation W3QS is to be open and extensible. This query language allows the user to plug in his or her own routines. However, the main disadvantage of this language is that it is very difficult to understand. WebSQL provides three kinds of dereferentiation: one for interior access within a document, one for local access to documents within the same server, and one for global access to documents outside the server. The motivation for abandoning location transparent access consists in the different costs imposed by the different accesses. WegLog is based on logic and is similar in spirit to Datalog. There is no notion of generalized path expressions. Instead, recursive Datalog-like rules replace regular expressions over attributes, attribute variables, and path variables. OQL-doc does not support non-transparent navigation, neither does it provide for an openness as exhibited by W3QL. Further, since it is an extension of OQL it follows another paradigm than the Datalog-like WebLog. Hence, OQL-doc does not share any of the distinguished features with the query languages for the Web. This should not come as a surprise since OQL-doc was not primarily designed to be a query language for the Web. However, since it features generalized path expressions to support navigation, it seems (yet another) good candidate to start with for developing a query language for the Web.

We already mentioned the relationship with the Lorel query language [QRS+95, AQM+96] developed at Stanford. Lorel was designed in the context of the TSIMMIS project [CGMH+94]. The goal of the TSIMMIS project is the integration of heterogeneous data sources. For this purpose, a new data model based on labeled trees (the object exchange model [PGMW95]) was introduced together with a query language working on trees [QRS+95, AQM+96]. Nevertheless Lorel is an extension of OQL as is OQL-doc. However, whereas our approach is to transform semi-structured documents into objects, the TSIMMIS approach is to transform all kinds of data into trees. Transformation is mainly performed via hand-coded mediators. However, first ideas for automatic mediator generation have been developed. Whereas pattern-matching forms their basis [PGMW95], our approach consists in exploiting context free grammars and in applying the idea of abstract syntax trees as used in compiler construction.

Further topics the TSIMMIS project concentrates on are the query language Lorel and the development of a light-weight object store [QWG+96]. The latter is opposed to our approach where we use the readily available O_2 database system. Concerning the former: Lorel and OQL-doc share many features, most notably, they are both OQL extensions. However, the efforts around OQL-doc have been centered around query optimization whereas Lorel provides reacher language primitives (see above).

The approach taken by the Araneus project at the University Roma Tre is quite different. Here, the idea is to provide semi-structured documents automatically generated from highly structured data [AMMT96]. This is done via Structured Web Servers. The core of this approach consists in a page-oriented data model (ARANEUS Data Model). Given a schema in this model, the ARANEUS View Language allows to define views (1) to pose relational queries and (2) to derive semi-structured documents. Hence, the problem of deriving structured data from semi-structured input as attacked in this paper is circumpassed. Another topic the ARANEUS project concentrates on is the development of an editor for structured documents [AM96].

Previous works on the integration of SGML [ISO86] documents and databases can be found in [BCK+94] for the relational model and in [YA94] for object databases.

3 Structuring schemas

In this section, we introduce informally the notion of structuring schema. To do that, we use a format that should be familiar to most readers, namely BibTeX. We then briefly consider SGML documents, a standard data exchange format quite popular in industry, that provided the main motivation for the work presented here. More generally, structuring schemas can be used for arbitrary data exchange formats, e.g., ASN.1 [ISO87] or HTML [Gra94].

As mentioned in the introduction, structuring schemas are used to provide database images of files. Their main role is thus to make explicit the logical structure of the information stored in the file, and thereby enable the usage of high level logical query languages. The database model that we use is the ODMG model, the standard for object database systems. The object model is based on values (e.g., string, image, sound), objects that allows to capture sharing and cycles, and constructors such as tuple, set and list. An alternative is clearly the relational model because of its popularity. However, documents have by nature a hierarchical structure that better fits the object model. Also references between or within documents play an important role and are a perfect match for the notion of object in the object model.

Our goal is to obtain a database representation of a piece of text that contains some inner structure. This inner structure is described in a natural way using a grammar. A first approach (that we will later abandon) is based strictly on the grammar. The connection with the database is obtained in a straightforward manner by linking each vertex of the parse-tree to a database object. In other words, the parse-tree of the string is stored in the database. We illustrate this approach next, and explain why it is unsatisfactory.

Example 1. Consider for example a bibliography file in the BibTEX format [Lam94]. An entry in the file is a string of the form:

$$
\begin{array}{ll}
@Inproceedings\ \{ & TLE90, \\
& author = \text{“}A.A.\ Toto\ and \\
& \ H.R.\ Lulu\ and\ A.B.C.D.\ Eux\text{”}, \\
& title = \text{”}\ On\ Weird\ Dependencies\text{”}, \\
& booktitle = stoc, \\
& year = \text{”1990”}, \\
& pages = \text{“}1 - 12\text{”} \\
& notes = \text{“}A\ later\ version\ of\ this \\
& \ paper\ appears\ in\ JIR98\text{”}\}.
\end{array}
$$

Consider the following partial grammatical description of a BibTEX file:

$$
\begin{array}{lll}
\langle Ref\_Set \rangle & \rightarrow & \langle Reference \rangle\ \ \langle Ref\_Set \rangle \\
& | & \epsilon \\
\langle Reference \rangle & \rightarrow & \text{“}@Inproceedings\{\text{”}\ \#String \\
& & \text{“}author = \text{”}\ \langle Author\_Set \rangle \\
& & \text{“}title = \text{”}\ \#String \\
& & \ldots \text{”}\}\text{”} \\
& | & \ldots
\end{array}
$$

Non-terminals of the grammar appear between brackets (e.g., $\langle Ref\_Set \rangle$). The grammar considers two kinds of lexical tokens: constant tokens between double quotes (e.g., "@Inproceedings {"), and other tokens that are prefixed by the # symbol (e.g., *#String*) and for which the lexical analyzer returns some typed values (e.g., string or integer).

A direct representation of the parse tree in the databases would lead to the use of the following class definitions (with standard methods such as display or edit attached to these classes):

$$
\begin{array}{ll}
class\ Ref\_Set & = tuple(\ reference : Reference, \\
& \qquad\qquad ref\_set : Ref\_Set) \\
class\ Reference & = tuple(\ string1 : String, \\
& \qquad\qquad author\_set : Author\_Set, \\
& \qquad\qquad string2 : String, \ldots) \\
& \quad + tuple(\ \ldots\ )
\end{array}
$$

Attribute names were generated automatically based on the names of non-terminals. The "plus" indicates union of types. It comes here by the fact that several rules define the same non-terminal with different structure patterns. A list of two references such as

$$
\begin{array}{ll}
@Inproceedings\ \{ & TLE90, \\
& author = \text{“}A.A.\ Toto\ and \\
& \ H.R.\ Lulu\ and\ A.B.C.D.\ Eux\text{”}, \\
& title = \text{“}On\ Weird\ Dependencies\text{”}, \\
& booktitle = stoc, \\
& year = \text{“1990”}, \\
& pages = \text{“}1 - 12\text{”} \\
& notes = \text{“}A\ later\ version\ of\ this \\
& \ paper\ appears\ in\ JIR98\text{”}\}. \\
@Inproceedings\ \{ & L91, \\
& author = \text{“}H.R.\ Lulu\text{”}, \\
& title = \text{”}More\ on\ Weird \\
& \ Dependencies\text{”}, \\
& booktitle = focs, \\
& year = \text{“1991”}, \\
& pages = \text{“}120 - 124\text{”}\}.
\end{array}
$$

is then represented by an object o, where the association between objects and values is partially described below:

class name	*oid*	*value*
Ref_Set :	o o_2 o_4	$[reference : o_1, ref\_set : o_2]$ $[reference : o_3, ref\_set : o_4]$ $\perp$
Reference	o_1 o_3	$[string1 : \text{“}TLE90\text{”}, author\_set : o_5, ..]$ $[string1 : \text{“}TLE90\text{”}, author\_set : o_6, ..]$

where object o_4 with undefined value ($\perp$) represents the empty string (i.e., an empty list of references).

