Full-Text Capabilities for Querying XML Repositories: a Formal Model

GIACOMO BURATTI
University of Camerino
Department of Mathematics and Informatics
Via Madonna delle Carceri 9, Camerino
ITALY

DANILO MONTESI
University of Bologna
Department of Computer Science
Mura Anteo Zamboni 7, Bologna
ITALY

Abstract: In this paper we address the important issue of establishing a formal background for the management of semi-structured data. We define a data model and propose an algebra for XML. The algebra, clearly inspired by relational algebra, is quite intuitive; nevertheless it is able to represent most of XQuery Full-Text expressions. Standard and full-text operators are tightly integrated and can be combined each other.

Key–Words: XQuery, Full-Text, XML, Data Model, Algebra.

1 Introduction and Related Work

In the last years the explosion of the Web and the ever higher interest for applications dealing with heterogeneous data sources revealed the inadequacy of the classical relational model. Semi-structured data [3] consequently gained growing attention and XML became the de facto standard for exchanging information over the Web and integrating heterogenous data sources. Several query languages for XML have been proposed until XPath and XQuery [6] have been widely accepted as the standard query languages.

The study of semi-structured data recently received a further boost from a new trend: the integration of structured, semi-structured and unstructured data into a more general framework. In the past, these three kinds of data have been extensively studied as separated worlds, leading to incompatible theories, models and languages. A convergence between these diverging theories is made necessary by the consideration that many today’s applications, like biological data [4], have to cope with data covering the entire spectrum. While XQuery is suitable to query a data-centric XML repository, searching relevant documents in a document-centric repository requires the use of Information Retrieval techniques; to this aim, W3C has recently published a Working Draft, mainly based on the previously proposed language TeXQuery [2], for extending XQuery with Full-Text operators [7].

Many different data models and algebras for XML [9, 8] have been presented in the last years; very few works [1], however, deal with the usage of IR-like techniques. Moreover, in our opinion, none of them fulfill all the requirements. Some, in fact, provide only simple, XPath-like, constructs, though restructuring constructs are of great importance in the XML context; others are based on concepts excessively diverging from classical relational algebra, thus making it difficult to (partially) reuse the work done in the relational context; on the contrary, others try to transform the problem of managing semi-structured data into that of managing structured data, thus losing the peculiarities of XML.

In this paper we propose a novel algebra for XML. Our algebra is a natural extension of the relational algebra, and is based on a simple data model in which trees and forests are the counterpart of relational tuples and relations. The algebra is quite intuitive and is able to represent many XQuery FLWOR expressions. Our data model and algebra also permit to represent full-text queries and ranked retrieval, and this is the main contribution of our work.

We begin in Sect. 2 by defining a data model for representing XML documents. Sect. 3 presents the algebraic operators, which are used in Sect. 4 to translate XQuery Full-Text expressions. Finally in Sect. 5 we draw some conclusions and outline the future work.

2 Data Model

In our data model an XML document is represented as a rooted, ordered, labeled tree. A tree is composed by a set of vertices, or elements, connected with arcs.

Formally, an element $e$ is a tuple $(k, n, A, v, p)$. Elements always have a name ($n$) and an identifier
The element identifier, actually, is not a “true” identifier, because newly created elements have a null identifier and multiple copies of an element share the same identifier; however, when an element is stored in an XML repository, the system is supposed to assign it a unique not-null identifier. Each element, except for the root element, also has a parent element (p). Moreover, an element can have a value (v) and an ordered list of attributes (A), where each attribute has a name and a value. We refer to each element property with the notation e.x., e.g. e.v is the value of the element e.

Value of elements, i.e. the text contained in them, is represented as a list of tokens, each of which is assigned a numeric position relative to the entire tree. Attributes values are instead separately tokenized: each attribute’s first token has position 1. An element can have mixed content, i.e. it can contain character data interspersed with child elements. We keep track of the position of child elements inside the text of an element by assigning, to the first token in a sub-element, the numeric position of the token immediately preceding it in the parent element plus one. In practice, the enumeration of tokens proceeds from the beginning of the XML document to the end, as it was the case in the example of tokenization over an XML document with mixed content is shown in Fig. 1.

