Optimizing generalized path expressions using full text indexes

Index plein-texte pour l'optimisation de requêtes avec expressions de chemin

Vassilis Christophides* — Sophie Cluet**
Guido Moerkotte* — Jérôme Siméon****

* *Institute of Computer Science, FORTH*
Vassilika Vouton, P.O.Box 1385, GR 711 10, Heraklion, Greece

** *Projet Verso, INRIA Rocquencourt*
78153 Le Chesnay cedex

*** *Lehrstuhl für praktische Informatik III*
Seminargebäude A5, Universität Mannheim, 68131 Mannheim, Germany

ABSTRACT. *Query languages for object bases became enriched by generalized path expressions that allow for attribute and path variables. Optimizing queries containing generalized path expressions attracted some interest. However, many interesting queries require still a full scan over the whole object base. This unbearable situation can be remedied best by utilizing index structures. However, traditional database indexes fail to support generalized path expressions. We propose to use a flexible mix of database and full text indexes in order to evaluate efficiently queries containing generalized path expressions. We introduce an algebraic framework for the optimization of such queries using full text indexes, and we report on a first prototype implementation including some performance results.*

RÉSUMÉ. *Étendre les langages de requêtes des SGBD objet avec des expressions de chemin généralisées permet d'interroger les données sans une connaissance exacte de leur structure. Cependant, l'évaluation efficace des requêtes contenant des expressions de chemin généralisées reste un problème ouvert et les techniques d'indexation classiques ne conviennent pas à ce nouveau contexte. Nous proposons d'utiliser conjointement index classiques et index plein-texte pour obtenir une évaluation efficace des requêtes avec expressions de chemin généralisées. Nous définissons un cadre algébrique permettant de combiner étroitement l'utilisation des index plein-texte et les techniques standard d'optimisation. Enfin, nous donnons quelques résultats sur les performances d'un premier prototype.*

KEY WORDS: *query optimization, generalized path expression, full-text index.*
MOTS-CLÉS : *optimisation de requêtes, expressions de chemin, index plein-texte.*

1. Introduction

Query languages for object bases have proven to be a very powerful tool for users. However, there exists good reasons to enrich them even further. The most useful extension proposes the introduction of generalized path expressions (GPE) containing attribute and path variables [KKS92]. These are very useful for several reasons. Firstly, if the schema is unknown to the user, he is still able to query the database. Secondly, if the schema includes some degree of heterogeneity, this might be overcome by attribute and path variables. Thirdly, some queries that cannot be stated otherwise or look very awkward, can be stated easily. Fourthly, attribute and path variables allow the querying of the schema and structure of an object base. Theses functionalities are crucial for a wide range of object base applications where the distinction between data and schema/type seems to disappear — at least for the end-user. As a witness, consider text applications that became even more popular with the advent of the Web.

Text applications have already received a lot of attention from the database community. Several commercial database systems (e.g., DB2, Sybase, Oracle, Informix, O_2) already offer tools for developing Web servers or Web applications. New database query languages have been proposed to deal with textual data (e.g., [BRG88, KKS92, QRS+95, CACS94]) and various optimization techniques have been introduced (e.g., [BRG88, CCM96, CS93, CM94b, CDY95]). However, there still lacks an approach that would encompass the full power of standard or extended database query languages — especially those containing generalized path expressions. More specifically, despite the new optimization techniques, many interesting queries still require a full scan of the object base. The most famous example of this kind is a query that resembles a Unix *grep*. This situation can only be remedied by proper use of well-suited indexes.

In this situation, our contribution is threefold. (1) We show how existing full text index systems (e.g., INQUERY [CCH92], Topic [Ver95], WAIS [Pfe95], etc.) can be used in order to optimize queries featuring generalized path expressions. The main idea is that the full text index should not only report where a certain item occurred but also the path that leads to this item. (2) We enhance an object algebra with appropriate operators and give equivalences between algebraic (sub)queries and queries using a full text index. A first version of the algebra allowing to optimize queries with GPEs was introduced in [CCM96]. However, some adjustments are necessary in order to allow for the introduction of full text indexes and state the equivalences enabling their utilization by the query optimizer. (3) We validate our approach by reporting performance measures on a first prototype. The main justification of our approach will be that queries that took hours without full text index utilization can now be answered within seconds.

Although there exists many proposals to integrate databases and information retrieval systems (e.g., INQUERY/IRIS [CS92], etc.), none is intended to use full text indexes to optimize standard or extended query language featuring GPEs.

Nevertheless, two approaches appear similar to ours [CM94b, CDY95]. Contrary to [CM94b], we do not exclude standard optimization techniques from our framework. Also, our use of full text indexing capabilities is much more flexible than that proposed in [CDY95] where only the textual parts of documents are indexed. Our approach allows the use of full text indexes on structured documents stored in files or in text databases, and this at various granularities, according to the application's need. Additionally, our approach is independent of the underlying full text indexing system.

The paper is organized as follows. Section 2 introduces GPE's and review (scarce) literature existing on the subject of their optimization. In Section 3, we show how full text indexes can be used to optimize GPE. Then, in Section 4, we show that full text indexes can also be used to optimize standard queries. Section 5 reports on a first prototype implementation. The paper is illustrated with some performance figures.