Observe that with this approach, *disjunctive types* naturally arise from non-terminals defined by several production rules. Clearly, the use of disjunctive types can be avoided using complex coding but then the database is a much less direct representation of the parse-tree. (Similarly, one could use also a data model without objects but again at the cost of widening the gap between the parse-tree and its database representation).

The database direct interpretation of the file leads to severe problems when querying the resulting database structure. One can consider two major origins for these problems:

1. *Presence of unnecessary information in the string/ grammar.* Everything from the file is explicitly represented in the database. In particular, the database will record parsing information with perhaps little informative content that will in fact hide the "natural" structure of the data. Also, this approach often results in the presence of many unnecessary objects. In the above example, the natural structure of the list is hidden and o_2 and o_4 have no meaningful semantics.
2. *Absence of some information in the string/grammar.* For instance, the string above contains a list of references. This may be interpreted as a database list, but also as a set or a bag. Clearly, the intended meaning is application dependent and the grammar gives no

indication to choose between these possible representations.

This approach based solely on the grammar clearly fails. This leads to consider a second approach requiring to explicitly state the links between grammatical constructs and their database counterparts. This is done again in a rather standard manner using a schema definition and an annotated grammar. We call the resulting couple a *structuring schema*. This solution is somewhat reminiscent of techniques used for automatic synthesis of editors for languages. They also use annotated grammars to construct editors and "unparsing" specifications to display programs. Of course, the problems studied are very different but some of the techniques they develop (e.g., incremental evaluation) are applicable to our context. (See [RT89].)

A *structuring schema* consists of a schema and an annotated grammar. The annotated grammar specifies the relationship between the grammar's non-terminals and their database representation. It also associates to each derivation rule $A \to A_1, \ldots, A_n$ a statement describing how the database representation of a word derived from this rule is constructed using the database representations of the sub-words derived from $A_1, \ldots, A_n$.

We illustrate the idea informally using an example. The example provides a possible (partial) structuring schema for a BibTeX file.

Example 2. (BibTeX continued) The schema is described first:

/* Non-terminals type definition */

$\textbf{type}\ \langle Reference\rangle = \textbf{tuple}(Key : string, Authors : ...)$
$\textbf{type}\ \langle Ref\_Set\rangle = \textbf{set}(\langle Reference\rangle)$
The annotated grammar is given by:

/* Annotated grammar */

$$
\begin{array}{ll}
\langle Ref\_Set\rangle & \to \langle Reference\rangle\ \langle Ref\_Set\rangle \\
 & \qquad \{\$\$:= \{\$1\} \cup \$2\} \\
 & \mid \epsilon \\
 & \qquad \{\$\$:= \{\}\} \\
\langle Reference\rangle & \to \text{“}@Inproceedings\{\text{”}\ \#String \\
 & \quad \text{”}author = \text{”}\ \langle Author\_Set\rangle \\
 & \quad \ldots \text{”}\}\text{”} \\
 & \qquad \{\$\$:= [Key : \$1, Authors : \$2 \ldots]\} \\
 & \mid \ldots
\end{array}
$$

Observe that a Yacc-like notation is used. In the example, the $$ symbol in the action part of a rule represents the data returned by this rule. A $i symbol represents the data returned by the i-th non-terminal or non-constant token in the right side of the rule. The non-constant tokens have values corresponding to database basic types that are returned by the lexical analyzer. The whole BibTeX file is represented by a set containing one element per bibliographical reference. A reference is represented by a tuple with one attribute per BibTeX field.

The schema describes the database types. One may argue that the type information can be partially or totally derived from the annotated grammar using type inference. However, the issue of type inference can be viewed in a larger context and is not the topic of the present paper.

We can use the structuring schema to parse the file and load the result into the database. In the BibTeX example, the database will then contain a set of tuples, one per reference. We may also use the structuring schema to define a (not materialized) "view" of the file. In this case, the set is not loaded into the database. In both cases, we would like to be able to query the references efficiently. If the data is fully loaded into a database, then the query optimization is done at the database level. On the other hand, when querying a view, we would like to avoid loading the entire file into the database, and need specific optimization techniques to achieve that. These aspects of optimization are the topic of Section 6.

A concrete application: representing SGML in O_2

Specific data exchange formats usually present particularities that require some adaptation of the approach. They typically include specific features that have to be modeled in their database counterparts. To conclude this section, we consider the representation of SGML documents in an implementation of ODMG, namely the O_2 object database system. We chose to consider SGML documents because of the wide usage of the SGML standard. In particular, HTML (the exchange standard for the WWW) can be viewed as an instance of SGML. We chose O_2 [BDK92] because it is an implementation of ODMG and in particular it is the first system to support the ODMG query language, OQL. The SGML hierarchical structure can be naturally represented using the complex values of O_2. Indeed, O_2 has a hybrid data model including objects and constructed values which turns to be very convenient for this particular application. The modeling would be more complicated in a pure object model.

We first explain briefly what parts of the SGML document model are easily handled by the ODMG database model, and what are the extensions that are needed to support the rest. (None of the existing object database management systems provides all the features needed to describe directly SGML documents.) Next we describe how the SGML documents are represented in the database, and how the document-to-database mapping can be defined using structuring schemas.

Documents often require multimedia types (e.g., images). These can easily be incorporated into any object database. They also often have cross references between logical components. These also are easily modeled in an object database using object identity. Furthermore, object sharing provided by object databases is useful for the manipulation of documents, notably in the support of their evolution (e.g., versions). On the other hand, modeling of SGML documents leads to some requirements that are not directly supported by ODMG. These are related to the use of SGML connectors and occurrence indicators, that are missing in O_2 and in other object database systems as well. The main modeling feature that is missing is the *union of types*. In SGML, alternative structures may be provided for the same element type. Union type is the appropriate solution for this. We felt that union type was an essential modeling feature to support SGML documents that was missing from the ODMG model. We therefore extended O_2 with this feature. This is a nontrivial task, and as we shall see this may be costly in terms of performances, which will lead us to develop appropriate optimization techniques. We use *marked* union, i.e., a tag specifies the chosen alternative type. Union types could have also been simulated using multiple inheritance but we did not adopt this solution that misuses the original inheritance semantics (specialization). In addition, this latter solution would needlessly increase the size of the class hierarchy.