![Figure 1: Tokenization of an XML document with mixed content.](image)


Ordering between elements is represented by a collection of ordered lists of element pointers: sibling elements have pointers to them contained in the same list, and the ordering of that list represents the ordering of the corresponding elements.

Given a tree, we can pick many subtrees from it. The concept of subtree is based on the notion of elements strict equality and on the order preservation property. Informally, two elements are exactly equal if they are the same element. For example, if we define two views over the same tree and an element is retained (without modifications) in both views, the two views will contain two elements which are strictly equal. The order preservation property also plays a crucial role; every operator of our algebra preserves ordering.

Trees are contained in forests, which are themselves ordered. In a certain way, trees and forests are the counterpart of tuples and relations in the relational model: our algebraic operators manipulate forests (that contain trees) and return a forest. However, trees and forests are ordered, while tuples and relations are not; moreover, trees contained in a forest are not required to share the same structure. Many subforests can be picked from a forest; as in the case of subtrees, the formal definition of subforest is based on the notion of tree strict equality (two trees are strictly equal if they are the same tree) and on the order preservation property.

3 Algebra Operators

In what follows we informally present the algebraic operators; due to space limitation, we omit the formal definitions and postpone examples to Sect. 4. The algebra is closed: all the operators (which are shown in Table 1) take in forest(s) and return a forest; consequently they can be composed with each other.

3.1 Set Operators

Our algebra is equipped with union and difference operators, which are quite similar to their relational counterparts.

Union takes in two forests and return a new forest composed by the trees contained in the two input forests; difference takes in two forests and returns a subforest of the first input forest, composed by those trees which are not included in the second input forest. Difference is based on the notion of trees strict equality presented in Sect. 2: a tree from the first input forest is not retained in the output if the second input forest contains a strictly equal tree. Both union and difference preserve tree ordering; this implies that union
Selection operates instead a horizontal putted, but only the attributes of interest are retained. In relational algebra, projection operates a vertical put. The selection condition can refer to any element properties and group input trees according to the selection condition. In fact what is needed (for the tree to be returned) is that at least of the subtrees satisfies the condition.

3.3 Product and Join

In relational algebra, product combines in every possible way tuples from the first relation with tuples from the second relation; the resulting relation has all the attributes of the first relation plus all the attributes of the second relation. Our product operator behaves similarly: it combines in every possible way all the trees from the first forest with all the trees from the second forest. Combination among two trees is done by creating a new root node called prod_root, whose left and right child are, respectively, the tree from the first input forest and the tree from the second input forest.

As in the relational world, join is a derived operator that combines a product and a selection; the selection condition compares a property value of an element of the first tree with a property value of an element of the second tree. Product and join operators preserve ordering, in the sense that the combination of trees occurs following the order of the input forests.

3.4 Deletion

Deletion takes in a forest and returns a new forest containing subtrees of the input trees, obtained by pruning from the original trees those subtrees that satisfy a deletion predicate. Deletion preserves ordering, either between trees or between elements of a tree.

Informally, this operator completes the features of projection and selection operators. In fact it permits to freely delete a portion of a tree (which is not possible using projection) and to delete every subtree that do not respect the selection condition (which, as already noted, does not occur using selection).

3.5 Grouping and Duplicate Elimination

Grouping allows to find distinct values of some element properties and group input trees according to those values. The output contains a distinct tree for each distinct combination of properties values found in the input forest; each root element (called group_root) has an attribute (named p1, p2 etc.) for each property of interest. The grouping predicate also permits to establish which part of the input trees should be retained in the group trees, through a list of