2. Optimizing Queries with GPE: State of the Art

In this section, we briefly introduce queries with generalized path expressions along with some first naive approaches to evaluate them [BRG88, CACS94, KKS92]. Next, we briefly review the algebraic approach we proposed in [CCM96]. We then use an example to show the lack in the current state of the art and the need for the introduction of full text indexes.

2.1. *Generalized Path Expressions*

Generalized path expressions (GPE) are very useful primitives that allow to query instance as well as schema in a uniform fashion. They were introduced as a means to query textual data [BRG88, CACS94], object schema [KKS92] or schema-less data [QRS+95]. Given these various motivations, they exist under different forms in the literature. However, all are more or less equivalent to the GPEs presented in this paper.

A GPE is a path expression containing, on top of more standard features, variables of two new kinds: path and attribute. As will be illustrated with different examples, these variables allow easy navigation through the composition graph of database objects.

Let us consider a first example. The language we use is OQL extended with GPEs [CACS94]. The schema is composed of five classes: Volume, Chapter, Section, Paragraph, Text. We are not interested here in the exact structure of the various classes. We just need to know that a volume has a set of chapters, a chapter a set of sections, a section a set of paragraphs and a paragraph has a text part of class Text. Finally, *Encyclopedia* is a database name identifying a set of volumes.

Example 2.1. The following query returns the couples of volumes referencing each other and such that the title of the first volume starts with "Digital". Note that this query could not be formulated using standard OQL.

`select`	v1, v2
`from`	Encyclopedia{v1} @P(x), Encyclopedia{v2} @Q(y)
`where`	v1.title like "Digital*" and x=v2 and y=v1

The query **from** clause contains two GPEs. Both allow to navigate from some volumes (v1 or v2) contained in the encyclopedia, following some path (P or Q) and ending in some value (x or y). Note that the "@" character is used to introduce path variables. □

Former (naive) approaches for evaluating GPEs were introduced in [BRG88, CACS94, KKS92] and implemented somehow an intuitive understanding of GPEs: (i) look for all possible instantiations of attribute and path variables, (ii) replace the attribute and path variables by their instantiations, (iii) eliminate the not well-typed alternatives, (iv) union the remaining instantiated queries, optimize and evaluate the resulting query.

In [CCM96], we showed the lacks of this naive approach (exponential input to the optimizer, no rewriting previous to or intermixed with the GPEs instantiation) and proposed an algebraic treatment that we review now.

2.2. *The Algebraic Approach*

In the naive approach, GPEs are processed from a schema perspective before being evaluated on the object base. Our approach relies on the fact that these two instantiations are somehow unavoidable but offers some optimization possibilities that should be exploited. The main idea of our technique 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, we extended the algebra of [CM93] with two new operators, *S_inst* and *D_inst*, that instantiate GPEs from a schema and a data perspective.

Let us consider the algebraic translation of Query 2.1 shown on the left side of Figure 1. Formal definitions of the *S_inst* and *D_inst* operators along with syntactical definitions of GPEs, path and patterns are given in [Chr96]. The definition of the standard algebraic operators can be found in [CM93]. We now explain this translation.

Operation (1) allows us to view Encyclopedia as a set of tuples with one attribute $v1$. 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:

{ [v1: o1], [v1: o2], ...}

where $o1, o2$ are Volume objects belonging to *Encyclopedia*.

Operation (2) finds all the possible schema paths matching the GPE (G1) and starting in $v1$. In order to differentiate them from data paths, we call schema

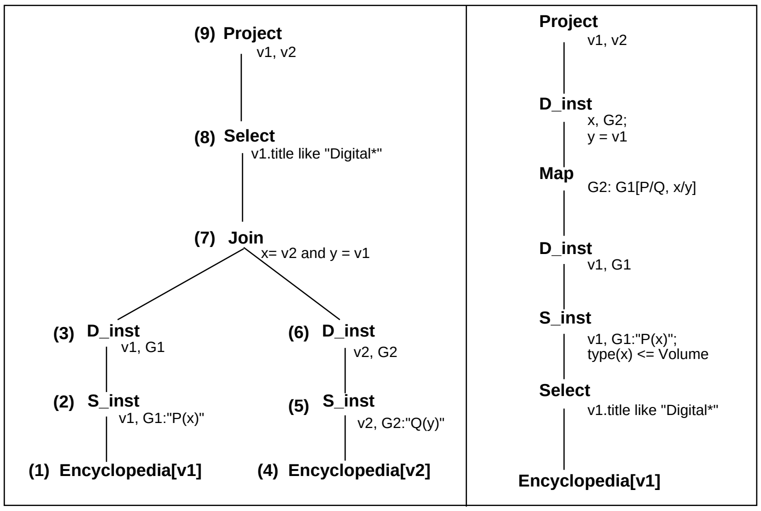

Figure 1. *Translation and optimization of query 2.1*

paths *patterns*. Operation (3) find all the data paths matching the patterns obtained by (2) on each volume of (1) and adds to (1) attributes corresponding to this evaluation (one per variable contained in the GPE). At the end of (3), we thus, have a set of the following form:

{ [v1: o1, x: "s1", P: p1], [v1: o2, x: "s2", P: p2], ...}

where *pi* is the database (i.e., instantiated) path going from *oi* to the string "*si*".

Operations (4,5,6) are similar to (1,2,3). Operation (7) is a join[1] which will result in a set with attributes *v1, x, P, v2, y, Q*. Operation (8) is a selection. The last operation is a projection.