Remark 1. Other more minor extensions have to be introduced as well. For example, the ordering of tuple components may be meaningful in SGML, although it is not in ODMG. Sometimes, the ordering is imposed; sometimes some flexibility is left. For such features, one again has the choice between "coding" them in the database model or extending the database model to have them explicitly in the database representation.

SGML documents are represented in the database as follows. Each SGML element definition in the DTD (i.e., the type description of the document) is interpreted as a class having a type, some constraints and a default behavior (i.e., standard display, read and write methods for each attribute). For instance, Figure 3 presents the O_2 classes corresponding to the element's definition in the DTD of Figure 1. The choice connector ("|" of SGML) is modeled by a union type (e.g., class Body). For unnamed SGML elements defined through nested parentheses (e.g., (title, body+) line 7, Figure 1), system supplied names are provided (e.g., a_1 in class Section, Figure 3). Observe the use of inheritance (in class Picture) and that of private attributes (e.g., status in class Article). The element components marked by a "+" or "*" occurrence indicator are represented by lists (e.g., attribute authors in class Article). Finally, note that constraints had to be introduced to capture certain aspects of occurrence indicators, the fact that some attributes are required and also the range restrictions. Constraints will not be considered in this paper.

The mapping between an SGML document and its O_2 image can be naturally defined using structuring schemas. The database schema of the document has the structure described above. Each element definition in the DTD can be viewed as a grammar rule, and the action associated with each such rule simply constructs the database image of the element. This is done in the style of the BibTeX example.

To conclude this section, it should be noted that the representation of SGML documents in an object database such as O_2 comes with some extra cost in storage. This is typically the price paid to improve access flexibility and performance.

```
 1. <!DOCTYPE    article [
 2. <!ELEMENT   article     - -     (title, author+, affil, abstract, section+, acknowl)>
 3. <!ATTLIST    article              status   (final | draft)   draft>
 4. <!ELEMENT   title         - -     (#PCDATA)>
 5. <!ELEMENT   author      - O     (#PCDATA)>
 6. <!ELEMENT   abstract    - O     (#PCDATA)>
 7. <!ELEMENT   section     - O     ((title, body+) |  (title, body*, subsectn+))>
 8. <!ELEMENT   subsectn   - O     (title, body+)>
 9. <!ELEMENT   body         - O     (figure | paragr)>
10. <!ELEMENT   figure       - O     (picture, caption?)>
11. <!ATTLIST     figure               label   ID   #IMPLIED>
12. <! ELEMENT  picture    - O     EMPTY>
13. <!ATTLIST     picture              sizex     NMTOKEN  "16cm"
                                       sizey     NMTOKEN  #IMPLIED
                                       file     ENTITY #IMPLIED>
14. <!ELEMENT    caption   O O     (#PCDATA)>
15. <!ENTITY        fig1    SYSTEM  "/u/christop/SGML/image1"   NDATA >
16. <!ELEMENT    paragr    - O     (#PCDATA)>
17. <!ATTLIST      paragr              reflabel   IDREF   #REQUIRED>
18. <!ELEMENT    acknowl  - O     (#PCDATA)>          ]>
```

Fig. 1. A DTD for a document of type article

```
<article status="final">
<title> From From Structured Documents to Novel Query Facilities </title>
<author> V. Christophides
<author> S. Abiteboul
<author> S. Cluet
<author> M. Scholl
...
<abstract> Structured documents (e.g., SGML) can benefit a lot from
database support and more specifically from object-oriented database
(OODB) management systems...
<section>
<title> Introduction </title>
...
<body><paragr> This paper is organized as follows. Section 2
introduces the SGML standard. The mapping from SGML to the O2
DBMS is defined in Section 3. Section 4 presents the extension ...
</body></section>
<section>
<title> SGML preliminaries </title>
<body><paragr> In this section, we present the main features of
SGML. (A general presentation is clearly beyond the scope of this paper.)
</body></section>
...
</article>
```

Fig. 2. An SGML document of type article

4 OQL-doc

In the previous section we explained how documents can be mapped to an object-oriented database. Once this is done the documents can be queried using a high level object query language. In this section, we present an extension of the OQL language, namely OQL-doc, that allows querying data in a very flexible manner. The presentation relies on examples. The running example uses the SGML document from Figures 1-2 and the corresponding O_2 schema of Figure 3. (It is straightforward to design a structuring schema for this mapping.)

As mentioned in the introduction, to query documents from a database, we must provide (i) sophisticated pattern matching facilities for the selection of documents according to their content relying on full-text indexing; and (ii) not require users to know the exact structure of documents to access data. To answer (i), it suffices

```
class Article public type tuple (title: Title, authors: list (Author), affil: Affil,
                                 abstract: Abstract, sections: list (Section),
                                 acknowl: Acknowl, private status: string)
          constraint:           title != nil, authors != list(), abstract != nil,
                                 sections != list(), status in set ("final", "draft")
class Title inherit Text
class Author inherit Text
class Affil inherit Text
class Abstract inherit Text
class Section public type union (a1: tuple (title: Title, bodies: list (Body)),
                                  a2: tuple (title: Title, bodies: list (Body),
                                             subsectns: list (Subsectn)))
          constraint:            (a1.title != nil, a1.bodies != list())
                                  | (a2.title != nil, a2.subsectns != list())
class Subsectn public type tuple (title: Title, bodies: list (Body))
          constraint: title != nil, bodies != list()
class Body public type union (figure: Figure, paragr: Paragr)
          constraint: figure != nil | paragr != nil
class Figure public type tuple (picture: Picture, caption: Caption,
                                private label: list (Object))
          constraint: picture != nil
class Picture inherit Bitmap
class Caption inherit Text
class Paragr inherit Text
          public type tuple (private reflabel: Object)
          constraint:  reflabel != nil
class Acknowl inherit Text
name Articles: list (Article)
```

Fig. 3. O_2 classes for documents of type article

to introduce more sophisticated string predicates in the OQL language. Then, to answer (ii), we introduce two new sorts (**path** and **att**) and their basic operations. Finally, we show how the extensions of the model impact the query language. In particular, we show how union type affects the OQL language.

4.1 Querying strings

The semantics of OQL relies on a functional approach. OQL is defined by a set of basic queries and a way of building new queries through composition and iterators. Thus, to add a new functionality to the language one often has only to add new basic queries. This is what we do here.

The next query introduces the predicate `contains` which offers pattern matching facilities.

Q1: *Find the title and the first author of articles having a section with a title containing the words "SGML" and "OODB".*

```
select  tuple (t: a.title, f: first(a.authors))
from    a in Articles, s in a.sections
where   s.title contains ("SGML" and "OODB")
```

The `contains` predicate allows to match a string with a pattern or a boolean combination of patterns. (Patterns are constructed using concatenation, disjunction, Kleene closure, etc.) Among other textual predicates, we only cite `near` that allows to check whether two words are separated by, at most, a given number of characters (or words) in a sentence.

From a linguistic viewpoint, this raises no issue. On the other hand, the integration of appropriate pattern matching algorithms and full-text indexing mechanisms are interesting technical issues that are considered in Section 6.2.

4.2 Union types

We next consider the introduction of the union type in the model. This is essential in the context of documents database. We believe more generally that, this is a useful extension to the ODMG model and OQL.