### Table 1: Algebraic operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union</td>
<td>$F \cup F'$</td>
</tr>
<tr>
<td>Difference</td>
<td>$F \setminus F'$</td>
</tr>
<tr>
<td>Projection</td>
<td>$\pi(F)$</td>
</tr>
<tr>
<td>Selection</td>
<td>$\sigma_{[\gamma]}(F)$</td>
</tr>
<tr>
<td>Product</td>
<td>$F \times F'$</td>
</tr>
<tr>
<td>Join</td>
<td>$F \bowtie F'$</td>
</tr>
<tr>
<td>Deletion</td>
<td>$\delta_{[\gamma]}(F)$</td>
</tr>
<tr>
<td>Grouping</td>
<td>$\sum_{(\lambda_1, \lambda_2, \ldots, \lambda_1', \lambda_2', \ldots)}(\gamma, x, \ldots)(F)$</td>
</tr>
<tr>
<td>Duplicate elimination</td>
<td>$\nu_{\lambda_1, \lambda_2, \ldots}(\gamma, x, \ldots)(F)$</td>
</tr>
<tr>
<td>Ordering</td>
<td>$\omega_{\lambda_1, \lambda_2, \ldots}(\gamma, x, \ldots)(F)$</td>
</tr>
<tr>
<td>Tree Construction</td>
<td>$\iota_{\lambda_1, \lambda_2, \ldots}(\gamma, x, \ldots)(F)$</td>
</tr>
<tr>
<td>Full-Text (FT) Selection</td>
<td>$\varsigma_{\lambda_1, \lambda_2, \ldots}(\gamma, x, \ldots)(F)$</td>
</tr>
<tr>
<td>FT Score Assignment</td>
<td>$\zeta_{\lambda_1, \lambda_2, \ldots}(\gamma, x, \ldots)(F)$</td>
</tr>
<tr>
<td>FT Selection with Score</td>
<td>$\xi_{\lambda_1, \lambda_2, \ldots}(\gamma, x, \ldots)(F)$</td>
</tr>
<tr>
<td>Top-K FT Selection</td>
<td>$\tau_{\lambda_1, \lambda_2, \ldots}(\gamma, x, \ldots)(F)$</td>
</tr>
<tr>
<td>Threshold FT Selection</td>
<td>$\omega_{\lambda_1, \lambda_2, \ldots}(\gamma, x, \ldots)(F)$</td>
</tr>
</tbody>
</table>

is not commutative an unavoidable deviation from relational algebra caused by the importance of order in the semi-structured data model.

3.2 Projection and Selection

In relational algebra, projection operates a vertical decomposition of the input relation: every tuple is outputted, but only the attributes of interest are retained. Selection operates instead a horizontal decomposition: only the tuples of interest are outputted, and every attribute is retained. Our projection and selection operators behave in a similar way: while every input tree contributes, with the subtrees of interest, to the projection output, only the trees of interest contribute, with their entire content, to the selection output. Both operators preserve order between elements and trees.

The subtrees of interest are specified in the projection predicate through a path expression, a concept almost identical to that used in XPath, except for the fact that it can not contain selection conditions. The selection predicate combines a path expression and a boolean expression specifying the selection condition. The path expression locates, for each input tree, a set of subtrees; in practice, it is used to operate a temporary projection on the input tree. Each subtree belonging to the temporary projection result is then checked: if at least one of them satisfies the selection condition, the original input tree is added to the selection output. The selection condition can refer to any element property, plus the value of some aggregate functions like position, count, max etc.

While in relational projection each input tuple corresponds to exactly one output tuple, in our projection each input tree corresponds to zero or more output trees; in fact a path expression could not be found in one input tree, or it could be found more than once. For what concerns selection, it is worth noticing that a returned tree could include a subtree, located by the path expression, not satisfying the selection condition; in fact what is needed (for the tree to be returned) is that at least of the subtrees satisfies the condition.
path expressions. Grouping preserves ordering: group trees appear in the output forest in the same order as the corresponding elements in the input forest.

Using the grouping operator, we define a derived duplicate elimination operator; it is identical to grouping, except that no list of path expressions identifying subtrees to retain is specified. Consequently, the resulting forest will contain trees composed by the only root element with its attributes.

### 3.6 Ordering

Ordering takes in a forest and returns a new forest containing the same trees, but arranged in a (possible) new order. The ordering predicate is a list of ordering directives, each of which specifies the element to consider (through a path expression), the property whose values must be compared and the ordering direction (ascending or descending).

### 3.7 Tree Construction

Tree construction permits to build new trees, specifying in the construction predicate name and value of elements to build, name and value of their attributes and the hierarchy of elements. The predicate is in fact a list of element construction specifications, each of which is formed by: the name of the element; the value (possibly null) of the element; a list (possibly empty) of attributes; a list (possibly empty) of child elements, each of which is an element constructor specification itself.