Let us now consider the optimized expression corresponding to query 2.1 that is shown on the right part of Figure 1. One can note the following changes: (i) The selection operation has been pushed down the query tree. This presents several advantages. Firstly, it is expected that the (expensive) *D_inst* operation is now applied to a smaller set. Secondly, this allows the use of an index to evaluate the selection. (ii) The first *S_inst* operation now uses some type information that should reduce the schema lookup phase. (iii) The second *S_inst* operation has disappeared to be replaced by a renaming (*Map*) of the result of the first. (iv) The join operation has been integrated to the second *D_inst* operation, which follows Q from the $v2$ data elements that can be found at the end

1. This join operation has actually required some rewriting after the translation process.

of P. This somehow takes care of the first join condition ($x = v2$). The second join condition ($y = v1$) is given to the *D_inst* operation that will consider only elements satisfying it.

This new expression looks considerably better than the first and, given some database index on the volumes title, one can expect reasonable response time. However, as we will see next, algebraic rewriting does not always lead to reasonable plans.

2.3. *Remaining Problems*

Let us consider a new example featuring a query that illustrates well the power of GPEs. The query performs a kind of Unix *grep* over the database.

Example 2.2. The query returns the paths starting from an attribute of "Encyclopedia" and ending in some string containing "Polypody".

```
select   A, P, X
from     Encyclopedia{V}.A P(X)
where    X contains "Polypody"
```

□

The algebraic expression corresponding to this query is presented in Figure 2.

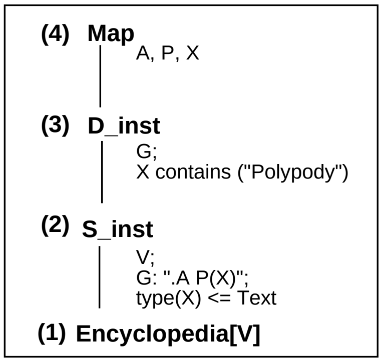

Figure 2. *Algebraic translation of the grep query*

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, we have to scan most of the database. This clearly implies that some alternative solutions must be found. Unsurprisingly, we propose to use full text indexes.

3. Optimizing GPEs using Full-Text Index

The work we presented in [CCM96] was preliminary. In this first proposition, we relied on a tree representation of schema instantiated GPEs that, as we will see, is not appropriate if one wants to use full text indexes. Also, two useful operations were hiding in the *D_inst* operator: data instantiation of a GPE and conversion into standard data. For these reasons, we had to modify the algebra. We introduced the concepts of paths and patterns, added functions to manipulate them, redefined the *S_inst* and *D_inst* operators and added two new ones *P_inst* and *FTI*.

In this section, we first illustrate the new framework by means of an example. We then show how query 2.2 can be rewritten using the new FTI operator and give some performance results obtained with a first prototype.

3.1. *The New Framework*

Let us first summarize our approach for optimizing GPEs. For this, let us consider Figure 3.

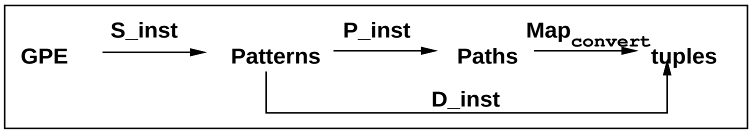

Figure 3. *How to process GPEs in the algebra*

As can be seen, GPEs are first transformed into patterns by means of the *S_inst* operator. A pattern thus corresponds to one possible schema instantiation of a given GPE. Then, there are two equivalent ways to transform patterns into (nearly) standard data (i.e., tuples eventually containing path and attribute values). The first direct method consists in applying the *D_inst* operator. The second consists in applying first the *P_inst* operator then the *convert* function to all the paths returned by *P_inst*. As we will see in Section 3.2, this cutting of the data instantiation process allows us to introduce the FTI operator.

We now illustrate each of the elements found on Figure 3. Formal definitions can be found in [Chr96].

Patterns. Patterns are syntactic elements used to generate (through instantiation), to filter or to interpret the paths involved in a generalized path expression. They are linear trees whose nodes represent data variables and whose edges represent access operations.

Example 3.1. The first part of Figure 4 shows three patterns, the last two corresponding to a possible schema instantiation of the GPE found in query 2.2. Pattern$\sharp$1 features two data variables ($X1$ and $X2$), one attribute variable (A) and one path variable (P). A node is labeled by a question mark when we are not interested in its possible data instantiations. Otherwise, a variable is used. A node is labeled with an α (resp. ω) marker to indicate the beginning (resp. end) of a path or attribute variable instantiation. □