The introduction of union types involves some serious modification of the type checking of OQL queries as well as of the evaluation of OQL iterators (**`select-from-where`**, **`exists`**, etc). We first explain the typing mechanism.

OQL imposes typing restrictions. For instance, when constructing a collection, we check that its elements have a common supertype (e.g., sets containing integers and characters are forbidden). We relax this constraint in our extension, we use markers. For instance, an heterogeneous set of integers and reals is allowed and could have type:

$$set(i : integer + r : real)$$

Observe the use of markers. Since we introduce union types in the model, we have to specify the subtyping rules. We briefly present the two main rules here.

1. There is no common supertype between a union type and a non-union type. Note that this forbids, for instance, to perform an intersection between a set of integers and a set of $(i : integer + r : real)$s. The set of integers has to be first coerced into a set of marked integers. This coercion can be achieved automatically.
2. Two union types have a common supertype if they do not have an attribute (marker) conflict, i.e., if they do not have the same marker with two incompatible types. The (least) common supertype of two union types is then defined in the straightforward manner. For instance, $(i : integer + c : char + s : string)$ is the least common supertype of $(i : integer + c : char)$ and $(c : char + s : string)$. Note that this second typing rule may result in a combinatorial explosion of types. However, (i) this should rarely happen and (ii) some semantic rules can be added to the OQL typing mechanism in order to control this inflation.

The following example shows how the OQL evaluation of iterators is modified to take into account union types.

Q2: *Find the subsections of articles containing the sentence "complex object".*

```
select  ss
from    a in Articles, s in a.sections,
        ss in s.subsections
where   text(ss) contains ("complex object")
```

Compared to Q1, the `contains` operation in query Q2 is evaluated not over individual data objects but over complex logical objects (e.g., subsections). For that we use a system supplied operator `text` performing an inverse mapping from a logical object (e.g., a subsection) to the corresponding portion of text [CACS94].

Recall now that, in the schema of Figure 3, sections have a union type: a section marked with a_1 corresponds to a tuple with attributes title and bodies and a section marked with a_2, to a tuple with attributes title, bodies and subsections. In query Q2, the variable ss ranges over subsections of sections marked with a_2. Two remarks are noteworthy. First, in the definition of variable ss in the `from` clause, the a_2 marker is omitted. This syntactic sugaring is required because users are not always aware of markers in union types. Second, we do not want the evaluation to fail on sections whose instance type does not correspond to the a_2 marker (e.g., sections that do not have subsections).

To deal with this problem, we introduce the notion of *implicit selectors*. Any operation on a variable ranging over a domain with union type implies an implicit selection. In the above query, the implicit selection is that $s.a_2$ should be defined. It must be noted that this mechanism stands only for variables and not for named instances. For example, suppose the existence of the name *my_section* which has been instantiated with a section marked with a_1. In this case, the query *my_section.subsections* will return a type error detected at runtime.

Let us now come back to the syntactic sugaring with a more complex example. Suppose that, instead of the subsections, we are now interested by the bodies of articles. The `from` clause of the query becomes:

`from` a `in` Articles, s `in` a.sections, b `in` s.bodies.
Variable b will range over the union of $s.a_1.bodies$ and $s.a_2.bodies$ and two implicit selectors will be used to avoid failure at evaluation. In this example, both attributes $bodies$ have the same type. However, this cannot be guaranteed in all cases. When this is not the case, a marked union type is generated by the system.

4.3 Querying paths

In order to query data without exact knowledge of their structure we introduce two new sorts: **path** and **att**. A value of the former denotes a *path* through complex objects/values (crossing objects, tuples, unions, lists, etc.) and a value of the latter represents an attribute name (of a tuple or of a marked union). For instance, *.sections[0].subsections[0]* is a path selecting the attribute sections of type list, then the first section, the attribute subsections of this section and finally the first subsection. *Paths can be queried like standard data.*

Like all types manipulated in OQL, **path** and **att** come with their variables and basic queries. To distinguish **path** or **att** variables from standard data variables, we prefix them by "$PATH\_$" and "$ATT\_$". For example, in the next query $PATH\_p$ is a path variable. The name *my_article* is a root of persistence representing an article.

Q3: *Find all titles in my_article.*

```
select t
from   my_article PATH_p.title(t)
```

The result is a set of titles reached by following various paths in the article structure (general title, title of a section, subsection, etc.).

In query Q3, the subquery *my_article PATH_p.title(t)* illustrates a new basic OQL query. It is a path expression with variables (here $PATH\_p$ and t), whose semantics is different from that traditionally used in path expressions without variables. These queries return a set of tuples with one attribute per variable. In the example, it returns a set of tuples with two attributes: t, $PATH\_p$. The attribute $PATH\_p$ ranges over all the path values that start from the root of *my_article* and that end with an attribute named title. The value of attribute t in tuple $(t, PATH\_p)$ corresponds to the title that can be reached from *my_article* following the path $PATH\_p$. Several points need further comments.

1. We may allow the syntactical sugared form
   `from` *my_article ... title(t)*
   if we are not interested in the actual values of path variables.
2. Note that the presence of path variables will often imply that the corresponding data variable is of a union type. Indeed, following different paths, one should expect reaching different types. This is particularly true if we query data from different sources, e.g., various authors may structure sections differently.
3. Path variables may be used outside a `from` clause without being defined in such a clause. For instance, the expression
   my_article PATH_p.title
   is a query that returns the set of paths to a title field.
4. The data type **path** comes equipped with functions. In particular, list functions can be used on paths. For instance, suppose that $PATH\_P$ is a path of value *.sections[0].subsections[0]* we can compute the length of $PATH\_P$: $\texttt{length}(PATH\_P) = 4$ and project $PATH\_P$ on its two first elements: $PATH\_P[0:1] = .sections[0]$.
5. When path variables are used in a path expression, there is always the possibility of cycles (in the schema and in the data). This may lead potentially to an infinite number of paths. Alternatives are:
   - handle this infinite number of paths. This is possible because of the "regularity" of the paths. (More precisely, the set of paths forms an infinite regular tree.)
   - restrict the semantics of a path variable to be a finite set of paths, e.g., by prohibiting the presence of the same object twice on a path, or of two objects from the same class. (This last approach is the one used in our implementation).

To see another example of the use of paths for querying data with incomplete knowledge of their structure, let us assume the existence of a name *my_old_article* representing an old version of *my_article*. Consider the query:

Q4: *Find the structural differences between two versions of my_article.*

my_article PATH_p1 − my_old_article PATH_p2

The left (resp. right) argument of the difference operation returns the set of paths starting from *my_article* (resp. *my_old_article*). Thus, the difference operation will return the paths that are in the new version of *my_article* and not in the old one. Supplementary conditions on data would allow the detection of possible updates or moves of individual textual objects within the document logical structure.