The tree construction predicate usually contains some reference to the input forest, in the form of path expressions identifying some elements, possibly followed by the name of an element property. The operator is applied separately to each input tree, in the order they appear in the input forest; consequently, at least one output tree is built for each input tree. If the predicate does not contain any reference to the input forest, the entire input forest is added to the newly created tree as child of the rightmost leaf element; this feature is useful if we want to create a root element that contains the entire output forest.

### 3.8 Full Text Operators

Our algebra is equipped with two basic full-text operators: full-text selection and full-text score assignment. The first one performs a full-text search using the boolean model: a tree satisfies the selection condition or it does not satisfy the condition at all. The second one performs instead a ranked retrieval.

Full-text selection behaves in a way similar to that of basic selection operator presented in Sect. 3.2: it operates a horizontal decomposition of the input forest, retaining only those trees having at least one subtree satisfying the full-text selection predicate. The full-text selection predicate we use is quite simple; it allows to search one or more words or phrases into the full-text value of an element or into the value of an attribute. Moreover, it supports proximity search, i.e. searching two or more words with a distance between one and another not greater than a defined threshold.

A typical IR system (and XQuery Full-Text too) provides many more options: use of stemming, thesaurus, stopwords etc; in this paper we do not deal with such options, because their usage has no impact on the nature of the algebraic operator.

On the other side, full-text score assignment does not perform a selection: each input tree is returned, without filtering. What it does is to assign to each tree a score value, that represents the level of satisfaction of the full-text condition; this value is stored in a newly created root attribute named score. The full-text condition is specified in the score assignment predicate, in the same way as in the full-text selection predicate. However, a weight can be assigned to each word or phrase in order to specify which words (or phrases) should highly influence score calculation. Moreover, an extra parameter \( f \), which can be thought of as a function pointer, specifies the way the score is calculated. The availability of such parameter let the user freely decide which technique to use among those provided by its system, hence providing a higher flexibility than that of XQuery Full-Text [7].

Using these two full-text operators plus some basic operators (namely, ordering and selection), some useful derived full-text operators can be defined:

- **full-text selection with score**: combines full-text selection and score assignment, thus permitting to select those trees that satisfy the condition, and distinguish among them those that “better” satisfy the condition;
- **top-K full-text selection**: assigns a score to each input tree and returns the \( k \) trees with highest score;
- **threshold full-text selection**: assigns a score to each input tree and returns those whose score is higher than a defined threshold \( \tau \).

### 4 Translating XQuery Expressions

In this section we show how most of XQuery expressions can be translated into algebraic expressions. We first show how each clause of a FLWOR expression is treated; then we translate a complex XQuery expression.
4.1 The for Clause

A for clause with a single variable binding specifies a list of path expressions $\lambda$, possibly interleaved with selection conditions (possibly a full-text condition, i.e. a $ftcontains$ expression) $\gamma$. Such a clause is translated into the following algebraic expression, where “1” is a special case of path expression that returns the first child of an element:

$$S_{1[\gamma]}(\pi_{1}[\lambda](\ldots(S_{1[\gamma]}(\pi[\lambda]("docname")))))).$$

Here $S$ is either the selection operator $\sigma$ or the full-text selection operator $\varsigma$, depending on the kind of selection condition present in the clause. If the clause also contains a score option, the full-text selection operator is substituted by the full-text selection with score operator $\varpi$.

If there are multiple variable bindings, each binding is translated as in the base case; the algebraic expressions are then assembled using the product operator. If one of the variable bindings refers to another variable, a further selection is applied, based on the strict equality between an element of the first tree (which represents the first variable binding) and an element of the second tree (which represents the second variable binding).

4.2 The let Clause

A let clause is similar to a for clause, but it binds a variable to the entire result of its associated expression, without iteration. In order to maintain this distinction, a let clause is translated into an algebraic expression which returns a single tree. This goal is achieved using the tree construction operator $\iota$, which creates a root node named $\text{let}_\text{root}$; the result of the expression associated with the let clause will be inserted as child of that node. A let clause is then translated into the following algebraic expression:

$$\text{let}_\text{root}\text{(null,null,null)}(\sigma_{1[\gamma]}(\pi_{1}[\lambda](\ldots(\sigma_{1[\gamma]}(\pi_{1}[\lambda]("docname"))))))).$$

A special case of let clause occurs when it contains a score variable declaration; in this case the translation is done using the full-text score assignment operator $\xi$.