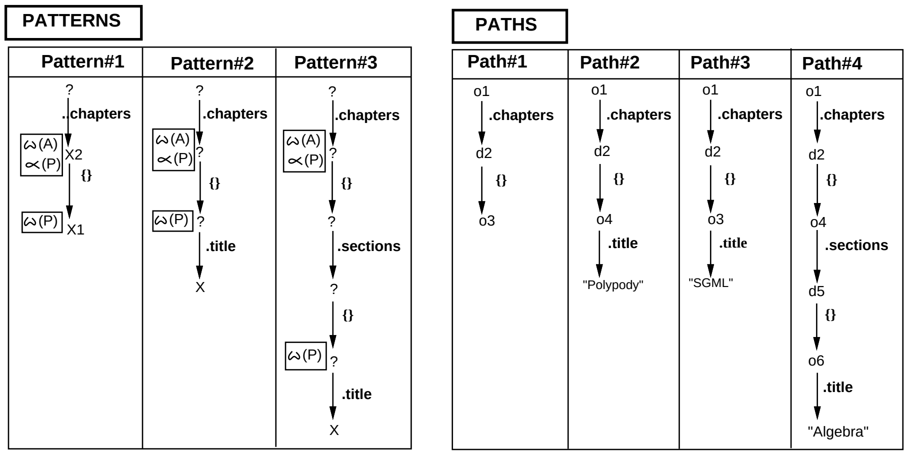

Figure 4. *Four paths and three patterns*

Paths: Paths are used to navigate through the objects and values of a database. They are linear trees whose nodes contain data items and whose edges indicate how we access one data item from another.

Example 3.2. Let us consider an object representing the volume of an encyclopedia, whose object identifier is "o1". This volume has a set of chapters identified by "d2". Let "o3" and "o4" be the identifiers of two of these chapters whose titles are "Polypody" and "SGML", respectively. The right part of Figure 4 shows some valid paths involving these data. For example, Path♯2 begins with object "o1", then goes through an attribute "chapters" to data "d2" which is a set containing "o4", etc. □

The S_inst (Schema Instantiation) Operator. The *S_inst* operator (first introduced in [CCM96]) takes a GPE as well as a set of tuples of a given type τ. Starting from that type, *S_inst* traverses the schema graph and returns the patterns within the schema graph that start at τ and match the GPE.

As an example, if we take the GPE "$AP.title(X)$" , and if we apply *S_inst* on the set of two volumes $\{[V : o1], [V : o2]\}$, the result is:

$$S\_inst_{V,G:\text{“}.AP.title(X)\text{”};\mathtt{length}(P)\leq 3}(\{[V:o1],[V:o2]\}) = \{[V:o1,G:Pattern\sharp 4],\\ [V:o1,G:Pattern\sharp 2],[V:o2,G:Pattern\sharp 4],[V:o2,G:Pattern\sharp 2],\ldots\}$$

The P_inst (Path Instantiation) Operator. The *P_inst* operator performs a walk through the composition graph of database objects. Given a set of values and a pattern, it returns all paths starting from one of the input values and matching the pattern. Note that this navigation through the composition graph is very similar to navigation mechanisms used in hypertext systems [AS92].

Example 3.3. As an example, consider the paths and patterns implied in the evaluation of query 2.2 (see Figure 4), we have:

$$P\_inst_{Pattern\sharp 2}(\{o1\}) = \{Path\sharp 2, Path\sharp 3\}$$

□

Convert, a Function to Interpret Paths: The *Convert* function links the path world to the standard data world. Given a pattern and a matching path, it returns a tuple whose attributes represent the pattern variables and their instantiation in the path. If the path does not match the pattern, *Convert* returns the undefined value (*nil*).

Example 3.4. Let us consider, once more, the paths and patterns of Figure 4. We obtain the following examples:

- $Convert_{Pattern\sharp 3}(Path\sharp 4) = [A:$ "chapters"$, P:$ "$\stackrel{\{\}}{\rightarrow} o4 \stackrel{.sections}{\longrightarrow} d5 \stackrel{\{\}}{\rightarrow} o6$"$, X:$ "Algebra"$]$
- $Convert_{Pattern\sharp 2}(Path\sharp 2) = [A:$ "chapters"$, P:$ "$\stackrel{\{\}}{\rightarrow} o4$"$, X:$ "Polypody"$]$
- $Convert_{Pattern\sharp 2}(Path\sharp 3) = [A:$ "chapters"$, P:$ "$\stackrel{\{\}}{\rightarrow} o3$"$, X:$ "SGML"$]$
- $Convert_{Pattern\sharp 1}(Path\sharp 1) = [X2: d2, A:$ "chapters"$, P:$ "$\stackrel{\{\}}{\rightarrow} o3$"$, X1: o3]$
- $Convert_{Pattern\sharp 3}(Path\sharp 3) = nil$

□

The D_inst (Data Instantiation) Operator. The *D_inst* follows all the paths corresponding to a given pattern, starting from a given data item. Then, it returns a set of tuples whose attributes represent all the possible instantiations of the pattern variables. It is obviously a combination of *P_inst* and *Convert* and is, as a matter of fact, defined as such. We keep it in the algebra because it offers some implementation possibilities worthwhile exploiting. As a matter of fact, and for the same reasons, we plan to add another operator called *G_inst* (GPE instantiation) performing both *S_inst* and *D_inst* operations.

Example 3.5. We consider the same example as above, from query 2.2, and with the specific condition X `contains` ("Polypody") (see Figure 2). If we apply the *D_inst* operator on the result of the previous *S_inst* operation, we obtain:

$$D\_inst_{V,G;X\,\texttt{contains}(\text{``}Polypody\text{''})}(\{[V:o1, G:Pattern\sharp 1], [V:o1, G:Pattern\sharp 2]\})$$
$$= \{[V:o1, G:Pattern\sharp 2, A:\text{``chapters''}, P:\text{``}\stackrel{\{\}}{\rightarrow} o4\text{''}, X:\text{``Polypody''}]\}$$

□

3.2. *Optimizing the Grep Query*

In order to optimize Query 2.2, we are now ready to introduce the FTI operator. The FTI operator represents the interface between the full text index mechanism and the algebra. We implemented this operator by coupling the Wais indexing mechanism with the O_2 database system. This first implementation is not optimal but allowed us to validate our optimization techniques. It is detailed in Section 5.

At this point, let us just say that given a textual predicate and a root of indexation, the FTI operator returns a set of couples (pattern, path) such that the paths matches their corresponding patterns, the root of the path is the root of indexation and the values at the end of the paths validate the predicate. Thus, the FTI operator somehow performs both *S_inst* and *D_inst* operations as well as a predicate evaluation. This is illustrated on Figure 5.

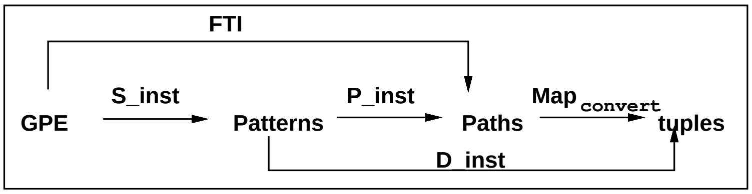

Figure 5. *Introducing FTI in the algebra*

Note that in order to obtain paths and patterns, we simply have to add to the full text indexing mechanism the ability to go back to some given root. We did that very simply using object identifiers, recursive calls and the SGML structure of the indexed documents (see Section 5). Still, with this rather trivial implementation we obtain good performances.

The use of the *FTI* operator on query 2.2 is illustrated on Figure 6. The equivalence used for the rewriting is given in [Chr96].

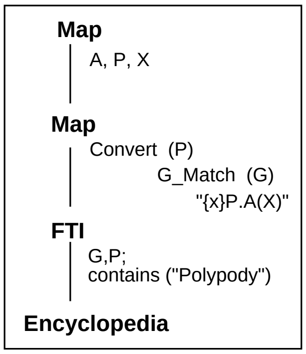

Figure 6. *Optimized expression of query 2.2*

The G and P parameters of the FTI operator will be used to name the resulting patterns (G) and paths (P). The G_match function used in the subscript of the $Convert$ operation allows to annotate the patterns returned by the FTI operator with the variables found in the given GPE.

Example 3.6. The following examples illustrate how the G_Match function works:

- $G\_Match_{\text{“}.AP.title(X)\text{”}}(? \overset{.chapters}{\rightarrow} ? \overset{\{\}}{\rightarrow} ?) = \emptyset$ (if the pattern does not match the GPE);
- $G\_Match_{\text{“}.AP(X)\text{”}}(? \overset{.chapters}{\rightarrow} ? \overset{\{\}}{\rightarrow} ? \overset{.title}{\rightarrow} ?) = \{? \overset{.chapters}{\rightarrow} ?^{\omega(A),\alpha(P)} \overset{\{\}}{\rightarrow} ? \overset{.title}{\rightarrow} X^{\omega(P)}\}$

Now, the result at the end of the FTI operation could be:

$$FTI_{G,P;X\,\mathtt{contains}(\text{“Polypody”})}(\{o1\}) = \{$$
$$[G : ? \overset{.chapters}{\rightarrow} ? \overset{\{\}}{\rightarrow} ? \overset{.title}{\rightarrow} ?, P : o1 \overset{.chapters}{\rightarrow} d2 \overset{\{\}}{\rightarrow} o4 \overset{.title}{\rightarrow} \text{“Polypody”}],$$
$$[G : ? \overset{.chapters}{\rightarrow} ? \overset{\{\}}{\rightarrow} ? \overset{.sections}{\rightarrow} ? \overset{\{\}}{\rightarrow} ? \overset{.title}{\rightarrow} ?,$$
$$P : o1 \overset{.chapters}{\rightarrow} d2 \overset{\{\}}{\rightarrow} o4 \overset{.sections}{\rightarrow} d5 \overset{\{\}}{\rightarrow} o6 \overset{.paragraph}{\rightarrow} \text{“Polypody”}], \ldots\}$$

□

To validate our approach, we evaluated the FTI algebraic expression on a database containing 10 chapters, 20 sections per chapter, 10 paragraphs per section and 1,500 words per paragraph (3,000,000 words at the leaves). The graphics on Table 1 presents the time spent (in seconds) to process both expressions on a Sparc20, one processor, 64M-bytes of RAM. The first line gives the selectivity (in %) of the "contains" predicate on the paragraphs. We did not evaluate the query without FTI. However, we expect it to take more or less the time of a dump (one and a half hours on our example database) and this, whatever the selectivity of the predicate. We see that FTI reduces this time to seconds.

Query's selectivity	1.5	3.0	4.5	6.0	7.5	9.0	10.5
Response time (in s)	9.1	18.2	27.3	36.4	45.5	57.6	84.8
Query's selectivity	12.0	13.5	15.0	30.0	45.0	60.0	75.0
Response time (in s)	87.9	112.1	124.2	227.3	375.8	487.9	609.1

Table 1. *Response time for the grep query using FTI*

4. Optimizing Standard Queries Using FTI

In the previous section, we have seen how the evaluation of queries featuring GPEs can be greatly optimized by using a FTI mechanism. But the FTI operator, which was first designed in order to evaluate GPEs in a reasonable time, can also be very useful for more standard query optimization.

In this section, we first demonstrate by means of an example, the limitations of current systems concerning the evaluation of text queries. We then demonstrate how we can use the new algebraic operations, defined in Section 3, for standard query optimization and we show that these operators — together with appropriate equivalences — allow the introduction of FTI evaluation on any part of a database query.

We will show that we are now able to combine full-text and database index. This mix of database and FTI evaluation has many advantages. On the functionality side, consider a database used as a server for existing documents (e.g., a front-end to a Web server). With our approach, the database can store only a *view* of the documents (e.g., some structural information, title, authors' names, etc.) and still answer database queries on their textual content. Part of these queries will be evaluated using the FTI. The result of the FTI evaluation will then be used to further evaluate the query. On the performance side, consider a database containing plenty of text. Queries with a `contains` predicate could be evaluated by the standard query evaluation mechanism, or by coupling the database with a full-text indexing mechanism. To conclude, the performance results we give at the end of this section favor the use of the FTI/database mix.

4.1. *A Standard Query on Text*

Example 4.1. The following query selects chapters along with their sections and paragraphs such that the paragraphs contain the word "Polypody" and the chapter is reviewed by "Dupont".

```
select   c, s, p
from     c in myencyclopedia.chapters, s in c.sections,
         p in s.paragraphs
where    p.text contains "Polypody"
         and c.reviewer contains "Dupont"
```

□

The query is translated into the algebraic expression shown on the left-hand side of Figure 7.

Operation (1) constructs a set of tuples with one attribute c taking its value in the set of chapters of the encyclopedia. Operation (2) is a dependency join (a join requiring nested loop evaluation because of some dependency of the second parameter on the first). It expands the tuples with a new attribute s taking its values in the set of sections of c. Operation (3) acts analogously for paragraphs. Operation (4) adds two attributes whose values are the reviewers' name and the paragraphs text. Operation (5) and (6) are the final selection and projection(Map).

D-joins are usually evaluated through nested loops. However, in [CM94a], we show how they can be rewritten into standard joins using class extents. Also, the selection can be pushed. Thus, the query can be rewritten into the expression

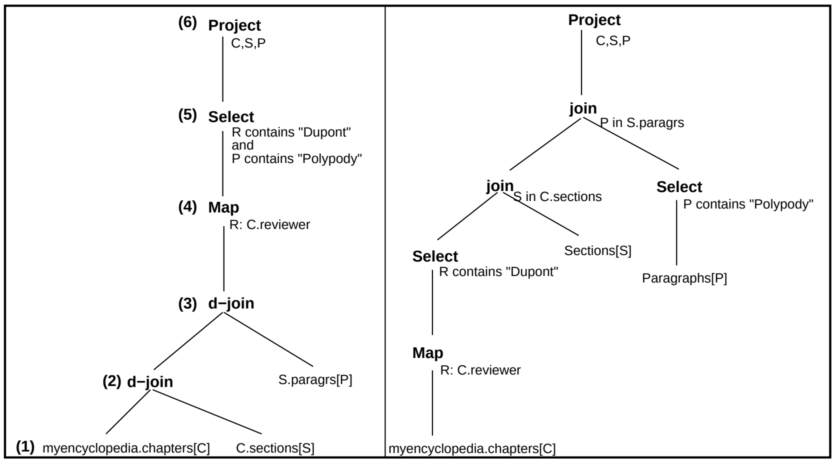

Figure 7. *Algebraic translation of query 4.1*

shown on the right-hand side of Figure 7 where Paragraphs and Sections denote the extents of classes **Section** and **Paragraph**.

Now, with appropriate indexes, the join between chapters and sections will be fast. However, in the absence of text indexing facilities, the selection on paragraphs will be very costly. Evaluating this selection at the end will not help much since there is still a join to be performed. Thus, a mix of both expressions might be the solution: first select the interesting chapters hoping there are few of them; then follow the path to the sections and paragraphs and check the condition. Obviously, unless the predicate on chapters is very selective or the database is very small, this solution will be very costly. This is one of the motivations which lead to make use of full text indexing facilities.

4.2. *Rewriting the Query with FTI*

Using appropriate algebraic equivalences (see formal equivalences and proofs in [Chr96]), we are able to rewrite the initial query into the two expressions given in Figure 8. The expression on the left-hand side (FTI+DB) uses FTI on the leaves to select the appropriate paragraphs. Then it finds the paths back to the root using database indexes. The expression on the right-hand side (FTI) uses only an FTI index which returns the paths going from the encyclopedia to the appropriate strings.

Now, let us give an intuition of how this rewriting works. In order to introduce the *FTI* operator and take advantage of the indexing mechanism, we need some equivalences that allow us to transform sequences of maps and *D-joins* into a path query. For this, the paths which occur in the form of a sequence of *D-joins* and *Map* operators have to be transformed (cut and pasted) into a pattern which is indexed by an FTI. The process of cutting and pasting the paths resembles very much the process of cutting and pasting database paths in order to introduce path indexes [KM90].

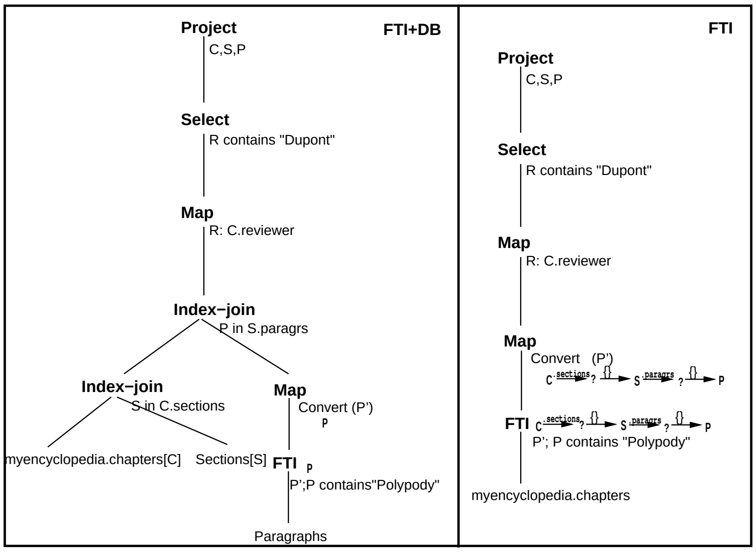

Figure 8. *FTI plan for Query 4.1*

To validate the mixed approach, we evaluated both expressions on the database introduced in the previous section. Resulting figures (in seconds) are presented in Table 2. We can draw two conclusions from these results. The first is that queries featuring FTI offer very good response times when less than 10% of the paragraphs are selected (the most probable cases) and reasonable ones in all cases. Note that we would perform better with a more efficient implementation. The second is that the mixed approach offers a very good response time. It would be interesting to see if we could derive a cost model in order to determine the appropriate mix between FTI and database indexes.

Query's selectivity	1.5	3.0	4.5	6.0	7.5	9.0	10.5
Response time (FTI only)	18.2	36.4	51.9	67.5	67.6	75.3	85.7
Response time (FTI+DB)	2.60	7.79	13.0	18.2	20.8	23.4	26.0
Query's selectivity	12.0	13.5	15.0	30.0	45.0	60.0	75.0
Response time (FTI only)	96.1	111.7	122.1	226.0	337.7	433.8	561.0
Response time (FTI+DB)	28.6	31.2	36.4	70.1	106.5	140.3	181.8

Table 2. *Response time for queries FTI and FTI+DB*

5. Coupling an OODB with a Full-Text Index

This section briefly describes the interface we developed in order to loosely couple O_2 and WAIS (which provided us with full-text indexing capabilities). Of course, such a loose coupling with an experimental — that is unoptimized — interface results in performance degradations not expected in a tightly integrated system. Nevertheless, even this testbed allowed us to validate our optimization techniques. A much better performance is expected for a tight coupling or an oodbms with integrated full-text indexing capabilities.

In the current prototype, we generate SGML documents corresponding to the objects stored in a database or a subset of them. For each object, a document is generated. All generated documents are then indexed. This step — called *dumping the database* — is described in the next subsection. Note that our optimization techniques can be used without changes on the converse approach (i.e., generating a database - or a database view - corresponding to some indexed documents). The last subsection describes how the interface interprets the query results from the full-text index.

5.1. *Dumping the Database*

There are two pre-requisites to the optimization techniques we introduced in previous sections: (i) correspondences between objects and documents representing the same entity must be maintained and (ii) a means to interpret the generated documents from a database perspective must be provided.

This can be done in different ways [ACM95]. Nevertheless, as the O_2 database system uses physical object identifiers, this means that they cannot be exported. As a consequence, we introduce logical identifiers and store within an O_2 database the correspondences between the logical and physical identifiers.

WAIS allows the full-text indexing of a set of documents. Given a retrieval predicate on strings, it returns all the documents satisfying the predicate. Furthermore, it returns also the offsets of the relevant strings occuring within the documents. As already mentioned, each database object is represented by a single document. This implies that, given a predicate, the WAIS system returns the logical identifiers of the objects as well as the offsets within the object/document where the strings occur.

Since database queries refer to, say, attribute values and not just strings occurring somewhere in the object, we have to validate that the string occurring in the document (representing an object) corresponds to the attribute referred to in the database query. Thus, we need to structure the documents. We do this by introducing SGML tags representing database (schema) information.

For instance, let us consider an object representing a section in a book whose logical identifier is "23". Further assume that the section is composed of a list of paragraphs and some other informations. Among the paragraphs, the first one has the logical identifier "24". The following string is our SGML representation of the section where tags are between brackets:

```
<doc><id>23<val><tuple><att>paragraphs<list><ob>24</ob>
....</val></doc>.
```

One of the WAIS most interesting features is its ability to index fields (i.e. parts of the documents between specific tags). We use this to index cross references between objects (by indexing the field between `<ob>` and `</ob>` tags).This specific index allows the retrieval of all objects which have a reference to a given object. More details can be found in [Sim95].

Let us finally give some numbers concerning the database dump. We dumped a database of 81 M-bytes. It took one and a half hours to browse the entire database and generate the appropriate text and data structures. The size of the dump was 17M-bytes. The difference in size is easily explained by the database overhead incurred on each string of a text. The WAIS index consumed 35M-bytes.

5.2. *Interpreting Results of WAIS Queries*

Given a string predicate, WAIS returns a set of documents and offsets in these documents. We now have to interpret this from a database point of view.

Using the SGML tags, we can find the path that goes from an object to the occurrence of the predicate string it contains. For example, we can infer in which logical part of the object the string occurs. This is a good start but we need more.

Typically, database queries are evaluated on some persistent root(s) (relations in relational systems, extents or names in oodb). Thus, knowing that an object satisfies a predicate is useful only if we can trace back a path from a root object occurring in the database query to the object retrieved by WAIS. As was explained before, we can find all the objects referencing a logical identifier (using the `<ob>` field). Performing this operation recursively, we are able to trace the object back to some root and build the corresponding path.

6. Conclusion

A new algebraic framework for the optimization of queries containing generalized path expressions was presented. The algebraic approach is very flexible and allows to use full-text indexing in conjunction with standard rewriting techniques. In this context, we show how full-text indexing can be used to optimize queries with generalized path expressions as well as standard OQL queries. The proposed optimization approach is quite general and can be applied even when the textual content of documents reside outside the database (e.g., handled by a traditional information retrieval engine). A first prototype adding full-text indexing capabilities in the O_2 database system was described. Some performance measures on this prototype validate the approach.

7. References

[ACM95] S. Abiteboul, S. Cluet, T. Milo. A Database Interface for Files Update. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, San Jose, California, 1995.

[AS92] B. Amman M. Scholl. Gram: A Graph Data Model and Query Language. In *ECHT'92*, Milano, Italy, December 1992.

[BRG88] E. Bertino, F. Rabitti, 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, M. Scholl. From Structured Documents to Novel Query Facilities. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Minneapolis, Minnesota, 1994.

[CCH92] J. Callan, B. Croft, S. Harding. The INQUERY Retrieval System. In *DEXA'92*, pages 78–83, Valencia, Spain, 1992. Springer-Verlag.

[CCM96] V. Christophides, S. Cluet, G. Moerkotte. Evaluating Queries with Generalized Path Expressions. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Montreal, Quebec, 1996.

[CDY95] S. Chaudhuri, U. Dayal, T. Yan. Join Queries with External Text Sources: Execution and Optimization. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 410–422, 1995.

[Chr96] V. Christophides. Document Structurés et Bases de Données Objet. PhD thesis, Conservatoire National des Arts et Métiers, October 1996.

[CM93] S. Cluet G. Moerkotte. Nested Queries in Object Bases. In *Proceedings International Workshop on Database Programming Languages*, 1993.

[CM94a] S. Cluet G. Moerkotte. Classification and Optimization of Nested Queries in Object Bases . In *BDA*, pages 331–349, 1994.

[CM94b] M. Consens T. Milo. Optimizing Queries on Files. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1994.

[CS92] W. B. Croft L. A. Smith. A Loosely-Coupled Integration of a Text Retrieval System and an Object-Oriented Database System. In *SIGIR'92*, pages 223–232, Copenhagen, Denmark, June 1992. ACM.

[CS93] S. Chaudhuri K. Shim. Query Optimization in the Presence of Foreign Functions. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 529–542, Dublin, Ireland, 1993.

[KKS92] M. Kifer, W. Kim, Y. Sagiv. Querying Object-Oriented Databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 393–402, 1992.

[KM90] A. Kemper G. Moerkotte. Advanced Query Processing in Object Bases Using Access Support Relations. In *Proceedings of the International Conference on Very Large Data Bases*, pages 294–305, 1990.

[Pfe95] U. Pfeifer. freeWAIS-sf. University of Dortmund, version 0.5, 1995.

[QRS+95] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, J. Widom. Querying Semistructured Heterogeneous Information. In *Proceedings of the International*

Conference on Deductive and Object-Oriented Databases (DOOD), pages 319–344, December 1995.

[Sim95] J. SIMÉON. Recherche en texte intégral et Bases de Données Orientées-Objet. Master's thesis, Université de Nancy I, 1995.

[ver95] Verity, Inc., California. *Topic User's Manual*, 1995.

Vassilis Christophides *received his PhD from the Conservatoire National des Arts et Métiers (CNAM) of Paris. From 1992 until 1996 he carried out his research work in the Verso group at INRIA in the topics of Object Database and Documents Management Systems. His main interests are query languages and optimization techniques for (semi)structured documents as well as intelligent mediation services for the integration of heterogeneous, distributed information. He is currently a researcher at the Institute of Computer Science, FORTH - Hellas and visiting professor at the Computer Science Department of Crete University.*

Sophie Cluet *obtained a PhD in Computer Science from the University of Paris-Sud in 1991. She is currently a researcher in the Verso group at INRIA. Her initial research concerned the design and optimization of query languages for object-oriented DBMS and she is one of the designers of OQL. Her other main topic of interest concerns communication between databases and other computer worlds (Unix file systems, WWW, SGML tools, etc.).*

Guido Moerkotte *studied computer science at the universities of Dortmund, Karlsruhe and Massachusetts at Amherst. He is currently a professor at the University of Mannheim. His research interests are object-oriented databases, data warehousing and queries against the web.*

Jérôme Siméon *is a PhD student in the Verso Group at INRIA. He obtained engineering degrees from École Polytechnique and Supelec, and his Master degree from the University of Nancy in 1995. His topics of interest are heterogeneous databases, data conversion and query optimization.*