In a last example, we illustrate the use of attribute variables:

Q5: *Find the attributes defined in my_article whose value contains the string "final".*

```
select unique name(ATT_a)
from   my_article PATH_p.ATT_a(val)
where  val contains ("final")
```

In this example, the data variable *val* ranges over data that can be accessed from *my_article* following a path denoted by *PATH_p* (which possibly is the empty path) and ending with the attribute denoted by the variable *ATT_a*. Accordingly, the initial type of *val* is the union of all the types found in the structure of *my_article*. The subquery *val* **contains** *("final")* uses the implicit selector *val.a* where $s : string$ represents an attribute of type string in the corresponding union type. As a result OQL restricts *val* to type *string*. An attribute is returned if for some occurrence of this attribute, the corresponding value contains the string "final". We believe that this is an important feature of the extension of OQL allowing the user to perform search operations like Unix **grep** inside an object database system. Finally, the function **name** returns a string, the name of an attribute.

5 Algebra

The main difficulty in the implementation of OQL-doc queries concerns the evaluation of generalized path expressions (GPE), i.e., expressions featuring path and attribute variables. There exists little literature on the subject [BRG88, CACS94]. A main approach has been proposed. It consists in transforming simple queries featuring GPEs into a union of queries in the following way:

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; and
4. union the remaining instantiated queries and evaluate the resulting query.

Let us illustrate this procedure using Query **Q5**. The transformed query is the following:

```
select title
from   my_article.title (val)
where  val contains ("final")
union
select title
from   my_article.sections[i].title(val)
where  val contains ("final")
union
...
```

An alternative, proposed[2] in [BRG88], is to use the Boolean connector *or* instead of a union. This second approach is only slightly different: an entry satisfying all conditions will be selected only once.

Note that the query resulting from the above procedure does not quite correspond to a standard OQL query. Omitting syntactical differences, there are still some variables (e.g., *i* in some **from** clauses) that are not defined over a given collection. To provide valuations for these variables, we need to evaluate (at least partially) the standard part of the query. This would allow us to find, for instance, how many sections contains *my_article*. Note also that the query does not contain explicitly the valuation of the attribute and path variables. In this particular example, it is not really a problem since these variables are not used outside the GPEs. However, queries featuring functions over path/attribute variables may occur and thus we need to compute the valuations of such variables.

Furthermore, this technique presents serious drawbacks from a performance viewpoint. The rewritten query may be very large thereby making the task of the optimizer almost impossible. The lack of flexibility (all paths are treated similarly) and possibly heavy redundancy in intermediate results can also handicap seriously performances.

Given this, we propose an algebraic treatment of GPEs. By extending the object algebra appropriately, we give control of GPE evaluation to the optimizer. This results in more evaluation alternatives and allows the use of adequate data structures.

Let us analyze what is going on in the "standard" rewriting strategy above. It first performs some kind of schema lookup and type inference to extract all possible valuations of the path and attribute variables based on the schema. It then eliminates those which result in incorrectly typed queries. The query is then rewritten, optimized and finally evaluated. This is therefore essentially a two phases process: schema lookup and type inference in a first phase, query optimization and evaluation in a second.

Now, in many applications, the schema itself can be reasonably large. Thus, it is interesting to apply query optimization techniques also to the schema lookup phase. Furthermore, in a number of cases, the separation into two phases is not appropriate. For instance, in many situations, inexpensive object base lookups can restrict the number of possible paths dramatically at a much lower cost than a schema lookup. The main idea of our technique is thus to integrate schema lookup and object base lookup, and thereby be able to apply optimization techniques in a homogeneous fashion to both lookups. For

[2] The method was proposed only for a very limited query language, but it is easy to generalize it.

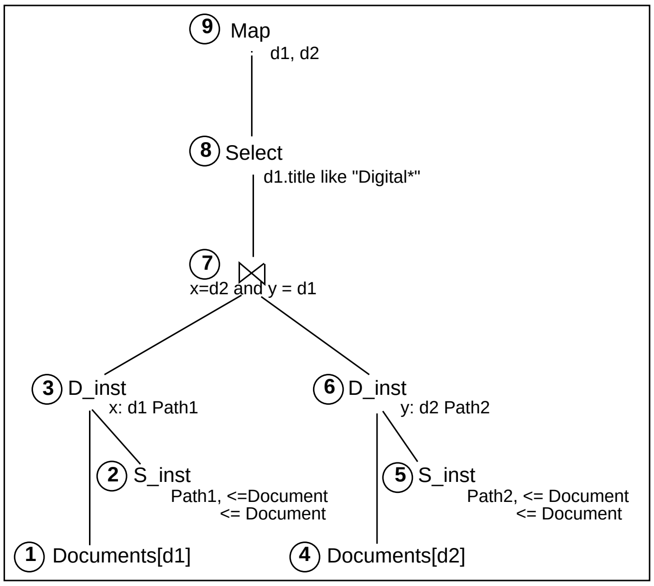

Fig. 4. Algebraic translation of Query Q6

this, we extend the algebra of [CM95] with two new operators, *S_inst* and *D_inst*, that instantiate GPEs from a schema and a data perspective.

The extended object algebra is described in detail in [CCM96]. We illustrate it here with a simple example. Query **Q6** returns pairs of documents referencing each other given that the title of the first document contains the word "digital".

Q6:select d1, d2
from Documents(d1)Path_1, Documents(d2)Path_2
where d1.title **like** "∗[Dd]igital∗" **and**
d1 Path_1 = d2 **and**
d2 Path_2 = d1

Let us now translate the query into the algebra. The result is given in Figure 4.

Operation (1) allows us to view Documents as a set of tuples with one attribute $d1$. This feature is essential to the algebra whose operators are, for the most part, defined on set of tuples. This gives nice properties to our operators. At the end of (1), we have a set of the following form:

{ [d1: o1], [d1: o2], ...}

where $o1, o2$ are objects belonging to *Documents*.

Operation (2) finds all the possible schema paths starting and ending in an object of class Document (or some subclass of Document). Note that in order to know the starting and ending types of the schema paths we performed some static type inference before/during the translation process. As we will see, this type information can be refined during the evaluation process to simplify the *S_inst* operation. One interesting issue concerns the representation of the schema paths. Trees allow a good factorization of schema paths and we favor this representation in [CCM96]. In this compact representation, the result of *S_inst* is a forest annotated with some informations allowing the interpretation of the different path/attribute/data variables.

Operation (3) evaluates all the paths obtained by (2) on each document of (1) and adds to (1) attributes corresponding to this evaluation. At the end of (3), we have a set of the following form:

{ [d1: o1, x: o1', Path1: p1], [d1: o2, x: o2', Path1: p2], ...}

where *pi* is the database (i.e., instantiated) path going from *oi* to oi'. Note that we loose here the factorization brought by the tree representation previously proposed. However, selections on paths and projections can be pushed into the *D_inst* operation. Thus, when the path instantiation is not needed in the query final result, we avoid the unnecessary information.

Operations (4,5,6) are similar to (1,2,3). Operation (7) is a join[3] which will result in a set with attributes *d1, x, Path1, d2, y, Path2*. Operation (8) is a selection. The last operation is a *map* that, in the present case, implements a simple projection. Other queries may require the application of functions, thus the use of the *map* operation in the translation process.

6 Optimization

In this section, we illustrate three different optimization techniques that can be applied on OQL-doc. First, we consider some algebraic rewriting for the new (non-standard) operators that we introduced in Section 5. Next, we consider the use of full-text indexing to evaluate GPE's. Finally, we present a technique that consists in loading the smallest possible portions of documents to answer a query when the documents are stored in external files.

6.1 Algebraic rewriting

We present by example some rewriting rules involving the new algebraic operators introduced in the previous section. Consider **Q6**. A possible rewriting is given in Figure 5.

One can see that the selection operation (8) has been pushed down the query tree. This presents several advantages. First, it is expected that the (expensive) *D_inst* operation (3) is now applied to a smaller set. Second, this allows the use of an index to evaluate the selection or of some less standard optimization techniques to be considered in the coming sections. Also, note that the join operation has been integrated to the second *D_inst* operation (6/7), which follows $Path2$ from the x data elements that can be found at the end of $Path1$. This takes care of the first join condition ($x = d2$). The second join condition ($y = d1$) is given to the *D_inst* operation that will consider only elements satisfying it.

This new expression is already preferable over the first one. Nevertheless, more improvements can be achieved. It is possible to factor the *S_inst* operations. We do not consider this alternative here, since it hinders

[3] This join operation has actually required some rewriting after the translation process.

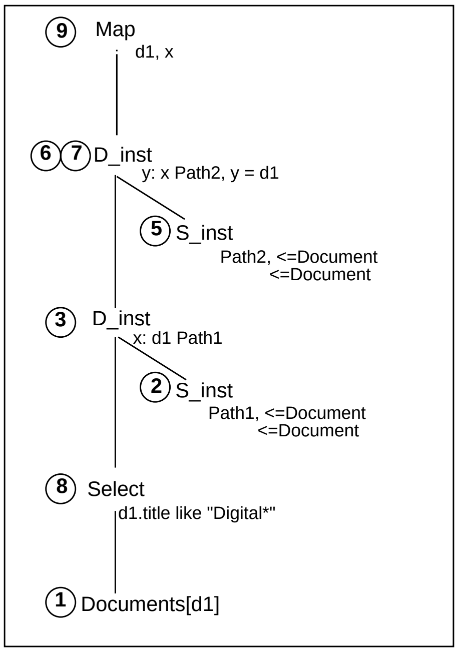

Fig. 5. Algebraic optimization of Query Q6

another optimization technique we want to illustrate. Let us suppose that class Document is the root of a rather large class hierarchy. Then, the *S_inst* operations are expensive. Now, let us further suppose that the selection operation returns only elements belonging to the class *Article*. Under this assumption, it is possible to simplify considerably both *S_inst* operations by, respectively, restricting the starting type of $Path1$ and ending type of $Path2$. If is not satisfactory, one can then reconsider the strategy and factor the *S_inst* operations.

6.2 Using full-text indexes

The documents in the database often come from external files. To compute a query, one can load the files into the database (using their structuring schemas), and then evaluate the (optimized) algebraic expression. However, in many cases one can do better by exploiting file services such as full-text indexing.

Let us suppose that the set of documents we are querying is not in the database but stored in some SGML files whose DTD is in correspondence with the database schema (see Section 3). Let us further suppose that these documents have been full-text indexed, and that the full-text indexer knows about the structure of the documents (tags). Obviously, we would then like to use this index in order to evaluate the selection operation (8) of Query **Q6**. This would (i) allow a fast evaluation of the selection operation and (ii) avoid the unnecessary load of irrelevant documents in the database.

This can be rather easily achieved. What is required is an interface between the full-text index (FTI) and the database. This interface is in charge of adding some semantics to the result of a full-text query. Typically, given a textual predicate, a full-text query returns the identifiers of the documents in which it found a string satisfying the predicate, as well as the offset where it found the string in the document. By a simple analysis of the document's SGML tags and their relationship with the database schema, the interface is then able to retrace the "database path" that goes from the document to the string. Let us consider the use of a full-text index operation to evaluate Operation (8) of Query **Q6**. This is shown on Figure 6 where FTI denotes the full-text index operator.

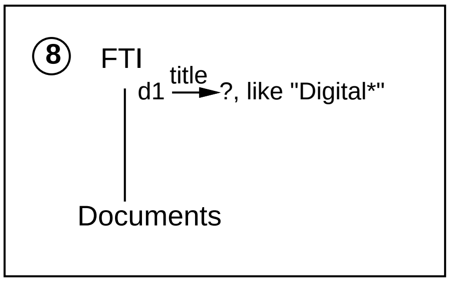

Fig. 6. Selection using a full-text index

As can be noted, FTI has two parameters. The second one is easily recognized as the selection predicate. The first one is a little less obvious. It specifies (i) the pattern we want to follow in order to reach the string satisfying the predicate and, also, (ii) what should be returned by FTI. In the present case, the pattern states that we start from some document and that we follow the attribute "title" to find the predicate string. It further states that we are interested in returning a set of tuples with one attribute $d1$ representing the documents. The question mark at the end of the pattern states that we are not interested in an attribute representing the documents title. All this information is translated by the full-text index/database interface in terms of SGML tags. The full-text index then returns the identifiers of documents featuring an appropriate string between the appropriate "title" SGML tags. The selected SGML documents are then translated and loaded into the database.

This ability of the interface to interpret database paths can be used for more complex operations. Let us consider Query **Q5** whose algebraic translation is given on the left part of Figure 7. In this query, the selection cannot be pushed down the query tree since it depends on the value found at the end of the GPE. Still, a full-text index on "my_article" could be of clearly great use to evaluate this query. The right part of the figure illustrates this point. The FTI* operator uses the full-text index/database interface in order to find the offsets of the strings satisfying the predicate. It then uses its ability to analyze the SGML tags and their correspondence with the database schema to trace back the database paths towards my_article. These paths are then converted into standard data through a *map* operation and used in the sequel of the query.

In [CCMS96], we give appropriate equivalences and also show how full-text indexes can be used to evaluate join queries.

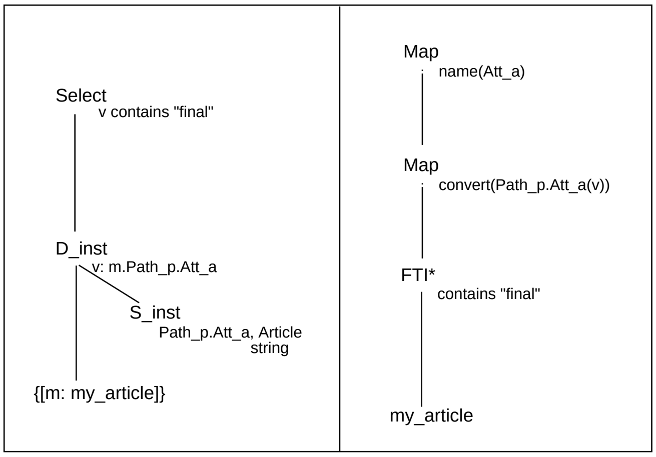

Fig. 7. Optimization of Query Q5 using full-text index

6.3 Pushing queries on files

Even if the files are not full-text indexed, there is still a better way to compute a query than simply load the whole file into the database and then evaluate the query. This is explained next.

The idea is that queries are often concerned with only a small portion of the data. The main contribution of this section is an optimization technique to avoid constructing the entire database to answer a query. Instead, the structuring schema is modified so that only the small portion that is relevant to the query evaluation is loaded. This is done using an optimization technique that we introduced in [ACM93]. It is based on "pushing queries" down the structuring schema in order to load only relevant data into the database. This optimization is limited to rather simple queries. However, using algebraic rewriting techniques, we can push all simple operations down the query tree and thus apply this new non-standard technique on the simple part of a query (e.g., the selection operation on Figure 5).

For simplicity, we illustrate the technique with a simple query. Assume that we are dealing with BibTeX files, having the structuring schema of Section 3, and that the file we are querying is neither loaded in the database nor full-text indexed. Suppose that we want to find the title of a paper of keyword TLE90. This can be formulated as a database query:

select r.Title
from r **in** References
where r.Key = "TLE90".

where *References* denotes the set returned by the parsing. The corresponding algebraic query is $\varphi(References)$ where:

$$\varphi \equiv \lambda X \cdot \Pi_{Title}(\sigma_{Key=\text{"}TLE90\text{"}}(X)).$$

Observe that we need to return only strings and that the construction of the entire database seems totally unnecessary. To reduce the data construction, we "push the selection down" inside the structuring schema in the way selections are pushed down in relational algebraic query expressions by optimizers [Ull88]. We leave the grammatical part of the structuring schema practically unchanged (the parser must still recognize the same file) whereas we modify the actions in an appropriate manner.

First, consider the rule used to compute the set of references:

$$\begin{aligned} \langle Ref\_Set\rangle \rightarrow\ & \langle Reference\rangle\ \langle Ref\_Set\rangle \\ & \{\$\$:= \{\$1\} \cup \$2\} \\ \mid\ & \epsilon \\ & \{\$\$:= \{\}\}. \end{aligned}$$

The desired result can be obtained using the rule:

$$\textbf{(1)}\quad \begin{aligned} \langle \varphi(Ref\_Set)\rangle \rightarrow\ & \langle Reference\rangle\ \langle Ref\_Set\rangle \\ & \{\$\$:= \varphi(\{\$1\} \cup \$2)\} \\ \mid\ & \epsilon \\ & \{\$\$:= \varphi(\{\})\} \end{aligned}$$

where $\langle \varphi(Ref\_Set)\rangle$ is a new non-terminal, indeed, the new start symbol.

A strict application of this structuring schema would result in creating the entire set of references and then applying the query to it. But, we can now apply some query rewriting to the action of rule 1. Observe first that

$$\varphi(\{\}) = \Pi_{Title}(\sigma_{Key=\text{"}TLE90\text{"}}(\{\})) \quad \rightsquigarrow \quad \{\}.$$

And in a less standard way, we have

$$\begin{gathered} \varphi(\{\$1\} \cup \$2) \rightsquigarrow \varphi(\{\$1\}) \cup \varphi(\$2) \rightsquigarrow \\ \Pi_{Title}(\sigma_{Key=\text{"}TLE90\text{"}}(\{\$1\})) \cup \varphi(\$2) \\ \rightsquigarrow \\ \{\Pi_{Title}(\sigma_{Key=\text{"}TLE90\text{"}}(\$1))\} \cup \varphi(\$2). \end{gathered}$$

Part of this is rather standard, e.g., distributivity of selection/projection w.r.t. union. Part is less so, e.g., pushing projection/selection onto a single element since \$1 is not a set but a tuple. For this, we extend the algebra to have such operations also operating on singleton elements. We obtain the new rewrite rules by viewing such an element as a singleton set. In particular, the selection on a single element is defined as follows: if it succeeds, the result is the element itself; and if it fails, the result is the single element $\perp$ (which is viewed algebraically as the empty set).

For instance, one rewrite rule that we used is:

$$\sigma_{Cond}(\{A\})) \rightsquigarrow \{\sigma_{Cond}(A)\}$$

which can now be accepted also if A is an atomic value. The standard set of rewrite rules [BK90b, CD92, SZ89, SÖzsu90] is extended in this manner.

The result of the rewriting of action of (1) leads to:

$$\textbf{(2)}\quad \begin{aligned} \langle \varphi(Ref\_Set)\rangle \rightarrow\ & \langle Reference\rangle\ \langle Ref\_Set\rangle \\ & \{\$\$:= \{\varphi(\$1)\} \cup \varphi(\$2)\} \\ \mid\ & \epsilon \\ & \{\$\$:= \{\}\} \end{aligned}$$

Continuing with the example, we next "push" the query "down into the grammar". The query on \$1 and on \$2 in (3) is pushed down to the corresponding non-terminals. More precisely, instead of calling $\langle Ref\_Set\rangle$ and filtering its result with φ, we use the new parser

$\langle\varphi(Ref\_Set)\rangle$ that also has the responsibility of applying φ. And similarly for $\langle Reference\rangle$. Thus, (2) is transformed into:

$$\text{(3)}\quad \begin{array}{l} \langle\varphi(Ref\_Set)\rangle \rightarrow \langle\varphi(Reference)\rangle\ \langle\varphi(Ref\_Set)\rangle \\ \qquad\qquad \{\$\$:= \{\$1\} \cup \$2\} \\ \qquad \mid \epsilon \\ \qquad\qquad \{\$\$:= \{\}\} \end{array}$$

We already have a definition for the non-terminal $\langle\varphi(Ref\_Set)\rangle$. We obtain a definition for $\langle\varphi(Reference)\rangle$ by transforming the rules for $\langle Reference\rangle$ in the following way:

$$\text{(4)}\quad \begin{array}{l} \langle\varphi(Reference)\rangle \rightarrow \text{"@}Inproceedings\{\text{"}\ \#String \\ \qquad \text{"}author = \text{"}\ \langle Author\_list\rangle \\ \qquad \text{"}title = \text{"}\ \langle\#String\rangle \\ \qquad \ldots \text{"}\}\text{"} \\ \qquad\quad \{\$\$:= \varphi([\ Key : \$1, \\ \qquad\qquad Authors : \$2, \\ \qquad\qquad Title : \$3 \ldots])\} \end{array}$$

Observe that here again we apply φ to a single element, i.e., a reference. We can use again a rewriting technique:

$$\text{(5)}\quad \begin{array}{l} \Pi_{Title}(\sigma_{Key=\text{"}TLE90\text{"}}([\ Key : \$1, \\ \qquad Authors : \$2, \\ \qquad Title : \$3 \ldots])) \\ \qquad \rightsquigarrow \\ \Pi_{Title}(\ [Key : \sigma_{\lambda\cdot x(x=\text{"}TLE90\text{"})}(\$1), \\ \qquad Title : \$3]). \end{array}$$

The selection operation has been pushed inside the tuple construction (that can be viewed as a product). We introduced a λ-expression in order to denote the element \$1 in the algebraic operation. It must be noted that, as before, we use selection on an element (here a string). If \$1 is not $TLE90$, the selection of \$1 is $\perp$, the tuple is also $\perp$ (since the standard product with an empty set is the empty set), and the value of the projection is $\perp$ as well (since the standard projection of an empty set is an empty set). Note also that the attribute *Authors* has been projected out and that \$2 is now simply ignored. The advantage of the above optimized construction is that it avoids the construction of irrelevant values.
The optimized structuring schema for a query φ is obtained as follows:

1. add the new start symbol (φ applied to the previous start symbol) and the new rule for it.
2. apply the rewrite rules to generate alternative execution plans:
   a) The *grammar rewriting rule* push queries down the grammar specification. If in a rule each occurrence of some $\$i$ is inside some unique algebraic query ψ (i.e., as $\psi(\$i)$), replace $\psi(\$i)$ by $\$i$ and replace the corresponding parser, say $\langle A\rangle$ by $\langle\psi(A)\rangle$. If such a rule does not exist yet, add a rule for parsing $\langle\psi(A)\rangle$ in the obvious way.
   b) Apply algebraic *query rewriting rules* to the algebraic expressions in the action parts of the rules.

Clearly, heuristics (such as selection pushing) have to be applied as in standard query optimization. The grammar rewriting rule must be applied whenever possible. In the best case (e.g., the previous example), the only values that are constructed are the ones returned by the query. In the worst case, the query remains at the root of the parse-tree and the whole database is constructed.

We conclude this section with a remark on the reduction.

Remark 2. The *grammar rewriting rules* introduce new non-terminals and thus new rules. Some non-terminals may become obsolete as shown by the BibTeX example (the non-terminal *Ref_Set*). Observe, however, that a non-terminal $\langle V\rangle$ may be replaced locally by $\langle\varphi(V)\rangle$ in a portion of the grammar but continues to be used in another portion.

7 Conclusion

The present work was done in the context of the OQL-doc project at INRIA, Rocquencourt. We used the O_2 object database system and developed a file loader for SGML documents using the Euroclyd SGML parser [Eur91], a Yacc-based loader [Her92] and a customizable SGML dumper of O_2 databases [Leb95]. Further, we implemented the loose coupling of O_2 with the WAIS system [Sim95]. The first OQL-doc interpreter is now running. Currently, it accepts most queries with generalized path expressions and some limited text predicates. A connection with the WAIS full-text mechanism has been implemented but more work is required in this area as well as more algebraic rewriting rules needed for generalized path expressions.

The present paper is meant to be an overview of some aspects of the OQL-doc project. More detailed and formal presentations of some particular aspects can be found in [ACM93, CACS94, ACM95, CCM96].

The language OQL is presented in [CACS94]. The optimization of the evaluation of *generalized path expressions* is considered in [CCM96]. The optimization allowing to load the relevant portions of the document files is described in [ACM93], and other optimizations in [CCM96]. In the present paper, we considered only queries. Updates for our framework are treated in [ACM95]

We also ignored here the issue of the graphic interface. In the OQL-doc project, we use O_2Web developed by O_2Technology to provide an access to documents stored in the database from any Web browser. (The URLs contain OQL-doc queries.) We want also to mention here some interesting interactions with the O_2Views system [dSAD94] which allows the customization of a database/document to a particular application.

The OQL-doc technology is now being tested in some applications. Part of it is already used with multimedia data in the projects *mediaculture* and *Aquarelle*.

References

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

[ACM95] S. Abiteboul, S. Cluet, and T. Milo. Updating the file. In *Proc. ACM Sigmod Conference on Management of Data*, 1995.

[AM96] P. Atzeni and G. Mecca. Cut & paste. Submitted for publication, 1996.

[AMMT96] P. Atzeni, G. Mecca, P. Merialdo, and E. Tabet. Structures in the web. Submitted for publication, 1996.

[AQM+96] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The lorel query language for semistructured data, 1996. `http://www-db.stanford.edu/`.

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

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

[BDS95] P. Buneman, S. Davidson, and D. Suciu. Programming constructs for unstructured data. In *Proc. DBPL*, 1995.

[BK90a] C. Beeri and Y. Kornatski. A logical query language for hypertext systems. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, 1990.

[BK90b] C. Beeri and Y. Kornatzky. Algebraic optimization of object-oriented query languages. In *Proc. ICDT, Paris, France*, 1990.

[BRG88] 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.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From Structured Documents to Novel Query Facilities. In *SIGMOD'94*, pages 313–324. ACM, 1994.

[Cat94] R.G.G. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, San Francisco, CA, 1994. with contributions from Tom Atwood et.al.

[CCM96] 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.

[CCMS96] V. Christophides, S. Cluet, G. Moerkotte, and J. Siméon. Query optimization using a cocktail of database and full text indexes. Technical report, Inria - Verso, 1996. To appear.

[CD92] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proc. Sigmod, San Diego, USA*, 1992.

[CGMH+94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The tsimmis project: Integration of heterogeneous information sources. In *IPSJ Conference*, pages 7–18, Tokyo, Japan, October 1994.

[CM89] Mariano P. Consens and Alberto O. Mendelzon. Expressing structural hypertext queries in graphlog. In *Proc. 2nd. ACM Conference on Hypertext*, pages 269–292, Pittsburgh, November 1989.

[CM95] S. Cluet and G. Moerkotte. Classification and optimization of nested queries in object bases. Technical Report 95-6, RWTH Aachen, 1995.

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

[Eur91] Euroclid. Le Parseur SGML d'Euroclid. Internal document, Euroclid, 12, Avenue des Prés 78180 Montigny le Bretonneux, 1991.

[Gra94] I. Graham. Html documentation and style guide, 1994. `http://www.utirc.utoronto.ca/HTMLdocs/`.

[Her92] L. Herr. O_2Yacc. Rapport de Stage de Maîtrise ENS, September 1992.

[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 *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 393–402, 1992.

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

[Lam94] L. Lamport. LaTeX*: A Document Preparation System*. AW Computer and Engineering Publishing Group, 1994. second edition.

[Leb95] L. Lebas. O_2Writer. Master's thesis, Universités Paris I et Paris XI, 1995.

[LSS96] 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.

[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*, pages 394–411. Springer, 1993.

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

[PAGM96] 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.

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

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

[QRS+95] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. In T.W. Ling, A.O. Mendelzon, and L. Vieille, editors, *Proceedings of the Fourth International Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 319–344, Singapore, 1995. Springer-Verlag.

[QWG+96] D. Quass, J. Widom, R. Goldman, K. Haas, Q. Luo, J. McHugh, S. Nestorov, A. Rajaraman, H. Rivero, S. Abiteboul, J. Ullman, , and J. Wiener. Lore: A lightweight object repository for semistructured data. In *Proc. ACM Sigmod Conference on Management of Data*, Montreal, Canada, June 1996. Demonstration description.

[RT89] T.W. Reps and T. Teitelboum. *The Synthesizer Generator, A system for Constructing language based editors.* Springer-Verlag, 1989.

[Sim95] J. Siméon. Recherche en Texte Intégral et Bases de Données Orientées-Objet. Master's thesis, Université de Nancy I, 1995.

[SÖzsu90] D. Straube and T. Özsu. Queries and Query Processing in Object-Oriented Database Sysems. Technical report, Department of computing science, university of Alberta, Edmonton, Alberta, Canada, 1990.

[SZ89] G. Shaw and S. Zdonik. Object-Oriented Queries: Equivalence and Optimization. In *Proc. DOOD, Kyoto, Japan*, December 1989.

[Ull88] J.D. Ullman. *Principles of Database and Knowledge Base Systems: Volume I and II.* Computer Science Press, 1988.

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

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