Algebraic expressions resulting from multiple let clauses are assembled using the product operation. In the same way they are assembled with algebraic expressions resulting from for clauses.

4.3 The where Clause

A simple where clause is of the form $\text{where } \$i \lambda[\gamma]$. Such clause is translated into the algebraic expression $S_{\forall \lambda'[\gamma]}(A)$, where $A$ is the algebraic expression representing the input forest, $\lambda'$ is a path expression that locates the nodes bound to the variable $\$i$ and $S$ is a selection $\sigma$ or a full-text selection $\varsigma$.

If the clause refers to two variables, a join operation is done between the forests representing the two variables. Existential quantifiers are treated as they were a for clause with reference to another variable binding, while universal quantifiers are translated by applying a selection with the inverted selection condition and operating a difference between the original forest and the forest resulting from the selection.

4.4 The order by Clause

Translation of an order by clause is done using the ordering operator. This translation is quite straightforward and does not need further analysis.

4.5 The return Clause

A typical return clause contains some element constructors plus some references to previously defined variables. It it naturally translated using the tree construction operator.

Sometimes a FLWOR expression can be nested inside a return clause. In this case the internal for, let, where and order by clauses can be translated as previously seen; the resulting expression should then be right joined with the expression resulting from the outer clauses. The outer join can be achieved using a product operation followed by a deletion of the left subtrees that does not satisfy the inner where clause; then a grouping on the basis of the right subtrees is done, and the left subtrees are maintained.

4.6 Translating Complex XQuery Expressions

We now present an example of translation of a complex XQuery Full-Text expression into our algebra; the example is taken from W3C XQuery Use Cases [5], namely from Use Case “XMP” Q4, and has been complicated adding a full-text condition. Suppose we have a bibliography XML file; for each author in the bibliography, we want to list the author’s name and the titles of all books by that author that contain some title. We have a bibliography XML file; for each author in the bibliography, we want to list the author’s name and the titles of all books by that author that contain some title.
\[ P = \gamma \{ \text{result} \} (\text{null}, \text{null}, (\text{"author"}, \text{null}, \text{null}, (\text{"last"} (//\text{group_root.A}[\text{p1}].v, \text{null}, \text{null}), (\text{"first"} (//\text{group_root.A}[\text{p2}].v, \text{null}, \text{null}))), //\text{group_root/title})\} \]

\[ P' = (/\text{prod_root}/\text{group_root.A}[\text{p1}].v, //\text{prod_root}/\text{group_root.A}[\text{p2}].v), //\text{prod_root/book/title}) \]

\[ P'' = //\text{prod_root/book/author}[\text{NOT}/\text{last}.v = //\text{prod_root/group_root.A}[\text{p1}].v \text{ AND } /\text{first}.v = //\text{prod_root/group_root.A}[\text{p2}].v] \]

where

\[ \delta/\text{prod_root/book}[\text{NOT}/(\text{author})] \]

\[ \delta/\text{group_root.A}[\text{p1}].v/\text{group_root.A}[\text{p2}].v \]

\[ \nu/\text{author}/\text{last}/\text{author}/\text{first}(\pi/\text{author}(\text{"bib.xml"}))) \]

\[ \pi/\text{bib/book}(\text{"bib.xml"}) \]

5 Conclusions and Future Work

In this paper we have presented a data model for representing XML documents; the model covers either data-centric or document-centric repositories. An algebra for querying these repositories has been defined; it performs either standard queries or full-text queries. The algebra is quite intuitive and is clearly inspired by the relational algebra; nevertheless, it is able to represent most of XQuery Full-Text expressions. We plan to carry on our work by investigating equivalence and containment properties our operators enjoy. This should be the first step towards an implementation of a working database system. To this aim, what is also needed is the definition of an implementative model, which should not necessarily be similar to the formal model presented here; moreover, an algorithm for the automatic translation of XQuery expressions into algebraic expressions should be developed.

Acknowledgements: The research was partially funded by PRIN Project ”Middleware basato su Java per la fornitura di servizi interattivi di TV digitale”.

References:


