Dottorato di Ricerca in Informatica

Università di Bologna, Padova

INF/01 INFORMATICA

A Model and an Algebra for Semi-Structured and Full-Text Queries

Giacomo Buratti

March 2007

Coordinatore:	Tutore:
Prof. Özalp Babaoğlu	Prof. Danilo Montesi

Abstract

The need for a convergence between semi-structured data management and Information Retrieval techniques is manifest to the scientific community. In order to fulfil this growing request, W3C has recently proposed XQuery Full Text, an IR-oriented extension of XQuery. However, the issue of query optimization requires the study of important properties like query equivalence and containment; to this aim, a formal representation of document and queries is needed. The goal of this thesis is to establish such formal background. We define a data model for XML documents and propose an algebra able to represent most of XQuery Full-Text expressions. We show how an XQuery Full-Text expression can be translated into an algebraic expression and how an algebraic expression can be optimized.

Contents

List of Tables

List of Figures

List of Algorithms

Part I

Preliminaries

Chapter 1

Introduction

In this chapter we present a synopsis of our doctoral work, which will serve as an introduction to the contents of the thesis. The research problem we have addressed is stated in Section 1.1. In Sections 1.2 and 1.3 we present our approach and main contributions. Then, in Section 1.4 we list the publications where we have disseminated some of the contents of the thesis and other related ideas. We conclude this introduction with the outline of the thesis.

1.1 Problem Statement

The semi-structured data paradigm [ASB99, Bun97, Suc98, Abi97] has gained growing attention in the last decade and XML [Con04] has become the *de facto* standard for exchanging information over the web and integrating heterogenous data sources. Several query languages for XML have been proposed [AQM+97, BDHS96, CCD+99, DFF+99, CRF00] until XPath [Con06a] and XQuery [Con06c] have received a general consensus, becoming the standard query languages.

The study of semi-structured data and XML received in the last years a further boost from a new trend: the integration of structured, semi-structured and unstructured data into a more general framework [INE]. In the past, these three kinds of data have been extensively studied as separated worlds, leading to incompatible models, languages and systems. A convergence between these diverging theories is made necessary by the con-

sideration that many today's applications, like biological data [BMBdlI05], have to cope with data covering the entire spectrum.

For what concerns the integration of structured data management and Information Retrieval techniques, some proposals, like BANKS [BHN+02] and DISCOVER [HP02], aims at enabling IR-like searches over relational databases. However, they are typically limited to simple keyword-based searches; no support for more complex queries (e.g. involving constraints on position of searched terms) is present.

For what concerns the integration of semi-structured data and Information Retrieval, XML plays a crucial role. In fact it permits to represent different kind of documents, ranging from *data-centric* documents (i.e. highly structured documents) to *document-centric* documents (i.e. loosely structured documents) [BYRN99]. However, a problem arose concerning the query language: 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. The easiest solution could be that of designing a system which accepts either XQuery expressions (managed by an XQuery engine) and IR-like searches (managed by an IR engine); however, such a splitting would made difficult expressing (and efficiently answering) queries that combines semi-structured and full-text queries. These considerations led to the definition of many query languages for XML with full-text capabilities [TG02, GSBS03, NDM+01, CMKS03, BG02, FG01]; lastly, W3C has published a Working Draft (mainly based on the previously proposed language TeXQuery [AYBS04]) for extending XQuery with Full-Text operators [Con06f].

While relational database systems and their language (SQL) were developed on a solid formal background (namely, relational model and relational algebra [Cod70]), in the semi-structured world efforts have been concentrated on practical problems, like defining suitable languages, leaving aside theoretical aspects. Only in the last few years important theoretical aspects, like the definition of a data model and an algebra for XML, have been tackled; these are central points for studying relevant properties of a query, like inclusion and equivalence, thus enabling the definition of rules for query optimization. Many different proposals [JLST01, FHP02] covering this issue have been presented; very few works [AKYJ03], however, deal with the further complexity introduced by the usage

of IR-like techniques in the semi-structured world. 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.

1.2 Our Proposal

In this thesis we propose AFTX (Algebra for Full-Text retrieval over XML repositories), a novel algebra for managing XML documents. It deeply integrates classical and full-text features, proposing itself as a valid framework for studying optimization techniques for XQuery Full-Text queries. The 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 the relational tuples and relations; AFTX is quite intuitive and is able to represent many XQuery FLWOR expressions, along with its full-text extensions.

The operators of our algebra enjoy some interesting algebraic properties, which are used to discover equivalence and containment between queries. This leads to the definition of rewriting rules for algebraic expression, whose purpose is to optimize query evaluation.

The definition of an algebra would be useless if such an algebra is not able to represent at least a significant fragment of the standard query languages for XML, which are XQuery and its IR extension XQuery Full-Text. Our algebra fulfills such a need.

1.3 Contributions

The contribution of our work is manifold:

- We present a new approach to the definition of a data model and an algebra for XML repositories; our approach is as close as possible to the classical relational theory,

with the necessary adaptations for dealing with the semi-structured paradigm. The data model and the algebra have a special emphasis on full-text retrieval capabilities, which are perfectly integrated with standard manipulation tasks.

- We develop an automatic translation algorithm from XQuery Full-Text expression to algebraic expressions. Numerous translation examples are presented, ranging from simple expressions composed by a single clause to quite complex expressions involving multiple variable bindings, nesting, content restructuring etc.

- We tackle the problem of efficient query evaluation by exploiting algebraic properties of our operators. This permits to study equivalence and containment of algebraic expressions and therefore to produce a set of rewriting rules aiming at transforming an expression into an optimized one.

1.4 Related Publications

The idea of defining a unified model for semi-structured and unstructured data, with particular focus on biological data, was included in a perspective article appeared in IEEE MMTC e-newsletter [BMBdlI05].

The AFTX algebra, which is the core of this thesis, has been presented in some published articles. A first version of the data model and the algebra, with support for standard XQuery-like queries has been presented in a paper accepted for the DEXA conference [BM06a]. Full-text support has been added in a paper accepted for a WSEAS conference [BM06c]. Query optimization issues have been tackled in a paper for the International Advanced Database Conference [BM06d] and an extended version has been published on a WSEAS journal [BM06b].

1.5 Outline of the Thesis

This thesis is organized in three main parts. The first part comprises this introduction and a review of related works in the area of semi-structured data. In particular we first

introduce the standard language for XML documents manipulation, XQuery, and its extensions towards IR tasks, XQuery Full-Text; then we review previously proposed data models and algebras, pointing out their strong points and weaknesses.

In the second part we present the core of our proposal, AFTX. Chapter 3 gives all the details about the data model we use to represent XML repositories and the operators used to manipulate them; we also relate our basic concepts to that used in XQuery. Chapter 4 shows how to translate an XQuery (Full-Text) expression into an AFTX expression; we first give informal hints for translation, then we present a formal algorithm for such a translation. The final goal of our algebra is to provide a method for efficiently evaluate queries; to this aim, in Chapter 5 we define interesting algebraic properties and demonstrate how they can be used in order to transform an algebraic expression into an optimized one.

Finally in the third part we draw out some conclusions and sketch future work.

Chapter 2

Related Works

In this chapter we do a survey of previous works that have significant connections with our thesis. We first analyze the standard query languages for XML; this study is preliminary to our work, because the algebra we propose to define must be able to express an expressive fragment of such languages. Then we review previous proposed algebras for XML, with and without full-text support. Their features are compared, and the critical points are highlighted.

2.1 Query Languages for Semi-Structured Data

Although many query languages for semi-structured data have been proposed during the last decade [AQM+97, BDHS96, CCD+99, DFF+99, CRF00], our work concentrates on the W3C's candidate standard XML query language XQuery [Con06c] and its full-text extension XQuery Full-Text [Con06f]. Consequently, in this section we only review these two languages, along with their corresponding data models.

2.1.1 XQuery

XQuery [Con06c] is the W3C's candidate standard XML query language; it is derived from a previous proposed language, Quilt [CRF00] and extends XPath 2.0 [Con06a].

The Data Model

The XQuery Data Model (XDM) [Con06d] is based on the concept of *sequence*. A sequence is an ordered list of zero or more *items*; an item can be:

- a *node*; a collection of nodes forms a tree, which consists of a root node plus all the nodes that are reachable directly or indirectly from the root node;
- an *atomic value*, i.e. a value of type atomic; an atomic type is a primitive simple type or a type derived by restriction from another atomic type.

Each node has a *unique identity*, while atomic values do not have identity. A *document order* is defined among all the nodes; document order is the order in which nodes appear in the XML serialization of a document.

The supported types are those defined in XML Schema [Con01] (`xs:string`, `xs:decimal`, `xs:datetime` etc.) plus five additional types: `xs:untyped` (an element node that has not been validated), `xs:untypedAtomic` (an untyped atomic value), `xs:anyAtomicType` (an atomic type that includes all atomic values), `xs:dayTimeDuration` (derived from `xs:duration` by restricting its lexical representation to contain only the days, hours, minutes and seconds components) and `xs:yearMonthDuration` (derived from `xs:duration` by restricting its lexical representation to contain only the year and month components).

There are seven kinds of nodes in the data model. The main kinds are *Document* (an entire XML document), *Element* (an XML element), *Attribute* (an XML attribute) and *Text* (XML character content); the other kinds are *Namespace* (the binding of a namespace URI to a namespace prefix), *Processing Instruction* (XML processing instructions) and *Comment* (XML comments).

A set of properties (called *accessors*) is defined on each node n; among them the most significant are:

- `dm:children`: the ordered list of child nodes of n;
- `dm:attributes`: the attributes of n; order of attributes of a node is implementation dependent;

- `dm:node-name`: the name of n;
- `dm:parent`: the parent of n;
- `dm:string-value`: the concatenation, in document order, of the string values of all text nodes descendants of n; for *attribute* and *text* nodes, it corresponds to the value of the node, because such nodes can not have descendants;
- `dm:typed-value`: the typed value of n;
- `dm:type-name`: the schema type of n.

Given those kinds of nodes and their accessors, a *document* is defined as a tree whose root node is a Document Node; a tree whose root node is not a Document Node, i.e. a subtree, is instead referred to as a *fragment*.

Example 2.1 Consider the XML document in Figure 2.1, taken from [Con06d]. Figure 2.2 shows how the document is represented using XDM; for the sake of simplicity only Document Nodes (Dx), Element Nodes (Ex), Attribute Nodes (Ax) and Text Nodes (Tx) are included. The Document Node D1, which represents the entire document, has one child Element Node (E1), corresponding to the XML element `catalog`. E1 has three child Attribute Nodes (corresponding to the XML attributes `xsi:schemaLocation`, `xml:lang` and `version`) and two child Element Nodes (corresponding to the XML elements `tshirt` and `album`). Note that the textual content of an Element Node is represented by a child Text Node (like T1, which corresponds to the content of the XML element `title`: "*Staind: Been Awhile Tee Black (1-sided)*"), while value of Attribute Nodes is not. The value of the `dm:string-value` property for the Element Node D1 is the concatenation of the string values of all its descendant Text Nodes: "*Staind: Been Awhile Tee Black (1-sided) Lyrics from the hit song 'It's Been Awhile' are shown in white, beneath the large 'Flock & Weld' Staind logo. 25.00 It's Been A While 10.99 Staind*".

The Language

The basic building block of XQuery is the *expression*; the result of an expression is affected by its *static context* (information about namespaces and schemas, defined variables

```
<?xml version="1.0"?>
<catalog xmlns="http://www.example.com/catalog"
         xmlns:html="http://www.w3.org/1999/xhtml"
         xmlns:xlink="http://www.w3.org/1999/xlink"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.example.com/catalog
                             dm-example.xsd"
         xml:lang="en" version="0.1">
  <tshirt code="T1534017" label=" Staind : Been Awhile "
          xlink:href="http://example.com/0,,1655091,00.html"
          sizes="M L XL">
    <title> Staind: Been Awhile Tee Black (1-sided) </title>
    <description>
      <html:p>
        Lyrics from the hit song 'It's Been Awhile'
        are shown in white, beneath the large
        'Flock &amp;amp; Weld' Staind logo.
      </html:p>
    </description>
    <price> 25.00 </price>
  </tshirt>
  <album code="A1481344" label=" Staind : Its Been A While "
         formats="CD">
    <title> It's Been A While </title>
    <description xsi:nil="true" />
    <price currency="USD"> 10.99 </price>
    <artist> Staind </artist>
  </album>
</catalog>
```

Figure 2.1: An XML document.

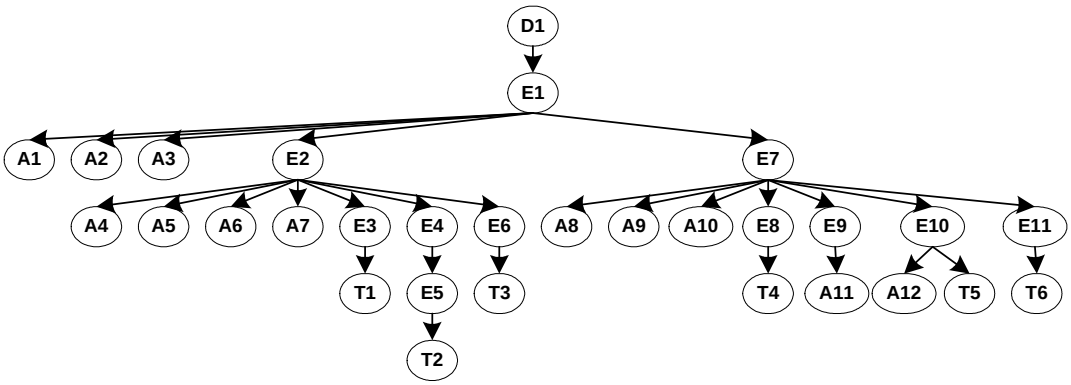


Figure 2.2: Graphical representation of the XML document in Figure 2.1 using XDM.

and functions, etc.) and by its *dynamic context*: the context item (the item currently being processed), the context position (the position of the context item within the sequence of items currently being processed), the context size (number of items in the sequence processed), the variable values, etc.

Two phases of processing are defined: the *static analysis phase* and the *dynamic evaluation phase*. During the static analysis phase, the query is parsed into an internal representation called the *operation tree*, which is then normalized; static type checking is performed. During the dynamic evaluation phase the value of the expression is computed; it depends on the operation tree, on the input data and on the dynamic context.

An expression is composed by one or more *single expression*s connected by the *comma* operator, which is used to form a sequence. Typically a single expression is a *FLWOR* expression; the name FLWOR is an acronym for the keywords `for`, `let`, `where`, `order by` and `return`. The `for` and `let` clauses in a FLWOR expression generate an ordered sequence of tuples of bound variables, called the *tuple stream*. The optional `where` clause serves to filter the tuple stream, retaining some tuples and discarding others. The optional `order by` clause reorders the tuple stream. The `return` clause constructs the result of the FLWOR expression; it is evaluated once for every tuple in the tuple stream, after filtering by the `where` clause, using the variable bindings in the respective tuples. The result of the FLWOR expression is an ordered sequence containing the results of these evaluations, concatenated as if by the comma operator.

The simplest example of a `for` clause contains one variable and an associated expression. For example the clause

```
for $d in fn:doc("depts.xml")/depts/deptno
```

iterates over all the departments in an input document, binding the variable `$d` to each department number in turn. The function `fn:doc` reads the XML document `depts.xml` and returns an XDM instance representing that document; then the path expression `/depts/deptno` looks for a child element named `deptno` of a child element named `depts` of the current item; initially the current item is the document node, that represents the entire document.

A `for` clause may also define multiple variables. In this case, the `for` clause iterates each variable over its binding sequence; the resulting tuple stream contains one tuple for each combination of values in the respective binding sequences. For example the clause

```
for $d in fn:doc("depts.xml")/depts/deptno,
    $e in fn:doc("emps.xml")/employees/employee
```

returns a tuple stream containing one tuple for each (*department number*, *employee*) pair.

Each variable bound in a `for` clause may have an associated *positional variable* that is bound at the same time. The name of the positional variable is preceded by the keyword `at`. As a variable iterates over the items in its binding sequence, its positional variable iterates over the integers that represent the ordinal positions of those items in the binding sequence, starting with 1. For example the clause

```
for $pet at $i in ("Cat", "Dog")
```

returns two tuples: *(“Cat”, 1)* and *(“Dog”, 2)*.

A `let` clause, like a `for` clause, binds one or more variables to a sequence; however, a `let` clause binds each variable to the entire result of its associated expression, without iteration. The variable bindings generated by `let` clauses are added to the binding tuples generated by the `for` clauses. For example the expression

```
for $d in fn:doc("depts.xml")/depts/deptno
let $e:=fn:doc("emps.xml")/employees/employee
```

returns a tuple stream containing one tuple for each department number; that tuple will contain the department number (bound to `$d`) and a sequence containing all the employees (bound to the variable `$e`).

A `where` clause serves as a filter for the tuples of variable bindings generated by the `for` and `let` clauses. The expression in the `where` clause is evaluated once for each tuple; if its boolean value is `true`, the tuple is retained and its variable bindings are used in an execution of the `return` clause; if the boolean value is `false`, the tuple is discarded. For example the expression

```
for $pet at $i in ("Cat", "Dog")
where $i mod 2 = 0
```

returns the tuple *("Dog", 2)*; the tuple *("Cat", 1)* is discarded because $1 \bmod 2 \neq 0$.

An `order by` clause contains one or more ordering specifications. For each tuple in the tuple stream, after filtering by the `where` clause, the ordering specifications are evaluated, using the variable bindings in that tuple. The relative order of two tuples is determined by comparing the values of their ordering specifications, working from left to right until a pair of unequal values is encountered. For example the expression

```
for $e in $employees
order by $e/salary descending
```

returns employees in descending order by salary.

The `return` clause of a FLWOR expression is evaluated once for each tuple in the tuple stream, and the results of these evaluations are concatenated, as if by the comma operator, to form the result of the FLWOR expression. A `return` clause typically use *constructors*, that create XML structures possibly referring to variables using enclosed expressions. For example, suppose to bind a `$b` variable to `book` elements having one or more `author` sub-elements; the clause

```
return  <book isbn="{$b/isbn}">
            <authors>{$b/author}</authors>
        </book>
```

for each input `book` element does the following:

- create a `book` element;
- create a `isbn` attribute;
- set the value of the `isbn` attribute to the value of the `isbn` child element of the input `book` element;
- create an `authors` element, whose parent is the `book` element;
- create a subtree of the `authors` element for each subtree rooted at `author` of the input `book` element.

In the previous examples we have seen that each clause use *path expression*s. A path expression consists of a series of one or more *steps*, separated by "`/`" or "`//`", and optionally beginning with "`/`" or "`//`". Each step generates a sequence of items and then filters the sequence by zero or more predicates. A predicate can test the value of an element (e.g. `/book[./price < 50]`: find all books with a price less than 50), the value of an attribute (e.g. `/book[@id = 1]`: find all books with an attribute `id` having value 1), the existence of an element (e.g. `/book[./author]`: find all books with at least one author), the existence of an attribute (e.g. `/book[@isbn]`: find all books with an attribute `isbn`), the context position (e.g. `/book/author[2]`: find the second author of each book).

Note that, in the expression `/book[./price < 50]`, an *atomization* operation is first performed, i.e. the typed value of the element `price` is extracted; then a comparison between such typed value and 50 is executed. XQuery provides three kinds of comparison expressions, called *value comparisons*, *general comparisons*, and *node comparisons*. The difference between value comparisons (`eq`, `ne`, `lt`, `le`, `gt`, and `ge`) and general

comparisons (=, !=, <, <=, >, and >=) is that the first ones operate only on single values, while the second ones operate also on sequences of values following an existential semantic; for example $(1, 2) = (2, 3)$, because there is a pair of values from the first and second sequence that are equal. Node comparisons are used to compare two nodes, by their identity (`is`) or by their document order (<< and >>).

XQuery permits the usage of *functions*, that must return an instance of XDM. The built-in functions supported by XQuery are defined in [Con06g]; additional functions may be declared in the prolog of the query, imported from a library module, or provided by the external environment as part of the static context. Among the built-in function, we cite:

- `distinct-values`: applied to a sequence of nodes, returns a sequence of atomic values containing only the distinct values found in the input sequence;
- `count`: applied to a sequence, returns the number of items in the sequence;
- `position`: returns the context position of the context node.

XQuery supports a conditional expression based on the keywords `if`, `then`, and `else`. For example the expression

```
if ($widget1/unit-cost < $widget2/unit-cost)
  then $widget1
  else $widget2
```

returns the sequence bound to either `$widget1` or `$widget2`, depending on the satisfaction of the test condition.

XQuery also supports universal and existential quantifiers. A quantified expression begins with a quantifier, which is either the keyword `some` or the keyword `every`, followed by one or more *in-clauses* that are used to bind variables, followed by the keyword `satisfies` and a test expression. Each in-clause associates a variable with an expression that returns a sequence of items; it generates tuples of variable bindings, including a tuple for each item that satisfies the test expression. For example the expression

```
some $emp in /emps/employee satisfies
     ($emp/bonus > 0.25 * $emp/salary)
```

returns true if at least one employee satisfies the given comparison expression; moreover it binds the variable `$emp` to the employees satisfying the condition.

XQuery allows expressions to be nested with full generality. For example, consider the XML document *book.xml*, shown in Fig. 2.3, which will be further used in the following chapters. The following query inverts the document hierarchy to transform a bibliography into an author list in which each author's name appears only once, followed by a list of titles of books written by that author:

```
<authlist>
 {
   for $a in fn:distinct-values($bib/book/author)
   order by $a
   return
     <author>
        <name> {$a} </name>
        <books>
          {
            for $b in $bib/book[author = $a]
            order by $b/title
            return $b/title
          }
        </books>
     </author>
 }
</authlist>
```

```
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="1992">
    <title>Advanced Programming in the Unix environment</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="2000">
    <title>Data on the Web</title>
    <author><last>Abiteboul</last><first>Serge</first></author>
    <author><last>Buneman</last><first>Peter</first></author>
    <author><last>Suciu</last><first>Dan</first></author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price>39.95</price>
  </book>
  <book year="1999">
    <title>Technology and Content for Digital TV</title>
    <editor>
      <last>Gerbarg</last><first>Darcy</first>
      <affiliation>CITI</affiliation>
    </editor>
    <publisher>Kluwer Academic Publishers</publisher>
    <price>129.95</price>
  </book>
</bib>
```

Figure 2.3: A working example of XML document.

2.1.2 XQuery Full-Text

The Data Model

The XQuery Data Model, based on the notion of sequence of nodes, is inadequate to support full-text searches over XML documents, because full-text search requires more information on the words contained in the document. In particular, at least the relative position of the words is needed; moreover, it could be worth representing which paragraph or sentence the word is contained in. XQuery Full-Text adds such information via a structure called *AllMatches*.

An *AllMatches* describes the possible results of a full-text selection; it contains zero or more *Match*es, each of which describes one result of the full-text selection. Each *Match* contains zero or more *StringInclude* and zero or more *StringExclude*, which describe the query: a *StringInclude* represents a searched token (i.e. a token that must be found in the document), a *StringExclude* represents an unwanted token (i.e. a token that must not be contained in the document). Finally, each *StringInclude*/*StringExclude* (known collectively as *StringMatch*) has an associated *TokenInfo*, that represents the word that matches the condition specified in the *StringMatch*. A *TokenInfo* is formed by a word, a unique identifier (*pos*) that captures the relative position of the word in document order and two more unique identifiers that represent the containing paragraph and sentence; these information are available thanks to a *tokenization* process that must be carried out before a full-text search can be evaluated.

Figure 2.4 shows an example of *AllMatches* (taken from [Con06f]) relative to the full-text selection "*Ford Mustang*". There are two possible results, represented by the two *Match*es. The first *Match* shows that the word "*Ford*" has been found at position 1 and the word *"Mustang"* has been found at position 2; the second *Match* shows that the words have been found at position 27 and 28.

Given their hierarchical nature, *AllMatches* structures can be represented as XML documents; therefore it is possible to define formal XQuery functions that represent the implementation of a full-text search condition. In this way full-text conditions can be composed with standard XQuery search conditions.

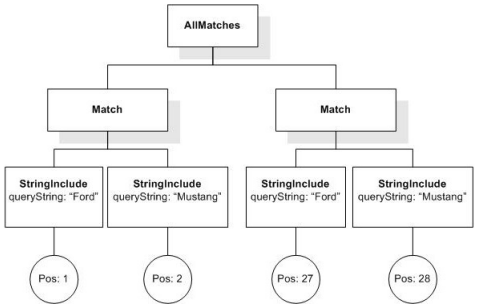


Figure 2.4: An example of *AllMatches*.

The Language

XQuery Full-Text extends XQuery and XPath by:

- adding a new expression called *FTContainsExpr*;
- enhancing the syntax of FLWOR expressions in XQuery and path expressions in XPath with optional score variables.

Wherever an XQuery comparison expression can be used, a *FTContainsExpr* can be used. A simple *FTContainsExpr* is of the form `ftcontains` *FTSelection*, where *FTSelection* represents the full-text condition. For example the XQuery Full-Text expression

```
for $b in doc("bib.xml")/books/book
where $b ftcontains "dog"
return $b/author
```

return all authors of books that contain somewhere the word *dog*. Note that the word is searched into the entire content of a *book* element, including the value of its sub-elements. The same full-text condition could also be included into a path expression; for example the following XQuery Full-Text expression is equivalent to the previous one:

```
for $b in doc("bib.xml")/books/book[. ftcontains "dog"]
return $b/author
```

The full-text condition can also be composed by multiple basic conditions, connected with the boolean operators && (and) or || (or). For example the XQuery Full-Text expression

```
for $b in doc("bib.xml")/books/book
where $b ftcontains "dog" && "cat"
return $b/author
```

return all authors of books that contain somewhere the word *dog* and *cat*.

Besides specifying a match of a full-text search as a boolean condition, full-text search applications typically also have the ability to associate *scores* with the results; scores express the relevance of those results to the full-text search conditions. XQuery Full-Text extends XQuery and XPath further by adding optional `score` variables to the `for` and `let` clauses of FLWOR expressions. For example consider the XQuery Full-Text expression:

```
for $b score $s in doc("bib.xml")/books/book
    [. ftcontains "dog" && "cat"]
return  <book>
            <title>{$b/title}</title>
            <score>{$s}</score>
        </book>
```

The evaluation of the expression following the `in` keyword not only determines the resulting sequence of the expression, i.e., the sequence of items which are iteratively bound to the `for` variable. It must also determine in each iteration the relevance score value of the current item and bind the `$s` variable to that value. The result is therefore a list of books containing somewhere the two searched words; for each book, the title and the score value is output.

The calculation of relevance is implementation-dependent, but score evaluation must follow these rules:

- score values are of type `xs:double` in the range [0, 1];

- for score values greater than 0, a higher score must imply a higher degree of relevance.

Similarly to the way they are used in a `for` clause, score variables may be specified in a `let` clause. A score variable in a `let` clause is also bound to the score of the expression evaluation, but in the `let` clause one score is determined for the complete result. The `let` variable may be dropped from the `let` clause, if the score variable is present. While when using the score option in a `for` clause the expression following the `in` keyword has the dual purpose of filtering, i.e., driving the iteration, and determining the scores, it is possible to separately specify expressions for filtering and scoring by combining a simple `for` clause with a `let` clause that uses scoring. For example consider the following XQuery Full-Text expression:

```
for $b in doc("bib.xml")/books/book
let score $s := $b ftcontains "dog" && "cat"
order by $s descending
return  <book>
            <title>{$b/title}</title>
            <score>{$s}</score>
        </book>
```

This query returns all the books, without any filter. However, a score is calculated for each book, and books are returned in descending order by score value.

Scoring may be influenced by adding *weight declarations* to search tokens. For example the `let` clause

```
let score $s := $b ftcontains ("dog" weight 0.2)
                && ("cat" weight 0.8)
```

instructs the system to give a higher importance to the word *cat* and a lower importance to the word *dog*; however the exact effect of weights on the result score is implementation-dependent.

Until now we have shown only basic full-text searches of single words. More complex full-text conditions can be written in XQuery Full-Text. Among them we cite the following possibilities:

- we can search for elements containing a phrase instead that a set of words (e.g. `/book[. ftcontains "Expert Reviews"]`);
- we can search for elements *not* containing a word or phrase (e.g. `/book[. ftcontains ! "usability"]`);
- we can state that searched words must be found in the same order as in the query (e.g. `/book/title ftcontains ("web site" && "usability") ordered`: find those titles that contain the phrase *web site* and, later, the word *usability*);
- we can state that searched words must be found in the same sentence or paragraph (e.g. `/book ftcontains "usability" && "Marigold" same sentence`);
- we can state that searched words must be found at a certain maximal distance (e.g.`/book ftcontains "web" && "site" distance at most 2 words`);
- we can state that searched words must appear at least n times (e.g. `/book[. ftcontains "usability" occurs at least 2 times]`);
- we can specify a set of match options that affect the result of a query: case-sensitive search, use of stemming, use of thesaurus, use of stopword etc. (e.g. `/book [. ftcontains "usability" with stemming with thesaurus default]`).

2.2 Algebras for XML

Many different algebras for semi-structured data have been proposed in the last few years. In this section we deeply analyze the two algebras that mainly influenced our work, TAX [JLST01] and XAL [FHP02]. Then we describe in fewer details further interesting proposals.

2.2.1 TAX

TAX [JLST01] (Tree Algebra for XML) is probably the most famous algebra for XML documents. In TAX data model an XML document is represented by an ordered labeled tree, which is the basic unit of information. Each node in a tree represents an XML element; a node can have a list of attributes (corresponding to XML attributes) plus the following special attributes:

- `tag`: the type of the element, i.e. its name;
- `content`: the value of the element;
- `pedigree`: it represents a sort of *history* of where a node came from; it has a different value for each element stored in an XML repository, but it is not a unique identifier; in fact, if a node is copied then both the copy and the original have the same pedigree, and when a new node is created it has a null pedigree.

Trees are grouped into collections; each TAX operator takes one or more collections as input and produces a collection as output.

The main innovation in TAX is the concept of *pattern trees*, which are essentially trees representing nodes and attributes of interest for an operator, plus a selection formula. Pattern tree nodes have a distinct integer as label; nodes are connected with `pc` (parent-child) or `ad` (ancestor-descendant) edges. The selection formula is a boolean combination of predicates applicable to nodes.

Two examples of pattern trees are shown in Figure 2.5. Pattern (a) asks for books published before 1988 and having at least one author; in fact it looks for an element

whose tag is `book`, having a child element named `year` with value less than 1988, and a descendant element named `author`. Pattern (b) asks for books publisher by a publisher whose name contains the string "Science" and written by Jack and Jill in that order.

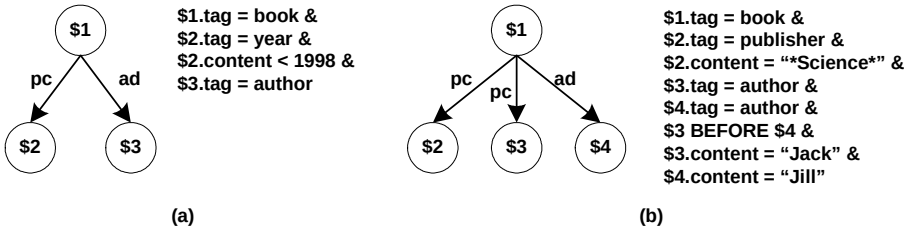


Figure 2.5: Two TAX pattern trees.

Given a collection of trees and pattern tree, a *witness* tree represents a possible map from the pattern tree to an input tree. It contains the nodes, corresponding to the nodes in the pattern tree, that satisfy the selection formula. Multiple witness tree can be obtained from a single input tree, if the pattern tree can be mapped in multiple ways. For example, consider the pattern tree in Figure 2.5(a); by applying it to the tree shown in Figure 2.6(a) two different witness trees, shown in Figure 2.6(b), are obtained, because two mappings exist from the pattern tree to the input tree.

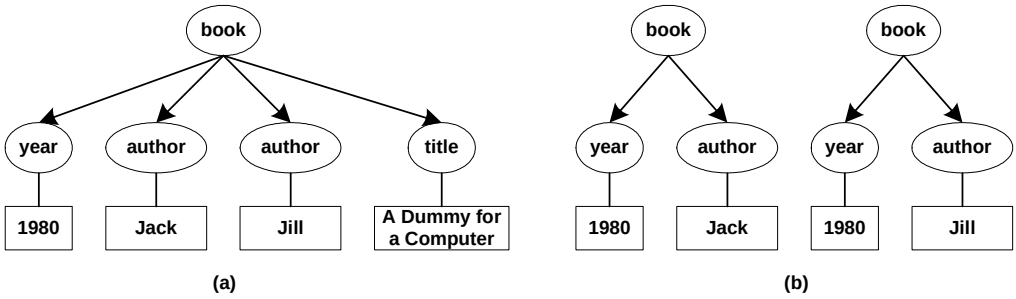


Figure 2.6: An input tree (a) and the resulting witness trees (b) obtained applying the pattern tree of Figure 2.5(a).

The notion of pattern tree is the basis for the definition of TAX operators, which are the following:

- *Selection* $\sigma_{\mathcal{P},\mathtt{SL}}(\mathcal{C})$: returns all possible witness trees corresponding to the pattern tree $\mathcal{P}$, enriched with all descendants of those nodes corresponding to pattern tree nodes included in the *adornment list* `SL`; for example, $\sigma_{\mathcal{P},\$1}(\mathcal{C})$, where $\mathcal{P}$ is the pattern tree in Figure 2.5(a) and $\mathcal{C}$ is a collection containing only the tree in Figure 2.6(a), returns two copies of the input tree, because the adornment list specifies to retain in the output the entire subtree rooted at `book`.

- *Projection* $\pi_{\mathcal{P},\mathtt{PL}}(\mathcal{C})$: for each witness tree, returns only those nodes which are included in the *projection list* `PL`, plus all descendants of those nodes corresponding to pattern tree nodes included in `PL` with a "*"; for example, using the same $\mathcal{P}$ and $\mathcal{C}$ of the selection example, $\pi_{\mathcal{P},\$1*}(\mathcal{C})$ returns the same result of $\sigma_{\mathcal{P},\$1}(\mathcal{C})$.

- *Product* $\mathcal{C} \times \mathcal{D}$: for each pair of trees $T_i \in \mathcal{C}$ and $T_j \in \mathcal{D}$ returns a new tree having a root node named `tax_prod_root`, whose left and right subtrees are a copy of T_i and T_j; join and outerjoin operators are derived by applying a selection condition to the result of a product.

- *Set Operators*: union, intersection, and difference act as in classical set theory; two trees are considered identical if all atttributes at corresponding nodes are identical.

- *Grouping* $\gamma_{\mathcal{P},\mathtt{GL},orfun}(\mathcal{C})$: groups witness trees, obtained by applying the pattern tree $\mathcal{P}$ to the trees in $\mathcal{C}$, by the value of elements/attributes in the *grouping list* `GL`; for each group, an output tree is built, having a root element named `tax_group_root` with two subtrees: 1) a `tax_grouping_list` element containing the nodes that form the grouping basis; 2) a `tax_group_subroot` element having as subtrees the witness trees in the group, ordered by the function *orfun*. Derived operators for duplicate elimination and ordering can be obtained using grouping and projection.

- *Aggregation* $\mathbf{A}_{\mathtt{aggAttr}=f_1(\$j.attr),pos}(\mathcal{C})$: each input tree is returned unchanged, except for the insertion of a new `tax_aggNode` node, having an attribute `aggAttr` whose value corresponds to the result of the aggregation function f_1 (min, max, count, sum etc.); the new node is inserted in the position specified by *pos*.

- *Renaming* $\rho_{\mathcal{P},\texttt{RS}}(\mathcal{C})$: each witness tree is returned with some nodes or attributes being renamed, according to the *renaming specification* RS; for example $\rho_{\mathcal{P},\$2\leftarrow\texttt{published}}(\mathcal{C})$, where $\mathcal{P}$ and $\mathcal{C}$ are the usual pattern tree and input tree, returns the same tree shown in Figure 2.6(a), except that the element `year` is renamed `published`.

- *Reordering* $\varrho_{\mathcal{P},f,\texttt{RL}}(\mathcal{C})$: for each input tree, the subtrees rooted at the element specified in the *reorder list* RL are reordered on the basis of the result of the function f applied to those subtrees.

- *Copy-and-Paste* $\kappa_{\mathcal{P},\texttt{CL},pos}(\mathcal{C})$: for each input tree, the nodes specified in the *copy list* CL (or the subtrees rooted at those nodes, if the node name includes a "*") are copied in the position specified by *pos*.

- *Value Updates* $\upsilon_{\mathcal{P},\texttt{US}}(\mathcal{C})$: each input tree is returned unchanged, except that some attribute values are changed, according to the *update specification* US; for example $\upsilon_{\mathcal{P},\$2:\texttt{content}\leftarrow\$2.\texttt{content}+1}(\mathcal{C})$, where $\mathcal{P}$ and $\mathcal{C}$ are the usual pattern tree and input tree, returns the same tree shown in Figure 2.6(a), except that the value of the element `year` is raised by 1.

- *Node Deletion* $\delta_{\mathcal{P},\texttt{DS}}(\mathcal{C})$: each input tree is returned unchanged, except that the nodes included in the *delete specification* DS (or the entire subtree rooted at those nodes) are deleted.

- *Node Insertion* $\iota_{\mathcal{P},\texttt{IS}}(\mathcal{C})$: each input tree is returned unchanged, except that a list of new nodes are inserted according to the *insert specification* IS; for example $\delta_{\mathcal{P},<\mathit{AfterLastChild}(\$1)>(\texttt{tag="publisher",content="Morgan Kaufman"})}(\mathcal{C})$ creates a new `publisher` node, having the value `Morgan Kaufman`, and insert as last child of the `book` node.

The authors claim that TAX is able to express any XQuery expression not involving recursion, function calls or tag variables, and such that the variables bound in the `let` clause are bound to an aggregate expression; some XQuery facilities, like quantifiers, are

not expressible in TAX, but the XQuery expression can be rewritten into an equivalent one not involving quantifiers. An informal procedure for the translation from XQuery to TAX is presented.

2.2.2 XAL

XAL [FHP02] (XML ALgebra) represents an XML document as a rooted connected directed graph with a partial order relation defined on its edges. Vertices are of type *element* (i.e. containing sub-elements) or *simple* (int, string, etc.); in the first case, the *value* property is the vertex identifier, in the second case it is the element content. Element containment edges model hierarchy between elements, and their name correspond to the child element name; attribute edges connect an element to its attributes; data edges connect an element to text data included in it. The order relation is defined only on element containment and data edges.

Three kinds of operators are defined: *extraction operators*, *meta-operators* and *construction operators*. Extraction operators retrieve information from the input XML documents and return a collection of vertices; they are:

- *Projection* $\pi[t, n](e)$: returns the collection of vertices that represent the targets of edges of type t and name n originating from vertices in e; for example $\pi[E, painter](e)$ returns all target nodes of element containment edges, originating from e, named *painter*.

- *Selection* $\sigma[cond](e)$: returns the collection of vertices that fulfill the condition *cond*, in which constants and projection operators can be used; for example $\sigma[\pi[A, name] = \text{“}Dali\text{”}](e)$ returns all vertices that have an attribute called *name* with the value “*Dali*”.

- *Distinct* $\delta(e)$: removes duplicates from a collection.

- *Sort* $\Sigma[expr](e)$: sorts a collection based on the value of expression *expr*; for example $\Sigma[\pi[A, name]](e)$ orders the input vertices by the value of their *name* attribute.

- *Join* $(x : expr1) \bowtie [cond](y : expr2)$ and *Product* $(x : expr1) \times (y : expr2)$: for each pair of vertices (x, y), where x and y are obtained by, respectively, *expr1* and *expr2*, if the pair fulfills the selection condition *cond* (or if the condition is not present, which is the case of product) then a new vertex is created; such a vertex has, as outgoing edges, first the outgoing edges of x, then the outgoing edges of y. For example $(x : \pi[E, person(people)) \bowtie [\pi[A, id](x) = \pi[A, name](y)](y : \pi[E, painter](painters))$ pairs person and painter vertices based on the equality of the *id* attribute of a *person* and the *name* attribute of a *painter*.

- *Union* $x \cup y$, *Difference* $x - y$, and *Intersection* $x \cap y$: these set operators have the classical semantics; they preserve ordering, thus union is not commutative.

Meta-operators apply a function to each element of the input collection. They are:

- *Map* $map[f](e)$: applies the function f to each element in e and concatenates the results in the output collection.

- *Kleene Star* $*[f, n](e)$: repeats the function f n times starting with the input e; at each iteration the results of the function are added to the next function input; if n is not present, the repetition continues (possibly infinite times) until a fix point is reached. For example, suppose that an XML document contains *painter* elements having *painter* sub-elements, these sub-elements having further *painter* sub-elements etc; then $\pi[A, name](*[\pi[E, painter]](root))$ gives the names of all painters.

Construction operators rearrange data previously extracted. They are:

- *Create Vertex* $vertex[t](v)$: creates a new vertex of type t and value v; for example $vertex[string](\text{“}Dali\text{”})$ creates a simple string element with value *Dali*, while $vertex[element](null)$ creates a complex element with null value.

- *Create Edge* $edge[t, n, p](c)$: adds to the graph an edge, named n, of type t from p to c; for example $edge[E, painter, vertex[element](null)](vertex[string](\text{“}Dali\text{”}))$ creates an element containment edge with name *painter* between the vertices created in the previous example.

A set of optimization laws for XAL expressions is presented. Some of them are based on similar relational algebraic optimization rules: selection decomposition, selection commutativity, projection and selection push-down, etc. Some useful optimization rules, like product commutativity, are not directly applicable, because their usage would change order between elements; however, there are cases when such order is not important, for example because a subsequent re-ordering must be applied, and therefore such query rewritings can be executed.

2.2.3 TAX and XAL: Features Comparison and Critical Points

In our opinion, TAX and XAL should be considered two very interesting proposals. TAX operators have a clear semantics, that is well suited to represent typical operations on semi-structured data, like those available using XQuery. On the other side XAL operators are defined in a way more similar to classical relational operators, which results in an easier definition of optimization rules. For what concerns the features exposed by the operators, XAL has the advantage of enabling recursion through the Kleene Star operator; on the other side, TAX is equipped with grouping and node deletion/update, which are not present in XAL.

Though many valuable ideas can be found in these two proposals, we believe they also have some important drawbacks.

For what concerns TAX, the concept of pattern trees (and the related concepts of *embedding* and *witness tree*), besides being probably its main innovation, is in our opinion not so intuitive; it represents a strong deviation from classical relational algebra, thus making it difficult to (partially) reuse the well-known equivalence rules for optimization purposes. Not surprisingly, the problem of query equivalence and containment is just mentioned and no formal rule is present.

Another critical point can be found in the definition of selection and projection operator. In relational algebra these two operators have a clear orthogonal semantics; in TAX the distinction is much less sharp. In fact, some results can be obtained by applying indifferently one of these two operators.

Finally, as already said, authors claim that almost any XQuery expression can be translated into a TAX expression. However, the presented translation algorithm is quite informal and not detailed, thus making it difficult to ratify such a claim; as an example, it is not clear if queries having more than two levels of nesting can be translated. Moreover, some limitations to the kind of XQuery expression that can be translated (like the fact that variables bound in a `let` clause must be bound to aggregate expressions) are rather severe.

For what concerns XAL, the main drawback is probably the fact that only extraction operators are closed; in fact meta-operators result depends on the function they apply to the input collection (for example, a list of simple values could be returned), while construction operators, according to the definition, do not even take as input any collection.

Another serious problem of XAL is that sometimes operators are not clearly defined. For example, it is not clear if projection retains only the vertices having a certain incoming edge, or if it retains also their sub-elements. Moreover, it looks like projection searches such vertices in the entire input collection, thus $\pi[E, painter](e)$ should be equivalent to the path expression `//painter`; what if we want to find only root *painter* elements? A further example of non-rigorous definition is that of the *Distinct* operator: which notion of equality does it use?

Finally, authors do not specify which part of XQuery can be expressed in XAL; at a first sight, it seems however that only very simple queries can be translated. No translation algorithm is present.

2.2.4 Other Algebras

The algebra presented in [SA02] uses a *path* operator to extract information from an XML database on the basis of a path expression, to build variable bindings and to store them in a relational-like structure; basic operators (selection, join etc.) then manipulate these relational structures and finally the *return* operator produces the resulting XML document. The use of such a relational structure enables to use classical relational optimization rules; however, a sorting operation is always needed before building the result, because the ordering of elements gets lost in the creation of the relational structure. Moreover, selection

operations, that could be interleaved with path expression evaluation, must be postponed to the end of the *path* operation; this need prohibits the use of some classical optimization rules, like selection push-down. This algebra shares many features with that presented in [CCS00], which however does not deal with the ordering of elements.

SAL [BT99] is a general algebra for semi-structured data that works on edge-labeled directed graphs; it is not specifically designed for handling XML features (e.g. it does not support attributes) and does not provide powerful restructuring operators.

XAT [ZPR02] is the algebra used in the Rainbow [DSR] XML data management system, which is based on XML views over relational data. Consequently, XAT optimization rules concentrate on moving as much computation as possible to the underlying relational engine, making it difficult to apply them to a more general framework.

The algebras we reviewed up to now fall into two camps; some of them are tuple-based algebras, while others are tree-based algebras. The algebra proposed in [RSF06] and used in the Galax XQuery engine [gal] borrows ideas from both camps. It is based on a data model in which values can be either an *XML value* (i.e. an ordered sequence of items) or a *table* (i.e. an ordered sequence of tuples containing XML values). Algebraic operators fall into three categories:

- *XML operators*, i.e. operators working on XML values; they can be further subdivided into:
  - *constructor operators*: they create sequences, elements, atomic values etc.;
  - *navigation operators*: they follow a path, possibly applying a node test;
  - *type operators*: they perform casting, validation, and type matching;
  - *functional operators*: they model function calls and conditional expressions;
  - *I/O operators*: they parse or serialize documents.
- *Tuple operators*, i.e. operators working on tuples; they can be further subdivided into:
  - *constructor operators*: they create or concatenate tuples;

  - *relational operators*: they perform typical relational operations (selection, join, etc.);
  - *map operators*: they perform functional map on tuples, i.e. apply some function on tuples;
  - *grouping and sorting*: they group or sort tuples.

- *XML / tuple operators*: these operators sit at the boundary between the tuple part or the algebra and XML part; they are used to transform tuples into items (and vice versa) and to express existential and universal quantifiers.

The main interesting feature of this algebra is the full coverage of XQuery Core expressions; authors proposes a set of compilation rules for transforming an XQuery expression into an algebraic expression. Moreover, some rewriting techniques are used to optimize the evaluation of a query, with particular emphasis on query un-nesting.

A different approach is followed in [MM06], which proposes to define logical database models by instantiating a general abstract model. The abstract model is equipped with a parametric algebra, which defines, in addition to standard operators like selection, projection etc., two distinctive operators: embedding, which extends objects with novel data, and splitting, which decomposes a single object into many objects. Algebraic operators work on collection of objects. The authors propose an instantiation of the abstract model in order to manage XML documents and show how an XQuery FLWOR expression can be translated in their algebra; however the model has some serious limitations: it can not manage path expressions containing selection conditions and it does not represent order between XML documents.

2.3 Full-Text Algebras

While there are a lot of proposals for algebras able to represent XQuery-like queries, to our knowledge the only algebra which integrates structured search with full-text capabilities is TIX [AKYJ03] (Text In XML). As the name suggests, TIX is an extension of TAX;

its data model is based on the concept of *scored tree*, which is a TAX tree extended with a *score* node attribute; the score of the root node represents the score of the tree.

A *scored pattern tree* is defined as a TAX pattern tree with the following extensions:

- besides `pc` (parent-child) and `ad` (ancestor-descendant) edges, a new `ad*` (self-or-descendant) relationship between pattern tree nodes can exist;
- a set of formulas specifies how to calculate the score of some nodes involved in IR-style search.

For example, consider the following query, expressed in natural language: *"Find document components that are part of an article written by an author with last name 'Doe'and are about 'search engine'. Relevance to 'internet'and 'information retrieval'is desirable but not necessary.*"; the corresponding scored pattern tree is shown in Figure 2.7.

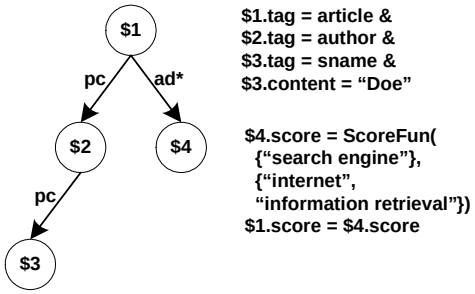


Figure 2.7: A TIX scored pattern tree.

Structural constraints (the presence of an `article` element having an `author` sub-element containing a `sname` sub-element) and value constraints (the last name of the author) are represented as in TAX pattern trees. The element `$4`, connected to `$1` with an `ad*` relationship, indicates that we are interested in articles or part of articles. The first score formula defines that the score of the element `$4` is calculated using a scoring function named `ScoreFun`. Furthermore, scored pattern trees require that each element having at least one sub-element involved in a scoring must also have a score; therefore the second score formula defines that the score of `$1` must be set to the same score value of `$4`.

Three TAX operators (selection, projection, and score) are redefined as *scored operators*; basically, their behavior is identical to that of the corresponding non-scored operator, but score is also calculated as specified by the score formulas. Two brand-new operators are also defined: *threshold* and *pick*.

Threshold operator $\tau'_{\mathcal{P},\mathtt{TC}}(\mathcal{C})$ returns those scored trees that satisfy each threshold condition in TC. A threshold condition refers to a node in $\mathcal{P}$ and asks for either 1) the score to be higher than a threshold V or 2) the rank to be higher than K; a tree satisfies a threshold condition if at least one of its referred nodes satisfies the condition.

Pick operator $\rho'_{\mathcal{P},\mathtt{PC}}(\mathcal{C})$ is essentially a way to remove from the output those nodes that are not expected to be relevant for the user. The pick conditions included in PC are typically a call to a pick function, that bases its decision of whether to pick or not a node on the scores of the node being considered and of some other node in the tree. For example, a pick function could specify that a node is picked either if it has a score higher than 0.8 or if at most 50% of its child elements have a score higher than 0.8.

The main drawback we see in TIX is that there is no effort in formalizing an algorithm for the translation of XQuery Full-Text expressions into TIX expressions. For example, it is not clear how to differentiate cases when a `score` variable is defined from cases when the `ftcontains` condition must be intended as mandatory. Actually, authors follow an inverse approach: they propose an extended version of XQuery (quite different from XQuery Full-Text) which is able to represent TIX expressions. Moreover, being an extension of TAX, TIX suffers the same limitations previously discussed.

In [AYCD06] an interesting algebraic approach to the representation of full-text predicates is presented. Starting from the observation that typical XML full-text languages share a common semantics, the authors define an algebra called XFT. It is based on the concept of *matching table*, which is a relational representation of the matchings found in an XML document for a full-text query. Each tuple of the matching table contains the node name where one or more matches have been found, the pattern (i.e. the searched keywords that have been found) and a list of matches (i.e. the position in which the keywords have been found). The defined algebraic operators works on matching tables:

- *get*: returns a table containing one tuple for each node with a non-empty set of

matches;

- *or*: returns the union of two matching tables;
- *and*: returns a matching table containing one tuple for each node found in both input matching tables;
- *minus*: returns a matching table containing one tuple for each node found in the first input forest and not found in the second input forest;
- *times*, *ordered*, *window*, *dist*: test various conditions (number of occurrences found, order between matches, size of the window in which matches are found, distance between each pair of adjacent patterns) on the matches.

Being based on a relational representation, the operators enjoy some of the well-known relational algebra equivalence properties, like selection commutativity, selection push-down etc. The article also presents:

- a scoring method, which permits to compute element scores incrementally from their descendants;
- some examples of translation of XQuery Full-Text predicates and NEXI [TS04] queries into XFT expressions;
- algorithms that implement the algebraic operators.

XFT is a powerful algebra for representing full-text search. However, it considers full-text tasks as a stand-alone subject, without integrating them with structured XML search tasks. Such an integration could be made difficult by the fact that XFT operators work on relational structures instead of tree-like structures.

An interesting approach to extending relational algebra with full-text concepts is that of FTA [BAYS06]. Like XFT, it does not integrate XML search tasks. It uses a data model where the basic building block is the concept of *node*, which could be a text document, and XML element, a relational tuple etc; each token in a node has an associated numeric position.

FTA operators work on *full-text relations*; each tuple in a full-text relation contains a node and a list of positions, which intuitively represents the token positions that satisfy the full-text condition. The operators are the following:

- $R_t(n, p)$: it returns a full-text relation containing a tuple for each node n that contains the searched token t at position p.
- $\pi_{n,p_1,p_2,...p_m}(R)$: it is the classical projection operator over a full-text relation R; the projection list should always include the node n.
- $R_1 \bowtie R_2$: it is the classical join operator, where the join condition is $R_1.n = R_2.n$; it ensures that positions in the same tuple are in the same node.
- $\sigma_{pred}(R)$: it is the classical selection predicate; $pred$ represents an arbitrary position-based predicate. FTA does not define any specific predicate, even if the authors propose some examples (*distance*, *ordered*, etc.).
- $R_1 - R_2, R_1 \cup R_2$: they are the classical set operators.

As an example, the following FTA expression returns the nodes that contain the keywords “assignment”, “district”, and “judge” in that order, where the keywords “district” and “judge” occur right next to each other, and the keyword “judge” appears within 5 words of the keyword “assignment”:

$$\pi_{\texttt{node}}(\sigma_{distance(att2,att3,5)}(\sigma_{ordered(att3,att1)}(\sigma_{ordered(att1,att2)}(\sigma_{distance(att1,att2,0)}(R_{district} \bowtie R_{judge}) \bowtie R_{assignment}))))$$

FTA (along with its equivalent calculus FTC) are used to define a notion of *completeness* for full-text languages; according to this definition, authors show that typical IR languages are not complete, as well as text region algebras [CM98]. Since a query evaluation algorithm for FTC queries would be polynomial in the size of data and exponential in the size of the query, authors propose a subset of FTC, including most common full-text predicates, that can be evaluated in a single pass over inverted lists. This result is obtained by observing that many full-text predicates (like *distance* and *ordered*) are true in a contiguous region of the position space; such predicates are defined as *positive*

predicates. An efficient query evaluation algorithm for positive predicates is presented, and experiments show that the performance of this algorithm scales linearly with the size of the query and the number of the context nodes.

There are a lot of other proposals for full-text algebras; some of them are quite interesting from an IR point of view, but lacks the powerful semi-structured constructs we expect from an algebra underlying XQuery Full-Text. For example, the algebra proposed in [PG04] contains a vague predicate, *about*, which defines a set of document parts within an XML document that fulfill a IR-style query; such a predicate can be combined with XPath-like expression, while there is no support for XQuery FLWOR expressions. As another example, the algebra in [MHBA04] has the same limitations, but it is based on the concept of region algebra and presupposes that XML documents are internally stored in a relational DBMS. Such proposals and similar ones are sometimes source of interesting ideas, but are too far away from our goal, so we do not treat them here in more details.

Part II

AFTX: an Algebra for Full Text Retrieval over XML Repositories

Chapter 3

The AFTX Algebra

In this chapter we present the core of our proposal: a data model for representing XML repositories and an algebra for data manipulation. In Section 3.1 we present two sample XML documents, which will be used in the following examples. Section 3.2 defines the data model, which is used by the algebraic operators shown in Section 3.3.

3.1 Motivating Examples

Throughout the rest of this chapter, we will use two working examples of XML documents. The first, shown in Figure 2.3 in textual form and graphically in Figure 3.1, is a data-centric XML document taken from XQuery Use Cases [Con06b]; the second, shown in Figure 3.2, is instead a document-centric XML document and comes from XQuery Full-Text Use Cases [Con06e].

3.2 The Data Model

In this section we present the data model forming the basic framework of our algebra. It is worth specifying that this data model should not be intended as the basis for an implementation of an XML Database System; rather, it should be considered as a formal description of the concepts that system is based on.

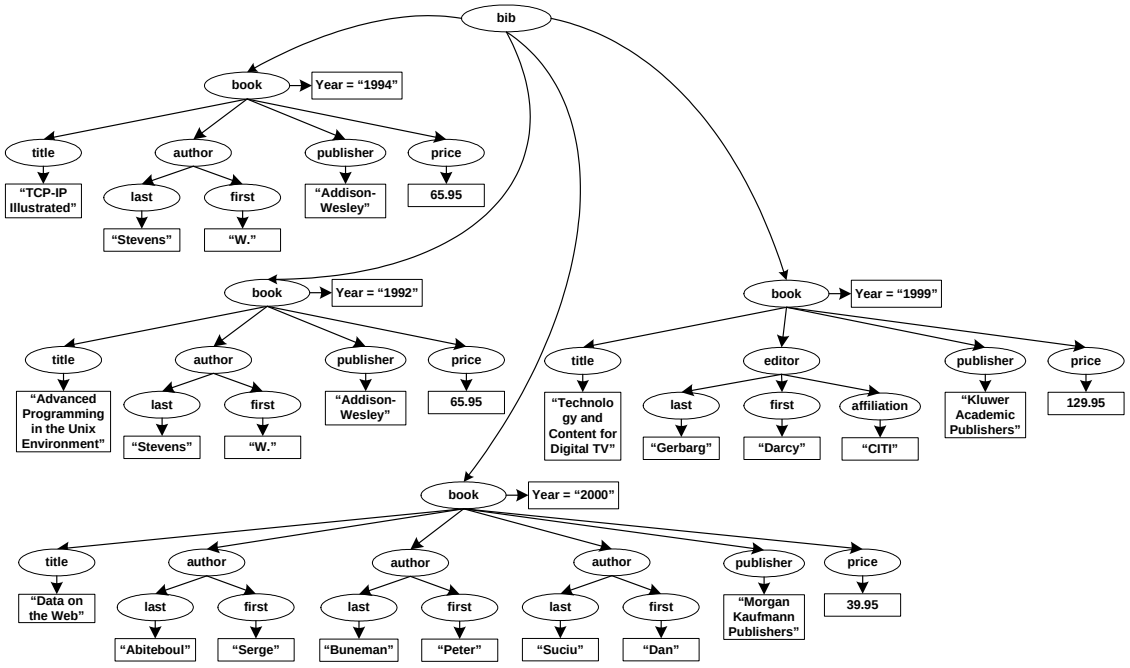


Figure 3.1: Graphical representation of the XML document in Figure 2.3.

3.2.1 Informal Overview

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.

Elements always have a *name* and an *identifier*. The notion of element identifier is similar to the notion of *pedigree* used in [JLST01]. It is not a "true" identifier; in fact multiple copies of an element share the same identifier, and elements can have a null identifier. However the identifier has the following properties:

- an element stored in an XML repository can not have a null identifier; when a tree is stored in the repository, the DBMS is supposed to assign to each element an identifier;
- two elements stored in an XML repository (in the same tree or in different trees) can not have the same identifier;
- when an algebraic operator creates a new element, it has a null identifier;

```
<book number="1">
   <metadata>
      ...
   </metadata>
   <content>
      ...
      <part number="1">
         ...
         <chapter>
            <title>Heuristic Evaluation</title>
            <p>Expert reviewers critique an interface to
            determine conformance with recognized
            usability principles. <footnote>One of the
            best known lists of heuristics is <citation
            url="http://www.useit.com/papers/heuristic
            /heuristic_list.html"> Ten Usability
            Heuristics by Jacob Nielson</citation>. Another
            is <citation url="http://usability.gov
            /guidelines/index.html"> Research-Based Web
            Design and Usability Guidelines</citation>
            </footnote></p>
         </chapter>
         ...
      </part>
      ...
   </content>
</book>
```

Figure 3.2: An XML document with elements having mixed content.

- the algebraic operators do not change the element identifier; this means that, if an algebraic expression creates multiple copies of an element, they all share the same identifier.

Each element, except for the root element, also has a *parent* element. Moreover, an element can have a *value* and a list of *attributes*, where each attribute has a name and a value.

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. The choice of leaving aside attribute values is motivated by the fact that XQuery Full-Text separately manages element values and attribute values. In fact any XQuery Full-Text expression must specify whether the full-text search of a word (or a phrase) has to be done over an element (and its sub-elements) or an attribute.

In the process of tokenization, various techniques typical of the Information Retrieval world can be used, like de-hyphenation, stopword elimination etc. In this dissertation we do not deal with such issues, because they have no impact on the operators of our algebra.

An element can have *mixed content*, i.e. it can contain character data interspersed with child elements. Such a situation is quite frequent in so-called *document-centric* XML documents, which are the main candidates for full-text retrieval. In order to manage such situations, we must keep track, in the data model, of the position of child elements inside the text of an element; we do it by numbering text tokens according to a preorder traversal of their containing tree.

Example 3.1 Consider the XML document in Figure 3.2. The tokenization of the subtree rooted at `chapter` is shown graphically in Figure 3.3; numeric position of tokens is indicated in square brackets. As we can see, de-hypenation is used and punctuation is not tokenized; these choices should not be considered as part of our model: they are instead just an example of the rules that could be followed in the tokenization phase.

Note that token enumeration proceeds from the element `title` to the element `p`, then to the element `footnote`; the fact that the first `citation` element is mixed inside the

text of `footnote` is represented assigning to its first token (`Ten`) the number immediately following the one assigned to the token (`is`) that precedes `citation` in the text of `footnote`. The two `url` attributes are separately tokenized, so their enumeration always starts from 1.

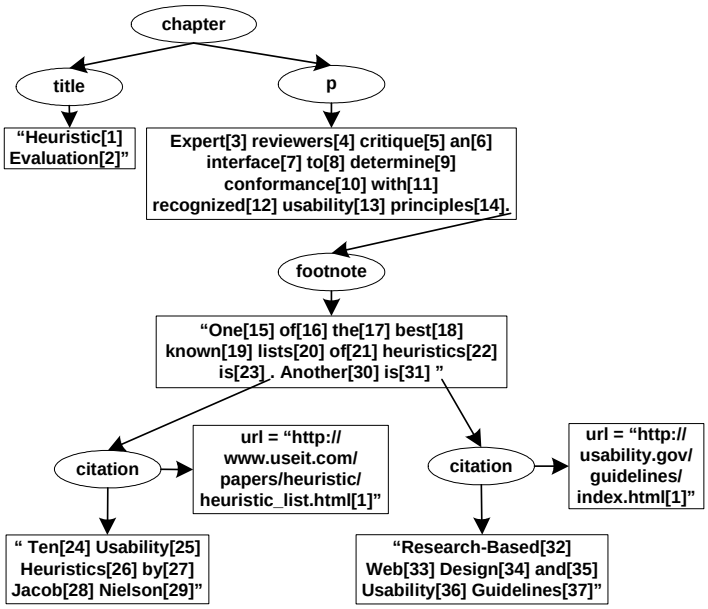


Figure 3.3: Tokenization of an XML document with mixed content.

Given the above tokenization, we can define a new property for our elements: the *fulltext*, which is the value of an element concatenated with the value of its sub-elements. For example, the fulltext of the `footnote` element in Figure 3.3 is *“One[15] of[16] the[17] best[18] known[19] lists[20] of[21] heuristics[22] is[23] Ten[24] Usability[25] Heuristics[26] by[27] Jacob[28] Nielson[29]. Another[30] is[31] Research-Based[32] Web[33] Design[34] and[35] Usability[36] Guidelines[37]”*.

Ordering between elements is represented by a property o, whose value is an integer ranging from 1 to the number of children of each element's parent. For example if we denote with e the *title* element in Figure 3.3 and with e' the *p* element, then $e.o = 1$ and $e'.o = 2$.

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 strictly equal if they are *the same* element. This means that two elements having the same name, the same value and the same attribute list are not necessarily strictly equal; contrariwise, 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 notion of strict equality is essential for the definition of some operators of our algebra, like difference, and is very useful in other situations, e.g. when we want to join a tree with itself. 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, in the same way relational algebra manipulates relations (that contain tuples) and returns a relation. However, some differences arise.

First of all, as already said, trees and forests are ordered: an order relationship between sibling elements is defined (and represented in our data model by the element property o), as well as between trees contained in a forest. In the relational world, on the other side, tuples and relations are not ordered; in fact we have no way of extracting the first tuple of a relation or the first attribute of a tuple, just because no order relationship is defined between attributes or between tuples.

An even more important difference is that trees contained in a forest are not required to share the same structure: forests are just ordered collection of trees, but there is no constraint on the structure of the trees contained in a forest; this choice is coherent with one of the distinguishing features of the semi-structured world: the *vagueness* of the schema. All the tuples contained in a relation have instead the same attribute list.

Provided that a tree is always contained in a forest, we can define for the root element a *count* property, which represents the number of trees contained in the forest; the value of such a property will obviously be the same for the root element of any tree contained in a forest.

Many *subforests* can be picked from a forest. As in the case of subtrees, the formal

definition of subforests 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. Informally, a forest is a subforest of another forest if it contains only trees strictly equal to trees of the original forest, and the relative order between pairs of trees remains unchanged. If a forest F is a subforest of G and G is a subforest of F, then the two forests are strictly equal: they contain the same trees in the same order.

3.2.2 Formal Definitions

We start with the definition of the basic building blocks of our data model: attributes and elements.

Definition 3.1 (Attribute) *An attribute a is a pair (n, V), where:*

1. *n is the name of the attribute;*
2. *V is a (possibly empty) ordered list $((t_1, 1), (t_2, 2), \ldots, (t_n, n))$ of pairs, where t_i is a token; V represents the value (possibly null) of the attribute.*

As already said, an attribute's tokens are always enumerated from 1 to n, where n is the number of tokens. We refer to each component of the tuple with the notation $a.x$, i.e. $a.n$ is the name of the attribute and $a.V$ is its value. With the notation $a.V[1]$ we indicate the first pair in the list V, while $a.V[1].t$ represents the first token.

Definition 3.2 (Element) *An element e is a tuple (k, n, A, V, p), where:*

- *k is a possibly null identifier;*
- *n is the name of the element;*
- *$A = \{a_1, a_2, \ldots, a_n\}$ is the set (possibly empty) of the element's attributes;*
- *V is a (possibly empty) ordered list of pairs (t, n), where t is a token and n is an integer value; V represents the value (possibly null) of the element;*
- *p is a pointer (possibly null) to its parent element.*

We refer to each component of the tuple with the notation $e.x$, e.g. $e.n$ is the name of the element e. Attributes are referred to using the notation $e.A[attname]$; for example, with $e.A[\texttt{id}].V$ we indicate the value of the `id` attribute of the element e. With the notation $e.V[1]$ we indicate the first pair in the list V, while $e.V[1].t$ represents the first token.

Now we define the concept of tree.

Definition 3.3 (Tree) *A tree T is a set of pairs (e, o), where:*

- *e is an element;*
- *o is an integer value.*

Each tree T satisfies the following properties:

- *Let $E_T = \{e \mid \exists (e, o) \in T\}$ the set of elements in T; then $\exists! e \in E_T$ such that $e.p = null$;*
- *For each $e \in E_T$, let $S_e = \{(e', o) \in E \mid e'.p = e.p\}$ and $O_{S_e} = \{o \mid (e', o) \in S_e\}$; then O_{S_e} is the set of the integer values between 1 and $|O_{S_e}|$;*
- *Let e be the first element in a preorder trasversal of the tree such that $e.V$ is not null; then $e.V[1].n = 1$;*
- *Let N be the total number of tokens found in elements' values of T; then each pair (t_i, n_i) is such that $1 \leq n_i \leq N$, and do not exist two pairs (t_i, n_i) and (t_j, n_j) such that $n_i = n_j$;*
- *For each $V = ((t_1, n_1), \ldots, (t_m, n_m))$, if (t_i, n_i) precedes (t_j, n_j) then $n_i < n_j$;*
- *Let $V = ((t_1, n_1), \ldots, (t_m, n_m))$ and $V' = ((t'_1, n'_1), \ldots, (t'_m, n'_m))$ be the values of two elements e and e' such that e is the parent of e'; then either $n_1 < n'_1 < n'_m < n_m$ or $n'_1 > n_m$.*

The first condition in this definition states that a tree always has exactly one root element; the second explains how order between elements is represented; the following four describe tokens enumeration. With $root(T)$ we denote the root element of the tree T, i.e. the element e such that $e.p = null$.

Example 3.2 Consider the XML document in Figure 2.3. In our data model, it is represented by the tree $T = ((e_1, 1), (e_2, 2), \ldots, (e_{36}, 36))$. In what follows we present the elements e_i; each element is of the form $e_i = (n, A, V, p)$. The identifier k is omitted (as stated, it can be thought of as a unique integer value assigned by the system to each element stored in the repository); the pointer to the parent element is represented with e_j.

e_1 = (“bib”, null, null, null)
e_2 = (“book”, ((“year”, ((“1984”, 1)))), null, e_1)
e_3 = (“title”, null, ((“TCP-IP”, 1), (“Illustrated”, 2)), e_2)
e_4 = (“author”, null, null, e_2)
e_5 = (“last”, null, ((“Stevens”, 3)), e_4)
e_6 = (“first”, null, ((“W.”, 4)), e_4)
e_7 = (“publisher”, null, ((“Addison-Wesley”, 5)), e_2)
e_8 = (“price”, null, ((“65.95”, 6)), e_2)
e_9 = (“book”, ((“year”, ((“1992”, 1)))), null, e_1)
e_{10} = (“title”, null, ((“Advanced”, 7), (“Programming”, 8), (“in”, 9), (“the”, 10), (“Unix”, 11), (“Environment”, 12)), e_9)
e_{11} = (“author”, null, null, e_9)
e_{12} = (“last”, null, ((“Stevens”, 13)), e_{11})
e_{13} = (“first”, null, ((“W.”, 14)), e_{11})
e_{14} = (“publisher”, null, ((“Addison-Wesley”, 15)), e_9)
e_{15} = (“price”, null, ((“65.95, 16)), e_9)
e_{16} = (“book”, ((“year”, ((“2000”, 1)))), null, e_1)
e_{17} = (“title”, null, ((“Data”, 17), (“on”, 18), (“the”, 19), (“Web”, 20)), e_{16})
e_{18} = (“author”, null, null, e_{16})
e_{19} = (“last”, null, ((“Abiteboul”, 21)), e_{18})

e_{20} = (“first”, null, ((“Serge”, 22)), e_{18})

e_{21} = (“author”, null, null, e_{16})

e_{22} = (“last”, null, ((“Buneman”, 23)), e_{21})

e_{23} = (“first”, null, ((“Peter”, 24)), e_{21})

e_{24} = (“author”, null, null, e_{16})

e_{25} = (“last”, null, ((“Suciu”, 25)), e_{24})

e_{26} = (“first”, null, ((“Dan”, 26)), e_{24})

e_{27} = (“publisher”, null, ((“Morgan”, 27), (“Kaufmann”, 28), (“Publishers”, 29)), e_{16})

e_{28} = (“price”, null, ((“39.95”, 30)), e_{16})

e_{29} = (“book”, ((“year”, ((“1999”, 1)))), null, e_1)

e_{30} = (“title”, null, ((“Technology”, 31), (“and”, 32), (“Content”, 33), (“for”, 34), (“Digital”, 35), (“TV”, 36)), e_{29})

e_{31} = (“editor”, null, null, e_{29})

e_{32} = (“last”, null, ((“Gerbarg”, 37)), e_{31})

e_{33} = (“first”, null, ((“Darcy”, 38)), e_{31})

e_{34} = (“affiliation”, null, ((“CITI”, 39)), e_{31})

e_{35} = (“publisher”, null, ((“Kluwer”, 40), (“Academic”, 41), (“Publishers”, 42)), e_{29})

e_{36} = (“price”, null, ((“129.95”, 43)), e_{29})

For the sake of convenience, we also define two derived element properties: the *text value* and the *full-text value*. Both are obtained by a *de-tokenization* of the textual content of an element; the first refers to the value of a single element, the second to the value of every element in a subtree. These two element properties are useful for the definition of other concepts and algebraic operators.

Definition 3.4 (Element Text Value) *Let e be an element and let $e.V = ((t_1, n_1), (t_2, n_2), \ldots, (t_m, n_m))$. The text value of e (denoted $e.v$) is the concatenation of the tokens $t_1 \ldots t_m$, separated by a white space.*

Definition 3.5 (Element Full-Text Value) *Let e be an element; let $firsttoken_e = min($ $\{n_i \mid \exists e'$, descendant of e, such that $(t_i, n_i) \in e'.V\})$ and $lasttoken_e = max(\{n_i \mid \exists e'$, descendant of e, such that $(t_i, n_i) \in e'.V\})$. The full-text value of e (denoted $e.fulltext$) is the concatenation of the tokens t_i from $firsttoken$ to $lasttoken$, separated by a white space.*

Example 3.3 Consider the element e_{29} in Example 3.2. The value of the derived property $e.v$ is *"Technology and Content for Digital TV"*; the value of the derived property $e.fulltext$ is *"Technology and Content for Digital TV Gerbarg Darcy CITI Kluwer Academic Publishers 129.95"*.

Now we define the concept of subtree, which is based on the notion of elements strict equality.

Definition 3.6 (Elements strict equality) *Two elements $e_1 = (k_1, n_1, A_1, V_1, p_1)$ and $e_2 = (k_2, n_2, A_2, V_2, p_2)$ are strictly equal (denoted $e_1 \equiv e_2$) if and only if all their components (except for parent element and tokens enumeration) are equal, i.e. $e_1.k = e_2.k$, $e_1.n = e_2.n$, $e_1.A = e_2.A$, $e_1.v = e_2.v$.*

In the previous definition, with $e_1.A = e_2.A$ we mean that the two attribute sets must be equal, i.e. each attribute in the first set must be present in the second set (with the same value) and viceversa.

Definition 3.7 (Subtree) *Given two trees T and T', let $E = \{e \mid (e, o) \in T\}$ and $E' = \{e \mid (e, o) \in T'\}$. T' is a subtree of T (denoted $T' \subset T$) if:*

- *$\forall e' \in E'$, $\exists e \in E$ such that $e' \equiv e$;*
- *$\forall e' \in E'$, let $e \in E$ be an element such that $e' \equiv e$; then either $e'.p = e.p$ or $e'.p = null$;*
- *$\forall (e_1, o_1) \in T, (e_2, o_2) \in T$ such that $e_1.p = e_2.p$ and $o_1 < o_2$, if $\exists (e'_1, o'_1) \in T', (e'_2, o'_2) \in T'$ such that $e'_1 \equiv e_1$ and $e'_2 \equiv e_2$, then $o'_1 < o'_2$.*

Let $T' \subset T$ and let $e \in E$ such that $e \equiv (root)(T')$. If, $\forall e_i \in E$ such that e_i is a descendant of e in T, $\exists e'_i \in E'$ such that $e'_i \equiv e_i$, then T' is a complete subtree of T (denoted $T' \subset^* T$).

The first condition says that each element of a subtree must come from the original tree; the second says that hierarchy can not be changed, except that a non-root element could become root of the subtree; the third states that order between elements must be preserved. The difference between a complete subtree and a non-complete subtree is shown in Figure 3.4: given the tree (a), the tree (b) is a complete subtree, while (c) is not complete.

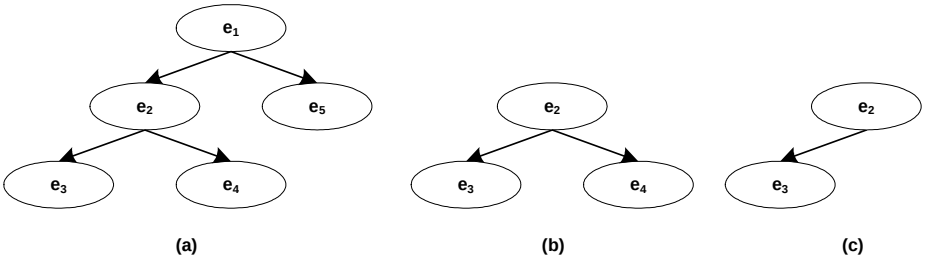


Figure 3.4: A tree (a), a complete subtree (b) and a non-complete subtree (c).

The last concepts to be defined are those of forest and subforest; for the definition of the subforest we need the notion of trees strict equality.

Definition 3.8 (Forest) *A forest $F = (T_1, T_2, ..., T_n)$ is an ordered list of distinct trees.*

Definition 3.9 (Trees strict equality) *Two trees T_1 and T_2 are strictly equal (denoted $T_1 \equiv T_2$) if $\exists f : T_1 \rightarrow T_2$ such that, $\forall (e, o) \in T_1$, $f((e, o)) = (e', o')$ is such that:*

- $e' \equiv e$;
- $e'.p = e.p$;
- $o' = o$.

Definition 3.10 (Subforest) *Given two forests $F = (T_1, T_2, \ldots, T_n)$ and $F' = (T'_1, T'_2, \ldots, T'_m)$, F' is a subforest of F (denoted $F' \subset F$) if:*

- $\forall T' \in F', \exists T \in F$ *such that* $T' \equiv T$*;*
- $\forall T_i, T_j \in F, i < j$*, if* $\exists T'_{i'}, T'_{j'} \in F'$ *such that* $T_i \equiv T'_{i'}$ *and* $T_j \equiv T'_{j'}$*, then* $i' < j'$*.*

The second condition in the last definition states that order between trees in a subforest must be identical to that between trees in the original forest. If two forests contain the same trees in the same order, then the two forests are said to be strictly equal.

Definition 3.11 (Forests strict equality) *Two forests F and F' are strictly equal (denoted $F \equiv F'$) if:*

- $F \subset F'$*;*
- $F' \subset F$*.*

As already said, the basic building blocks in our data model are elements (possibly with attributes), which are contained into trees, which are contained into forests. Some element properties (like name and text value) do not depend on the tree the element is contained in; on the contrary, other properties (like full-text value) do depend on the tree the element is contained in. It is useful to define other two properties, which can be thought of as *tree properties*; they depend on the forest the tree is contained in. For consistency, we define these properties as element properties, but they make sense only for the root element of a tree.

Definition 3.12 (Element Count) *Let e be the root element of a tree T and let F be the forest that contains T. The count of e (denoted $e.count$) is the number of trees contained into F.*

Definition 3.13 (Element Position) *Let e be the root element of a tree T and let F be the forest that contains T. The position of e (denoted $e.pos$) is the position of the tree T in the forest F.*

It should be clear that the value of the *count* property value will be the same for each root element of the trees contained in a forest, while the *position* property value varies from 1 to n, where n is the number of trees in the forest.

Example 3.4 Suppose to have a forest (shown in Figure 3.5) composed by the complete subtrees rooted at book that can be extracted from the XML document in Figure 2.3 (we will see in Section 3.3 how to obtain such a forest using an operator of our algebra); in this figure and in all the following figures representing a forest, trees are ordered from left to right and from top to bottom. Let e be the book element that corresponds to the book *"Advanced Programming in the Unix Environment"*; then $e.count = 4$ and $e.pos = 2$.

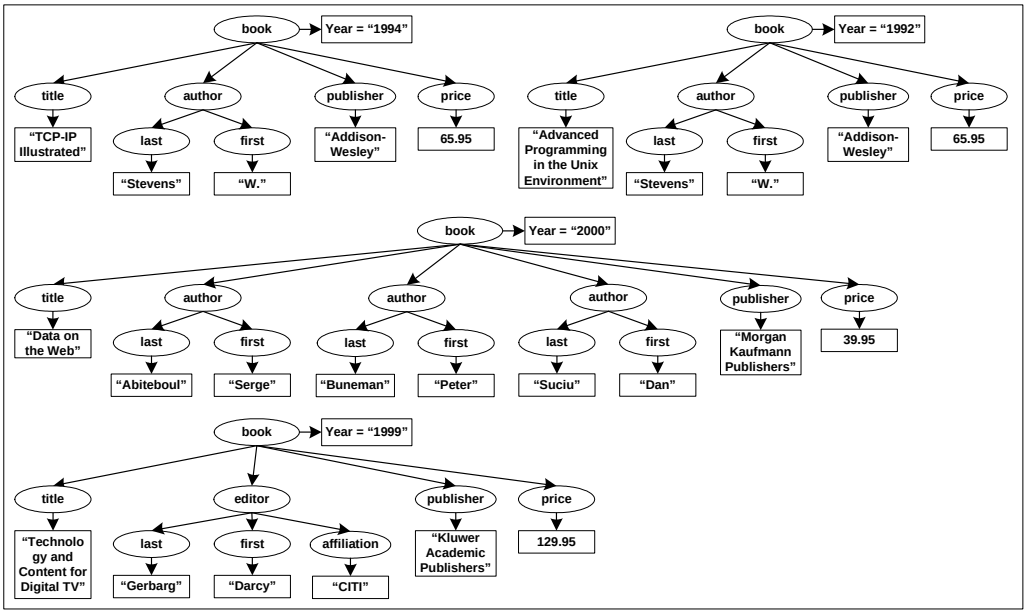


Figure 3.5: Graphical representation of a forest.

Finally, we define another element property: the *score* property. The value of this property (initially set to a default value) will be changed by some full-text algebraic operators of our algebra. We will discuss the meaning of this property in Section 3.3; here we just define the possible values it can assume.

Definition 3.14 (Score) *Let e be an element of a tree T. The score of e (denoted $e.score$) is a value in the range $[0, 1]$.*

3.2.3 A Comparison with XQuery (and XQuery Full-Text) Data Model

Our data model presents some differences with respect to the XQuery Data Model; such differences (which are summarized in Table 3.1) result in a simplification of the data model.

Table 3.1: Comparison between XQuery Data Model and AFTX Data Model.

Concept	**XQuery Data Model**	**AFTX Data Model**
Basic building block	Items: nodes or atomic values	Elements
Collection	Sequences	Forests
Types	XML Schema types plus five additional types	No type information
Identity	Nodes have unique identity, atomic values have not	Element identity through identifier
Node kinds	Document, Element, Attribute, Text, Namespace, Processing Instruction, Comment	Trees and elements (with attributes and value)
Element properties	`dm:children`, `dm:attributes`, `dm:node-name`, `dm:string-value`, `dm:typed-value`, `dm:type-name`	A, n, $fulltext$

In XDM, the basic concept is that of sequence, which is composed by nodes (i.e. a single node or a tree formed by nodes) and atomic values. The concept of forest present in our data model is equivalent to the XDM sequence, but forests contain only trees: we do not consider the case of atomic values. XDM node identity concept corresponds to our strict equality notion.

Every node in XDM has a type; in our data model we do not consider types. Like XDM, we provide element identity, using the element identifier k.

XDM nodes are of seven kinds. Document nodes correspond to our concept of tree, while Element nodes correspond to our elements. The counterparts of Attribute and Text nodes in our model are respectively the element properties A and v. We do not consider namespaces, processing instructions and comments.

Every XDM node has a set of accessors, which represent the properties of the node. There is a correspondence between some accessors and some element properties of our model: `dm:attributes` corresponds to A, `dm:node-name` corresponds to n, `dm:string-value` corresponds to $fulltext$. No correspondence exists for `dm:typed-value` and `dm:type-name`, because our data model does not take types into account; `dm:children` also is not present in our data model, even if a corresponding derived property could be easily defined.

Like XQuery Full-Text Data Model, our data model is based on a tokenization of the source document, that assigns a numeric value that represents the relative position of the word in document order. We do not deal with paragraph and sentences enumeration, because our algebraic full-text operators (which will be shown in Section 3.3) do not provide such search options.

In the XQuery Full-Text Data Model score values are represented as a variable; it is not clear, however, how the value of the score variable is bound to the sequence of items that generate that score. In our data model score values are represented by an element property; this gives an immediate and easy to understand correspondence between a tree and its score value.

3.3 Algebraic Operators

3.3.1 Informal Overview

In this section we define the AFTX operators, which can be categorized into *basic* operators (which cover classical data manipulation tasks) and *full-text* operators (which perform IR-style queries). For each of them, we give an informal overview of the characteristics and one or more basic examples; later we present the formal definitions.

The operators of our algebra (which are summarized in Table 3.2) can be unary or binary. Unary operators take in a forest and return a forest; their general form is

$$\alpha_P(F)$$

where α is an operator, P is a predicate and F is the input forest. Binary operators take in two forests and return a forest; their general form is

$$\alpha_P(F, G) \ .$$

In order to improve readability, binary operators can also be represented using the equivalent infix notation

$$F \alpha_P G \ .$$

Sometimes we write $\alpha_P(T)$, where T is a tree. This expression must be intended as the application of the operator α to a forest containing the single tree T.

The algebra is *closed*: all the operators take in forest(s) and return a forest. Consequently the operators can be composed with each other. In an algebraic expression, wherever an input forest is expected, it is possible to find:

- an algebraic operator; for instance in the expression $\alpha(\alpha'(\ldots))$ the operator α takes in the output forest of the operator α';
- a new forest, obtained by reading an XML document; for instance in the expression $\alpha(\text{“}docname\text{”})$ the operator α takes in the forest (containing a single tree) obtained by reading the document *docname*.

Union

The union operator is quite similar to its relational counterpart; it takes in two forests and returns a new forest composed by the trees contained in the two input forests. Union preserves ordering: the output forest will contain the trees coming from the first input forest (in the same order as they were in the input forest), followed by the trees coming from the second input forest. This implies that union is not commutative; this is an unavoidable deviation from relational algebra, due to the importance of order in semi-structured data

Table 3.2: AFTX algebraic operators.

Operator	**Usage**
Union	$F \cup F'$
Difference	$F - F'$
Projection	$\pi_{\lambda}(F)$
Selection	$\sigma_{\lambda[\gamma]}(F)$
Product	$F \times F'$
Join	$F \bowtie_{\lambda[\gamma]} F'$
Deletion	$\delta_{\lambda[\gamma]}(F)$
Grouping	$\Sigma_{((\lambda_1 p_1,n_1),(\lambda_2 p_2,n_2),...),(\lambda'_1,\lambda'_2,...)}(F)$
Duplicate elimination	$\nu_{(\lambda_1 p_1,n1),(\lambda_2 p_2,n_2),...}(F)$
Ordering	$o_{\lambda_1 p_1\ a_1,\lambda_2 p_2\ a_2,...}(F)$
Tree Construction	$\iota_{e_1,e_2,...}(F)$
Full-Text Selection	$\varsigma_{\lambda a[\gamma,x,\texttt{stem},\texttt{thes},\texttt{stop}]}(F)$
Full-Text Score Assignment	$\xi_{\lambda a[\gamma,x,\texttt{stem},\texttt{thes},\texttt{stop}]f}(F)$
Full-Text Selection with Score	$\overline{\varsigma}_{\lambda a[\gamma,x,\texttt{stem},\texttt{thes},\texttt{stop}]f}(F)$
Top-K Full-Text Selection	$\top_{\lambda a[\gamma,x,\texttt{stem},\texttt{thes},\texttt{stop}]f,k}(F)$
Threshold Full-Text Selection	$\omega_{\lambda a[\gamma,x,\texttt{stem},\texttt{thes},\texttt{stop}]f,\tau}(F)$

model. The behavior of the union operator is shown graphically in Figure 3.6; rounds with label T_i represents trees.

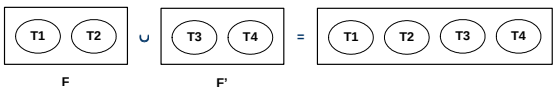


Figure 3.6: The behavior of AFTX union operator.

Example 3.5 Suppose to have two XML documents, named “CSbooks.xml” and “Mathbooks.xml”, with a structure similar to the XML document shown in Figure 2.3. The

query

$$\text{“}\texttt{CSbooks.xml}\text{”} \cup \text{“}\texttt{Mathbooks.xml}\text{”}$$

returns a forest containing two trees: the tree contained in “CSbooks.xml” followed by the tree contained in “Mathbooks.xml”. In this example the two expected input forests are obtained by reading two XML documents.

Difference

Like union, the difference operator is analogous to the relational difference operator. It 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 strict equality notion presented in Section 3.2.2: a tree from the first forest is retained in the output if the second forest does not contain a strictly equal tree.

Difference preserves ordering between trees: it returns trees in the same order they were in the first input forest. The behavior of the difference operator is shown graphically in Figure 3.7; the fact that a tree in the second forest is strictly equal to a tree in the first forest is indicated by using the same label (T_2 in the example) for both trees.

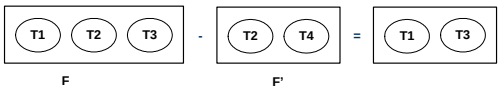


Figure 3.7: The behavior of AFTX difference operator.

Example 3.6 Let A and B be two algebraic expressions that take in the XML document shown in Figure 2.3 and return, respectively, a forest containing all the subtrees rooted at *book* and a forest containing all the subtrees rooted at *book* such that the attribute *year* has the value *“1992”* (we will see later how to obtain such forests using projection and selection operators). The query

$$A - B$$

returns all the books except those written in 1992.

Projection

In relational algebra, projection performs a *vertical* decomposition of the input relation: every tuple is output, but only the attributes of interest are retained. AFTX projection operator behaves in a similar way: every input tree contributes to the output, but only the subtrees of interest are retained. A graphical representation of the behavior of the AFTX projection operator compared to the relational counterpart is shown in Figure 3.8, where grey parts of relations and trees are those retained after projection.

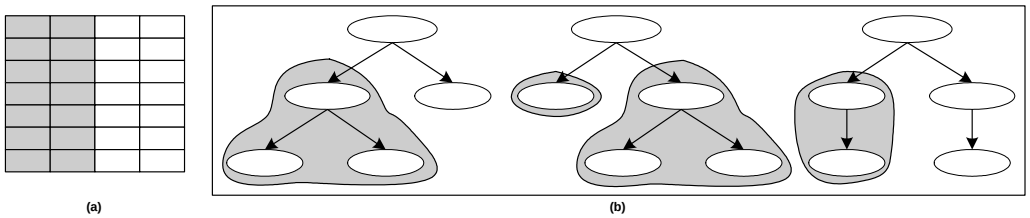


Figure 3.8: The behavior of relational projection operator (a) compared to AFTX projection operator (b).

The subtrees of interest are specified in the projection predicate through a *path expression* λ. The concept of path expression is almost identical to that used in XPath: there are child ("/") and descendant ("//") axis, while the elements to retrieve can be specified by the name, the special string "*" (meaning "any name") or by an integer specifying the position of the element. For example the path expression `/book/3//*` retrieves any element that is descendant of the third child of a *book* element; for each element satisfying the path expression, the output will contain the complete subtree rooted at it.

The main difference between relational and AFTX projection is the cardinality of the output: while in relational algebra each input tuple corresponds to exactly one output tuple (and multiple tuples can collapse in a single output tuple), in AFTX projection each input tree corresponds to zero, one or more output trees. The fact that an input tree can not have a corresponding output tree is due to one of the main distinguishing characteristic of the semi-structured model: the *vagueness* of the schema; this consideration leads, in our model, to an heterogeneity of the forests, as already noted in Section 3.2.1. Consequently,

it is possible that a path expression can not be found in one input tree, thus excluding that tree from the output of a projection. On the other side, an input tree can have multiple corresponding trees in the projection output: while in the relational world a tuple contains only one attribute with a given name, in the semi-structured world an element can have multiple child elements with the same name. Moreover, multiple subtrees can be obviously returned by a projection if the path expression contains a wildcard ("*") or a descendant axes ("//").

The projection operator preserves order between elements and trees, i.e.:

- if a tree T_1 precedes a tree T_2 in the input forest, then each subtree T_1^i of the tree T_1 precedes each subtree T_2^j of the tree T_2 in the output forest;
- if an element e_1 precedes an element e_2 in an input tree T, then a subtree T_1 rooted at $e_1' \equiv e_1$ precedes a subtree T_2 rooted at $e_2' \equiv e_2$ in the output forest.

Example 3.7 Consider the XML document in Figure 2.3. We want to retrieve the title of all the books. The following expression answers to the query:

$$\pi_{\texttt{/bib/book/title}}(\text{“}\texttt{books.xml}\text{”}) \ .$$

The result of the algebraic expression is shown graphically in Figure 3.9. Note that, in this case, four output trees correspond to one input tree, because the input tree contains four subtrees reachable by following the path `/bib/book/title`.

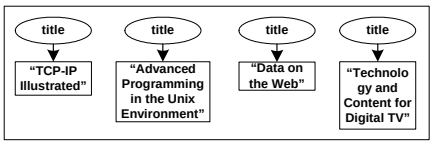


Figure 3.9: Graphical representation of the result of a projection.

It is worth noticing that the evaluation of a path expression always starts from the root element of each input tree. For example, suppose to perform a projection over the XML document in Figure 2.3 using the predicate `/bib/book`; now we have a forest of trees

rooted at `book`. Later, we want to project again this result in order to obtain trees rooted at `title`; the second projection predicate must be `/book/title`, that is interpreted as *"find those elements named* `title` *having a parent root element named* `book`*"*. In this case the same result would be obtained using the predicate `//title`; using the predicate `/title` would instead result in an empty forest, because there are no trees having a root element named `title`.

Selection

In relational algebra, selection performs a *horizontal* decomposition of the input relation: only the tuples of interest are output, and every attribute of those tuples is retained. AFTX selection operator behaves in a similar way: only the trees of interest contribute to the output, and those trees are entirely retained. A graphical representation of the behavior of the AFTX selection operator compared to the relational counterpart is shown in Figure 3.10; again, grey parts of relations and forests are those retained in the output.

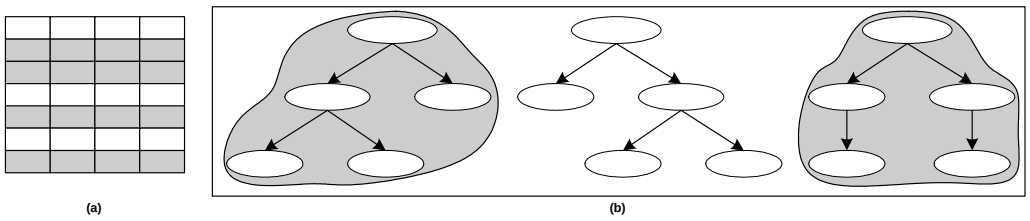


Figure 3.10: The behavior of relational selection operator (a) compared to AFTX selection operator (b).

The trees of interest are specified through a selection predicate, which is formed by an optional path expression λ and, enclosed in square brackets, an optional *selection condition* γ. If present, the path expression locates, for each input tree, a set of subtrees; in practice, it is used to generate a temporary projection on the input tree. Each subtree belonging to the temporary projection result is then checked: if at least one of those subtrees satisfies the selection condition, the original input tree belongs to the output of the selection. If the selection condition is not present, the selection predicate must be

intended like *"find those trees having at least one subtree reachable following the path expression λ"*. If the path expression is not present, the selection condition must be checked against the original trees.

In relational algebra the selection condition can only refer to the value of some attribute; in AFTX it can refer to any element property: its name and value, the name and value of one of its attributes, plus the value of some aggregate functions; aggregate functions are calculated considering each subtree belonging to the temporary projection result. For example, suppose to have an input forest composed by trees rooted at a *book* element (e.g. the result of $\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”})$); then the selection predicate $/\texttt{book}/\texttt{author}[.\texttt{count} > 2]$ means that all the books having more than two authors should be returned. In fact the selection operator behaves according to the following steps:

1. consider T_1, the first input tree;
2. build the forest $F_1 = \pi_{/\texttt{book/author}}(T_1)$;
3. count the number of trees in F_1; if it is greater than 2, add T_1 to the output;
4. repeat steps 1–3 for the other input trees.

Moreover, the selection condition can also refer to the *identity* of a subtree, i.e. it is possible to specify that a subtree must be strictly equal to another subtree for the input tree to be returned. This feature is useful when we want to join a tree with itself (we will see later how to perform a join operation and what this means).

It is worth noticing that not every subtree located by the path expression is required to satisfy the selection condition for the tree to be returned. Instead, AFTX projection has an *existential* semantic: a tree is returned if at least one subtree located by the path expression satisfies the selection condition. For example, consider the selection predicate $/\texttt{book}/\texttt{author}/\texttt{last}[.\texttt{v} = \text{“}\texttt{Suciu}\text{”}]$; it means that each book having an author whose last name is *"Suciu"* should be retained. Then, the book *"Data on the Web"* in Figure 2.3 satisfies the selection condition, because one of its authors is Dan Suciu, even if it has two more authors with a different last name.

Example 3.8 Consider the XML document in Figure 2.3. We want to retrieve all the books whose price is greater than 50. The following expression answers to the query:

$$\sigma_{/\texttt{book}/\texttt{price}[.\texttt{v}>50]}(\pi_{/bib/book}(\text{“}\texttt{books.xml}\text{”}))\ .$$

The result of the algebraic expression is shown graphically in Figure 3.11.

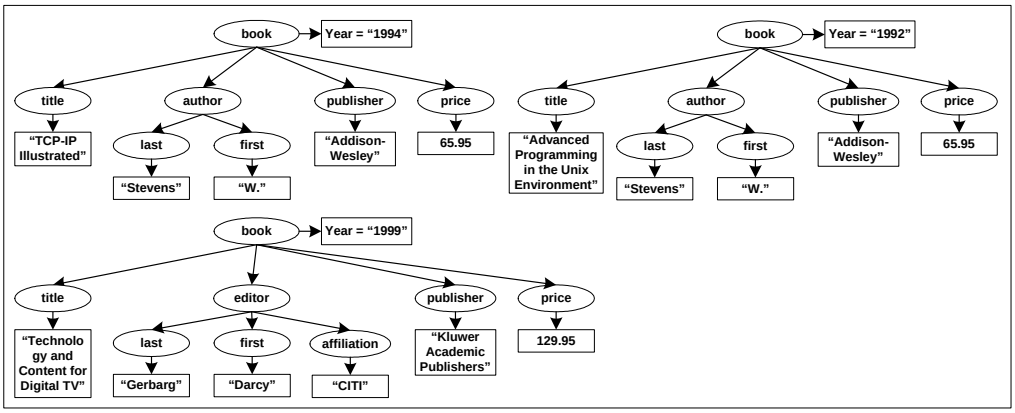


Figure 3.11: Graphical representation of the result of the expression in Example 3.8.

The inner projection returns a forest of trees rooted at *book*; the selection retains those whose price is greater than 50. As previously stated, the selection condition is optional; thus, the following query

$$\sigma_{/\texttt{book}/\texttt{price}}(\pi_{/bib/book}(\text{“}\texttt{books.xml}\text{”}))$$

is valid and retrieves all the books which have an associated price (in this case, all the four input trees).

Product

In relational algebra the product operator 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. AFTX product operator behaves similarly: it combines in every possible way all the trees from the first forest F with all the trees of the second forest F'. The combination among two trees is

obtained by creating a new root node called *prod_root*, whose left and right child will be, respectively, the tree from the first input forest and the tree from the second input forest. A graphical representation of the behavior of the AFTX product operator compared to its relational counterpart is shown in Figure 3.12; in this case, grey coloring distinguishes the first input tuple (or tree) from the second one.

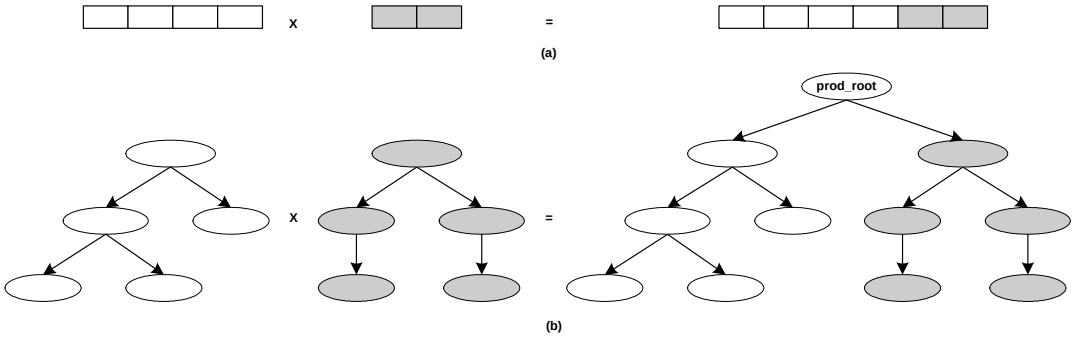


Figure 3.12: The behavior of relational product operator (a) compared to AFTX product operator (b).

The product operator preserves ordering, in the sense that the combination of trees occurs following the order of the input forests. For example, if $F = (T_1, T_2)$ and $F' = (T'_1, T'_2)$, then the first tree in $F \times F'$ will be the combination of the input trees T_1 and T'_1.

Example 3.9 Consider the XML document in Figure 2.3. We want to retrieve, for each author, its name and the books written by him. The following expression answers to the query:

$$\sigma_{/\texttt{prod\_root}[/\texttt{author}\equiv/\texttt{book/author}]}\big(\\ \pi_{/\texttt{bib/book/author}}(\text{“}\texttt{books.xml}\text{”}) \times \pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”})\big)$$

The two projections return two forests containing, respectively, all the subtrees rooted at `author` and all the subtrees rooted at `book`. Then product combines each author with each book. Finally selection retains only the pairs (author, book) such that the author is one of the authors of the book; this is done using an identity test, which restricts the result to those pairs (author, book) such that the tree rooted at `author` is a subtree of the tree

rooted at `book`. A partial result of the algebraic expression (limited to the author *"W. Stevens"*) is shown graphically in Figure 3.13.

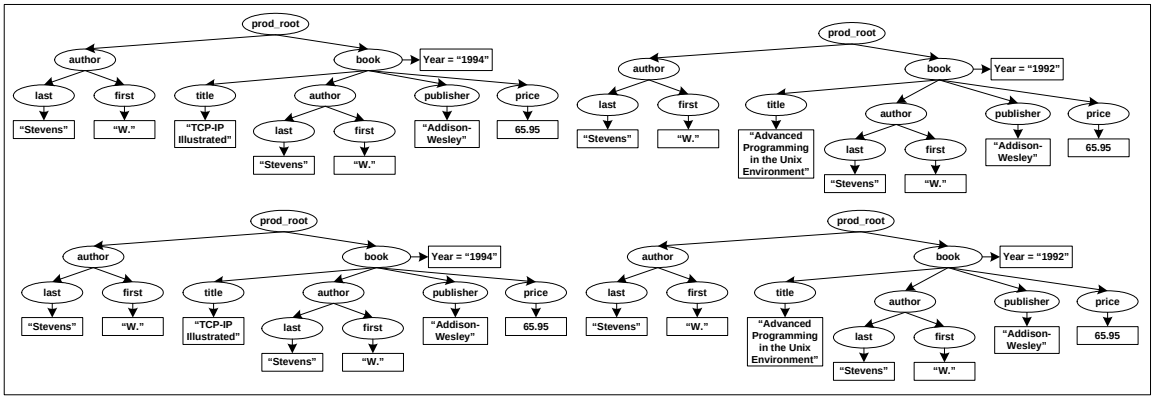


Figure 3.13: Graphical representation of the result of an algebraic expression involving product.

The previous expression does not return the answer one probably wants. First of all, each pair (author, book) is repeated as many times as the number of books written by that author. For example, *"W. Stevens"* has written two books, then there are two subtrees of the input tree reachable by following the path `/bib/book/author`, and these two subtrees will be part of the result of the first projection; they will be combined with each tree belonging to the result of the second projection (i.e. the subtrees rooted at `book`), thus resulting in two output trees for each pair (*"W. Stevens"*, book); the subsequent selection removes unwanted pairs (i.e. pairs involving books not written by *"W. Stevens"*), but it does not resolve the duplicates problem.

Moreover, one probably wants to retain only a part of the information relative to a book (e.g. the title and the publisher), and the books written by an author should be somewhere *grouped*, so that the name of an author appears just once. Finally, the `prod_root` element should be probably eliminated or renamed. Figure 3.14 shows the result one probably wants; we will see later how to reach this goal combining product with other operators of our algebra.

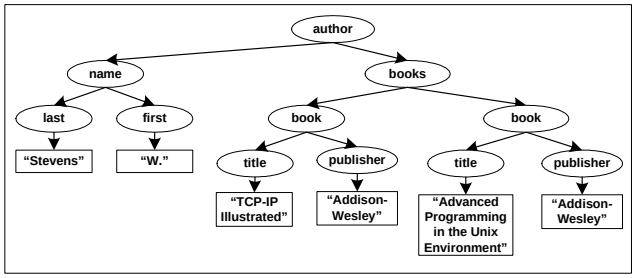


Figure 3.14: Graphical representation of the expected result of a product.

Join

Having fixed the concept of product, the definition of the join operator is quite straightforward. As in the relational world, AFTX 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; alternatively, the selection condition can also be an identity condition.

Example 3.10 Consider the query of Example 3.9. Using the derived join operator, it can be answered using the following expression:

$$\pi_{/\texttt{bib/book/author}}(\text{“}\texttt{books.xml}\text{”}) \bowtie_{[/\texttt{author}\equiv/\texttt{book/author}]} \pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”}))$$

The result of the expression does not change, and is therefore that shown in Figure 3.13.

Deletion

All the operators presented until now have some similarity with the corresponding relational operators; we now introduce a brand-new operator. The deletion operator takes in a forest and returns a new forest containing non-complete subtrees of the input trees, obtained by pruning from the original trees those subtrees that satisfy a deletion predicate.

Why do we need this operator? Informally, it completes the features of the projection and selection operators. Remember that the projection operator permits to identify one

or more elements and returns the complete subtrees rooted at those elements; if we want instead to freely delete a portion of a tree, we need the deletion operator. On the other hand, introducing the selection operator we noticed that not every subtree located by the path expression is required to satisfy the selection condition for the tree to be returned; the deletion operator enables, if needed, the deletion of those subtrees that do not respect the selection condition. Figure 3.15 shows graphically two input trees and, in grey, the subtrees retained after deletion. It should be clear that the same results can not be achieved using selection or projection.

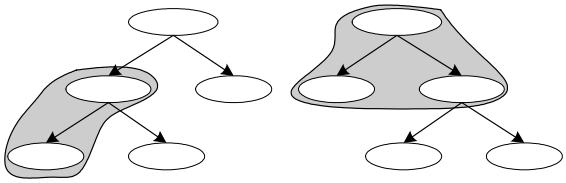


Figure 3.15: Two examples of deletion.

Even if the deletion operator is able to transform the input forest in a way not obtainable using projection and selection, it is not a substitute for those operators. In fact:

- the root element can not be deleted, because such a deletion would delete the entire tree; thus, there is no way to obtain the same result as $\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”})$ using the deletion operator;
- deletion eliminates some subtrees of the input trees, but it does not filter trees: each input tree is retained (with some modification) in the output.

Finally, it should be noticed that deletion preserves ordering, either between trees (they appear in the output forest in the same order as they appear in the input forest) or between elements of a tree (they appear in an output tree in the same order as they appear in the corresponding input tree).

Example 3.11 Consider the XML document in Figure 2.3. We want to retrieve the last name of the first author of each book. The following algebraic expression answers to the

query:

$$\pi_{/\texttt{book/author/last}}(\delta_{/\texttt{book/author}[.\texttt{pos}>1]}(\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”}))) \ .$$

It is worth noticing that the previous query can not be answered using the selection predicate; in fact the query

$$\pi_{/\texttt{book/author/last}}(\sigma_{/\texttt{book/author}[.\texttt{pos}=1]}(\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”})))$$

would return, for each book having at least one author, the last name of each author. This is because the selection $\sigma_{/\texttt{book/author}[.\texttt{pos}=1]}(\ldots)$ is interpreted as follows: “among all the books, return only those which have at least one author satisfying the condition to be the first author”; in practice, this is true for all the books having at least one author. On the other hand the query

$$\sigma_{[.\texttt{pos}=1]}(\pi_{/\texttt{bib/book/author}}(\text{“}\texttt{books.xml}\text{”})))$$

would return only the very first author found in the document. This is because the selection $\sigma_{[.\texttt{pos}=1]}(\ldots)$ is a special case of selection, in which the selection predicate does not contain a path expression; therefore the *working forest* for position check is the forest resulting from inner projection, and selection is interpreted as follows: “among all the authors, return only those who satisfy the condition to be the first author”; in practice, this is true only for the first author in the input forest.

Grouping

Although relational algebra does not have a grouping operator, we decided to insert it into AFTX. Grouping, in fact, is useful in many situations in the semi-structured world; for example, it is a convenient way to express inversion of hierarchy.

AFTX grouping operator is a very powerful construct; through the grouping predicate it is possible to specify a list of one or more element properties $\lambda_i p_i$: the value of those properties drives the process of grouping. In fact, the output will contain a distinct tree for each distinct combination of property values found in the input forest; for each of such trees, the root element (called `group_root`) will have as many attributes as the number of element properties in the grouping list, and the value of those attributes will represent

the properties value for that group. For example, suppose we want to group the books in Figure 2.3 by price and publisher. Then, the books *"TCP-IP Illustrated"* and *"Advanced Programming in the Unix Environment"* will be grouped together in a single tree; the root of the group tree will have two attributes: `price` with value `65.95` and `publisher` with value `Addison-Wesley`. Note that the name of the newly created attributes can be specified in the grouping predicate using the parameters n_i.

In the example just proposed grouping is done on the basis of the *value* property of the elements `/book/price` and `/book/editor`; however, any element property can be used in the grouping predicate, as shown in Example 3.12.

The grouping predicate also permits to establish which part of the input trees should be retained in the group trees, through a list of path expressions λ'_i. For example, using the path expression `/book/title`, only the title of each book will be retained in group trees.

Grouping preserves ordering: group trees appear in the output forest in the same order as the corresponding elements appear in the input forest.

Example 3.12 Consider the XML document in Figure 2.3. We want to retrieve all the book titles, grouped by the number of authors of the book. The following expression answers to the query:

$$\Sigma_{((\texttt{/book/authors.count},\text{“}\texttt{numAuthors}\text{”})),(\texttt{/book/title})}(\pi_{\texttt{/bib/book}}(\text{“}\texttt{books.xml}\text{”}))\ .$$

The result of the algebraic expression is shown in Figure 3.16 and graphically in Figure 3.17.

Duplicate Elimination

The problem of retrieving the different values of an element property in the input forest can be easily solved using the grouping operator previously defined. In fact, the elimination of duplicate values of an element property (or a list of element properties) corresponds to a grouping operation by that element property. Provided that the only information we want to retain is the list of values, we will specify an empty list of subtrees to attach to the group trees.

```
<group_root numAuthors="1">
  <title>TCP/IP Illustrated</title>
  <title>Advanced Programming in the Unix Environment</title>
</group_root>
<group_root numAuthors="0">
  <title>Technology and Content for Digital TV</title>
</group_root>
<group_root numAuthors="3">
  <title>Data on the Web</title>
</group_root>
```

Figure 3.16: The result of an algebraic expression involving grouping.

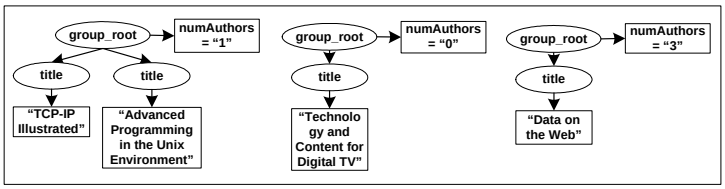


Figure 3.17: Graphical representation of the result of an algebraic expression involving grouping.

For the sake of convenience, we define a derived duplicate elimination operator; it is identical to the grouping operator, except that no list of path expressions identifying subtrees is specified. Consequently, the resulting forest will contain trees composed by the only `group_root` element; that element, as in the case of grouping, will have as many attributes as the number of element properties of interest. The number of trees in the output forest will be obviously equal to the number of combinations of element properties' different values found in the input forest.

Example 3.13 Consider the XML document in Figure 2.3. We want to retrieve the last name and first name of all the authors; each author should appear just once in the result.

The following expression answers to the query:

$$\nu_{(/\texttt{author/last.v},\text{“}\texttt{last}\text{”}),(/\texttt{author/first.v},\text{“}\texttt{first}\text{”})}(\pi_{/\texttt{bib/book/author}}(\text{“}\texttt{books.xml}\text{”}))\ .$$

The result is shown in Figure 3.18.

```
<group_root last="Stevens" first="W."/>
<group_root last="Abiteboul" fist="Serge"/>
<group_root last="Buneman" fist="Peter"/>
<group_root last="Suciu" fist="Dan"/>
```

Figure 3.18: The result of an algebraic expression involving duplicate elimination.

Example 3.14 Consider again the query of Example 3.9. Using a product and a subsequent selection (or, equivalently, a join), we obtained a result (shown in Figure 3.13 limited to the author *“W. Stevens”*) which is not in the form one probably wants. Using duplicate elimination and grouping it is possible to obtain a result in which each author appears just once and books written by an author are grouped. The following expression:

$$\Sigma_{((/\texttt{prod\_root/group\_root/.A[last].v},\text{“}\texttt{last}\text{”}),(/\texttt{prod\_root/group\_root/.A[first].v},\text{“}\texttt{first}\text{”})),/\texttt{prod\_root/book}}($$
$$\sigma_{/\texttt{prod\_root}[/\texttt{group\_root.A[last].v}=/\texttt{book/author/last.v AND /group\_root.A[first].v}=/\texttt{book/author/first.v}]}($$
$$\nu_{(/\texttt{author/last.v},\text{“}\texttt{last}\text{”}),(/\texttt{author/first.v},\text{“}\texttt{first}\text{”})}(\pi_{/\texttt{bib/book/author}}(\text{“}\texttt{books.xml}\text{”}))\times$$
$$\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”})))$$

gives the result shown graphically (again limited to the author “*W. Stevens*”) in Figure 3.19.

Let us examinate the behavior of this expression, limited to the author “*W. Stevens*”. The grouping operator searches all the possible combinations of values for `/author/last` and `/author/first` in the trees resulting from the first projection (i.e. subtrees rooted at `author`). For the author *Stevens* an output tree is built; that output tree is composed by a root element named `group_root`, with two attributes named `last` (with value “*Stevens*”) and `first` (with value “*W.*”). Such a tree is then combined

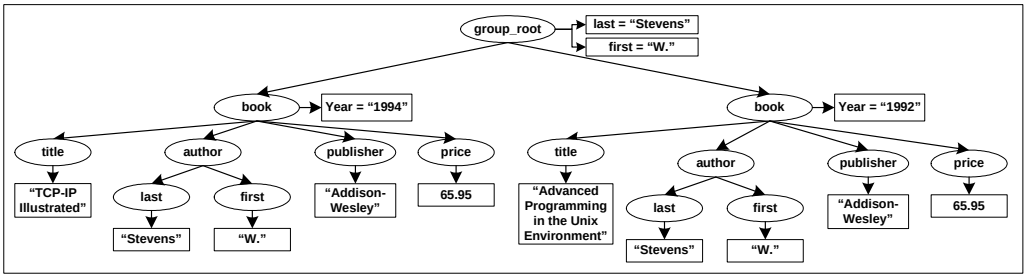


Figure 3.19: Refinement of the result of an expression using grouping.

with all the books using product; then selection discard pairs containing a book not written by Stevens. Finally, grouping groups again by first and last name; the second part of the grouping predicate indicates that each subtree rooted at `book` should be retained in the output group trees.

Ordering

We have seen that every AFTX operator preserves ordering of trees and elements. Sometimes, however, we need to change the order of trees. To this aim we introduce the ordering operator: it takes in a forest and returns a new forest containing the same trees as the original one, 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 λ), the property p whose values must be compared and the ordering direction a (ascending or descending).

Example 3.15 Consider again the query of Example 3.13. We now want the authors to be in alphabetical order. The following expression answers to the query:

$$o_{/\texttt{group\_root.A}[\text{“}\texttt{last}\text{”}]\texttt{.v ASC},/\texttt{group\_root.A}[\text{“}\texttt{first}\text{”}]\texttt{.v ASC}}\Big(\nu_{/\texttt{author/last.v},/\texttt{author/first.v}}\Big(\pi_{/\texttt{bib/book/author}}(\text{“}\texttt{books.xml}\text{”})\Big)\Big)$$

The result is shown in Figure 3.20.

```
<group_root last="Abiteboul" first="Serge"/>
<group_root last="Buneman" first="Peter"/>
<group_root last="Stevens" first="W."/>
<group_root last="Suciu" first="Dan"/>
```

Figure 3.20: The result of an algebraic expression involving ordering.

Tree Construction

The operators presented up to now permit, in various ways, to filter and modify the input trees. What still lacks is a way to build *new* trees, possibly using the data contained in the input forest; the tree construction operator accomplishes this function.

The tree construction predicate enables to specify name and value of elements to build, name and value of their attributes and the hierarchy of elements. It is in fact a list $(e_1, \ldots, e_n)$ of *element construction* specification, where each element construction specification is formed by:

- the name n of the element;
- the value v (possibly null) of the element;
- the list A (possibly empty) of attributes, where each attribute is, as one could expect, a pair (name, value);
- the list $(e'_1, \ldots, e'_m)$ (possibly empty) of child elements, each of which is an element constructor specification itself.

For example, if we want to create a `book` element with an attribute `publishingYear` whose value is `"1994"` and a child `title` element whose value is `"TCP-IP Illustrated"`, we do it using the tree construction predicate `"book"(null, (("publishingYear", "1994")), ("title"("TCP-IP Illustrated", null, null)))`.

In this example the tree construction predicate contains all the data needed to build the output tree. In most cases, however, some of these data must be picked from the input

forest; to this aim, the tree construction predicate can contain some *reference* to the input forest. Such references are path expressions identifying some elements of the input forest, possibly followed by the name of an element property. For example, suppose we want to retrieve the year of publication and the title of the book from the input forest; in this case we use the tree construction predicate `"book"(null, (("publishingYear", /book.A[year].v)), (/book/title))`.

Here the reference `/book.A[year].v` states that the value of the attribute `publishingYear` should be set to the value of the attribute `year` in the input tree; the reference `/book/title` states the newly created `book` element should have as children every `/book/title` element of the input tree. Note that, in this case, an element construction specification (the one that builds the child elements of root `book` element) is not of the form $n(v, A, (e_1, \ldots, e_n))$; it is instead of the form λ. This case is also possible for the root element construction specification, i.e. it is perfectly legal to write an expression like $\iota_{/\texttt{book}/\texttt{author}}(A)$.

Usually the input forest contains more than a single tree. The tree construction operator is applied separately to each input tree, in the order they appear in the input forest. For example, if the input forest in the previous example would contain five trees, the output forest would contain five trees rooted at `book`.

Typically one output tree is built for each input tree. There are however cases in which no output tree is built for an input tree or more than one output tree is built for an input tree.

Suppose to use the construction predicate `/bib/book/title`. If an input tree does not contain such a path, no output tree corresponds to that input tree. On the other side, if four `/bib/book/title` elements are found in an input tree, four output trees correspond to that input tree. It should be clear, however, that this is a degenerate example: the same result, in fact, could be obtained simply using the projection operator instead of the tree construction operator.

There is another case in which an input tree can generate more than one output tree: it can occur when the tree construction predicate states to build a root element whose value must be picked from the input forest. For example, consider to project the XML

document in Figure 2.3 using the projection predicate `/bib/book`, then to build the output using the tree construction predicate `"lastname"(/book/author/last.v, null, null)`; if a book has more than one author, multiple values of `/book/author/last` are found. In this case multiple `lastname` elements are built, one for each last name found.

The tree construction predicate can also contain more than one outer element construction specification. For example the predicate `"books"(...), "authors"(...)` means that, for each input tree, two kinds of output trees must be built: one rooted at a `books` element, the other rooted at an `authors` element.

Finally, if the tree construction specification 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 the output forest to be composed by a single tree, as shown in the following example.

Example 3.16 Consider the XML document in Figure 2.3. We want to retrieve the first and last name of each author and return them as sub-elements of an element named `name`, which in turn should be sub-element of an element named `author`. The following expression answers to the query:

$$\iota_{\text{“author”(null,null,“name”(null,null,(/author/first,/author/last)))}}\Big(\pi_{\text{/bib/book/author}}(\text{“books.xml”})\Big)$$

Here the input forest (i.e. the result of the projection operation) is composed by trees rooted at `author`, as shown in Figure 3.21(a). An output tree is created for each input tree, i.e. for each author. Suppose now we want the same result, but with an `authors` element containing all the `author` elements. If we indicate with A the previous algebraic expression, the following expression answers to the modified query:

$$\iota_{\text{“authors”(null,null,null)}}(A) \ .$$

In this case the construction predicate does not contain any reference to the input forest; consequently the entire input forest is inserted in the output tree as child of the root

`authors` element, as shown in Figure 3.21(b). It is important to notice that it is impossible to obtain this result without nesting a construction predicate inside another; in fact the expression

$$\iota_{\text{“}\texttt{authors}\text{”}(\texttt{null},\texttt{null},\text{“}\texttt{author}\text{”}(\texttt{null},\texttt{null},\text{“}\texttt{name}\text{”}(\texttt{null},\texttt{null},(/\texttt{author}/\texttt{first},/\texttt{author}/\texttt{last}))))}\Big($$
$$\pi_{/\texttt{bib}/\texttt{book}/\texttt{author}}\big(\text{“}\texttt{books.xml}\text{”}\big)\Big)$$

would return a forest containing as many trees (with a root element named `authors`) as the number of trees in the projection result, i.e. the number of `author` elements in the input XML document.

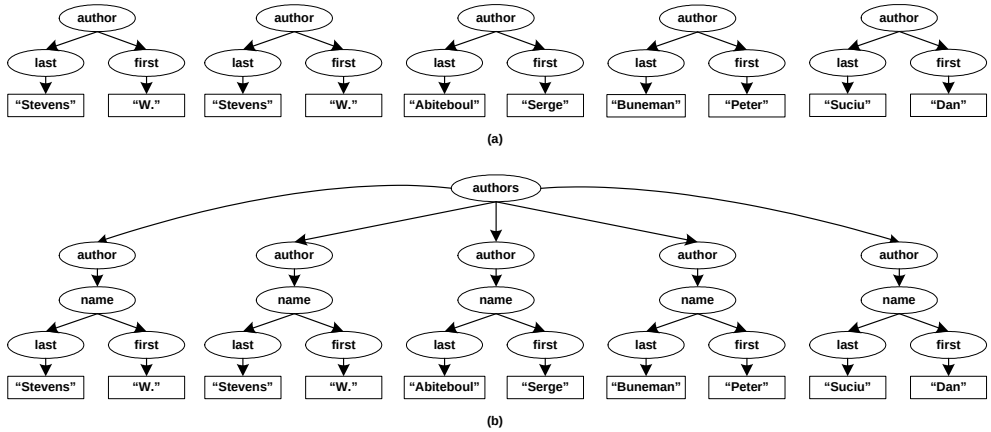


Figure 3.21: The input forest for the tree construction operator of Example 3.16 (a) and the result of the tree construction operation (b).

Example 3.17 Consider again the query of Example 3.14. Having defined the tree construction operator, we are ready to write an expression whose result is that shown (limited to the author *Stevens*) in Figure 3.14:

$$\iota_{\text{“}\texttt{author}\text{”}(\texttt{null},\texttt{null},(\text{“}\texttt{name}\text{”}(\texttt{P}_1),\text{“}\texttt{books}\text{”}(\texttt{P}_2)))}(F)$$

where

$$P_1 = \texttt{null},\texttt{null},(\text{“}\texttt{last}\text{”}(/\texttt{group\_root.A}[\texttt{last}].\texttt{v},\texttt{null},\texttt{null}),$$
$$\text{“}\texttt{first}\text{”}(/\texttt{group\_root.A}[\texttt{first}].\texttt{v},\texttt{null},\texttt{null}))\ ,$$
$$P_2 = \texttt{null},\texttt{null},(\text{“}\texttt{book}\text{”}(\texttt{null},\texttt{null},$$
$$(/\texttt{group\_root}/\texttt{book}/\texttt{title},/\texttt{group\_root}/\texttt{book}/\texttt{publisher})))\ ,$$

and F is the algebraic expression of Example 3.14.

Full-Text Selection

Up to now, we have presented the *basic* operators of our algebra. Now we present the full-text operators, starting with the full-text selection operator.

The full-text selection operator behaves in a way similar to that of basic selection operator previously presented: it performs a horizontal decomposition of the input forest, retaining only those trees having at least one subtree satisfying the full-text selection predicate. Full-text selection operates according to a *boolean* model. This means that a binary judgement (relevant / non-relevant) is made on every tree in the input forest: relevant trees are retained, not relevant ones are discarded.

The full-text selection predicate allows to search one or more words or phrases (specified by the parameter γ, which is a list of words or phrases connected with boolean operators) into the full-text value of an element (reachable from the root element by following the path λ) or into the value of an attribute a. Moreover, it supports *proximity search*, i.e. searching two or more words with a distance between one and another not greater than a threshold x. Finally, using the parameters `stem`, `thes`, and `stop`, the user can instruct the system to use stemming, thesaurus, and stopwords.

Having a behavior similar to the selection operator, even full-text selection preserves ordering. Moreover, it enjoys the same algebraic properties as selection; we will see this in more details in Chapter 5.

Example 3.18 Consider the XML document in Figure 2.3. We want to retrieve all the books with a title containing the words *Web* and *Data* at a distance not greater than 3. The following expression answers to the query:

$$\varsigma_{\texttt{/book/title}[\text{“}\texttt{Web}\text{”}\ \texttt{AND}\ \text{“}\texttt{Data}\text{”},\ \texttt{3}]}\big(\pi_{\texttt{/bib/book}}(\text{“}\texttt{books.xml}\text{”})\big)\ .$$

The previous query returns the book “Data on the Web”; in fact its title contains the two searched words and $pos(\text{“}Web\text{”}) - pos(\text{“}Data\text{”}) = 3$. The same result would be obtained using the following expression:

$$\varsigma_{\texttt{/book}[\text{“}\texttt{Web}\text{”}\ \texttt{AND}\ \text{“}\texttt{Data}\text{”},\ \texttt{3}]}\big(\pi_{\texttt{/bib/book}}(\text{“}\texttt{books.xml}\text{”})\big)\ .$$

In fact, even if no `book` element contains the searched words, the third `book` element has a child `title` element that contains such words; therefore the full-text value of the third `book` element contains the words. Anyway the two expressions are clearly not equivalent, even if in this special case they yield the same result.

Full-Text Score Assignment

The full-text score selection performs a full-text search using the *boolean* model: a tree satisfies the selection condition or it does not satisfy the condition at all. If we want to perform *ranked* retrieval over our forest we must use the full-text score assignment operator.

This operator 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 score value is represented by the element property *score*, which is set for the root element.

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 (within the parameter γ) in order to specify which words (or phrases) should highly influence score calculation. The weight values must be in the range $[0, 1]$, and their sum must be equal to 1; if no weight is specified, the system should consider each word as equally important. For example, if there are four searched words and no weight is explicitly specified, each word should have a weight of 0.25.

How is the score calculated? The score assignment predicate provides an extra parameter f, which can be thought of a *function pointer*, i.e. a pointer to the function that is in charge of score calculation; if the parameter f is not present in the score assignment predicate, a default (implementation dependent) score function should be used. The availability of such parameter lets the user freely decide which technique to use among those provided by its XML database system. It should be noted that the choice of defining a parameterized operator provides a higher flexibility than that present in the W3C Working Draft for XQuery Full-Text [Con06f], which just states that score values are in the range

[0, 1] and a higher score must imply a higher degree of relevance, without any indication about the techniques to use in the process of score calculation.

Example 3.19 Consider the XML document in Figure 2.3. Suppose we are looking for a book about web programming written by Stevens. This informational need is *translated* into the task of assigning a score to each book on the basis of the containment, somewhere in the book description, of the words *"Web"*, *"Programming"* and *"Stevens"*; the word *"Stevens"* must have a weight of 0.4, while the words *"Web"* and *"Programming"* must have a weight of 0.3. The following expression answers to the query:

$$\xi_{/\texttt{book}[0.4\text{“}\texttt{Stevens}\text{”}\ \texttt{AND}\ 0.3\text{“}\texttt{Web}\text{”}\ \texttt{AND}\ 0.3\text{“}\texttt{Programming}\text{”}]\texttt{f}}(\pi_{/\texttt{bib}/\texttt{book}}(\text{“}\texttt{books.xml}\text{”}))\ .$$

In this example we left undefined the function f used for score calculation. As previosly said, it should be chosen by the user among those provided by the system. For example, a simple scoring function could be

$$root(T).score = \sum_{i,j} tf_{t_i,e_j} * w_i$$

where:

- T is the tree whose score we want to calculate;
- tf_{t_i,e_j} is the *term frequency* of the word (or phrase) t_i (included in the query) relative to the full-text value of the element e_j, which is the root of a subtree reachable from $root(T)$ by following the path λ;
- w_i is the weight assigned in the query to the word (or phrase) t_i.

Using this scoring function, the score value of the four *book* elements would be (element names are those used in Example 3.2):

$$e_2.score = 0.167 * 0.4 + 0 + 0 = 0.067$$
$$e_9.score = 0.1 * 0.4 + 0 + 0.1 * 0.3 = 0.07$$
$$e_{16}.score = 0 + 0.071 * 0.3 + 0 = 0.021$$
$$e_{29}.score = 0 + 0 + 0 = 0$$

Full-Text Selection with Score

We have seen that full-text selection and score assignment absolve two different needs: the first is used to select those trees that satisfy a full-text selection condition, the second is used to assign to each tree a score value. We may want to combine those two features; informally speaking, we would like to select those trees that satisfy the condition, and distinguish among them those that "better" satisfy the condition.

For example, suppose two documents contain the searched word s; the first one contains one occurrence of s, while the second one contains ten occurrences of s. Both the documents satisfy the selection condition, but the second document is more likely to be relevant.

The derived full-text selection with score operator behaves in the following way: first, a full-text selection is done, thus removing those trees that do not satisfy the selection condition; then, a score value is assigned to each retained tree.

Example 3.20 Consider again the informational need of Example 3.19. We now want to express the fact that the book description must contain the word *"Stevens"* and at least one of the words *"Programming"* and *"Web"*; a score should be assigned to each book and the three searched words should have the same weights as in the previous example. The following expression answers to the query:

$$\overline{\varsigma}_{\texttt{/book[0.4"Stevens" AND (0.3"Programming" OR 0.3"Web")]}}\left(\pi_{\texttt{/bib/book}}(\texttt{"books.xml"})\right) \ .$$

The score assigned to each book will not change; however, only the book *"Advanced Programming in the Unix Environment"* will be returned, because the other three books will be filtered out by the full-text selection condition.

Top-K and Threshold Full-Text Selection

Until now, we have presented two full-text operators dealing with score: score assignment and selection with score. Both calculate a score, but they do not use in any way such a score.

Typical full-text searches, instead, use scores in order to filter and order input trees. There are two classical operations we want to deal with:

- *"find the k most relevant results and return them in score order"*;
- *"find all the results whose relevance is higher than a defined threshold and return them in score order"*.

.

In order to answer similar queries we define two ad-hoc derived operators: top-K full-text selection and threshold full-text selection. An explicit definition of such operators can be very useful for optimization purposes; in fact, specialized algorithms can be developed, e.g. in order to limit the number of resulting trees interested by the expensive ordering operation.

The top-K full-text selection operator takes in a forest, assigns each input tree a score and returns a subset of the input forest, containing the k trees with higher score, ordered by score value. Top-K selection combines a score assignment operation with a subsequent ordering and a final selection of the k best results; the predicate is a score predicate, augmented with a k stating the number or trees to return.

Example 3.21 Consider again the query of Example 3.19. We now want to retrieve only the 2 most relevant books. The following expression answers to the query:

$$\top_{\texttt{/book[0.4"Stevens" AND 0.3"Programming" AND 0.3"Web"]f,2}}(\pi_{\texttt{/bib/book}}(\text{“}\texttt{books.xml}\text{”})) \ .$$

The threshold full-text selection operator takes in a forest, assigns each input tree a score and returns a subset of the input forest, containing the trees with a score not less than a specified threshold τ, ordered by score value. Threshold selection combines a score assignment operation with a subsequent selection of the most relevant trees and a final ordering; the predicate is a score predicate, augmented with a τ stating the threshold score under which trees should be discarded.

Example 3.22 Consider again the query of Example 3.19. We now want to retrieve only the books with a score higher than 0.05. The following expression answers to the query:

$$\omega_{\texttt{/book[0.4"Stevens" AND 0.3"Programming" AND 0.3"Web"]f,0.05}}(\pi_{\texttt{/bib/book}}(\text{“}\texttt{books.xml}\text{”})) \ .$$

3.3.2 Formal Definitions

We now give the formal definitions for the operators informally presented in Section 3.3.1. When necessary, we also give a conversational explaination of the definitions.

Set Operators

Our algebra is equipped with two set operators: union, which returns all the trees from the first input forest followed by the trees from the second input forest, and difference, which returns each tree from the first input forest that is not present in the second input forest.

Definition 3.15 (Union) *Given two forests $F = (T_1, T_2, ..., T_n)$ and $F' = (T'_1, T'_2, ..., T'_m)$, the union operator $F \cup F'$ returns the forest $H = (T_1, T_2, ..., T_n, T'_1, T'_2, ..., T'_m)$.*

Definition 3.16 (Difference) *Given two forests $F = (T_1, T_2, ..., T_n)$ and $F' = (T'_1, T'_2, ..., T'_m)$, the difference operator $F - F'$ returns a forest $G \subset F$ such that, $\forall T_i \in F$, if $T_i \notin G$ then $\exists T' \in F'$ such that $T' \equiv T$.*

Projection

The projection operator returns all the subtrees of the input trees that can be reached following a path expression. We first define the notion of path expression, which is used by many other operators of our algebra, then the projection predicate and the projection operator.

Definition 3.17 (Path expression) *A path expression λ is an expression of the form*

$$\alpha_1\beta_1\alpha_2\beta_2 \ldots \alpha_m\beta_m$$

where:

- *α_i is either "`/`" or "`//`";*
- *β_i is either a string or an integer or the special string "`*`".*

Let T be a tree and $\lambda = \alpha_1\beta_1$ a path expression. A complete subtree $T' \subset^* T$ is reachable from $root(T)$ by following the path λ if one of the following conditions holds:

- *α_1 is "/" and one of the following conditions holds:*
  - *β_1 is a string and $root(T).n = \beta_1$;*
  - *β_1 is the integer 1;*
  - *β_1 is the special string "*".*

  In this case T' corresponds to T.
- *α_1 is "//" and one of the following conditions holds:*
  - *β_1 is a string and $root(T'_1).n = \beta_1$;*
  - *β_1 is the integer i and $root(T')$ is the i-th element (in pre-order enumeration) of the tree T;*
  - *β_1 is the special string "*".*

Let T be a tree, $\lambda = \alpha_1\beta_1\alpha_2\beta_2 \ldots \alpha_{m-1}\beta_{m-1}$ a path expression and $T' \subset^* T$ reachable from $root(T)$ by following the path λ. A subtree $T'' \subset^* T'$ is reachable from $root(T)$ by following the path $\lambda\alpha_m\beta_m$ if one of the following conditions holds:

- *α_m is "/", $root(T'').p = root(T')$, and one of the following conditions holds:*
  - *β_m is a string and $root(T'').n = \beta_m$;*
  - *β_m is the integer i and $root(T'').o = i$;*
  - *β_m is the special string "*".*
- *α_m is "//" and one of the following conditions holds:*
  - *β_m is a string and $root(T'').n = \beta_m$;*
  - *β_m is the integer i and $root(T'')$ is the i-th element (in pre-order enumeration) of the tree T';*
  - *β_m is the special string "*".*

Definition 3.18 (Projection Predicate) *A projection predicate P is a path expression λ. A subtree $T' \subset^* T$ satisfies the projection predicate P if it can be reached from $root(T)$ by following the path λ.*

Definition 3.19 (Projection) *Given a forest $F = (T_1, T_2, ..., T_n)$ and a projection predicate P, the projection operator $\pi_P(F)$ returns a forest $G = G_1 \cup G_2 \cup \ldots \cup G_n$ such that:*

- $\forall G_i = (T_i^1, T_i^2, \ldots, T_i^m), T_i^k \subset^* T_i, \forall k$;
- $\forall T_i^k \in G_i$, T_i^k *satisfies the projection predicate* P.

Note that projection preserves ordering between trees; in fact each subtree of T_1 satisfying the projection predicate will be in G_1, and consequently will precede in the output forest each subtree of T_2.

Selection

The selection operator returns each tree in the input forest that satisfies the selection predicate. The selection predicate can check the value of different element properties and can use different comparison operators. The element properties that can be checked are those presented in Section 3.2.2. In what follows we define the available comparison operators; then we define the selection predicate and the selection operator.

Definition 3.20 (Comparison Operators) *Given two trees T_1 and T_2, two kinds of comparison operators between $root(T_1)$ and $root(T_2)$ are defined:*

- *value comparison: given two element properties p_1 and p_2, the usual comparison operators $=, <, >, \neq$ etc. are defined between $root(T_1)p_1$ and $root(T_2)p_2$;*
- *strict equality comparison: $root(T_1) \equiv root(T_2)$ is true if $T_1 \equiv T_2$.*

Definition 3.21 (Selection Predicate) *A selection predicate P is an expression of the form $\lambda[\gamma]$, where:*

- *λ is an optional path expression;*
- *γ is a list of zero or more base conditions $\gamma_1, \gamma_2, \ldots, \gamma_n$ connected with boolean operators (*AND, OR, NOT*); each base condition γ_i is of one of following forms:*
  - *$\lambda' \equiv \lambda''$, where λ' is an optional path expression and λ'' is a path expression;*
  - *$\lambda' p' \theta x$, where:*
    - *λ' is an optional path expression;*
    - *p' is an element property;*
    - *θ is a value comparison operator;*
    - *x is a constant or is of the form $\lambda'' p''$, where λ'' is a path expression and p'' is an element property.*

Let G be the forest of subtrees of a tree T that can be reached from $root(T)$ by following the path λ, i.e. $G = \pi_\lambda(T)$ (if λ is omitted, then $G =$T). The tree T satisfies the selection predicate P if $\exists T_1 \in G$ such that $root(T_1)$ satisfies the boolean expression γ. The evaluation of each base condition γ_i is done as follows:

- *if γ_i is of the form $\lambda' \equiv \lambda''$, γ_i is satisfied if $\exists T' \in G' = \pi_{\lambda'}(T_1)$ and $T'' \in G'' = \pi_{\lambda''}(T_1)$ such that $root(T') \equiv root(T'')$*
- *if γ_i is of the form $\lambda' p' \theta x$ and x is a constant, γ_i is satisfied if $\exists T' \in G' = \pi_{\lambda'}(T_1)$ such that $root(T') p' \theta x$;*
- *if γ_i is of the form $\lambda' p' \theta x$ and if x is of the form $\lambda'' p''$, γ_i is satisfied if $\exists T' \in G' = \pi_{\lambda'}(T_1), T'' \in \pi_{\lambda''}(T_1)$ such that $root(T') p' \theta root(T'') p''$.*

Definition 3.22 (Selection) *Given a forest $F = (T_1, T_2, ..., T_n)$ and a selection predicate P, the selection operator $\sigma_P(F)$ returns a subforest $G \subset F$ such that each tree in G satisfies the selection predicate P.*

Product and Join

The product operator combines each tree from the first input forest with each tree from the second input forest. The join operator is derived from product and selection.

Definition 3.23 (Product) *Given two forests $F = (T_1, T_2, \ldots, T_n)$ and $F' = (T'_1, T'_2, \ldots, T'_m)$, the product operator $F \times F'$ returns a forest $F'' = (T''_{11}, T''_{12}, \ldots, T''_{1m}, T''_{21}, T''_{22}, \ldots, T''_{2m}, \ldots, T''_{n1}, T''_{n2}, \ldots, T''_{nm})$ such that, for each i, j, T''_{ij} is a tree built as follows:*

- $root(T''_{ij}) = (\texttt{null}, \texttt{prod\_root}, \texttt{null}, \texttt{null}, \texttt{null})$*;*
- $root(T''_{ij})$ *has two children;*
- *let* $L''_{ij} = \pi_{\texttt{prod\_root}/1}(T''_{ij})$ *be the left subtree of* $root(T''_{ij})$*; then* $L''_{ij} \equiv T_i$*;*
- *let* $R''_{ij} = \pi_{\texttt{prod\_root}/2}(T''_{ij})$ *be the right subtree of* $root(T''_{ij})$*; then* $R''_{ij} \equiv T'_j$*;*

The formal definition of join predicate is quite similar to that of selection predicate; the definition of the operator clarifies its derived nature.

Definition 3.24 (Join) *Given two forests $F = (T_1, T_2, \ldots, T_n)$ and $F' = (T'_1, T'_2, \ldots, T'_m)$ and a join predicate $P = \lambda[\gamma]$, the join operator $F \bowtie_P F'$ returns a forest $F'' = (T''_1, T''_2, \ldots, T''_k)$ such that, for each T''_i:*

- $T''_i \in F \times F'$*;*
- T''_i *satisfies the selection predicate* $P' = /\texttt{prod\_root}\lambda[\gamma]$.

Join is a derived operator; in fact the following equation holds:

$$F \bowtie_P G = \sigma_{P'}(F \times G) \ .$$

Deletion

The deletion operator purges from each input tree those subtrees that satisfy the deletion predicate.

Definition 3.25 (Deletion Predicate) *A selection predicate P is an expression of the form $\lambda[\gamma]$, where:*

- *λ is an optional path expression;*
- *γ is a list of zero or more base conditions $\gamma_1, \gamma_2, \ldots, \gamma_n$ connected with boolean operators (*AND, OR, NOT*); each base condition γ_i is of one of following forms:*
  - *$\lambda' \equiv \lambda''$, where λ' is an optional path expression and λ'' is a path expression;*
  - *$\lambda' p' \theta x$, where:*
    - *λ' is an optional path expression;*
    - *p' is an element property;*
    - *θ is a value comparison operator;*
    - *x is a constant or is of the form $\lambda'' p''$, where λ'' is a path expression and p'' is an element property.*

Let G be the forest of subtrees of a tree T that can be reached from $root(T)$ by following the path λ, i.e. $G = \pi_\lambda(T)$ (if λ is omitted, then $G = (T)$. The tree T satisfies the deletion predicate P if $\exists T_1 \in G$ such that $root(T_1)$ satisfies the boolean expression γ. The evaluation of each base condition γ_i is done as follows:

- *if γ_i is of the form $\lambda' \equiv \lambda''$, γ_i is satisfied if $\exists T' \in G' = \pi_{\lambda'}(T_1)$ and $T'' \in G'' = \pi_{\lambda''}(T)$ such that $root(T') \equiv root(T'')$*
- *if γ_i is of the form $\lambda' p' \theta x$ and x is a constant, γ_i is satisfied if $\exists T' \in G' = \pi_{\lambda'}(T_1)$ such that $root(T') p' \theta x$;*
- *if γ_i is of the form $\lambda' p' \theta x$ and if x is of the form $\lambda'' p''$, γ_i is satisfied if $\exists T' \in G' = \pi_{\lambda'}(T_1), T'' \in \pi_{\lambda''}(T)$ such that $root(T') p' \theta root(T'') p''$.*

The deletion predicate is almost equal to a selection predicate; however a slight but important difference arise. When the second part of a base condition is not a constant (i.e. it is of form λ or λp), the path expression must be evaluated considering as base forest the input forest, instead of the forest resulting from the projection caused by the first path expression of the predicate.

For example, consider the deletion predicate `prod_root/book/author`[.v $\neq$ `/prod_root/author`.v]. For each input tree T, the deletion operator operates as follows:

- calculate $F = \pi_{\texttt{/prod\_root/book/author}}(T)$
- calculate $G = \pi_{\texttt{/prod\_root/author}}(T)$
- for each tree $T' \in F$:
  - if $\exists T'' \in G$ such that $root(T').v \neq root(T'').v$, remove T' from T.

Definition 3.26 (Deletion) *Given a forest $F = (T_1, T_2, \ldots, T_n)$ and a deletion predicate $P = \lambda[\gamma]$, the deletion operator $\delta_P(F)$ returns a forest $F' = (T'_1, T'_2, \ldots, T'_n)$ such that:*

- $\forall i, T'_i \subset T_i$*;*
- $\forall i$*, if $T_i^k \subset T_i$ is reachable from $root(T_i)$ by following the path λ (i.e. $T_i^k \in \pi_\lambda(T_i)$) and T_i^k satisfies the selection condition γ, then T_i^k is not present in T'_i.*

Grouping and Duplicate Elimination

The grouping operator creates an output tree for each possible combination of some properties values found in the input trees. Each output tree will have a `group_root` root element, with as many attributes as the number of properties involved in grouping.

Definition 3.27 (Grouping predicate) *A grouping predicate P is of the form*

$$((\lambda_1 p_1, n_1), (\lambda_2 p_2, n_2), \ldots, (\lambda_n p_n, n_n)), (\lambda'_1, \lambda'_2, \ldots, \lambda'_m)$$

where:

- λ_i *and* λ'_i *are path expressions;*
- p_i *is an element property;*
- n_i *is a string.*

Definition 3.28 (Grouping) *Given a forest* F *and a grouping predicate*

$$P = ((\lambda_1 p_1, n_1), (\lambda_2 p_2, n_2), \ldots, (\lambda_n p_n, n_n)), (\lambda'_1, \lambda'_2, \ldots, \lambda'_m)$$

the grouping operator $\Sigma_P(F)$ *returns a forest* F' *such that:*

- $\forall T' \in F', root(T') = (\texttt{null}, \text{“}\texttt{group\_root}\text{”}, \texttt{A}, \texttt{null}, \texttt{null})$*;*
- $root(T').A = (a_1, a_2, \ldots, a_n)$*, where* $a_i.n = n_i$*;*
- $\forall T' \in F', \exists T \in F$ *such that* $\exists T'' \in F_T = \underbrace{(\ldots((}_{(n-2) \text{ times}} \pi_{\lambda_1}(T) \times \pi_{\lambda_2}(T)) \times \pi_{\lambda_3}(T)) \times \ldots) \times \pi_{\lambda_n}(T)$ *such that:*
  - $root(T').A[n_1].v = e_1 p_1$*, where*

    $$e_1 = root(\pi_{\underbrace{\texttt{prod\_root/prod\_root/}\ldots\texttt{/prod\_root}}_{(n-1) \text{ times}}/1}(T'')) \ ;$$

  - $root(T').A[n_2].v = e_2 p_2$*, where*

    $$e_2 = root(\pi_{\underbrace{\texttt{prod\_root/prod\_root/}\ldots\texttt{/prod\_root}}_{(n-1) \text{ times}}/2}(T'')) \ ;$$

  - $root(T').A[n_3].v = e_3 p_3$*, where*

    $$e_3 = root(\pi_{\underbrace{\texttt{prod\_root/prod\_root/}\ldots\texttt{/prod\_root}}_{(n-2) \text{ times}}/2}(T'')) \ ;$$

  - $\ldots$*;*
  - $root(T').A[n_n].v = e_n p_n$*, where*

    $$e_n = root(\pi_{\texttt{prod\_root}/2}(T'')) \ ;$$

- $\pi_{\texttt{group\_root}/*}(T') = \pi_{\lambda'_1}(T) \cup \pi_{\lambda'_2}(T) \cup \ldots \cup \pi_{\lambda'_m}(T)$;

- $\forall T \in F$, *let* $F_T = \underbrace{(\ldots((}_{(n-2) \text{ times}} \pi_{\lambda_1}(T) \times \pi_{\lambda_2}(T)) \times \pi_{\lambda_3}(T)) \times \ldots) \times \pi_{\lambda_n}(T)$; $\forall T'' \in F_T, \exists T' \in F'$ *such that:*

  - $root(T').A[n_1].v = e_1 p_1$, *where*

    $$e_1 = root(\pi_{\underbrace{\texttt{prod\_root}/\texttt{prod\_root}/\ldots/\texttt{prod\_root}}_{(n-1) \text{ times}}/1}(T'')) \ ;$$

  - $root(T').A[n_2].v = e_2 p_2$, *where*

    $$e_2 = root(\pi_{\underbrace{\texttt{prod\_root}/\texttt{prod\_root}/\ldots/\texttt{prod\_root}}_{(n-1) \text{ times}}/2}(T'')) \ ;$$

  - $root(T').A[n_3].v = e_3 p_3$, *where*

    $$e_3 = root(\pi_{\underbrace{\texttt{prod\_root}/\texttt{prod\_root}/\ldots/\texttt{prod\_root}}_{(n-2) \text{ times}}/2}(T'')) \ ;$$

  - $\ldots$;

  - $root(T').A[n_n].v = e_n p_n$, *where*

    $$e_n = root(\pi_{\texttt{prod\_root}/2}(T'')) \ ;$$

  - $\pi_{\texttt{group\_root}/*}(T') = \pi_{\lambda'_1}(T) \cup \pi_{\lambda'_2}(T) \cup \ldots \cup \pi_{\lambda'_m}(T)$;

- $\forall T'_1, T'_2 \in F', root(T'_1) \not\equiv root(T'_2)$.

This definition is quite complex and deserves an in-depth analysis. The first two conditions explain how the root element of the trees resulting from grouping must be named and which attributes they must have. The third condition says that the value of each `group_root` element's attribute comes from the value of some properties of the input trees; the product between multiple projection over a tree is needed in order to find each possible combination of properties values. Moreover, the third condition explains that some subtrees of the input trees are retained in the output as children of the `group_root`

element. The fourth condition is just the opposite of the third one, thus stating that each possible combination of properties values is found in some output tree. Finally, the fifth condition says that each possible combination of properties values is found just once in the output trees.

The duplicate elimination operator derives from the grouping operation; in practice, eliminating duplicate values of some element properties means grouping by that properties without returning any subtree of the input trees.

Definition 3.29 (Duplicate elimination predicate) *A duplicate elimination predicate P is of the form $(\lambda_1 p_1, n_1), (\lambda_2 p_2, n_2), \ldots, (\lambda_n p_n, n_n)$, where:*

- *λ_i is a path expression;*
- *p_i is an element property;*
- *n_i is a string.*

Definition 3.30 (Duplicate elimination) *Given a forest F and a duplicate elimination predicate $P = (\lambda_1 p_1, n_1), (\lambda_2 p_2, n_2), \ldots, (\lambda_n p_n, n_n)$, the derived duplicate elimination operator $\nu_P(F)$ returns the forest $F' = \Sigma_{(P),\texttt{null}}(F)$.*

Ordering

Formal definitions of ordering predicate and ordering operator are quite easy to understand and are given in what follows.

Definition 3.31 (Ordering predicate) *An ordering predicate $P = P_1, P_2, \ldots, P_n$ is of the form $\lambda_1 p_1\ a_1, \lambda_2 p_2\ a_2, \ldots, \lambda_n p_n\ a_n$, where:*

- *λ_i is a path expression;*
- *p_i is an element property;*
- *a_i is either* ASC *or* DESC.

Definition 3.32 (Ordering) *Given a forest F and an ordering predicate*

$$P = \lambda_1 p_1, \lambda_2 p_2, \ldots, \lambda_n p_n \ ,$$

the ordering operator $o_P(F)$ returns a forest F' such that:

- *$\forall T' \in F'$, $\exists T \in F$ such that $T' \equiv T$;*
- *$\forall T \in F$, $\exists T' \in F'$ such that $T \equiv T'$;*
- *$\forall T'_1, T'_2 \in F'$ such that T'_1 precedes T'_2, $\exists k, 1 \leq k \leq n$ such that:*
  - *$\forall j < k$, if $T''_1 \subset T'_1$ is reachable from $root(T'_1)$ by following the path λ_j and $T''_2 \subset T'_2$ is reachable from $root(T'_2)$ by following the path λ_j, then $root(T''_1)p_j = root(T''_2)p_j$;*
  - *if $T''_1 \subset T'_1$ is reachable from $root(T'_1)$ by following the path λ_k and $T''_2 \subset T'_2$ is reachable from $root(T'_2)$ by following the path λ_k, then:*
    - *if a_k is* ASC*, then $root(T''_1)p_k < root(T''_2)p_k$;*
    - *if a_k is* DESC*, then $root(T''_1)p_k > root(T''_2)p_k$.*

Tree Construction

The tree construction operator is used for building new elements, possibly using parts of the input trees. We give the formal definition of predicate and operator. Then, in order to clarify the definition, we present an algorithm for tree construction.

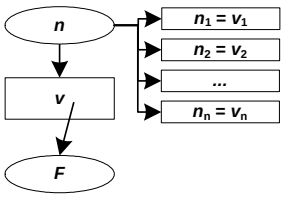
Definition 3.33 (Tree Construction Predicate) *A tree construction predicate P is of the form $e_1, e_2, \ldots, e_n$, where each e_i (named* element construction specification*) can be:*

- *a path expression;*
- *an expression of the form $n(v, A, (e'_1, e'_2, \ldots, e'_m))$, where:*
  - *n is a string;*
  - *v is either a string or a path expression followed by an element property;*

- *A is either a path expression followed by the property $.A$ or an expression of the form $A = ((n_1, v_1), (n_2, v_2), \ldots, (n_n, v_n))$, where each n_i is a string and each v_i is either a string or a path expression followed by an element property;*
- *$(e'_1, e'_2, \ldots, e'_m)$ is a list of element construction specifications.*

Definition 3.34 (Tree Construction) *Given a forest $F = (T_1, T_2, \ldots, T_n)$ and a tree construction predicate $P = e_1, e_2, \ldots, e_n$, the tree construction operator $\iota_P(F)$ returns a forest $G = (G_1, G_2, \ldots, G_n)$, where each G_i contains trees built according to the tree construction specification e_i as follows:*

- *if e_i is an expression of the form $n(v, ((n_1, v_1), (n_2, v_2), \ldots, (n_m, v_m)), null)$ such that n, v, each n_i and each v_i are not path expressions, a single tree is built as follows:*



- *if e_i contains some path expression, a forest G_k containing one or more trees is built for each input tree T_k as follows:*
  - *if e_i is a path expression λ, $G_k = \pi_\lambda(T_k)$;*
  - *if e_i is an expression of the form $n(v, A, (e'_1, e'_2, \ldots, e'_m))$:*
    - *an element e^o named n is built;*
    - *its value is set as follows:*
      - *if v is a string, $e^o_i.v$ is set to v;*
      - *if v is an expression of the form λp, as many copies of $e^o.v$ as the number of trees $T' \in \pi_\lambda(T_k)$ are created, and each copy is assigned the value $(root)(T')p$;*

- *for each e^o, its attribute list is set as follows:*
  - *if A is of the form $((a_1, a_2, \ldots, a_m)$, for each $a_i = (n_i, v_i) \in A$ an attribute is built; its name is set to n_i and its value is set to v_i (if v_i is a string) or $\pi_\lambda(T_k)p$ (if v_i is of the form λp[1];*
  - *if A is of the form $\lambda.A$, $e_i^o.A$ is set to $\lambda.A$[2];*
- *the tree construction specifications $e'_1, e'_2, \ldots, e'_m$ are treated as previously seen, and the elements built are made children of each e^o.*

Algorithm 1 explains the behavior of the tree construction operator. It uses the procedure *SimpleSpecification*, which is shown in Algorithm 2.

Algorithm 1 Algorithm TreeConstruction

Input: a forest F and a tree construction predicate $e_1, e_2, \ldots, e_n$

Output: a forest F'

$F' \leftarrow ()$ {F' is initialized to the empty list}
for all element construction specification e_i **do**
  for all tree $T_j \in F$ **do**
    $F' \leftarrow F' \cup$ SimpleSpecification(e_i, T_j)
  if e_i does not contain any path expression **then**
    insert F as subtree of the rightmost leaf element of F'

Full-Text Selection

The full-text selection operator returns those trees having at least a subtree satisfying the full-text selection condition. Such a condition, specified using the full-text selection predicate, is a list of one or more words or phrases that must be found in the full-text value of the root element of a subtree reachable following a path expression. It is also possible to specify a window option, i.e. to constrain the searched words (or phrases) to have a distance between one and another not greater than a specified value. Moreover, stemming, thesaurus and stopwords can be used.

[1] This expression must return a single value.
[2] This expression must return a single value.

Algorithm 2 Algorithm SimpleSpecification

Input: a tree T and an element construction specification e

Output: a forest F

if e is of the form λ **then**

return $\pi_\lambda(T)$

else $\{e$ is of the form $n(v, A, (e_1, e_2, \ldots, e_n))\}$

if v is a string **then**

build new element e'; $e'.n \leftarrow n$; $e'.v \leftarrow v$

else $\{v$ is of the form $\lambda p\}$

for all tree $T' \in \pi_\lambda(T)$ **do**

build new element e'; $e'.n \leftarrow n$; $e'.v \leftarrow root(T')p$

for all element e' just built **do**

if A is of the form $\lambda.A$ **then**

$T' \leftarrow \pi_\lambda(T)$; $e'.A \leftarrow T'.A$

else $\{A$ is of the form $((a_1, v_1), \ldots (a_n, v_n))\}$

for all pair (n_i, v_i) **do**

build new attribute a and assign it to e'; $a.n \leftarrow n_i$

if v_i is a string **then**

$a.v \leftarrow v_i$

else $\{v_i$ is of the form $\lambda p\}$

$T' \leftarrow \pi_\lambda(T)$; $a.v \leftarrow root(T')p$

for all sub-element construction specification e_i **do**

$F \leftarrow$ SimpleSpecification(T)

for all tree $T \in F$ **do**

$root(T).p = e'$

return a forest of trees having e' as root element

In what follows we give the formal definitions of full-text selection predicate and full-text selection operator.

Definition 3.35 (Full-Text Selection Predicate) *A full-text selection predicate P is an expression of the form $\lambda a[\gamma,\ x, \texttt{stem}, \texttt{thes}, \texttt{stop}]$, where:*

- *λ is a path expression;*
- *a is optional and, if present, is of the form* `.A[`attname`]`*;*
- *γ is a list of one or more base conditions $\gamma_1, \gamma_2, \ldots, \gamma_n$ connected with boolean operators (*`AND`, `OR`, `NOT`*); each base condition γ_i is of the form "s_i", where s_i is a word or phrase;*
- *x is an optional integer value;*
- `stem`, `thes`, *and* `stop` *are optional.*

Let F' be the forest of subtrees of a tree T that can be reached from $root(T)$ by following the path λ, i.e. $F' = \pi_\lambda(T)$. The tree T satisfies the full-text selection predicate P if $\exists T' \in F'$ such that T' satisfies the boolean expression γ and, if present, the window option x. If one or more of the parameters `stem`, `thes`, *and* `stop` *are present, the full-text selection satisfaction must be decided using, respectively, stemming, thesaurus, and stopwords.*

Let $t = root(T')a.v$ if a is present, or let $t = root(T').fulltext$ if a is not present; each base condition γ_i is satisfied if t contains the word or phrase s_i.

Let $S = \{(t_1, t_2, \ldots, t_n) \mid t_i$ is a token (or a list of consecutive tokens) present in t such that $t_i = s_i\}$. Let $pos(t_i[j])$ be the position of the j-th token in the token list t_i (if t_i is a single token, only $t_i[1]$ is defined). T' satisfies the window option x if $\exists (t_1, t_2, \ldots, t_n) \in S$ such that, for each pair (t_k, t_w) of elements contained in $(t_1, t_2, \ldots, t_n)$ such that $pos(t_k[1]) < pos(t_w[1])$, $pos(t_w[1]) - pos(t_k[m]) \leq x$, where m is the length of the token list t_k.

In this definition t represents the *scope* of the full-text search. If the optional a is used, it is the value of the attribute being checked; otherwise, it is the full-text value of the root element of the subtree being checked.

The last part of the definition explains the meaning of the window option. Informally, each pair of searched words is checked, confronting their position; if we are searching for (say) two phrases instead that two single words, the position of the last word in the first phrase is confronted with the position of the first word in the second phrase.

Definition 3.36 (Full-Text Selection) *Given a forest $F = (T_1, T_2, ..., T_n)$ and a full-text selection predicate P, the full-text selection operator $\varsigma_P(F)$ returns a subforest $G \subset F$ such that each tree in G satisfies the full-text selection predicate P.*

Example 3.23 Suppose to have an AFTX expression A returning a forest containing, among the others, the tree T shown graphically in Figure 3.3. Let us write the following expression:

$$\varsigma_{/\texttt{chapter}[\text{“}\texttt{Usability Heuristic}\text{” AND “}\texttt{Web Design}\text{”,}10\texttt{,stem}]}(A) \ .$$

We are looking for chapters containing the phrases “*Usability Heuristic*”and “*Web Design*”, at a distance not greater than 10; stemming must be used. When T is checked, only one subtree rooted at `chapter` is found; actually, it corresponds to the entire tree T. The full-text value of the root element contains the searched phrases: the first one is found at position 25–26, the second one at position 33–34; note that the element actually contains the phrase “*Usability Heuristics*”instead of “*Usability Heuristic*”, but the usage of stemming allows to consider it as a match. In order to check the satisfaction of the window option the position of the last word in the first phrase (“*Heuristics*”) is confronted with the position of the first word in the second phrase (“*Web*”); $34 - 26 = 8 \leq 10$, therefore the subtree satisfies the selection condition.

Full-Text Score Assignment

The full-text score assignment operator calculates a score for each input tree, on the basis of the full-text conditions specified in the predicate. It does not filter out any tree, it

just calculates the score. The predicate is identical to that of full-text selection, with the following exceptions:

- weights can be assigned to each searched word;
- the function to use for score calculation can be specified.

In what follows we give the formal definitions of predicate and operator.

Definition 3.37 (Full-Text Score Predicate) *A full-text score predicate P is an expression of the form $\lambda a[\gamma, x, \texttt{stem}, \texttt{thes}, \texttt{stop}]f$, where:*

- *λ is a path expression;*
- *a is optional and, if present, is of the form* `.A[`attname`]`*;*
- *γ is a list of one or more base conditions $\gamma_1, \gamma_2, \ldots, \gamma_n$ connected with boolean operators (*`AND`, `OR`, `NOT`*); each base condition γ_i is of the form w_i"s_i", where:*
  - *s_i is a word or phrase;*
  - *w_i is an optional decimal value representing the weight assigned to the word (or phrase) s;*
- *x is an optional integer value;*
- `stem`, `thes`, *and* `stop` *are optional;*
- *f is a function pointer.*

Definition 3.38 (Full-Text Score Assignment) *Given a forest $F = (T_1, T_2, ..., T_n)$ and a full-text score assignment predicate P, the full-text score assignment operator $\xi_P(F)$ returns a forest $G = (T'_1, T'_2, ..., T'_n)$ such that, for each i, $T'_i \equiv T_i$, with the exception that $root(T')$.`score` has a new value calculated by the function f considering the full-text selection predicate $\lambda a[\gamma, x, \texttt{stem}, \texttt{thes}, \texttt{stop}]$ and the weights w_i assigned to each word or phrase included in γ_i.*

Full-Text Selection with Score

The full-text selection with score operator combines the features of full-text selection and full-text score assignment: it filters out those trees that do not satisfy the selection condition and assigns to each retained tree a score.

The full-text selection with score predicate is actually a full-text score predicate. Consequently, we directly give the formal definition of the operator.

Definition 3.39 (Full-Text Selection with Score) *Given a forest $F = (T_1, T_2, ..., T_n)$ and a full-text score predicate $P = \lambda a[\gamma, x, \texttt{stem}, \texttt{thes}, \texttt{stop}]f$, the full-text selection with score operator $\overline{\varsigma}_P(F)$ returns a forest F' such that, for each tree $T' \in F'$:*

- *$\exists T_i \in F$ such that $T' \equiv T_i$, with the exception that $root(T')$ has a new value of* `score` *representing the level of satisfaction of the full-text score predicate P;*
- *T' satisfies the full-text selection predicate $P' = \lambda a[\gamma', x, \texttt{stem}, \texttt{thes}, \texttt{stop}]$, where γ' is obtained by removing weights from γ.*

The full-text selection with score operator is a derived operator; in fact the following equation holds:

$$\overline{\varsigma}_P(F) = \xi_P(\varsigma_{P'}(F)) \ .$$

Top-K and Threshold Full-Text Selection

The top-K full-text selection is a derived operator: it applies to the input forest the score assignment operator, thus assigning each tree a score value; then it orders the trees by the score value just computed and retains only the k trees with highest score.

The top-K full-text selection predicate is identical to the score assignment predicate, augmented with a k stating the number of trees to retain. Formal definitions of predicate and operator follow.

Definition 3.40 (Top-K Full-Text Selection Predicate) *A top-k full-text selection predicate P is an expression of the form $\lambda a[\gamma, x, \texttt{stem}, \texttt{thes}, \texttt{stop}]f, k$, where:*

- *λ is a path expression;*

- a *is optional and, if present, is of the form* `.A[`attname`]`*;*
- γ *is a list of one or more base conditions* $\gamma_1, \gamma_2, \ldots, \gamma_n$ *connected with boolean operators (*`AND`*,* `OR`*,* `NOT`*); each base condition* γ_i *is of the form* w_i *"*s_i*", where:*
  - s_i *is a word or phrase;*
  - w_i *is an optional decimal value representing the weight assigned to the word (or phrase)* s_i*;*
- x *is an optional integer value;*
- `stem`*,* `thes`*, and* `stop` *are optional;*
- f *is a function pointer;*
- k *is an integer representing the number of trees to return.*

Definition 3.41 (Top-K Full-Text Selection) *Given a forest* $F = (T_1, T_2, ..., T_n)$ *and a top-K full-text selection predicate* P*, the top-K full-text selection operator* $\top_P(F)$ *returns a forest* $G = (T'_1, T'_2, ..., T'_k)$ *such that:*

- $\forall T'_i \in G$*,* $\exists T''_j \in \xi_{P'}(F)$ *such that* $T'_i \equiv T''_j$*, where* P' *is obtained from* P *by removing* k*;*
- $\forall T''_j \in \xi_{P'}(F)$ *such that* $T''_j \notin G$*,* $root(T'_i)$.`score` $\geq$ $root(T''_j)$.`score`*, for each* $T'_i \in G$*;*
- G *is in descending order by the value of the* `score` *property of the trees' root element.*

The top-K full-text selection operator is a derived operator; in fact the following equation holds:

$$\top_P(F) = \sigma_{[\texttt{.pos} \leq \texttt{k}]}(o_{/\texttt{1.score DESC}}(\xi_{P'}(F)))$$

where P' *is obtained from* P *by removing* k*.*

The threshold full-text selection is also a derived operator: it applies to the input forest the score assignment operator, thus assigning each tree a score value; then it selects the trees with a score higher than a treshold τ and returns them sorted by score.

The top-K full-text selection predicate is identical to the score assignment predicate, augmented with a τ stating the score threshold under which trees must be discarded. Formal definitions of predicate and operator follow.

Definition 3.42 (Threshold Full-Text Selection Predicate) *A threshold full-text selection predicate P is an expression of the form $\lambda a[\gamma\ x, \texttt{stem}, \texttt{thes}, \texttt{stop}]f, \tau$, where:*

- *λ is a path expression;*
- *a is optional and, if present, is of the form* `.A[`attname`]`*;*
- *γ is a list of one or more base conditions $\gamma_1, \gamma_2, \ldots, \gamma_n$ connected with boolean operators (*`AND`, `OR`, `NOT`*); each base condition γ_i is of the form w_i "s_i", where:*
  - *s_i is a word or phrase;*
  - *w_i is an optional decimal value representing the weight assigned to the word (or phrase) s;*
- *x is an optional integer value;*
- `stem`, `thes`, *and* `stop` *are optional;*
- *f is a function pointer;*
- *τ is a decimal value representing the minimum score of the trees to return.*

Definition 3.43 (Threshold Full-Text Selection) *Given a forest $F = (T_1, T_2, ..., T_n)$ and a threshold full-text selection predicate P, the threshold full-text selection operator $\omega_P(F)$ returns a forest $G = (T'_1, T'_2, ..., T'_m)$ such that:*

- *$\forall T'_i \in G$, $\exists T''_j \in \xi_{P'}(F)$ such that $T'_i \equiv T''_j$, where P' is obtained from P by removing τ;*

- $\forall T'_i \in G$, $root(T'_i)$.`score` $\geq \tau$;
- $\forall T''_j \in \xi_{P'}(F)$ *such that* $T''_j \notin G$, $root(T'_i)$.`score` $< \tau$;
- G *is in descending order by the value of the* `score` *attribute of the trees' root element.*

The threshold full-text selection operator is a derived operator; in fact the following equation holds:

$$\omega_P(F) = o_{/1.\texttt{score DESC}}(\sigma_{/1[.\texttt{score}\geq\tau]}(\xi_{P'}(F)))$$

where P' *is obtained from* P *by removing* τ.

Chapter 4

Translating XQuery (Full-Text) Expressions

In this chapter we show how an XQuery (Full-Text) expression can be translated into an AFTX expression. In Section 4.1 we show how each clause of a FLWOR expression (without full-text extensions) can be translated; informal translation rules, examples and a formal translation algorithm are presented. Then in Section 4.2 we deal with full-text extensions; again we provide an informal overview, translation examples and the translation algorithm. In Section 4.3 we translate more complex XQuery (Full-Text) expressions, taken from W3C XQuery Use Cases [Con06b] and XQuery Full-Text Use Cases [Con06e]. Finally in Section 4.4 we briefly introduce a proposed extension of XQuery with update capabilities and informally discuss how the new XQuery expressions could be translated into AFTX expressions.

4.1 XQuery Translation Rules

4.1.1 Informal Overview

The `for` Clause

A `for` clause with a single variable binding is of the form

`for $i in doc("`*docname*`")`$\lambda_1[\gamma_1]\lambda_2[\gamma_2]\ldots\lambda_n[\gamma_n]$

where *docname* is the input XML document, λ_i is a path expression and γ_i is a condition. A `for` clause with a single variable binding is translated into the following algebraic expression:

$$\sigma_{/1[\gamma_n]}(\pi_{/1\lambda_n}(\ldots(\sigma_{/1[\gamma_2]}(\pi_{/1\lambda_2}(\sigma_{/1[\gamma_1]}(\pi_{\lambda_1}(\text{“}docname\text{”}))))))) \ .$$

Projection is used to follow a path, selection represents the filter predicates. For example, the clause

```
for $i in doc("books.xml")/bib/book[@year=2000]/author
```

applies to the document `books.xml` the path expression `/bib/book`, followed by the condition `@year=2000`, followed by the path expression `/author`. This clause is translated into the following algebraic expression:

$$\pi_{/1/\texttt{author}}(\sigma_{/1[.\texttt{A}[\texttt{year}].\texttt{v}=2000]}(\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”}))) \ .$$

Remember that `/1` is a special case of path expression, that selects the first child of the current element. If such an expression is found in the first step of a path expression, it selects the root element, therefore the previous query may equivalently be written as:

$$\pi_{/\texttt{book/author}}(\sigma_{/\texttt{book}[.\texttt{A}[\text{“}\texttt{year}\text{”}].\texttt{v}=2000]}(\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”}))) \ .$$

A `for` clause could contain the definition of a positional variable, through the use of the reserved keyword `at`. The presence of a positional variable does not change the algebraic expression associated with the for clause; if later in the query there is a reference to the positional variable, that reference is translated using the element property `.pos`. For example the clause

```
for $i at $p in doc("books.xml")/bib/book[@year=2000]
    /author
```

is translated into the same algebraic expression as before.

A `for` clause can contains a `distinct-values` function calls, like the following:

```
for $i in distinct-values(doc("books.xml")//author/last)
```

This clause is translated using the duplicate elimination operator as follows:

$$\nu_{(\text{/last.v, "last"})}(\pi_{\text{//author/last}}(\text{"books.xml"})) \ .$$

Conditions can be nested. A `for` clause with a nested condition is of the form:

`for $i in` $\ldots \lambda[\gamma_1$ `and` $[\lambda_2\gamma_2]]$

Such a condition is translated into the following algebraic expression:

$$\sigma_{\lambda[\gamma_1 \text{ AND } \lambda_2\gamma_2]}(A)$$

where A is the algebraic expression representing `for $i in` ....

A `for` clause with multiple variable bindings is of the form

`for $`i_1 `in doc("`$docname_1$`")`$\lambda_1^1[\gamma_1^1]\lambda_2^1[\gamma_2^1]\ldots\lambda_n^1[\gamma_n^1]$,
`$`i_2 `in doc("`$docname_2$`")`$\lambda_1^2[\gamma_1^2]\lambda_2^2[\gamma_2^2]\ldots\lambda_{n'}^2[\gamma_{n'}^2]$,
...,
`$`i_m `in doc("`$docname_m$`")`$\lambda_1^m[\gamma_1^m]\lambda_2^m[\gamma_2^m]\ldots\lambda_{n''}^m[\gamma_{n''}^m]$

where i_k is a variable name, $docname_k$ is an input XML document, λ_i^k is a path expression and γ_i^k is a condition. A `for` clause with multiple variable binding is translated in the following algebraic expression:

$$((((A_1 \times A_2) \times A_3)\ldots) \times A_m)$$

where A_k is the algebraic expression corresponding to the k-th variable binding, obtained as seen in the case of a `for` clause with a single variable binding. For example the clause

```
for $i in doc("books.xml")/bib/book[@year=2000],
  $j in doc("authors.xml")/authors/author[/first="John"]
```

is translated into the following algebraic expression:

$$\sigma_{\text{/1[.A[year].v=2000]}}(\pi_{\text{/bib/book}}(\text{"books.xml"})) \times$$
$$\sigma_{\text{/1/first[.v="John"]}}(\pi_{\text{/authors/author}}(\text{"authors.xml"}))$$

In a `for` clause with multiple variable bindings, a variable binding can refer to another variable. A clause of the form

`for $i in ...,`

`$j in $i`$\lambda[\gamma]$

is translated into the following algebraic expression:

$$A \bowtie_{\lambda_1[\lambda \equiv \lambda_2]} \sigma_{/1[\gamma]}(\pi_{/1\lambda}(A)))$$

where A is an algebraic expression representing the first variable binding and λ_1 (respectively λ_2) is the path expression representing the variable $\$i$ (respectively $\$j$). Informally speaking, each tree resulting from the first binding is joined with those subtrees rooted at λ that respect the selection condition γ. For each resulting tree, the left subtree of the root will represent the variable $i, while the right subtree will represent the variable $j. For example, the clause

```
for $i in doc("books.xml")/bib/book[@year=2000],
    $j in $i/author[./first="Serge"]
```

is translated in the following algebraic expression:

$$\sigma_{/1[.\texttt{A[year]}.\texttt{v}=2000]}(\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”})) \bowtie_{/\texttt{book}[/\texttt{author}\equiv/\texttt{author}]} \sigma_{/1/\texttt{first}[.\texttt{v}=\text{“}\texttt{Serge}\text{”}]}(\pi_{/1/\texttt{author}}(\sigma_{/1[.\texttt{A[year]}.\texttt{v}=2000]}(\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”}))))$$

The result of this expression, where `books.xml` is the XML document of Figure 2.3, is displayed in Figure 4.1. An efficient implementation of the algebra should first calculate the left input forest for the join ($\sigma_{/1[.\texttt{A[year]}.\texttt{v}=2000]}(\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”}))$), then use this partial result to calculate the right input forest (by applying $\sigma_{/1/\texttt{first}[.\texttt{v}=\text{“}\texttt{Serge}\text{”}]}(\pi_{/1/\texttt{author}}$ to the partial result), and finally calculate the join of the two forests.

The `let` Clause

A typical `let` clause is of the form

`let $i := doc("`*docname*`")`$\lambda_1[\gamma_1]\lambda_2[\gamma_2]\ldots\lambda_n[\gamma_n]$

```
<prod_root>
  <book year="2000">
    <title>Data on the Web</title>
    <author>
      <last>Abiteboul</last><first>Serge</first>
    </author>
    <author>
      <last>Buneman</last><first>Peter</first>
    </author>
    <author>
      <last>Suciu</last><first>Dan</first>
    </author>
    <publisher>Morgan Kaufmann Publishers</publisher>
      <price>65.95</price>
  </book>
  <author>
    <last>Abiteboul</last><first>Serge</first>
  </author>
</prod_root>
```

Figure 4.1: The result of a `for` clause with 2 variable bindings, where the second variable references the first one.

where *docname* is the input XML document, λ_i is a path expression and γ_i is a condition.

Unlike a `for` clause, a `let` clause binds a variable to the result of its associated expression, without iteration. This difference is also present in our algebra: while a `for` clause is translated into an algebraic expression which returns a different tree for each possible binding, a `let` clause is translated into an algebraic expression which returns a single tree. This goal is achieved using the tree construction operator ι, which creates a root node named `let_root`; the result of the expression associated with the `let` clause will be inserted as subtree of that root node. A `let` clause with a single variable binding is thus translated into the following algebraic expression:

$$\iota_{\text{“let\_root”(null,null,null)}}\big(\\ \sigma_{/1[\gamma_n]}(\pi_{/1\lambda_n}(\ldots(\sigma_{/1[\gamma_2]}(\pi_{/1\lambda_2}(\sigma_{/1[\gamma_1]}(\pi_{\lambda_1}(\text{“}docname\text{”}))))))))$$

Often a `let` clause is used in conjunction with a `for` clause. This case can be treated in the same way as a `for` clause with multiple variable bindings: the expression representing the `for` clause is combined, using the product operator, with the expression representing the `let` clause. It must be pointed out, however, that the algebraic expression representing the `let` clause will return a single tree, that will be the right subtree of each root element resulting from the product operation. For example the (partial) query

```
for $a in doc("authors.xml")//author
let $b := doc("books.xml")//book
```

is translated into the following algebraic expression:

$$\pi_{//\texttt{author}}(\text{“}\texttt{authors.xml}\text{”}) \times \iota_{\text{“let\_root”(null,null,null)}}(\pi_{//\texttt{book}}(\text{“}\texttt{books.xml}\text{”}))\ .$$

Sometimes a `let` clause is used a simple “alias” for a complex expression; in this case the `let` clause is of the form

```
let $i := $jλ
```

where λ is a path expression and `$j` is previously defined variable. Such a clause does not need to be explicitly translated into an algebraic expression; in fact, when the variable `$i` will be referred to (e.g. in a `return` clause), that reference will be substituted with a reference to `$j`λ. For example the query

```
for $b in doc("books.xml")/bib/book
let $c := $b/author
return  <book>
          {$b/title, <count>{ count($c) }</count>}
        </book>
```

is translated into the following algebraic expression:

$$\iota_{\text{“book”}(\texttt{null},\texttt{null},(/\texttt{book}/\texttt{title},\text{“count”}(/\texttt{book}/\texttt{author.count},\texttt{null},\texttt{null})))}(\pi_{/\texttt{bib}/\texttt{book}}(\text{“}\texttt{books.xml}\text{”}))) \ .$$

The `where` Clause

A simple `where` clause is of the form

`where $i`$\lambda\gamma$

where λ is a path expression and γ is a condition. Such clause is translated into the following algebraic expression:

$$\sigma_{\lambda'\lambda[\gamma]}(A)$$

where:

- A is the algebraic expression representing the input forest;
- λ' is a path expression that locates the nodes bound to the variable $i.

A key point in the translation process is the need to keep track of the path expression that locates the nodes bound to a variable. Whenever a `for` or `let` clause is translated, the translator creates a new pair (*variable*, path), which will be later used when the variable is referred in the XQuery expression. For example the (partial) query

```
for $i in doc("books.xml")/bib/book[@year=2000]
where $i/price > 50
```

is translated into the following algebraic expression:

$$\sigma_{\texttt{/book/price[.v>50]}}(\sigma_{\texttt{/book[.A[year].v=2000]}}(\pi_{\texttt{/bib/book}}(\text{“}\texttt{books.xml}\text{”}))) \ .$$

After the translation of the `for` clause, a pair (`$i`, `/book`) must have been created. The meaning of this pair is: “*whenever in the translation a reference to $i is found, it must be substituted with the path expression* `/book`”. In fact, when the expression `$i/price` is encountered, it is translated as `/book/price`.

A `where` clause can also refer to two variables; in this case the clause is of the form

`where $i`$\lambda_1 p_1 \theta$`$j`$\lambda_2 p_2$

where λ_i is a path expression, p_i is an element property and θ is a comparison operator. Such a clause is translated into the following algebraic expression:

$$\sigma_{\lambda_0[\lambda'_1 \lambda_1 p_1 \theta \lambda'_2 \lambda_2 p_2]}(A)$$

where:

- λ_0 is the longest common path expression between the path expressions that locate the nodes bound to the variables $i and $j;
- λ'_1 and λ'_2 are the path expressions that locate the nodes bound to the variables $i and $j, excluding the common part considered in λ_0;
- A is an algebraic expression representing the input forest.

For example the (partial) query

```
for $i in doc("books.xml")//book,
  $j in doc("authors.xml")//author
where $i/author = $j/@id
```

is translated into the following algebraic expression:

$$\sigma_{\texttt{/prod\_root[/book/author.v=/author.A[id].v]}}(\pi_{\texttt{//author}}(\text{“}\texttt{authors.xml}\text{”}) \times \pi_{\texttt{//book}}(\text{“}\texttt{books.xml}\text{”})) \ .$$

This example shows that the path expression that locates the nodes bound to a variable can change during the translation process. In particular, when a product operator is inserted into the algebraic expression, the path expression that locates the nodes bound to the variables involved in the product must be changed by adding a leading `/prod_root`. In this example, initially the two variable bindings in the `for` clause are translated and two pairs (`$i`, `/book`) (`$j`, `/author`) are created. Then a product must be inserted between the two algebraic expressions; therefore at the end of the `for` translation `$i` is located by `/prod_root/book` and `$j` is located by `/prod_root/author`.

Since two variables are involved in the kind of `where` clause we are considering, the algebraic expression A will necessarily contain a product operator; an alternative way to express the `where` condition is to substitute, in A, the product operator with a join operator. Therefore the previous expression could be rewritten as follows:

$$\pi_{//\texttt{author}}(\text{“}\texttt{authors.xml}\text{”}) \bowtie_{/\texttt{book}/\texttt{author}[.\texttt{v}=/\texttt{author.A}[\texttt{id}].\texttt{v}]} \pi_{//\texttt{book}}(\text{“}\texttt{books.xml}\text{”}) \ .$$

Quantifiers can be used in a `where` clause. A `where` clause involving an existential quantifier is of the form

`where some $i in $j`λ_1 `satisfies`($\$i\lambda_2\gamma$)

where λ_1 and λ_2 are path expressions and γ is a condition. Such a `where` clause is translated into the following algebraic expression:

$$\sigma_{\lambda'\lambda_1[\lambda_2\gamma]}(A)$$

where:

- λ' is a path expression that locates the nodes bound to the variable $j;
- A is the algebraic expression representing the input forest.

On the other side, a `where` clause involving a universal quantifier is of the form

`where every $i in $j`λ_1 `satisfies`($\$i\lambda_2\gamma$)

where, as before, λ_1 and λ_2 are path expressions and γ is a condition. Such a `where` clause is translated into the following algebraic expression:

$$A - \sigma_{\lambda'\lambda_1[\neg\lambda_2\gamma]}(A)$$

where A is the algebraic expression representing the input forest. Informally, in order to check that every subtree reachable from $j following the path λ_1 satisfies the selection predicate $\lambda_2\gamma$, we subtract from the input forest those trees that have at least one subtree reachable from $j following the path λ_1 that satisfies the inverted selection predicate $\neg\lambda_2\gamma$. It is worth noticing that this translation is coherent with the semantics of XQuery universal quantifier; in fact the resulting forest will also contain the trees that do not have a subtree reachable following the path $\lambda_1\lambda_2$, and this is exactly equal to the XQuery behavior.

The `order by` Clause

An `order by` clause is of the form

`order by` $\$i_1\lambda_1 x_1\ a_1, \$i_2\lambda_2 x_2\ a_2, \ldots, \$i_n\lambda_n x_n\ a_n$

where:

- $\$i_k$ is a variable name;
- λ_k is a path expression;
- x_k is one of the form *@attname* (indicating the value of the attribute named *attname*), .count (indicating the number of elements with name n, where n is the name of the current element) or the empty string (indicating the value of the current element);
- a_k is `ASCENDING` or `DESCENDING`.

An `order by` clause is translated in the following algebraic expression:

$$o_{\lambda'_1\lambda_1 p_1\ a_1, \lambda'_2\lambda_2 p_2\ a_2, \ldots, \lambda'_n\lambda_n p_n\ a_n}(A)$$

where:

- A is the algebraic expression representing the input forest;
- λ'_k is a path expression that locates the nodes bound to the variable $\$i_k$;
- p_k is an element property;
- a_k is ASC or DESC.

For example the (partial) query

```
for $i in doc("books.xml")/bib/book
  order by $i/title ascending
```

is translated into the following algebraic expression:

$$o_{/\texttt{book/title.v ASC}}(\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”}))\ .$$

As a more complex example, consider again the query of Example 3.9. We want to retrieve, for each author, the last name and the books written by him; in addition, the result should be sorted by author's last name and book title. The XQuery expression

```
for $i in doc("books.xml")/bib/book/author/last,
  $j in doc("books.xml")/bib/book
where $i=$j/author/last
order by $i ascending, $j/title ascending
```

is translated into the following algebraic expression:

$$o_{/\texttt{prod\_root/last.v ASC},/\texttt{prod\_root/book/title.v ASC}}(\\ \sigma_{/\texttt{prod\_root}[/\texttt{last.v}=/\texttt{book/author/last.v}]}(\\ \pi_{/\texttt{bib/book/author/last}}(\text{“}\texttt{books.xml}\text{”})\times\\ \pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”})))$$

The `return` Clause

The simplest form of a `return` clause is the following:

`return` $\{$`$i`$\lambda\}$

where:

- $i is a variable name;
- λ is a path expression.

This clause does not actually build any new tree; what it does is to project the input forest into the path λ. Consequently it can be translated into the following algebraic expression:

$$\pi_{\lambda}(A)$$

where A is the algebraic expression representing the input forest. For example the query

```
for $i in doc("books.xml")/bib/book/author
order by $i/last, $i/first
return {$i/last}
```

is translated into the following algebraic expression:

$$\pi_{\texttt{/author/last}}(o_{\texttt{/author/last.v ASC,/author/first.v ASC}}(\pi_{\texttt{/bib/book/author}}(\text{“}\texttt{books.xml}\text{”})))\ .$$

As soon as the `return` clause contains more than one reference to the input forest, it is necessary to use the tree construction predicate. A `return` clause that does not contain element constructors is of the form

`return` $\{\$i_1\lambda_1\}\{\$i_2\lambda_2\}\ldots\{\$i_n\lambda_n\}$

and is translated into the following algebraic expression:

$$\iota_{\lambda'_1\lambda_1,\lambda'_2\lambda_2,\ldots,\lambda'_n\lambda_n}(A)$$

where:

- A is the algebraic expression representing the input forest;
- λ'_k is a path expression that locates the nodes bound to the variable $\$i_k$.

For example the query

```
for $i in doc("books.xml")/bib/book
return {$i/author}{$i/editor}
```

that returns all the authors plus all the editors, is translated into the following algebraic expression:

$$\iota_{\texttt{/book/author},\texttt{/book/editor}}(\pi_{\texttt{/bib/book}}(\text{“}\texttt{books.xml}\text{”}))\ .$$

The construction predicate is obviously necessary when the `return` clause contains an element constructor, even if there is only one reference to the input forest. For example the query

```
for $i in doc("books.xml")/bib/book
return <book title={$i/title}></book>
```

that returns the title of all books as attribute of a *title* element, is translated into the following algebraic expression:

$$\iota_{\text{“}\texttt{book}\text{”}(\texttt{null},((\text{“}\texttt{title}\text{”},\texttt{/book/title.v})),\texttt{null})}(\pi_{\texttt{/bib/book}}(\text{“}\texttt{books.xml}\text{”}))\ .$$

Sometimes a `return` clause may refer more than one variable. Consider again the query of Example 3.9. As before, we want to retrieve the last name of each author and the books written by him, but the only information we want about books is the title. Moreover, each pair (author,title) should appear just once and the name of the author should be an attribute of the element *author*. The corresponding XQuery expression

```
for $i in distinct-values(doc("books.xml")/bib/book
  /author/last),
  $j in doc("books.xml")/bib/book
where $i=$j/author/last
return  <author name={$i}>
```

```
    <book>{$j/title/text()}</book>
  </author>
```

is translated in the following algebraic expression:

$$\iota_{\text{“author”}(\texttt{null},((\text{“name”},/\texttt{prod\_root/group\_root.A[last].v})),(\text{“book”}(/\texttt{prod\_root/book/title.v},\texttt{null},\texttt{null})))}\Big($$
$$\sigma_{/\texttt{prod\_root}[/\texttt{group\_root.A[last].v}=/\texttt{book/author/last.v}]}\Big($$
$$\nu_{(/\texttt{last.v},\text{“last”})}\big(\pi_{/\texttt{bib/book/author/last}}(\text{“books.xml”})\big) \times$$
$$\pi_{/\texttt{bib/book}}(\text{“books.xml”})\big)\Big) \ .$$

The previous XQuery expression, however, does not return the result one probably wants; in fact, if an author wrote five books, the resulting forest will contain 5 trees with the same author, one for each book written by him. Consider now the following nested expression:

```
for $i in doc("authors.xml")/authors/author/last
return <author name={$i}>
  {
    for $j in doc("books.xml")/bib/book
    where $j/author/last=$i
    return <book>{$j/title/text()}</book>
  } </author>
```

Thanks to the nesting, now each author is returned just once. This XQuery expression, however, requires that each author present in the document *authors.xml* must be returned, even if he has not written any book. A simple selection with a selection predicate like $/\texttt{prod\_root}/\texttt{book}/\texttt{author}/\texttt{last.v} = /\texttt{prod\_root}/\texttt{author}/\texttt{last.v}$ is therefore not usable, because, deleting any tree that does not respect the condition, it would cancel an author that has not written any book. What we need is a sort of *left outer join*. This goal can be reached using deletion and grouping:

$$\iota_{\text{“author”}(\texttt{null},((\text{“name”},/\texttt{group\_root/last.v})),(\text{“book”}(/\texttt{group\_root/book/title.v},\texttt{null},\texttt{null}))}($$
$$\delta_{/\texttt{group\_root}/*[.\texttt{k}=/\texttt{group\_root.A}[\text{“treeIdentity”}].\texttt{v AND .pos}>1]}($$
$$\Sigma_{((/\texttt{prod\_root/1.k},\text{“treeIdentity”})),(/\texttt{prod\_root/1},/\texttt{prod\_root/2})}($$
$$\delta_{/\texttt{prod\_root/book}[/\texttt{author/last}]}($$
$$\delta_{/\texttt{prod\_root/book/author}[/\texttt{last.v}\neq/\texttt{prod\_root/author/last.v}]}($$
$$\pi_{/\texttt{authors/author/last}}(\text{“}\texttt{authors.xml}\text{”})\times$$
$$\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”}))))$$

Let us examine this expression, from the inner part to the outer part. The first deletion prunes each `last` element in the right subtree whose value is not equal to that of the `last` element in the left subtree. The second deletion prunes each `book` element in the right subtree does not have a child `author` element having a child `last` element. Now a tree whose left subtree represents an author that has not written the book represented by the right subtree has been reduced to a tree without a `book` subtree. When we group by node identity of the left subtree root element (i.e. `last` elements), an author that has not written any book is still present in the result, obviously without any associated book. Finally the last deletion deletes multiple `last` subtrees, retaining just the first one.

4.1.2 Formal Translation Algorithm

We have informally seen in Section 4.1.1 how most of XQuery expressions can be translated into AFTX expressions. We now want to formally state which part of XQuery can be expressed into AFTX and how such a translation is carried out.

The fragment of XQuery expressible in our algebra is shown by the following grammar:

```
Expr              ::= ExprSingle ("," ExprSingle)*
ExprSingle        ::= FLWORExpr | Constructor
FLWORExpr         ::= (ForClause | LetClause | ForClause
                      LetClause) WhereClause?
                      OrderByClause? "return" Constructor
ForClause         ::= "for" VarRef PositionalVar? "in"
```

```
                        ForLetContext ("," VarRef
                        PositionalVar? "in" ForLetContext)*
VarRef              ::= "$" Name
PositionalVar       ::= "at" VarRef
ForLetContext       ::= DVFunction | DVContext
DVFunction          ::= "distinct-values" "(" DVContext ")"
DVContext           ::= DocFunction PathExpr
                        | VarRef PathExpr
DocFunction         ::= "doc" "(" Literal ")"
LetClause           ::= "let" VarRef ":=" ForLetContext
                        ("," VarRef ":=" ForLetContext)*
WhereClause         ::= "where" (ComparisonExpr |
                        QuantifiedExpr) ("and"
                        (ComparisonExpr | QuantifiedExpr))*
OrderByClause       ::= "order" "by" OrderSpec
                        ("," OrderSpec)*
OrderSpec           ::= VarRef (AxisStep QName)* ("@" QName)?
                        ("ascending" | "descending")?
AxisStep            ::= "/" | "//"
QuantifiedExpr      ::= ("some" | "every") VarRef "in"
                        VarRef PathExpr "satisfies" "("
                        ComparisonExpr ")"
ComparisonExpr      ::= UnaryExpr GeneralComp UnaryExpr
UnaryExpr           ::= (VarRef PathExpr) | Literal
                        | CountPosFunction
GeneralComp         ::= "=" | "!=" | "<" | "<=" | ">" | ">="
PathExpr            ::= AxisStep RelativePathExpr
RelativePathExpr    ::= StepExpr (AxisStep StepExpr)*
                        ("/" FinalStepExpr)?
StepExpr            ::= NameTest Predicate*
```

```
NameTest            ::= QName | "*"
Predicate           ::= "[" ComparisonExpr2 "]"
ComparisonExpr2     ::= UnaryExpr1 (GeneralComp UnaryExpr2)?
UnaryExpr1          ::= PathExpr | CountPosFunction
UnaryExpr2          ::= PathExpr | (VarRef PathExpr)
                        | Literal | CountPosFunction
FinalStepExpr       ::= ("@" NameTest) | ("text" "(" ")")
CountPosFunction    ::= "count" "(" ForLetContext ")"
                        | "position" "(" ")"
Constructor         ::= DirElemConstructor | EnclosedExpr*
DirElemConstructor  ::= "<" QName DirAttribute*
                        ("/>" | (">" DirElemContent* "</"
                        QName ">"))
DirAttribute        ::= (QName "=" DirAttributeValue)
DirAttributeValue   ::= """ Literal """ | PathExpr2
DirElemContent      ::= DirElemConstructor | EnclosedExpr
                        | Literal
EnclosedExpr        ::= "{" (FLWORExpr | PathExpr2) "}"
PathExpr2           ::= VarRef (AxisStep QName)*
                        ("/" FinalStepExpr)?
```

With respect to the XQuery specifications, our fragment has the following main limitations:

- no prolog exists in a query;
- each single expression can only be a FLWOR expression or a constructor;
- nesting is permitted only inside a `return` clause;
- no function calls are permitted, except for the functions `count`, `pos` and `distinct-values`;
- the *if-then-else* construct is not supported.

We now present, step by step, the formal translation algorithm. The main function is *XQuery2AFTX*, presented in Algorithm 3. It checks every single expression in the query. For each expression, if it is a FLWOR expression, the procedure *FLWORExpr* is called.

Algorithm 3 Function XQuery2AFTX

Input: an XQuery expression e

Output: an AFTX expression A

1: **for all** ExprSingle $e_i \in e$ **do**
2: **if** e_i is a FLWORExpr **then**
3: $A_i \leftarrow$ ‘’; $V_i \leftarrow$ emptylist
4: FLWORExpr$(e_i, A_i, V_i,$ **true**$)$
5: $A \leftarrow A + A_i$
6: **else** $\{e_i$ is a Constructor$\}$
7: $A \leftarrow A +$ ‘$\iota_{\text{Constructor}(e_i)}$(’
8: **for all** FLWORExpr e'_i in e_i **do**
9: $A_i \leftarrow$ ‘’; $V_i \leftarrow$ emptylist
10: FLWORExpr$(e'_i, A_i, V_i,$ **true**$)$
11: $A \leftarrow A + A_i$
12: **if** e'_i is the last FLWORExpr **then**
13: $A \leftarrow A +$ ‘)’
14: **else**
15: $A \leftarrow A +'\cup'$
16: **if** e_i is not the last ExprSingle **then**
17: $A \leftarrow A +$ ‘ $\cup$ ’
18: **return** A

If the expression is a constructor, the function *Constructor* is called; we will analyze this function later. If the constructor contains some inner FLWOR expressions, for each of them the procedure *FLWORExpr* is called, and the resulting AFTX expressions are fed to the union operator. Note that this operation is done only for the outermost FLWOR expressions, because innermost FLWOR expressions are treated by the called *FLWORExpr* procedure; we mean that, if an XQuery expression is of the form

```
<tagname1> ...
  {
    for ...
    return <tagname2>...{for ...}</tagname2>
  }
</tagname1>
```

then *XQuery2AFTX* first calls *Constructor*, then call just once *FLWORExpr*, passing as input the outer FLWOR expression. The inner FLWOR expression is managed by *FLWORExpr* as we will see soon.

Finally, each AFTX expression representing a single expression (being either a FLWOR expression or a constructor) is fed to the Union operator.

The procedure *FLWORExpr*, presented in Algorithm 4, takes as input the FLWOR expression to manage, the AFTX expression built up to now, a variable binding list and a boolean value. The variable binding list is a list (initially empty) of elements that associate, to each variable used in the XQuery expression, the path expression that locates, in the forest resulting from an AFTX expression, the elements bound to that variable; it is populated when a `for` or `let` is managed, and it is used during the translation process. The boolean value tells the procedure whether it must apply a tree construction operator to the AFTX expression or it must only build a tree construction predicate and pass it back to the calling procedure; it is set to *true* when an outermost FLWOR expression is being translated.

FLWORExpr first calls the procedures *ForClause*, *LetClause, WhereClause*, and *OrderByClause*. Each procedure takes as input the clause of interest, the AFTX expression built up to now, and the variable binding list; they modify the AFTX expression and the variable binding list.

Then *FLWORExpr* calls the function *ReturnClause*, which will return a tree construction predicate. If the boolean input parameter *addTreeConstr* is *true*, this predicate is used in a tree construction operator which is added to the AFTX expression previously created; in any case the predicate is returned to the calling procedure.

Algorithm 4 Function FLWORExpr

Input: a FLWOR expression e, an AFTX expression A, a variable binding list V, a boolean $addTreeConstr$

Output: a tree constructor predicate t

1: **if** e contains a ForClause F **then**
2: ForClause(F, A, V)
3: **if** e contains a LetClause L **then**
4: LetClause(L, A, V)
5: **if** e contains a WhereClause W **then**
6: WhereClause(W, A, V)
7: **if** e contains a OrderByClause O **then**
8: OrderByClause(O, A, V)
9: $t \leftarrow$ ReturnClause(C, A, V) $\{C$ is the Constructor$\}$
10: **if** $addTreeConstr$ **then**
11: $A \leftarrow$ '$\iota_t($' $+ A +$ '$)$'
12: **return** t

The procedure *ForClause*, presented in Algorithm 5, cycles over each variable binding. For each one, a new element of the variable binding list is created.

The first thing to check is whether the associated expression contains a predicate that references a previously defined variable; in this case such expression is split into three parts: the part before the predicate, the predicate, and the part after the predicate. For example, if the associated expression is `doc("bib.xml")/book[/author = $a]/title`, it is split into: 1) `doc("bib.xml")/book`; 2) `[/author = $a]`; 3) `/title`. Such a splitting is performed by the procedure *SplitPathExpr*, shown in Algorithm 6.

If the first part of the associated expression (or the entire associated expression, if it has not been split) starts with `fn:doc`, the algebraic expression corresponding to the variable is initialized to the name of the XML document. Then the procedure *PathExpr*, shown in Algorithm 7, is called.

It checks each step in the path expression and adds a projection to the AFTX expression. If there are some predicates in the step, for each of them a selection is added to the AFTX expression; the selection predicate is created by calling the *Predicate* function, that we will analyze later. Note that *PathExpr* also sets the path expression that locates the elements bound to the variable defined in the `for` clause; it is set to the last `NameTest` found in the XQuery path expression. For example, if the clause is `for $t in doc("bib.xml")/book[@year="1999"]/title`, the variable binding list will contain the pair (`$t`, `/title`).

Now *ForClause* checks if a `distict-values` function is applied to the expression just translated. If this is the case (and the expression has not been split), a leading duplicate elimination operator is added to the AFTX expression, and the variable binding list element is updated.

Finally, a call to the procedure *CreateProduct*, shown in Algorithm 8, is used in order to introduce a product between the AFTX expression just created and the AFTX expression previously created. *CreateProduct* also updates the variable binding list by adding a leading `/prod_root` to each path expression.

Instead of starting with a `fn:doc`, the first part of the expression associated with a

Algorithm 5 Procedure ForClause

Input: a `for` clause F, an AFTX expression A, a variable binding list V

1: **for all** variable binding in F relative to a variable $\$i$ **do**
2: $V_{\$i} \leftarrow$ ''; SplitPathExpr(PE, PE', P, PE'') {PE is the PathExpr in DVContext}
3: **if** PE' starts with `fn:doc("docname")` **then**
4: $A_i \leftarrow$ '"`docname`"'; PathExpr(PE', A_i, V)
5: **if** ForLetContext is a DVFunction and P is null **then**
6: $A_i \leftarrow$ '$\nu_{(V_{\$i}+\text{'.v,"'}+V_{\$i}\text{ without heading '/'}+\text{'")}}$(' + A_i + ')'
7: $V_{\$i} \leftarrow$ '/`group_root.A`[' + $V_{\$i}$ without heading '/' + ']'
8: CreateProduct(A_i, A, V)
9: **else** {the expression starts with a reference to a variable `$j`}
10: $A_i \leftarrow A_j$; $V_{\$i} \leftarrow V_{\$j}$; PathExpr($PE', A_i, V$); CreateJoin($A_i, A, V, V_{\$j}$ + '[' + concat. of (AxisStep + NameTest) in PE' + ' $\equiv$ ' + $V_{\$i}$ + ']')
11: **if** ForLetContext is a DVFunction and P is null **then**
12: $A \leftarrow$ '$\Sigma_{((V_{\$i}+\text{'.v,"'}+\text{last NameTest in }PE'+\text{'")),(/\texttt{prod\_root/1})}$(' + A + ')'
13: $V_{\$i} \leftarrow$ '/`group_root.A`[' + last NameTest in PE' + ']'
14: **for all** variable binding $V_{\$k}$ in V excluding $V_{\$i}$ **do**
15: replace initial '`/prod_root`' in $V_{\$k}$ with '`/group_root`'
16: **if** P is not null **then**
17: $A \leftarrow$ '$\sigma_{\text{Predicate}(P,V)}$(' + A + ')'
18: **if** PE'' is not null **then**
19: $\lambda \leftarrow$ '/`prod_root/2`'+concatenation of all (AxisStep + NameTest) in PE''
20: **if** ForLetContext is a DVFunction **then**
21: $A \leftarrow$ '$\Sigma_{((\lambda+\text{'.v,"'}+\text{last NameTest in }\lambda+\text{'"}),(/\texttt{prod\_root/1.k,"nodeIdentity"})),(/\texttt{prod\_root/1})}$(' +$A$ + ')'
22: $V_{\$i} \leftarrow$ '/`group_root.A`[' + last NameTest in λ + ']'
23: **for all** variable binding $V_{\$k}$ in V excluding $V_{\$i}$ **do**
24: replace initial '`/prod_root`'in $V_{\$k}$ with '`/group_root`'
25: **else**
26: $A \leftarrow$ '$\iota_{\text{"/prod\_root"(null,null,(/prod\_root/1,}\lambda\text{))}}$(' + A + ')'
27: $V_{\$i} \leftarrow$ '/`prod_root/2`'

Algorithm 6 Procedure SplitPathExpr

Input: a PathExpr PE, a string PE', a string P, a string PE''

1: **if** PE contains a predicate that reference a previously defined variable $j **then**
2: $P \leftarrow$ the predicate
3: $PE' \leftarrow$ the part of PE' before the predicate
4: $PE'' \leftarrow$ the part of PE' after the predicate
5: **else**
6: $PE' \leftarrow PE$

`for` clause could starts with a reference to another variable previously defined, i.e. the `for` clause could be of the form `$i in $j...`. In this case the expression corresponding to the referred variable is copied; then each step in the path expression is translated as before and the resulting expression is joined with the expression corresponding to the previous variable bindings. The join, which is produced by the procedure *CreateJoin* shown in Algorithm 9, is based on a strict equality comparison predicate.

If the reference to another variable is preceded by a calls to `distinct-values` (i.e. the `for` clause is of the form `$i in distinct-values($j...)`), we must extract the distinct values from the root elements of the trees in the second forest, while the first forest must be maintained unchanged. This result is obtained by adding a grouping operator; the group is done on the basis of the root elements' value of the trees in the second forest, and the subtrees `/prod_root/1`, corresponding to the trees from the first forest, are retained in the output. Such subtrees, which were previously reachable by following the path `/prod/root/...`, are now reachable by following the path `/group_root/...`; the variable binding list is accordingly updated.

If the expression contained in the `for` clause has not been split (because no predicate contains a reference to a previously defined variable), the translation is finished. Otherwise, a selection representing the predicate ([/author = $a] in our example) is added to the AFTX expression; also in this case the selection predicate is built by the function *Predicate*. Such a selection must be done after the product or join, because it refers two variables.

Algorithm 7 Procedure PathExpr

Input: a PathExpr PE, an AFTX expression A_i, a variable binding list V

1: $\lambda \leftarrow V_{\$i}$
2: **for all** Step in PE **do**
3: $\lambda \leftarrow \lambda +$ AxisStep $+$ NameTest
4: **if** there is some predicate **then**
5: $A_i \leftarrow$ '$\pi_{\lambda}($' $+ A_i +$ '$)$'
6: $V_{\$i} \leftarrow$ '$/$' $+$ NameTest
7: **for all** predicate P in Step **do**
8: $A_i \leftarrow$ '$\sigma_{\text{Predicate}(P,V)}($' $+ A_i +$ '$)$'
9: $\lambda \leftarrow$ '$/1$'
10: **if** this is the last step **then**
11: $V_{\$i} \leftarrow$ '$/$' $+$ NameTest
12: **if** no predicate has been found in the last step **then**
13: $A_i \leftarrow$ '$\pi_{\lambda}($' $+ A_i +$ '$)$'

Now, if the predicate is followed by a path expression (`/title` in our example), we must check if a `distinct-values` function is applied to the entire expression. If it is the case, a grouping operator is added. Grouping is based on the value of the expression (the value of `title` in our example) and on the identifier of the first subtree, i.e. the subtree that corresponds to AFTX expression built before starting to consider the `for` clause. This way a tree is built for each distinct pair (*result of the previously built AFTX expression, value of the variable binding being translated*).

If there is not a `distinct-values` function call, the path expression following the predicate is translated using the tree construction operator. In order to understand why it is not possible to use a simple projection, we must consider the situation we are in, which is shown graphically in Fig. 4.2. Tree (a) is an example of the result of the AFTX expression built up to now; the path expression we must translate is intended to maintain only the grey part of the right subtree (which is the subtree corresponding to the variable binding we are translating), leading to the resulting tree (b). There is no way to obtain

Algorithm 8 Procedure CreateProduct

Input: an AFTX expression A_i, an AFTX expression A, a variable binding list V

1: **if** A is the empty string **then**
2: $A \leftarrow A_i$
3: **else**
4: $A \leftarrow$ '(' $+ A +$ ' $\times$ ' $+ A_i +$ ')'
5: **for all** variable binding $\$k$ in the variable binding list **do**
6: $V_{\$k} \leftarrow$ '/`prod_root`' $+ V_{\$k}$

Algorithm 9 Procedure CreateJoin

Input: an AFTX expression A_i, an AFTX expression A, a variable binding list V, a join predicate P

1: $A \leftarrow$ '(' $+ A +$ ' $\bowtie_P$ ' $+ A_i +$ ')'
2: **for all** variable binding $\$k$ in the variable binding list **do**
3: $V_{\$k} \leftarrow$ '/`prod_root`' $+ V_{\$k}$

such a result using projection; using tree construction, instead, it is possible to build a tree having: 1) a root element named `prod_root`; 2) a left subtree corresponding to the left subtree of tree (a); 3) a right subtree corresponding to the grey part of the right subtree of tree (b).

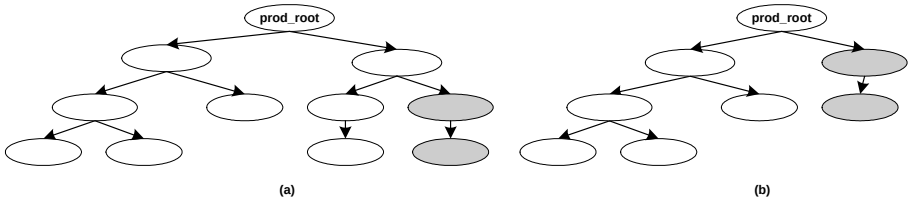


Figure 4.2: An input tree (a) and the tree that must be obtained (b).

The function *Predicate*, presented in Algorithm 10, must deal with two possible predicates: a comparison predicate between element or attribute values, or a predicate that checks the existence of an element or attribute. Its goal is to produce a selection predicate.

Algorithm 10 Function Predicate

Input: a Predicate P, a variable binding list V

Output: a selection predicate γ

1: **if** UnaryExpr2 contains a reference to a variable **then**
2: $\gamma \leftarrow$ the longest common path expression between the path expressions corresponding to the two variables $+$'['; $openCondition \leftarrow$ **false**
3: **else**
4: $\gamma \leftarrow$ ''; $openCondition \leftarrow$ **true**
5: **if** UnaryExpr1 is a PathExpr PE **then**
6: $\gamma \leftarrow \gamma + V_{\$i}$ (excluding the part already added in line 2) $+$UnaryExpr1$(PE, V, openCondition)$
7: **if** there is a GeneralComp **then**
8: $\gamma \leftarrow \gamma +$ '.v'
9: **else** {UnaryExpr1 is a CountPosFunction}
10: $\gamma \leftarrow \gamma + V_{\$i}$ (excluding the part already added in line 2) $+$UnaryExpr1$(PE, V, openCondition) +$ '.count' or '.pos'
11: **if** there is a GeneralComp **then**
12: $\gamma \leftarrow \gamma +$ GeneralComp
13: **if** UnaryExpr2 is a PathExpr PE' **then**
14: $\gamma \leftarrow \gamma +$ UnaryExpr1$(PE', V,$ **false**$) +$ '.v'
15: **else if** UnaryExpr2 is a VarRef PathExpr (\$j PE') **then**
16: $\gamma \leftarrow \gamma + V_{\$j}$ (excluding the part already added in line 2) $+$UnaryExpr1$(PE', V,$ **false**$) +$ '.v'
17: **else if** UnaryExpr2 is a Literal **then**
18: $\gamma \leftarrow \gamma +$ Literal
19: **else** {UnaryExpr2 is a CountPosFunction}
20: **if** the internal PathExpr starts with a variable $V_{\$j}$ **then**
21: $\gamma \leftarrow \gamma + V_{\$j}$ (excluding the part already added in line 2) $+$UnaryExpr1$(PE', V,$ **false**$) +$ '.count' or '.pos'
22: **else**
23: $\gamma \leftarrow \gamma +$ UnaryExpr1$(PE', V,$ **false**$) +$ '.count' or '.pos'
24: $\gamma \leftarrow \gamma +'\,]'$
25: **return** γ

The function, at line 5, checks the first unary expression; it can be:

- a path expression; function *UnaryExpr1*, shown in Algorithm 11, adds axis and name steps to the AFTX selection predicate, until a predicate is reached; then the function is called recursively on that predicate;
- a `count` or `position` function; the inner path expression is translated as previously seen; a `.count` or `.pos` is concatenated.

Algorithm 11 Function UnaryExpr1

Input: a UnaryExpr UE, a variable binding list V, a boolean $openCondition$

Output: a partial selection predicate γ

1: $\gamma \leftarrow$ ''
2: **for all** StepExpr **do**
3: $\gamma \leftarrow \gamma +$ AxisStep $+$ NameTest
4: **if** there is some predicate **then**
5: **for all** predicate P in UE **do**
6: $\gamma \leftarrow \gamma +$ '[' $+$ Predicate(P, V)
7: **if** P is not the last predicate **then**
8: $\gamma \leftarrow \gamma +$ ' `AND` '
9: $\gamma \leftarrow \gamma +$ ']'
10: **if** there is a FinalStepExpr F **then**
11: **if** F is of the form '@'+ NameTest **then**
12: **if** $openCondition$ **then**
13: $\gamma \leftarrow \gamma +$ '[.`A`[' $+$ NameTest $+$ ']'
14: **else**
15: $\gamma \leftarrow \gamma +$ '.`A`[' $+$ NameTest $+$ ']'
16: **else**
17: **if** $openCondition$ **then**
18: $\gamma \leftarrow \gamma +$ '['
19: **return** γ

The unary expression can be followed by a comparison operator and a second unary expression; in this case the comparison operator is concatenated to the predicate, then the second unary expression is translated. That expression can be, other than a path expression and a `count` or `position` function, the following:

- a literal; the literal is concatenated to the selection predicate;
- a reference to a variable followed by a path expression; the path expression is translated as previously seen; a leading path expression representing the referenced variable is added, by reading from the list of variable bindings.

A special care must be dedicated to the case where the second unary expression is a reference to a variable followed by a path expression. Consider first a predicate like `/price > 50`; it can be translated using a selection with predicate '/book/price[.v $>$ 50'] (supposing that the variable binding list element corresponding to the clause being translated has the value `/book`, e.g. if the complete `for` clause is `for $b in doc("bib.xml")/bib/book[/price > 50]`). Consider now a predicate involving a variable, like `/author = $a.name` (supposing that the variable binding list element for $b is `/prod_root/book` and the variable binding list element for $a is `/prod_root/author`); if we use the same technique, we would obtain a predicate '/prod_root/book/author[.v = prod_root/author/name.v]', but this solution is not correct. In fact the selection operator would first perform a temporal projection using the path expression prod_root/book/author, then it would search a prod_root/author /name path inside the obtained subtrees, thus leading to an empty result. In such cases, the correct selection predicate is therefore '/prod_root[/book/author.v = /author /name.v]'; lines 1–4 of *Predicate* take care of this issue.

Example 4.1 Consider again the basic `for` clause translation examples presented in Section 4.1.1; we now see that the presented algorithm behaves as expected. Let us start with the clause

```
for $i in doc("books.xml")/bib/book[@year=2000]/author
```

The function *ForClause* is called. Here are how the AFTX expression is built step by step; whenever the AFTX expression or some variable is modified, we show the function name, the line number and the new value. Moreover we show function or procedure calls.

- *ForClause*, 2: $V_{\$i} \leftarrow$ ‘’;
- *ForClause*, 2: calls to *SplitPathExpr*, obtaining $PE' \leftarrow$ ‘doc(“books.xml”)/bib /book[@year = 2000]/author’;
- *ForClause*, 4: $A_{\$i} \leftarrow$ ‘“books.xml”’;
- *ForClause*, 4: calls to *PathExpr*:
  - *PathExpr*, 1: $\lambda \leftarrow$ ‘’;
  - *PathExpr*, 3: $\lambda \leftarrow$ ‘/bib’;
  - *PathExpr*, 3: $\lambda \leftarrow$ ‘/bib/book’;
  - *PathExpr*, 5: $A_{\$i} \leftarrow$ ‘$\pi_{\texttt{/bib/book}}$(“books.xml”)’;
  - *PathExpr*, 5: $V_{\$i} \leftarrow$ ‘/book’;
  - *PathExpr*, 7: calls to *Predicate*:
    - *Predicate*, 4: $\gamma \leftarrow$ ‘’
    - *Predicate*, 6: calls to *UnaryExpr1*:
      - *UnaryExpr*, 1: $\gamma \leftarrow$ ‘’;
      - *UnaryExpr*, 13: $\gamma \leftarrow$ ‘[.A[year];
      - *UnaryExpr*, 19: return γ to *Predicate*;
    - *Predicate*, 6: $\gamma \leftarrow$ ‘/book[.A[year]’
    - *Predicate*, 8: $\gamma \leftarrow$ ‘/book[.A[year].v’
    - *Predicate*, 12: $\gamma \leftarrow$ ‘/book[.A[year].v =’
    - *Predicate*, 18: $\gamma \leftarrow$ ‘/book[.A[year].v = 2000’
    - *Predicate*, 24: $\gamma \leftarrow$ ‘/book[.A[year].v = 2000]’
    - *Predicate*, 25: return γ to *PathExpr*;

  - *PathExpr*, 8: $A_{\$i} \leftarrow$ '$\sigma_{/\texttt{book}[.\texttt{A}[\texttt{year}].\texttt{v}=2000]}(\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”}))$';
  - *PathExpr*, 9: $\lambda \leftarrow$ '$/\texttt{1}$';
  - *PathExpr*, 3: $\lambda \leftarrow$ '$/\texttt{1/author}$';
  - *PathExpr*, 11: $V_{\$i} \leftarrow$ '$/\texttt{author}$';
  - *PathExpr*, 13: $A_{\$i} \leftarrow$ '$\pi_{/\texttt{1/author}}(\sigma_{/\texttt{book}[.\texttt{A}[\texttt{year}].\texttt{v}=2000]}(\pi_{/\texttt{bib/book}}($ “books.xml”$)))$';

- *ForClause*, 11: calls to *CreateProduct*;
  - *CreateProduct*, 2: $A \leftarrow$ '$\pi_{/\texttt{1/author}}(\sigma_{/\texttt{book}[.\texttt{A}[\texttt{year}].\texttt{v}=2000]}(\pi_{/\texttt{bib/book}}($ “books.xml”$)))$'.

The translation has been completed, and the result is that expected. Moreover, the variable binding list now contains one item, stating the variable $i is reachable following the path `/author`.

Example 4.2 Consider now the following `for` clause, that involves two variable bindings:

```
for $i in doc("books.xml")/bib/book[@year=2000],
    $j in doc("authors.xml")/authors/author[/first="John"]
```

The two variable binding are translated in the same way as in the previous example, thus leading to this partial result:

- $A \leftarrow$ '$\sigma_{/\texttt{book}[.\texttt{A}[\texttt{year}].\texttt{v}=2000]}(\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”}))$';
- $V_{\$i} \leftarrow$ '$/\texttt{book}$';
- $A_{\$j} \leftarrow$ '$\sigma_{/\texttt{author/first}[.\texttt{v}=\text{“}\texttt{John}\text{”}]}(\pi_{/\texttt{authors/author}}(\text{“}\texttt{authors.xml}\text{”}))$';
- $V_{\$j} \leftarrow$ '$/\texttt{author}$'.

Now at line 11 *ForClause* calls CreateProduct:

- *CreateProduct*, 4: $A \leftarrow (\sigma_{/\texttt{book}[.\texttt{A}[\texttt{year}].\texttt{v}=2000]}(\pi_{/\texttt{bib}/\texttt{book}}(\text{“books.xml”})) \times \sigma_{/\texttt{author}/\texttt{first}[.\texttt{v}=\text{“}\texttt{John}\text{”}]}(\pi_{/\texttt{authors}/\texttt{author}}(\text{“}\texttt{authors.xml}\text{”})))$';
- *CreateProduct*, 6: $V_{\$i} \leftarrow$ '$/\texttt{prod\_root}/\texttt{book}$';
- *CreateProduct*, 6: $V_{\$j} \leftarrow$ '$/\texttt{prod\_root}/\texttt{author}$'.

The translation has been successfully completed. Note that, after applying the product operator, the paths in the variable binding list have been correctly updated by adding a heading `/prod_root`.

Example 4.3 Consider now the case of a `for` clause with multiple variable bindings where a variable binding refers to another variable, like the following one:

```
for $i in doc("books.xml")/bib/book[@year=2000],
    $j in $i/author[/first="Serge"]
```

The first variable binding is translated as usual; the second is translated as follows:

- ForClause, 13: $A_{\$j} \leftarrow$ '$\sigma_{/\texttt{book}[.\texttt{A}[\texttt{year}].\texttt{v}=2000]}(\pi_{/\texttt{bib}/\texttt{book}}(\text{“}\texttt{books.xml}\text{”}))$';
- ForClause, 13: calls to PathExpr, that does the following:
  - $A_{\$j} \leftarrow$ '$\sigma_{/\texttt{author}/\texttt{first}[.\texttt{v}=\text{“}\texttt{Serge}\text{”}]}(\pi_{/\texttt{book}/\texttt{author}}(\sigma_{/\texttt{book}[.\texttt{A}[\texttt{year}].\texttt{v}=2000]}(\pi_{/\texttt{bib}/\texttt{book}}(\text{“}\texttt{books.xml}\text{”}))))$';
  - $V_{\$j} \leftarrow$ '$/\texttt{author}$';
- *ForClause*, 13: calls to *CreateJoin*:
  - *CreateJoin*, 1: $A \leftarrow$ '$(\sigma_{/\texttt{book}[.\texttt{A}[\texttt{year}].\texttt{v}=2000]}(\pi_{/\texttt{bib}/\texttt{book}}(\text{“}\texttt{books.xml}\text{”})) \bowtie_{/\texttt{book}[/\texttt{author}\equiv/\texttt{author}]} \sigma_{/\texttt{author}/\texttt{first}[.\texttt{v}=\text{“}\texttt{Serge}\text{”}]}(\pi_{/\texttt{book}/\texttt{author}}(\sigma_{/\texttt{book}[.\texttt{A}[\texttt{year}].\texttt{v}=2000]}(\pi_{/\texttt{bib}/\texttt{book}}(\text{“}\texttt{books.xml}\text{”})))))$';
  - *CreateJoin*, 3: $V_{\$i} \leftarrow$ '$/\texttt{prod\_root}/\texttt{book}$';
  - *CreateJoin*, 3: $V_{\$j} \leftarrow$ '$/\texttt{prod\_root}/\texttt{author}$'.

Example 4.4 Consider now a `for` clause with a `distinct-values`, like the following:

```
for $i in distinct-values(doc("books.xml")/bib/book/author
    /last)
```

The `DVContext` is translated as usual, leading to the partial result:

- $A_{\$i} \leftarrow$ '$\pi_{\texttt{/bib/book/author/last}}$(“books.xml”)';
- $V_{\$i} \leftarrow$ '/last'.

Then the translation goes on as follows:

- *ForClause*, 9: $A_{\$i} \leftarrow$ '$\nu_{(\texttt{/last.v,“last”})}(\pi_{\texttt{/bib/book/author/last}}$(“books.xml”))';
- *ForClause*, 10: $V_{\$i} \leftarrow$ 'group_root.A[last]';
- *ForClause*, 11: calls to CreateProduct, obtaining $A \leftarrow A_i$.

The procedure *LetClause*, presented in Algorithm 12, is quite similar to the procedure *ForClause*. It should be noted, however, that line 9 adds a leading tree construction operator, in order to create an AFTX expression that returns a single tree rooted at a `let_root` element; the same line also modifies the variable binding list element just created, by adding a leading `/let_root`.

Moreover, line 1 tells to cycle over each variable binding, except for those that simply build an alias for a complex expression. Finally, it must be pointed out that line 12 builds a copy of the expression corresponding to a referenced variable excluding a possible ι“let_root”(null, null, null) leading expression.

Example 4.5 Consider the following partial query:

```
for $a in doc("authors.xml")//author
let $b := doc("books.xml")//book
```

The `for` clause is translated as usual, thus leading the following partial result:

Algorithm 12 Procedure LetClause

Input: a `let` clause L, an AFTX expression A, a variable binding list V

1: **for all** var. binding in L relative to a var. $\$i$ (not of the form `let $i:=$j/`λ) **do**
2: $V_{\$i} \leftarrow$ '' {create a new variable binding element}
3: **if** DVContext starts with `fn:doc("docname")` **then**
4: $A_i \leftarrow$ '"`docname`"'
5: PathExpr(PE, A_i, V) {PE is the PathExpr in DVContext}
6: **if** ForLetContext is a DVFunction **then**
7: $A_i \leftarrow$ '$\nu_{(V_{\$i}+\text{'.v,"'}+V_{\$i} \text{ without heading '/'}+\text{'")'}}$(' $+ A_i +$ ')'
8: $V_{\$i} \leftarrow$ '/`group_root.A`[' $+ V_{\$i}$ without heading '/' $+$ ']'
9: $A_i \leftarrow$ '$\iota_{\text{"let\_root"(null,null,null)}}$(' $+ A_i +$ ')'; $V_{\$i} \leftarrow$ '/`let_root`' $+ V_{\$i}$
10: CreateProduct(A_i, A, V)
11: **else** {the expression starts with a reference to a variable `$j`}
12: $A_i \leftarrow A_j$ {copy the AFTX expression built for $j, excluding ι}
13: $V_{\$i} \leftarrow V_{\$j}$
14: PathExpr(PE, A_i, V)
15: $\lambda' \leftarrow$ concatenation of all AxisStep + NameTest inPE
16: CreateJoin$(A_i, A, V, V_{\$j} +$ '[' $+ \lambda' +$ ' $\equiv$ ' $+ V_{\$i} +$ ']');

- $A \leftarrow$ '$\pi_{//\texttt{author}}(\text{“}\texttt{authors.xml}\text{”})$';
- $V_{\$a} \leftarrow$ '$/\texttt{author}$'.

Then the procedure *LetClause* is called. The DVContext is initially translated as in case of `for` clauses, leading to:

- $A_{\$b} \leftarrow$ '$\pi_{//\texttt{book}}(\text{“}\texttt{books.xml}\text{”})$';
- $V_{\$b} \leftarrow$ '$/\texttt{book}$'.

Then line 9 adds the tree construction operator:

- $A_{\$b} \leftarrow$ '$\iota_{\text{“}\texttt{let\_root}\text{”}(\texttt{null},\texttt{null},\texttt{null})}(\pi_{//\texttt{book}}(\text{“}\texttt{books.xml}\text{”}))$';
- $V_{\$b} \leftarrow$ '$/\texttt{let\_root}/\texttt{book}$'.

Finally, a product is created as usual, leading to the final result:

- $A \leftarrow$ '$(\pi_{//\texttt{author}}(\text{“}\texttt{authors.xml}\text{”}) \times \iota_{\text{“}\texttt{let\_root}\text{”}(\texttt{null},\texttt{null},\texttt{null})}(\pi_{//\texttt{book}}(\text{“}\texttt{books.xml}\text{”})))$';
- $V_{\$a} \leftarrow$ '$/\texttt{prod\_root}/\texttt{author}$';
- $V_{\$b} \leftarrow$ '$/\texttt{prod\_root}/\texttt{let\_root}/\texttt{book}$'.

The procedure *WhereClause*, shown in Algorithm 13, cycles over each single clause. These clauses can be a comparison between two expressions or a quantified expression.

In the first case, if the comparison expression does not refer to a variable defined in an outer FLWOR expression, the translation is done by simply applying a selection to the AFTX expression built up to now. The selection predicate is returned by the function *Predicate2*.

In the second case, the first thing to do is to create a new variable binding list element. If the quantified expression contains an existential quantifier and the expression does not refer to a variable defined in an outer FLWOR expression, a selection is then applied to the AFTX expression built up to now. If the quantified expression contains instead a universal quantifier, a difference operator is applied between the AFTX expression built up to

Algorithm 13 Procedure WhereClause

Input: a `where` clause W, an AFTX expression A, a variable binding list V

1: **for all** clause w_i **do**
2: **if** the clause is a ComparisonExpr CE **then**
3: **if** CE refers to a variable defined in an outer FLWOR expression **then**
4: CreateOuterJoin(CE, A, V)
5: **else**
6: $A \leftarrow$ '$\sigma_{\text{Predicate2}(CE,V)}($' $+ A +$ ')'
7: **else** {the clause is a QuantifiedExpr `some`/`every $i in $j`λ}
8: $V_{\$_i} \leftarrow V_{\$_j} + \lambda$;
9: **if** the QuantifiedExpr is of the form `some $i in $j`λ **then**
10: **if** CE refers to a variable defined in an outer FLWOR expression **then**
11: CreateOuterJoin(CE, A, V)
12: **else**
13: $A \leftarrow$ '$\sigma_{\text{Predicate2}(CE,V)}($' $+ A +$ ')'
14: **else** {the QuantifiedExpr is of the form `every $i in $j`λ}
15: $A \leftarrow$ '$A - \sigma_{\text{Predicate2inv}(CE,V)}($' $+ A +$ ')'

now and the result of a selection on that expression, using an inverted selection predicate. The function *Predicate2Inv*, which builds such inverted predicate, is not shown. However its behavior should be clear; for example, if the quantified expression is `every $a in $b//author satisfies ($a/name = "John")`, *Predicate2Inv* create a predicate corresponding to `NOT $b//author/name = "John"`.

Either if the clause is a comparison expression or if it is a quantified expression, the expression could refer to a variable defined in an outer FLWOR expression, like in the following example:

```
for $i in ...
...
return
  {
    for $j in ...
    where $j/...=$i/...
    ...
  }
```

In such cases translating the clause using the selection predicate is not correct; in fact each element bound to `$i` should be part of the result, even if there are no elements bound to `$j` that satisfy the `where` clause. What we need is a sort of *left outer join*. Such a join is created, using the technique already discussed in Section 4.1.1, by procedure *CreateOuterJoin*, shown in Algorithm 15.

The function *Predicate2*, shown in Algorithm 14, is almost identical to the function *Predicate* already presented; consequently we do not discuss it here.

Example 4.6 Consider the partial query

```
for $i in doc("books.xml")/bib/book[@year=2000]
where $i/price > 50
```

The `for` clause is translated as usual, leading to the following partial result:

Algorithm 14 Function Predicate2

Input: a ComparisonExpr CE, a variable binding list V

Output: a selection predicate γ

1: **if** both UnaryExpr are VarRef PathExpr or CountPosFunction **then**
2: $\gamma \leftarrow$ the longest common path expression between the path expressions corresponding to the two variables $+$'['; $openCondition \leftarrow$ **false**
3: **else**
4: $\gamma \leftarrow$ ''; $openCondition \leftarrow$ **true**
5: **if** the first UnaryExpr is a VarRef PathExpr (\$i PE) **then**
6: $\gamma \leftarrow \gamma + V_{\$i}$ (excluding the part already added in line 2) $+$UnaryExpr1$(PE, V, openCondition) +$ '.v'
7: **else if** the first UnaryExpr is a CountPosFunction **then**
8: $\gamma \leftarrow \gamma + V_{\$i}$ (excluding the part already added in line 2) $+$UnaryExpr1$(PE, V, openCondition) +$ '.count' or '.pos'
9: **else** {the first UnaryExpr is a Literal}
10: $\gamma \leftarrow \gamma +$ Literal
11: $\gamma \leftarrow \gamma +$ GeneralComp
12: **if** the second UnaryExpr is a VarRef PathExpr (\$j PE') **then**
13: $\gamma \leftarrow \gamma + V_{\$j}$ (excluding the part already added in line 2) $+$UnaryExpr1$(PE', V,$ **false**$) +$ '.v'
14: **else if** the second UnaryExpr is a CountPosFunction **then**
15: $\gamma \leftarrow \gamma + V_{\$j}$ (excluding the part already added in line 2) $+$UnaryExpr1$(PE', V,$ **false**$) +$ '.count' or '.pos'
16: **else** {the second UnaryExpr is a Literal}
17: $\gamma \leftarrow \gamma +$ Literal
18: $\gamma \leftarrow \gamma +$ ']'
19: **return** γ

Algorithm 15 Procedure CreateOuterJoin

Input: a ComparisonExpr CE, an AFTX expression A, a variable binding list V

if the first UnaryExpr refers to a variable \$out defined in a outer FLWOR expression **then**
    $UEOut \leftarrow$ first UnaryExpr; $UEIn \leftarrow$ second UnaryExpr
else
    $UEOut \leftarrow$ second UnaryExpr; $UEIn \leftarrow$ first UnaryExpr
for all StepExpr in $UEIn$ **do**
    if this is the last step **then**
        $\gamma' \leftarrow \gamma +$ AxisStep + NameStep
        $\gamma \leftarrow \gamma +$ '[`NOT`'
    $\gamma \leftarrow \gamma +$ AxisStep + NameStep
if $UEIn$ is a VarRef PathExpr that uses an inner variable \$in **then**
    $\gamma \leftarrow V_{\$\text{in}} + \gamma +$ '.`v`'
else {$UEIn$ is a CountPosFunction that uses an inner variable \$in}
    $\gamma \leftarrow V_{\$\text{in}} + \gamma +$ '.`count`' or '.`pos`'
$\gamma \leftarrow \gamma +$ GeneralComp
for all StepExpr in $UEOut$ **do**
    $\gamma'' \leftarrow \gamma'' +$ AxisStep + NameStep
if $UEOut$ is a VarRef PathExpr **then**
    $\gamma \leftarrow \gamma + V_{\$\text{out}} + \gamma'' +$ '.`v`]'
else if $UEOut$ is a CountPosFunction **then**
    $\gamma \leftarrow \gamma + V_{\$\text{out}} + \gamma'' +$ '.`count`]' or '.`pos`'
$g \leftarrow$ '((/`prod_root`/1.`k`, "`treeIdentity`")), (/`prod_root`/1, /`prod_root`/2)'
$d \leftarrow$ '/`group_root`/ $*$ [.`k` $=$ /`group_root`.`A`["`treeIdentity`"].`vAND`.`pos` > 1]
$A \leftarrow$ '$\delta_d(\Sigma_g(\delta_{V_{\$in}[\gamma']}(\delta_\gamma($' $+ A +$ '$)$'
for all variable binding $V_{\$k}$ in V **do**
    replace initial '/`prod_root`'in $V_{\$k}$ with '/`group_root`'

- $A \leftarrow$ '$\sigma_{/\texttt{book}[.\texttt{A}[\texttt{year}].\texttt{v}=2000]}(\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”}))$';
- $V_{\$i} \leftarrow$ '$/\texttt{book}$'.

The `where` clause is translated as follows:

- *WhereClause*, 6: calls to *Predicate2*;
  - *Predicate2*, 4: $\gamma \leftarrow$ '';
  - *Predicate2*, 6: calls to *UnaryExpr1*, that returns '$/\texttt{price}[$';
  - *Predicate2*, 6: $\gamma \leftarrow$ '$/\texttt{book/price}[.\texttt{v}$';
  - *Predicate2*, 11: $\gamma \leftarrow$ '$/\texttt{book/price}[.\texttt{v} >$ ';
  - *Predicate2*, 17: $\gamma \leftarrow$ '$/\texttt{book/price}[.\texttt{v} > 50$';
  - *Predicate2*, 18: $\gamma \leftarrow$ '$/\texttt{book/price}[.\texttt{v} > 50]$';
  - *Predicate2*, 19: returns γ to *WhereClause*
- WhereClause, 6: $A \leftarrow$ '$\sigma_{/\texttt{book/price}[.\texttt{v}>50]}(\sigma_{/\texttt{book}[.\texttt{A}[\texttt{year}].\texttt{v}=2000]}($ $\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”})))$.

The procedure OrderByClause, shown in Algorithm 16, creates an ordering predicate by defining an AFTX ordering directive for each order specification. Each ordering specification is composed by the path expression of the variable binding list element corresponding to a variable, followed by an optional path expression, followed by an optional attribute name, followed by the element property $.v$, followed by the ordering direction ASC or DESC. A heading ordering operator, using the ordering predicate just built, is then added to the AFTX expression built before the `order by` clause.

Example 4.7 Consider the partial query

```
for $i in doc("books.xml")/bib/book
order by $i/title ascending
```

The `for` clause is translated as usual. The `order by` clause is translated as follows:

Algorithm 16 Procedure OrderByClause

Input: a `order by` clause O, an AFTX expression A, a variable binding list V

1: $o \leftarrow$ ‘’
2: **for all** OrderSpec O_i in O that refers to a variable $i **do**
3: $o \leftarrow o + V_{\$i}$
4: **for all** (AxisStep QName) **do**
5: $o \leftarrow o +$ AxisStep $+$ QName
6: **if** there is a (“@”QName) **then**
7: $o \leftarrow o +$ ‘.`A`[’ $+$ QName $+$ ‘]’
8: $o \leftarrow o +$ ‘$.v$’
9: **if** ordering direction is “`descending`” **then**
10: $o \leftarrow o +$ ‘`DESC`’
11: **else** {ordering direction is “`descending`”or is not present}
12: $o \leftarrow o +$ ‘`ASC`’
13: **if** O_i is not the last OrderSpec **then**
14: $o \leftarrow o +$ ‘, ’
15: $A \leftarrow$ ‘o’$_o +$ ‘(’ $+ A +$ ‘)’

- OrderByClause, 1: $o \leftarrow$ '';
- OrderByClause, 3: $o \leftarrow$ '/book';
- OrderByClause, 5: $o \leftarrow$ '/book/title';
- OrderByClause, 8: $o \leftarrow$ '/book/title.v';
- OrderByClause, 12: $o \leftarrow$ '/book/title.v ASC';
- OrderByClause, 15: $A \leftarrow$ '$o_{\texttt{/book/title.v ASC}}(\pi_{\texttt{/bib/book}}(\text{“}\texttt{books.xml}\text{”}))$';

Example 4.8 Consider the partial query

```
for $i in doc("books.xml")/bib/book/author/last,
    $j in doc("books.xml")/bib/book
where $i=$j/author/last
order by $i ascending, $j/title ascending
```

The `for` clause is translated as usual, thus leading to the following partial result:

- $A \leftarrow$ '$\pi_{\texttt{/bib/book/author/last}}(\text{“}\texttt{books.xml}\text{”}) \times \pi_{\texttt{/bib/book}}(\text{“}\texttt{books.xml}\text{”})$';
- $V_{\$i} \leftarrow$ '/prod_root/last';
- $V_{\$j} \leftarrow$ '/prod_root/book'.

Now we show how the `where` and `order by` clauses are translated:

- *WhereClause*, 6: calls to *Predicate2*
  - *Predicate2*, 2: $\gamma \leftarrow$ '/prod_root[';
  - *Predicate2*, 6: calls to *UnaryExpr1*, which returns the empty string;
  - *Predicate2*, 6: $\gamma \leftarrow$ '/prod_root[/last.v';
  - *Predicate2*, 11: $\gamma \leftarrow$ '/prod_root[/last.v =';
  - *Predicate2*, 13: calls to *UnaryExpr1*, which returns '/author/last';

  - *Predicate2*, 13: $\gamma \leftarrow$ '/prod_root[/last.v = /book/author/last.v';
  - *Predicate2*, 18: $\gamma \leftarrow$ '/prod_root[/last.v = /book/author/last.v]';
  - *Predicate2*, 19: return γ to *WhereClause*;

- *WhereClause*, 6: $A \leftarrow$ '$\sigma_{\text{/prod\_root[/last.v=/book/author/last.v]}}($ $\pi_{\text{/bib/book/author/last}}(\text{“books.xml”}) \times \pi_{\text{/bib/book}}(\text{“books.xml”}))$';
- *OrderByClause*, 1: $o \leftarrow$ '';
- *OrderByClause*, 3: $o \leftarrow$ '/prod_root/last';
- *OrderByClause*, 8: $o \leftarrow$ '/prod_root/last.v';
- *OrderByClause*, 12: $o \leftarrow$ '/prod_root/last.v ASC';
- *OrderByClause*, 14: $o \leftarrow$ '/prod_root/last.v ASC,';
- *OrderByClause*, 3: $o \leftarrow$ '/prod_root/last.v ASC, /prod_root/book';
- *OrderByClause*, 5: $o \leftarrow$ '/prod_root/last.v ASC, /prod_root/book/title';
- *OrderByClause*, 8: $o \leftarrow$ '/prod_root/last.v ASC, /prod_root/book/title.v';
- *OrderByClause*, 12: $o \leftarrow$ '/prod_root/last.v ASC, /prod_root/book/title.v ASC';
- *OrderByClause*, 15: $A \leftarrow$ '$o_{\text{/prod\_root/last.v ASC,/prod\_root/book/title.v ASC}}($ $\sigma_{\text{/prod\_root[/last.v=/book/author/last.v]}}($ $\pi_{\text{/bib/book/author/last}}(\text{“books.xml”}) \times \pi_{\text{/bib/book}}(\text{“books.xml”})))$'.

The function *ReturnClause*, shown in Algorithm 17, first checks if the clause contains some inner FLWOR expression. If this is the case, the function *FLWORExpr* is called, passing as input:

- the inner FLWOR expression;
- the AFTX expression built up to now;

- the variable binding list;
- the boolean value *false*, stating that *FLWORExpr* must build a tree construction predicate but it must not apply a tree construction operator.

Algorithm 17 Function ReturnClause

Input: a Constructor C, an AFTX expression A, a variable binding list V

Output: a tree constructor predicate t

1: **if** C contains somewhere some FLWOREXPR **then**
2: **for all** FLWORExpr e_i **do**
3: $t_i \leftarrow$ FLWORExpr$(e_i, A, V,$ **false**$)$
4: **if** C is a DirElemConstructor **then**
5: $t \leftarrow$ DirElemConstructor$(C, V, T)\{T$ is the list of computed $t_i\}$
6: **else** $\{C$ is a list of EnclosedExpr$\}$
7: **for all** EnclosedExpr E **do**
8: **if** E is a PathExpr2 **then**
9: $t \leftarrow t +$ PathExpr2$(E, V,$ **false**$)$
10: **else** $\{E$ is the FLWORExpr $e_i\}$
11: $t \leftarrow t + t_i$
12: **if** E is not the last EnclosedExpr **then**
13: $t \leftarrow t +$ ', '
14: **return** t

Then the function checks if the constructor is a direct element constructor (e.g. `<result>...</result>`) or a list of enclosed expressions (i.e. something of the form `{...}`). In the first case the function *DirElemConstructor* is called, passing as input the constructor, the variable binding list, and the list of tree construction predicates that have been built calling *FLWORExpr* for the inner FLWOR expressions. In the second case, the enclosed expression can be:

- a path expression (e.g. `{$b/title}`): the function *PathExpr2*, which is shown in Algorithm 18, builds the corresponding tree construction predicate, which is appended to the predicate built up to now;

- a FLWOR expression: the tree construction predicate built by the previously called *FLWORExpr* function is appended to the predicate built up to now.

Algorithm 18 Function PathExpr2

Input: a PathExpr2 P, a variable binding list V, a boolean $isValue$

```
1:  s ← V_VarRef
2:  for all (AxisStep QName) do
3:      s ← s + AxisStep + QName
4:  if there is a FinalStepExpr then
5:      if FinalStepExpr is of the form "@"NameTest then
6:          s ← s + '.A[' + NameTest + '].v'
7:      else {FinalStepExpr is of the form '/text()'}
8:          s ← s + '.v'
9:  else
10:     if isValue then
11:         s ← s + '.v'
12: return s
```

The function *DirElemConstructor*, shown in Algorithm 19, first checks if the constructor contains some attributes. If this is the case, for each of them the function *DirAttribute*, which is shown in Algorithm 20, builds a string, which will be the A part of the resulting tree construction operator $n(v, a, c)$. Remember that n is the name of the element, v is the value, a is the list of attributes, and c is the list of child elements.

Then *DirElemConstructor* analyzes the content of the constructor being translated. It is a list of:

- direct element constructors: for each of them the function *DirElemConstructor* is called recursively, and its result is added, using the procedure *AddChild* shown in Algorithm 21, to the list c of child elements;

- enclosed expressions: each of them can be:

Algorithm 19 Function DirElemConstructor

Input: a DirElemConstructor C, a variable binding list V, a list of tree construction predicate T

Output: a tree constructor predicate t

1: **if** there is at least one DirAttribute **then**
2: **for all** DirAttribute D **do**
3: $a \leftarrow a +$ DirAttribute(D, V)
4: **if** DirAttribute is not the last one **then**
5: $a \leftarrow a +$ ‘, ’
6: $a \leftarrow$ ‘(’ $+ a +$ ‘)’
7: **else**
8: $a \leftarrow$ ‘`null`’
9: **for all** DirElemContent C_i **do**
10: **if** C_i is a DirElemConstructor **then**
11: AddChild$(c,$ DirElemConstructor$(C_i, A, V))$
12: **else if** C_i is an EnclosedExpr **then**
13: **if** C_i is the FLWORExpr e_i **then**
14: AddChild(c, t_i)
15: **else** $\{C_i$ is a PathExpr2$\}$
16: **if** C_i contains a FinalStepExpr **then**
17: $v \leftarrow v +$ PathExpr2$(C_i, V,$ **false**$)$
18: **else**
19: AddChild$(c,$ PathExpr2$(C_i, V,$ **false**$))$
20: **else** $\{C_i$ is a literal$\}$
21: $v \leftarrow v + C_i$
22: **if** v is the empty string **then**
23: $v \leftarrow$ ‘`null`’
24: **if** c is the empty string **then**
25: $c \leftarrow$ ‘`null`’
26: **else**
27: $c \leftarrow c +$ ‘)’
28: $t \leftarrow$ ‘“’ $+$ QName $+$ ‘”(’ $+ v +$ ‘, ’ $+ a +$ ‘, ’ $+ c +$ ‘)’
29: **return** t

Algorithm 20 Function DirAttribute

Input: a DirAttribute D, a variable binding list V

Output: an attribute construction specification a

1: **if** DirAttributeValue is a PathExpr2 **then**
2: $a \leftarrow$ PathExpr2(DirAttributeValue, V, **true**)
3: **else** {DirAttributeValue is a Literal}
4: $a \leftarrow$ '"' + Literal + '"'
5: $a \leftarrow$ '(" + QName + '",' + a + ')'

  - a FLWOR expression: the tree construction predicate resulting from the previous call to *FLWORExpr* for that FLWOR expression is added to the list c of child elements;
  - a path expression: the result of *PathExpr2* is either added to the list c of child elements (if the XQuery path expression results in an element) or appended to the value v of the element (if the XQuery path expression results in a value, e.g. if it is {`$b/title/text()`});

- literals: each of them is appended to the value v of the element.

Finally the complete tree construction predicate is built, using the values a, v, and c just computed.

Algorithm 21 Procedure AddChild

Input: a child element list specification c, a child element specification c'

1: **if** c is an empty string **then**
2: $c \leftarrow$ '(' + c'
3: **else**
4: $c \leftarrow c +$ ',' + c'

Example 4.9 Consider the following XQuery expression:

```
for $i in doc("books.xml")/bib/book/author
```

```
order by $i/last, $i/first
return {$i/last}
```

The `for` and `order by` clauses are translated as previously seen, leading to the partial result

- $A \leftarrow$ '$o_{\texttt{/author/last.v ASC,/author/first.v ASC}}(\pi_{\texttt{/bib/book/author}}(\text{“}\texttt{books.xml}\text{”}))$';
- $V_{\$_i} \leftarrow$ '$/\texttt{author}$'.

Now we show how the `return` clause is translated:

- *FLWORExpr*, 9: calls to *ReturnClause*:
  - *ReturnClause*, 9: calls to *PathExpr2*:
    - *PathExpr2*, 1: $s \leftarrow$ '$/\texttt{author}$';
    - *PathExpr2*, 3: $s \leftarrow$ '$/\texttt{author}/\texttt{last}$';
    - *PathExpr2*, 12: return s to *ReturnClause*;
  - *ReturnClause*, 9: $t \leftarrow$ '$/\texttt{author}/\texttt{last}$';
  - *ReturnClause*, 14: return t to *FLWORExpr*;
- *FLWORExpr*, 9: $t \leftarrow$ '$/\texttt{author}/\texttt{last}$';
- *FLWORExpr*, 11: $A \leftarrow$ '$\iota_{\texttt{/author/last}}(o_{\texttt{/author/last.v ASC,/author/first.v ASC}}($ $\pi_{\texttt{/bib/book/author}}(\text{“}\texttt{books.xml}\text{”})))$'.

Note that the resulting AFTX expression is correct, even if, as already seen in Section 4.1.1, the same result could be obtained by using the projection operator instead of the tree construction operator.

Example 4.10 Consider the following XQuery expression:

```
for $i in doc("books.xml")/bib/book
return {$i/author}{$i/editor}
```

The `for` clause is translated as usual, leading to the partial result

- $A \leftarrow \pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”})$;
- $V_{\$i} \leftarrow$ ‘$/\texttt{book}$’.

Now we show how the `return` clause is translated:

- *FLWORExpr*, 9: calls to *ReturnClause*:
  - *ReturnClause*, 9: calls to *PathExpr2*:
    - *PathExpr2*, 1: $s \leftarrow$ ‘$/\texttt{book}$’;
    - *PathExpr2*, 3: $s \leftarrow$ ‘$/\texttt{book}/\texttt{author}$’;
    - *PathExpr2*, 12: return s to *ReturnClause*;
  - *ReturnClause*, 9: $t \leftarrow$ ‘$/\texttt{book}/\texttt{author}$’;
  - *ReturnClause*, 13: $t \leftarrow$ ‘$/\texttt{book}/\texttt{author},$’;
  - *ReturnClause*, 9: calls to *PathExpr2*:
    - *PathExpr2*, 1: $s \leftarrow$ ‘$/\texttt{book}$’;
    - *PathExpr2*, 3: $s \leftarrow$ ‘$/\texttt{book}/\texttt{editor}$’;
    - *PathExpr2*, 12: return s to *ReturnClause*;
  - *ReturnClause*, 9: $t \leftarrow$ ‘$/\texttt{book}/\texttt{author}, /\texttt{book}/\texttt{editor}$’;
  - *ReturnClause*, 14: return t to *FLWORExpr*;
- *FLWORExpr*, 9: $t \leftarrow$ ‘$/\texttt{book}/\texttt{author}, /\texttt{book}/\texttt{editor}$’;
- *FLWORExpr*, 11: $A \leftarrow$ ‘$\iota_{/\texttt{book/author},/\texttt{book/editor}}(\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”}))$’.

Example 4.11 Consider the following XQuery expression:

```
for $i in doc("books.xml")/bib/book
return <book title={$i/title}></book>
```

The `for` clause as in the previous example. Now we show how the `return` clause is translated:

- *ReturnClause*, 5: calls to *DirElemConstructor*:

  - *DirElemConstructor*, 3: calls to *DirAttribute*:
    - *DirAttribute*, 2: calls to *PathExpr2*, obtaining $a \leftarrow$ '/book/title.v';
    - *DirAttribute*, 5: $a \leftarrow$ '("title", /book/title.v)'
  - *DirElemConstructor*, 3: $a \leftarrow$ '("title", /book/title.v)';
  - *DirElemConstructor*, 6: $a \leftarrow$ '(("title", /book/title.v))';
  - *DirElemConstructor*, 23: $v \leftarrow$ 'null';
  - *DirElemConstructor*, 25: $c \leftarrow$ 'null';
  - *DirElemConstructor*, 28: $t \leftarrow$ '"book"(null, (("title", /book/title.v)), null)';

- *ReturnClause*, 5: $t \leftarrow$ '"book"(null, (("title", /book/title.v)), null)';
- *FLWORExpr*, 11: $A \leftarrow$ '$\iota_{\text{"book"(null,(("title",/book/title.v)),null)}}(\pi_{\text{/bib/book}}($ "books.xml"))'.

Example 4.12 Consider the following XQuery expression:

```
for $i in distinct-values(doc("books.xml")/bib/book/author
    /last),
  $j in doc("books.xml")/bib/book
where $i=$j/author/last
return  <author name={$i}>
          <book>{$j/title/text()}</book>
        </author>
```

The first variable binding is translated as seen in Example 4.4. The translation goes on as follows:

- the second variable binding is translated as usual, thus leading to the partial result:
  - $A \leftarrow$ '$(\nu_{\text{(/last.v,"last")}}(\pi_{\text{/bib/book/author/last}}(\text{"books.xml"})) \times \pi_{\text{/bib/book}}(\text{"books.xml"}))$';

  - $V_{\$i} \leftarrow$ '/prod_root/group_root.A[last]';
  - $V_{\$j} \leftarrow$ '/prod_root/book';

- the `where` clause is translated as usual, leading to the partial AFTX expression $A \leftarrow$ '$\sigma_{\text{prod\_root[/group\_root.A[last].v=/book/author/last.v]}}(($ $\nu_{(\text{/last.v,“last”})}(\pi_{\text{/bib/book/author/last}}(\text{“books.xml”})) \times \pi_{\text{/bib/book}}(\text{“books.xml”})))$';
- *ReturnClause*, 5: calls to *DirElemConstructor*:
  - *DirElemConstructor*, 3: calls to *DirAttribute*:
    - *DirAttribute*, 2: calls to *PathExpr2*, obtaining $a \leftarrow$ '/prod_root/group_root.A[last].v';
    - *DirAttribute*, 5: $a \leftarrow$ '(“Name”, /prod_root/group_root.A[last].v)';
  - *DirElemConstructor*, 3: $a \leftarrow$ '(“Name”, /prod_root/group_root.A[last].v)';
  - *DirElemConstructor*, 6: $a \leftarrow$ '((“Name”, /prod_root/group_root.A[last].v))';
  - *DirElemConstructor*, 11: calls to *DirElemConstructor*, passing as input the constructor <book>{$j/title/text()}</book>:
    - *DirElemConstructor*, 8: $a \leftarrow$ 'null';
    - *DirElemConstructor*, 17: calls to *PathExpr2*, obtaining $v \leftarrow$ '/prod_root/book/title.v';
    - *DirElemConstructor*, 25: $c \leftarrow$ 'null';
    - *DirElemConstructor*, 28: $t \leftarrow$ '“book”(/prod_root/book/title.v, null, null)';
    - *DirElemConstructor*, 29: returns t to the calling *DirElemConstructor*;
  - *DirElemConstructor*, 11: calls to *AddChild*, obtaining $c \leftarrow$ '(“book”(/prod_root/book/title.v, null, null)';
  - *DirElemConstructor*, 23: $v \leftarrow$ 'null';
  - *DirElemConstructor*, 27: $c \leftarrow$ '(“book”(/prod_root/book/title.v, null, null))';

  - *DirElemConstructor*, 28: $t \leftarrow$ ‘“author”(null, ((“name”, /prod_root/group_root.A[last].v)), (“book”(/prod_root/book/title.v, null, null)));
  - *DirElemConstructor*, 29: returns t to *ReturnClause*;

- *ReturnClause*, 5: $t \leftarrow$ ‘“author”(null, ((“Name”, /prod_root/group_root.A[last].v)), (“book”(/prod_root/book/title.v, null, null)));
- *ReturnClause*, 14: returns t to *FLWORExpr*;
- *FLWORExpr*, 11: $A \leftarrow$ ‘$\iota_t(\sigma_{\text{prod\_root[/group\_root.A[last].v=/book/author/last.v]}}(($ $\nu_{(\text{/last.v},\text{“last”})}(\pi_{\text{/bib/book/author/last}}(\text{“books.xml”}))\times\pi_{\text{/bib/book}}(\text{’“books.xml”}))))$’.

Example 4.13 Consider the following XQuery expression:

```
for $i in doc("authors.xml")/authors/author/last
return <author name={$i}>
  {
    for $j in doc("books.xml")/bib/book
    where $j/author/last=$i
    return <book>{$j/title/text()}</book>
  }
  </author>
```

The outer `for` clause is translated as usual, leading to a partial result

$$A \leftarrow \text{‘}\pi_{\text{/authors/author/last}}(\text{“authors.xml”})\text{’}$$

while $V_{\$i} \leftarrow$ ‘/last’. The translation goes on as follows:

- at line 9, *FLWORExpr* calls *ReturnClause*, passing as input the outer `return` clause ;
- at line 3, *ReturnClause* calls *FLWORExpr*, passing as input the inner FLWOR expression;

- at line 2, *FLWORExpr* calls *ForClause* as usual, obtaining:
  - $A \leftarrow$ '$\pi_{/\texttt{authors/author/last}}(\text{“}\texttt{authors.xml}\text{”}) \times \pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”})$';
  - $V_{\$i} \leftarrow$ '$/\texttt{prod\_root}/\texttt{last}$'
  - $V_{\$j} \leftarrow$ '$/\texttt{prod\_root}/\texttt{book}$'
- at line 6, *FLWORExpr* calls *WhereClause*, passing as input the clause `$j/author/last = $i`;
- at line 4, *WhereClause* calls *CreateOuterJoin*, obtaining:
  - $A \leftarrow$ '$\delta_{/\texttt{group\_root}/*[\texttt{.k}=/\texttt{group\_root.A}[\texttt{treeIdentity}]\texttt{.v AND .pos}>1]}($
    $\Sigma_{((/\texttt{prod\_root}/1\texttt{.k},\text{“}\texttt{treeIdentity}\text{”})),(/\texttt{prod\_root}/1,/\texttt{prod\_root}/2)}($
    $\delta_{/\texttt{prod\_root/book}[/\texttt{author/last}]}($
    $\delta_{/\texttt{prod\_root/book/author}[\texttt{NOT }/\texttt{last.v}=/\texttt{prod\_root/last.v}]}($
    $\pi_{/\texttt{authors/author/last}}(\text{“}\texttt{authors.xml}\text{”}) \times \pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”})))))$';
  - $V_{\$i} \leftarrow$ '$\texttt{group\_root}/\texttt{last}$';
  - $V_{\$j} \leftarrow$ '$\texttt{group\_root}/\texttt{book}$';
- at line 9, *FLWORExpr* calls *ReturnClause*, passing as input the inner constructor `<book>{$j/title/text()}</book>`;
- at line 5, *ReturnClause* calls *DirElemConstructor*, obtaining a partial tree constructor predicate $t \leftarrow$ '$\text{“}\texttt{book}\text{”}(/\texttt{group\_root}/\texttt{book}/\texttt{title.v}, \texttt{null}, \texttt{null})$';
- *ReturnClause* passes back t to *FLWORExpr*;
- *FLWORExpr* passes it back to the outer *ReturnClause* without modifying the AFTX expression (because the input parameter $addTreeConst$ is false);
- at line 5, *ReturnClause* calls *DirElemConstructor*, passing as input the outer constructor (including the inner FLWOR expression) and a list of tree construction predicates that now contains t;

- at line 14, *DirElemConstructor* set c to '"book"(/group_root/book/title.v, null, null)' (because the DirElemContent of the DirElemConstructor is a FLWOR-Expr); the final value of t is therefore '"author"(null, (("name", group_root /last.v)), ("book"(/group_root/book/title.v, null, null)))';

- *ReturnClause* passes back t to *FLWORExpr*;

- at line 11, *ReturnClause* builds the final AFTX expression:
  $\iota_t(\delta_{/\text{group\_root}/*[.\text{k}=/\text{group\_root.A}[\text{treeIdentity}].\text{v AND .pos}>1]}($
  $\Sigma_{((/\text{prod\_root}/1.\text{k},\text{“treeIdentity”})),(/\text{prod\_root}/1,/\text{prod\_root}/2)}($
  $\delta_{/\text{prod\_root}/\text{book}[/\text{author}/\text{last}]}($
  $\delta_{/\text{prod\_root}/\text{book}/\text{author}[\text{NOT } /\text{last.v}=/\text{prod\_root}/\text{last.v}]}($
  $\pi_{/\text{authors}/\text{author}/\text{last}}(\text{“authors.xml”}) \times \pi_{/\text{bib}/\text{book}}(\text{“books.xml”})))))$'.

Up to now, we have seen what happens when the XQuery expression corresponds to a FLWOR expression. However, there are cases when the XQuery expression is instead a constructor, which includes an inner FLWOR expression; this is the case when we want to include the result of a FLWOR expression in an enclosing XML element. In such cases, the function XQuery2AFTX calls, at line 7, the function *Constructor*, shown in Algorithm 22.

Constructor is a simplified version of the function *DirElemConstructor* already analyzed. It also builds a tree construction predicate, but it ignores inner FLWOR expressions; the result is therefore a predicate without any path expression.

XQuery2AFTX now adds a tree construction operator using the predicate just created; then it calls *FLWORExpr* for each enclosed FLWOR expression.

Example 4.14 Consider the following XQuery expression:

```
<authors>
  {
    for $a in doc("books.xml")//author
    return {$a}
  }
```

Algorithm 22 Function Constructor

Input: a Constructor C

Output: a tree constructor predicate t

1: **if** there is at least one DirAttribute **then**
2: **for all** DirAttribute D **do**
3: $a \leftarrow a +$ '(“' + QName + '”, “' + Literal + '”)'
4: **if** DirAttribute is not the last one **then**
5: $a \leftarrow a +$ ', '
6: $a \leftarrow$ '(' $+ a +$ ')'
7: **else**
8: $a \leftarrow$ '`null`'
9: **for all** DirElemContent C_i **do**
10: **if** C_i is a DirElemConstructor **then**
11: AddChild$(c,$ Constructor$(C_i))$
12: **else if** C_i is a literal **then**
13: $v \leftarrow v + C_i$
14: **if** v is the empty string **then**
15: $v \leftarrow$ '`null`'
16: **if** c is the empty string **then**
17: $c \leftarrow$ '`null`'
18: **else**
19: $c \leftarrow c +$ ')'
20: $t \leftarrow$ '“' + QName + '”(' $+ v +$ ', ' $+ a +$ ', ' $+ c +$ ')'
21: **return** t

```
</authors>
```

The translation is carried out as follows:

- *XQuery2AFTX*, at line 7, calls *Constructor*;
- *Constructor* builds the predicate $t \leftarrow$ '"`authors`"(`null`, `null`, `null`)';
- *XQuery2AFTX* builds the partial AFTX expression $A \leftarrow$ '$\iota_{\text{"authors"(null,null,null)}}$(';
- *XQuery2AFTX*, at line 10, calls *FLWORExpr* passing as input the inner FLWOR expression, obtaining the result $A_i \leftarrow \iota_{/\texttt{author}}(\pi_{//\texttt{author}}(\text{"}\texttt{books.xml}\text{"}))$;
- *XQuery2AFTX*, at line 13, obtains the final result $A \leftarrow$ '$\iota_{\text{"authors"(null,null,null)}}($ $\iota_{/\texttt{author}}(\pi_{//\texttt{author}}(\text{"}\texttt{books.xml}\text{"})))$'.

4.2 XQuery Full-Text Translation Rules

4.2.1 Informal Overview

XQuery Full-Text provides two kinds of full-text search:

- *boolean retrieval*: an element satisfies the full-text condition or it does not satisfy the condition at all;
- *ranked retrieval*: each element in the context is assigned a score reflecting the level of satisfaction of the full-text condition.

Boolean retrieval is done by inserting a `ftcontains` expression, either in a path expression (of a `for` or `let` clause) or in a `where` clause. Such expression is translated into an AFTX expression using the full-text selection predicate. For example the partial query

```
for $b in doc("bib.xml")/books/book
where $b ftcontains "dog"
```

is translated into the following AFTX expression:

$$\varsigma_{/\texttt{book}[\text{“}\texttt{dog}\text{”}]}(\pi_{/\texttt{books}/\texttt{book}}(\text{“}\texttt{bib.xml}\text{”}))\ .$$

The `for` clause

```
for $b in doc("bib.xml")/books/book[. ftcontains "dog"]
```

is translated into the same AFTX expression; this is not surprising, because the second XQuery expression is equivalent to the first one.

If the `ftcontains` expression is formed by two words (or phrases) connected with a boolean operator, also the full-text selection operator will have a predicate composed by two basic full-text conditions connected with a boolean operator. For example the partial query

```
for $b in doc("bib.xml")/books/book
where $b ftcontains "dog" && "cat"
```

is translated into the following AFXT expression:

$$\varsigma_{/\texttt{book}[\text{“}\texttt{dog}\text{”}\ \texttt{AND}\ \text{“}\texttt{cat}\text{”}]}(\pi_{/\texttt{books}/\texttt{book}}(\text{“}\texttt{bib.xml}\text{”}))\ .$$

Ranked retrieval is instead done by adding a `let` clause that defines a score variable. Such a `let` clause is translated using the full-text score assignment operator. For example the partial query

```
let score $s := $b ftcontains "dog" && "cat"
order by $s descending
```

is translated into the following AFTX expression:

$$o_{/\texttt{1.score}\ \texttt{DESC}}(\xi_{/\texttt{1}[\text{“}\texttt{dog}\text{”}\ \texttt{AND}\ \text{“}\texttt{cat}\text{”}]}(A))\ ,$$

where A is the algebraic expression representing the variable \$b. The score assignment operator assigns a value to the *score* property of the root element of each input tree; the subsequent ordering operator uses such score to order the forest.

If the score variable is in a `for` clause (that must contains a `ftcontains` expression), a full-text selection must be executed; then each retained tree must be assigned a score value. This is exactly the behavior of the derived full-text selection with score operator. Therefore a clause like

```
for $b score $s in doc("bib.xml")/books/book
    [. ftcontains "dog" && "cat"]
```

is translated into the following AFTX expression:

$$\overline{\varsigma}_{/\texttt{book}[\text{“}\texttt{dog}\text{”}\ \texttt{AND}\ \text{“}\texttt{cat}\text{”}]}\left(\pi_{/\texttt{books/book}}(\text{“}\texttt{bib.xml}\text{”})\right) \ .$$

Scoring may be influenced by adding *weight* specifications to search tokens. If this is the case, weights are added to the used AFTX operator, either if it is the score assignment operator or if it is the full-text selection with score operator. For example the `let` clause

```
let score $s := $b ftcontains ("dog" weight 0.2)
    && ("cat" weight 0.8)
```

is translated into the following AFTX expression:

$$\xi_{/1[0.2\ \text{“}\texttt{dog}\text{”}\ \texttt{AND}\ 0.8\ \text{“}\texttt{cat}\text{”}]}(A) \ ,$$

where A is the algebraic expression representing the variable $b.

A `ftcontains` expression could state that searched words must be found at a certain maximal distance between one and another. AFTX also provides such an option, thus the translation is straightforward. For example a clause like

```
for $b in doc("bib.xml")//book
    [. ftcontains "web" && "site" distance at most 2]
```

is translated into the following AFTX expression:

$$\varsigma_{/\texttt{book}[\text{“}\texttt{web}\text{”}\ \texttt{AND}\ \text{“}\texttt{site}\text{”},2]}\left(\pi_{//\texttt{book}}(\text{“}\texttt{bib.xml}\text{”})\right) \ .$$

Finally, AFTX also permits to express part of the match options provided by XQuery Full-Text, namely the usage of stemming, thesaurus and stopwords. For example the clause

```
for $b in doc("bib.xml")//book [. ftcontains "the web site"
    && "usability" with stemming with thesaurus default
    with default stop words]
```

is translated into the following AFTX expression:

$$\varsigma_{/\texttt{book}[\text{“}\texttt{the web site}\text{”}\ \texttt{AND}\ \text{“}\texttt{usability}\text{”},\texttt{stem},\texttt{thes},\texttt{stop}]}\left(\pi_{//\texttt{book}}(\text{“}\texttt{bib.xml}\text{”})\right) \ .$$

It should be noted that, in the translation examples involving score, we have never inserted the parameter f, which defines the score function to use. This is because the availability of such a parameter is an AFTX feature not present in XQuery Full-Text. Therefore the default score function will be used when translating XQuery Full-Text expressions.

4.2.2 Formal Translation Algorithm

In Section 4.1.2 we have presented the partial XQuery grammar that can be expressed in AFTX. That grammar must be expanded in order to represent full-text extensions:

```
ForClause           ::= "for" VarRef PositionalVar?
                        FTScoreVar? "in" ForLetContext ("," 
                        VarRef PositionalVar? FTScoreVar?
                        "in" ForLetContext)*
LetClause           ::= (("let" VarRef := ForLetContext)
                        | ("let" FTSCoreVar ":=" VarRef
                        (AxisStep NameStep)* "ftcontains"
                        FTSelection)) ("," VarRef ":="
                        ForLetContext)*
FTScoreVar          ::= "score" VarRef
WhereClause         ::= "where" (ComparisonExpr |
                        QuantifiedExpr | (VarRef PathExpr
                        "ftcontains" FTSelection)) ("and"
                        (ComparisonExpr | QuantifiedExpr
```

```
                        | (VarRef PathExpr "ftcontains"
                        FTSelection)))*
ComparisonExpr2     ::= (UnaryExpr1 (GeneralComp UnaryExpr2)?)
                        | (PathExpr "ftcontains" FTSelection)
FTSelection         ::= FTOr (FTMatchOption)*
FTOr                ::= FTAnd ( "||" FTAnd )*
FTAnd               ::= FTUnaryNot ( "&&" FTUnaryNot )*
                        FTDistance?
FTUnaryNot          ::= ("!")? Literal ("weight" Number)?
FTMatchOption       ::= FTStemOption | FTThesaurusOption
                        | FTStopwordOption
FTStemOption        ::= "with" "stemming"
FTThesaurusOption   ::= "with" "thesaurus" "default"
FTStopwordOption    ::= "with" "default" "stop" "words"
FTDistance          ::= "distance" "at" "most" Number "words"
```

As we can see, the `for` and `let` clauses now permit to define a score variable; in a `let` clause, we can define *normal* variables or score variables. `ComparisonExpr` and `ComparisonExpr2`, that are used respectively in a predicate of a `for`/`let` clause and in a `where` clause, has been modified in order to provide, besides *normal* predicates, full-text predicates. `FTSelection` and all the following production rules define how a full-text predicate can be formed.

With respect to the XQuery Full-Text specifications, our grammar has the following limitations:

- a `let` clause cannot contain more than one score variable definition; if a single clause defines a score variable, it cannot also define a *normal* variable;
- the only supported match options are stemming, thesaurus and stop word;
- only the default thesaurus and the default list of stop words can be used;
- *mild not* operator is not supported;

- distance option can only be of the type *at most*;
- scope option is not supported;
- ignore option is not supported.

Algorithm 23 shows the only modification that must be done over the procedure *ForClause*. Between lines 2 and 3 we add a conditional expression: if the clause contains the definition of a score variable, a new element is added to the variable binding list. That element will represent the score value.

Algorithm 23 Changes to the procedure ForClause

2.1: **if** DVContext contains a score variable $s **then**
2.2: $V_{\$s} \leftarrow$ '/1.score';

The procedure *ForClause*, as seen in Section 4.1.2, calls *PathExpr* for each variable binding. *PathExpr* calls *Predicate* for each predicate found in the path expression. However, if the predicate is a full-text predicate it must instead call *FTSelection*, that will be analyzed soon, as shown in Algorithm 24. Then a full-text selection or a full-text selection with score is added to the AFTX expression built up to now.

Algorithm 24 Changes to the procedure PathExpr

7.1: **if** P is a full-text predicate **then**
7.2: $\lambda \leftarrow$ '/1'
7.3: **for all** (AxisStep NameStep) **do**
7.4: $\lambda \leftarrow \lambda +$ AxisStep + NameStep
7.5: $\lambda \leftarrow \lambda +$ '[' + FTSelection(FT) + ']' {FT is the full-text condition}
7.6: **if** the clause contains a score variable **then**
7.7: $A \leftarrow$ '$\overline{\varsigma}_{\lambda}$(' $+ A +$ ')'
7.8: **else**
7.9: $A \leftarrow$ 'ς_{λ}(' $+ A +$ ')'
7.10: **else**
7.11: continue with normal algorithm

Algorithm 25 shows the modifications that must be done over the procedure *Let-Clause*. After line 1 we add a conditional expression: if the clause contains the definition of a score variable a score assignment operator is added to the AFTX expression built up to now, otherwise the clause is treated as in the original algorithm. The full-text predicate is created by the function *FTSelection*.

Algorithm 25 Changes to the procedure LetClause

2.1: **if** $i is a score variable **then**
2.2: $V_{\$i} \leftarrow$ '`/1.score`'
2.3: $\lambda \leftarrow V_{\text{VarRef}}$
2.4: **for all** (AxisStep NameStep) **do**
2.5: $\lambda \leftarrow \lambda +$ AxisStep $+$ NameStep
2.6: $\lambda \leftarrow \lambda +$ '[' $+$ FTSelection$(FT) +$ ']' $\{FT$ is the full-text condition$\}$
2.7: $A \leftarrow$ '$\xi_\lambda($' $+ A +$ ')'
2.8: **else**
2.9: continue with normal algorithm

Algorithm 26 shows the modifications that must be done over the procedure *Where-Clause*. After line 1 we add a new case to the if-then-else expression: the clause can be a `ftcontains` expression. If this is the case, we add a full-text selection operator to the AFTX expression built up to now; also in this case the full-text predicate is created by the function *FTSelection*.

Algorithm 26 Changes to the procedure WhereClause

1.1: **if** the clause is a FTContains expression **then**
1.2: $\lambda \leftarrow V_{\text{VarRef}}$
1.3: **for all** (AxisStep NameStep) **do**
1.4: $\lambda \leftarrow \lambda +$ AxisStep $+$ NameStep
1.5: $\lambda \leftarrow \lambda +$ '[' $+$ FTSelection$(FT) +$ ']' $\{FT$ is the full-text condition$\}$
1.6: $A \leftarrow$ '$\varsigma_\lambda($' $+ A +$ ')'
1.7: **else**
1.8: continue with normal algorithm

The function *FTSelection*, presented in Algorithm 27, build the full-text condition. The translation process is quite straightforward, so we do not analyze it in more details.

Example 4.15 Consider the following partial query:

```
for $b in doc("bib.xml")/books/book
where $b ftcontains "dog" && "cat"
```

The `for` clause is translated as usual, thus leading to the following partial result:

- $A \leftarrow$ '$\pi_{\texttt{/books/book}}(\text{“}\texttt{bib.xml}\text{”})$';
- $V_{\$b} \leftarrow$ '$/\texttt{book}$'.

Then the procedure *WhereClause* is called:

- *WhereClause*, 1.2: $\lambda \leftarrow$ '$/\texttt{book}$';
- *WhereClause*, 1.5: calls to *FTSelection*:
  - *FTSelection*, 7: $\gamma \leftarrow$ '“`dog`”';
  - *FTSelection*, 9: $\gamma \leftarrow$ '“`dog`” `AND` ';
  - *FTSelection*, 7: $\gamma \leftarrow$ '“`dog`” `AND` “`cat`”';
  - *FTSelection*, 21: returns γ to *WhereClause*;
- *WhereClause*, 1.5: $\lambda \leftarrow$ '$/\texttt{book}[\text{“}\texttt{dog}\text{”}\ \texttt{AND}\ \text{“}\texttt{cat}\text{”}]$';
- *WhereClause*, 1.6: $A \leftarrow$ '$\varsigma_{\texttt{/book}[\text{“}\texttt{dog}\text{”}\ \texttt{AND}\ \text{“}\texttt{cat}\text{”}]}(\pi_{\texttt{/books/book}}(\text{“}\texttt{bib.xml}\text{”}))$'.

Example 4.16 Consider the following partial query:

```
for $b in doc("bib.xml")/books/book
let score $s := $b ftcontains ("dog" weight 0.2)
                && ("cat" weight 0.8)
```

The `for` clause is translated as in the previous example; then the procedure *LetClause* is called:

Algorithm 27 Function FTSelection

Input: a full-text selection FT

Output: a partial full-text predicate γ

1: **for all** FTAnd in FTOr **do**
2: **for all** FTUnaryNot in FTAnd **do**
3: **if** there is a 'weight' **then**
4: $\gamma \leftarrow \gamma +$ 'Number'
5: **if** there is a '!' **then**
6: $\gamma \leftarrow \gamma +$ '`NOT`'
7: $\gamma \leftarrow \gamma + Literal$
8: **if** this is not the last FTUnaryNot **then**
9: $\gamma \leftarrow \gamma +$ '`AND`'
10: **if** there is a FTDistance **then**
11: $\gamma \leftarrow \gamma +$ ',' $+ Number$
12: **if** this is not the last FTAnd **then**
13: $\gamma \leftarrow \gamma +$ '`OR`'
14: **for all** FTMatchOption in FT **do**
15: **if** it is a FTStemOption **then**
16: $\gamma \leftarrow \gamma +$ ', `stem`'
17: **else if** it is a FTThesaurusOption **then**
18: $\gamma \leftarrow \gamma +$ ', `thes`'
19: **else** {it is a FTStopWordOption}
20: $\gamma \leftarrow \gamma +$ ', `stop`'
21: **return** γ

- *LetClause*, 2.2: $V_{\$s} \leftarrow$ '/1.score';
- *LetClause*, 2.3: $\lambda \leftarrow$ '/book';
- *LetClause*, 2.6: calls to *FTSelection*:
  - *FTSelection*, 4: $\gamma \leftarrow$ '0.2';
  - *FTSelection*, 7: $\gamma \leftarrow$ '0.2 "dog"';
  - *FTSelection*, 9: $\gamma \leftarrow$ '0.2 "dog" AND ';
  - *FTSelection*, 4: $\gamma \leftarrow$ '0.2 "dog" AND 0.8';
  - *FTSelection*, 7: $\gamma \leftarrow$ '0.2 "dog" AND 0.8 "cat"';
  - *FTSelection*, 21: returns γ to *LetClause*;
- *LetClause*, 2.6: $\lambda \leftarrow$ '/book[0.2 "dog" AND 0.8 "cat"]';
- *LetClause*, 2.7: $A \leftarrow$ '$\xi_{\text{/book[0.2 "dog" AND 0.8 " cat"]}}(\pi_{\text{/books/book}}(\text{"bib.xml"}))$'.

4.3 Complex Translation Examples

4.3.1 XQuery Expressions

In this section we present a series of examples of translation of complex XQuery expressions into AFTX expressions. These examples are taken from W3C XQuery Use Cases [Con06b] and demonstrate that almost any XQuery expression can be translated into AFTX.

For each example, we present the query requirements (expressed in natural language), the solution in XQuery and the solution in AFTX.

Example 4.17 [Use Case "XMP" Q1] List books published by Addison-Wesley after 1991, including their year and title.
XQuery solution:

```
<bib>
  {
    for $b in doc("bib.xml")/bib/book
    where $b/publisher = "Addison-Wesley"
      and $b/@year > 1991
    return
      <book year="{ $b/@year }">
        { $b/title }
      </book>
  }
</bib>
```

The query in this example and all the following has been slightly modified, by shortening the name of the input XML document. This is done just for the sake of brevity and does not affect in any way the translation process.

AFTX translation:

$$\iota_{\text{“bib”}(\texttt{null},\texttt{null},\texttt{null})}\Big(\iota_{\text{“book”}(\texttt{null},((\text{“year”},/\texttt{book.A[year].v})),(/\texttt{book/title}))}\Big(\sigma_{/\texttt{book}[.\texttt{A}[\texttt{year}].\texttt{v}>1991]}\Big(\sigma_{/\texttt{book}[/\texttt{publisher.v}=\text{“Addison}-\text{Wesley”}]}\Big(\pi_{/\texttt{bib/book}}(\text{“}\texttt{bib.xml}\text{”})\Big)\Big)\Big)$$

This example shows how a `where` clause with conditions connected with `AND` is translated: two subsequent selection operations are applied to the input forest.

Example 4.18 [Use Case “XMP” Q2] Create a flat list of all the title-author pairs, with each pair enclosed in a “result” element.

XQuery solution:

```
<results>
  {
    for $b in doc("bib.xml")/bib/book,
      $t in $b/title,
```

```
      $a in $b/author
    return
      <result>
        { $t }
        { $a }
      </result>
  }
</results>
```

AFTX translation:

$$\iota_{\text{“results”}(\texttt{null},\texttt{null},\texttt{null})}($$
$$\iota_{\text{“result”}(\texttt{null},\texttt{null},(/\texttt{prod\_root}/\texttt{prod\_root}/\texttt{title},/\texttt{prod\_root}/\texttt{author}))}($$
$$((\pi_{/\texttt{bib}/\texttt{book}}(\text{“}\texttt{bib.xml}\text{”}) \bowtie_{/\texttt{book}[/\texttt{title}\equiv/\texttt{author}]}$$
$$\pi_{/\texttt{book}/\texttt{title}}(\pi_{/\texttt{bib}/\texttt{book}}(\text{“}\texttt{bib.xml}\text{”}))) \bowtie_{/\texttt{prod\_root}/\texttt{book}[/\texttt{author}\equiv/\texttt{author}]}$$
$$\pi_{/\texttt{book}/\texttt{author}}(\pi_{/\texttt{bib}/\texttt{book}}(\text{“}\texttt{bib.xml}\text{”})))))$$

Example 4.19 [Use Case “XMP” Q3] For each book in the bibliography, list the title and authors, grouped inside a “result” element.

XQuery solution:

```
<results>
  {
    for $b in doc("bib.xml")/bib/book
    return
      <result>
        { $b/title }
        { $b/author }
      </result>
  }
</results>
```

AFTX translation:

$$\iota_{\text{“results”}(\texttt{null},\texttt{null},\texttt{null})}($$

$$\iota_{\text{“result”}(\texttt{null},\texttt{null},(/\texttt{book}/\texttt{title},/\texttt{book}/\texttt{author}))}\Big($$
$$\pi_{/\texttt{bib}/\texttt{book}}(\text{“}\texttt{bib.xml}\text{”}))\Big)$$

Example 4.20 [Use Case “XMP” Q4] For each author in the bibliography, list the author's name and the titles of all books by that author, grouped inside a “result” element.
XQuery solution:

```
<results>
  {
    for $last in distinct-values(doc("bib.xml")
        //author/last),
      $first in distinct-values(doc("bib.xml")
        //author[last=$last]/first)
    order by $last, $first
    return
      <result>
        <author>
          <last>{ $last }</last>
          <first>{ $first }</first>
        </author>
        {
          for $b in doc("bib.xml")/bib/book
          where some $ba in $b/author satisfies
            ($ba/last = $last and $ba/first=$first)
          return {$b/title}
        }
      </result>
  }
</results>
```

The query in this example has been slightly modified in order to be consistent with the grammar defined in Section 4.1.2. The original query had an initial `let` clause binding a

variable $a to the `authors` element.

AFTX translation:

$$
\begin{array}{l}
\iota_{\text{“}\texttt{results}\text{”}(\texttt{null},\texttt{null},\texttt{null})}(\\
\quad \iota_t(\\
\quad\quad \delta(/\texttt{group\_root}/ * [.\texttt{pos} > 1 \texttt{ AND } .\texttt{k} = /\texttt{group\_root.A}[\texttt{treeIdentity}].\texttt{v}](\\
\quad\quad\quad \Sigma_{((/\texttt{prod\_root}/1.\texttt{k},\text{“}\texttt{treeIdentity}\text{”})),(/\texttt{prod\_root}/1,/\texttt{prod\_root}/2)}(\\
\quad\quad\quad\quad \delta_{/\texttt{prod\_root/book/author}[\texttt{NOT } /\texttt{last}]}(\\
\quad\quad\quad\quad\quad \delta_{/\texttt{prod\_root/book/author}[\texttt{NOT } (P)]}(\\
\quad\quad\quad\quad\quad\quad o_{\texttt{group\_root/group\_root.A}[\texttt{last}].\texttt{v},/\texttt{group\_root.A}[\texttt{first}].\texttt{v}}(\\
\quad\quad\quad\quad\quad\quad\quad \Sigma_{((/\texttt{prod\_root}/2/\texttt{first},\text{“}\texttt{first}\text{”}),(/\texttt{prod\_root}/1.\texttt{k},\text{“}\texttt{treeIdentity}\text{”})),(/\texttt{prod\_root}/1)}(\\
\quad\quad\quad\quad\quad\quad\quad\quad \sigma_{/\texttt{prod\_root}[/\texttt{author/last.v}=/\texttt{group\_root.A}[\texttt{last}].\texttt{v}]}(\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \nu_{(/\texttt{last.v},\text{“}\texttt{last}\text{”})}(\pi_{//\texttt{author/last}}(\text{“}\texttt{bib.xml}\text{”})) \times \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \pi_{//\texttt{author}}(\text{“}\texttt{bib.xml}\text{”})))) \times \pi_{/\texttt{bib/book}}(\text{“}\texttt{bib.xml}\text{”}))))))))'
\end{array}
$$

where

- $t =$ ‘“`result`”(`null`, `null`, (“`author`”(`null`, `null`, (“`last`”(/`group_root`/`group_root`/`group_root.A`[`last`], `null`, `null`), “`first`”(/`group_root`/`group_root.A`[`first`], `null`, `null`))), `group_root`/`book`/`title`))’;
- $P =$ ‘/`last.v` = /`prod_root`/`group_root`/`group_root.A`[`last`].v `AND` /`first.v` = /`prod_root`/`group_root.A`[`first`].v.

This translation is quite complex and deserves an in-depth analysis. The final AFTX expression, which is shown graphically in Figure 4.3, is obtained through the following steps:

- the query starts with an element constructor, thus at line 7 *XQuery2AFTX* calls *Constructor*, which creates the tree constructor predicate `"results"(null, null, null)`; the initial AFTX expression is therefore ‘$\iota_{\text{“}\texttt{results}\text{”}(\texttt{null},\texttt{null},\texttt{null})}($’;
- the constructor contains an inner FLWOR expression, thus at line 10 *XQuery2AFTX* calls *FLWORExpr*, which must build a completely unrelated AFTX expression;
- at line 2, *FLWORExpr* calls *ForClause*;

- *ForClause* translates the first variable binding as usual, obtaining:
  - $A \leftarrow$ '$\nu_{(/\texttt{last.v},\text{“}\texttt{last}\text{”})}(\pi_{//\texttt{author/last}}(\text{“}\texttt{bib.xml}\text{”}))$';
  - $V_{\$\texttt{last}} \leftarrow$ '/group_root.A[last]';
- the second variable binding contains a predicate that references `$last`, thus *ForClause* translates the first part of the relative path expression, obtaining:
  - $A \leftarrow$ '$\nu_{(/\texttt{last.v},\text{“}\texttt{last}\text{”})}(\pi_{//\texttt{author/last}}(\text{“}\texttt{bib.xml}\text{”})) \times \pi_{//\texttt{author}}(\text{“}\texttt{bib.xml}\text{”})$';
  - $V_{\$\texttt{last}} \leftarrow$ 'prod_root/group_root.A[last]';
  - $V_{\$\texttt{first}} \leftarrow$ 'prod_root/author';
- at line 20 *ForClause* calls Predicate, obtaining:
  $A \leftarrow$ '$\sigma_{/\texttt{prod\_root}[/\texttt{author/last.v}=/\texttt{group\_root.A}[\texttt{last}].\texttt{v}]}($
  $\nu_{(/\texttt{last.v},\text{“}\texttt{last}\text{”})}(\pi_{//\texttt{author/last}}(\text{“}\texttt{bib.xml}\text{”})) \times \pi_{//\texttt{author}}(\text{“}\texttt{bib.xml}\text{”}))$';
- the `for` clause contains a call to `distinct-values`, thus *ForClause* at lines 24–27 obtains:
  - $A \leftarrow$ '$\Sigma_{((/\texttt{prod\_root/2/first},\text{“}\texttt{first}\text{”}),(/\texttt{prod\_root/1.k},\text{“}\texttt{treeIdentity}\text{”})),(/\texttt{prod\_root/1})}($
    $\sigma_{/\texttt{prod\_root}[/\texttt{author/last.v}=/\texttt{group\_root.A}[\texttt{last}].\texttt{v}]}($
    $(\nu_{(/\texttt{last.v},\text{“}\texttt{last}\text{”})}(\pi_{//\texttt{author/last}}(\text{“}\texttt{bib.xml}\text{”})) \times \pi_{//\texttt{author}}(\text{“}\texttt{bib.xml}\text{”})))$';
  - $V_{\$\texttt{first}} \leftarrow$ '/group_root.A[first]';
  - $V_{\$\texttt{last}} \leftarrow$ '/group_root/group_root.A[last]';
- at line 8 *FLWORExpr* calls *OrderByClause*, obtaining:
  $A \leftarrow$ '$o_{\texttt{group\_root/group\_root.A}[\texttt{last}].\texttt{v},/\texttt{group\_root.A}[\texttt{first}].\texttt{v}}($
  $\Sigma_{((/\texttt{prod\_root/2/first},\text{“}\texttt{first}\text{”}),(/\texttt{prod\_root/1.k},\text{“}\texttt{treeIdentity}\text{”})),(/\texttt{prod\_root/1})}($
  $\sigma_{/\texttt{prod\_root}[/\texttt{author/last.v}=/\texttt{group\_root.A}[\texttt{last}].\texttt{v}]}($
  $\nu_{(/\texttt{last.v},\text{“}\texttt{last}\text{”})}(\pi_{//\texttt{author/last}}(\text{“}\texttt{bib.xml}\text{”})) \times \pi_{//\texttt{author}}(\text{“}\texttt{bib.xml}\text{”}))))$';
- at line 9 *FLWORExpr* calls *ReturnClause*;

- the constructor contains an inner FLWOR expression, thus at line 3 *ReturnClause* calls *FLWORExpr* passing as input the inner FLWOR expression;
- at line 2 *FLWORExpr* calls *ForClause*, that translates the `for` clause as usual obtaining:
  - $A \leftarrow$ '$o_{\texttt{group\_root/group\_root.A[last],/group\_root.A[first]}}($
    $\Sigma_{((\texttt{/prod\_root/2/first,“first”}),(\texttt{/prod\_root/1.k,“treeIdentity”})),(\texttt{/prod\_root/1})}($
    $\sigma_{\texttt{/prod\_root[/author/last.v=/group\_root.A[last].v]}}($
    $\nu_{(\texttt{/last.v,“last”})}(\pi_{\texttt{//author/last}}(\texttt{“bib.xml”})) \times \pi_{\texttt{//author}}(\texttt{“bib.xml”}))) \times$
    $\pi_{\texttt{/bib/book}}(\texttt{“bib.xml”})$';
  - $V_{\$\texttt{b}} \leftarrow$ '/prod_root/book';
  - $V_{\$\texttt{first}} \leftarrow$ '/prod_root/group_root.A[first]';
  - $V_{\$\texttt{last}} \leftarrow$ '/prod_root/group_root/group_root.A[last]';
- at line 6 *FLWORExpr* calls *WhereClause*, that translates the `where` clause using *CreateOuterJoin* obtaining:
  - $A \leftarrow$ '$\delta_{\texttt{/group\_root/*[.pos>1 AND .k=/group\_root.A[treeIdentity].v]}}($
    $\Sigma_{((\texttt{/prod\_root/1.k,“TreeIdentity”})),(\texttt{/prod\_root/1,/prod\_root/2})}($
    $\delta_{\texttt{/prod\_root/book/author[NOT /last]}}($
    $\delta_{\texttt{/prod\_root/book/author[NOT }(P)\texttt{]}}($
    $o_{\texttt{group\_root/group\_root.A[last],/group\_root.A[first]}}($
    $\Sigma_{((\texttt{/prod\_root/2/first,“first”}),(\texttt{/prod\_root/1.k,“treeIdentity”})),(\texttt{/prod\_root/1})}($
    $\sigma_{\texttt{/prod\_root[/author/last.v=/group\_root.A[last].v]}}($
    $\nu_{(\texttt{/last.v,“last”})}(\pi_{\texttt{//author/last}}(\texttt{“bib.xml”})) \times \pi_{\texttt{//author}}(\texttt{“bib.xml”}))) \times$
    $\pi_{\texttt{/bib/book}}(\texttt{“bib.xml”}))))$'
    where $P =$ '/last.v = /prod_root/group_root/group_root.A[last].v AND
    /first.v = /prod_root/group_root.A[first].v';
  - $V_{\$\texttt{b}} \leftarrow$ '/group_root/book';
  - $V_{\$\texttt{first}} \leftarrow$ '/group_root/group_root.A[first]';

  - $V_{\$\texttt{last}} \leftarrow$ '/group_root/group_root/group_root.A[last]';

- at line 9 *FLWORExpr* calls *ReturnClause*, which returns $t \leftarrow$ 'group_root/book /title';

- *FLWORExpr* passes back t to the calling *ReturnClause*;

- *ReturnClause*, through multiple nested calls to *DirElemConstructor* and using the previously built t, builds a tree construction predicate $t \leftarrow$ '"result"(null, null, ("author"(null, null, ("last"(/group_root/group_root/group_root.A[last], null, null), "first"(/group_root/group_root.A[first], null, null))), group_root/book/title))';

- *ReturnClause* passes back t to the calling *FLWORExpr*, which build the AFTX expression '$\iota_t(A)$', where t is the tree construction predicate just built and A is the AFTX expression built until now;

- the control passes back to the calling *XQuery2AFTX*, which at lines 11–13 build the final AFTX expression '$\iota_{\text{"results"(null,null,null)}}(A)$', where A is the AFTX expression built in the previous step.

Example 4.21 [Use Case "XMP" Q5] For each book found at both *bstore1.example.com* and *bstore2.example.com*, list the title of the book and its price from each source.
XQuery solution:

```
<books-with-prices>
  {
    for $b in doc("bib.xml")//book,
      $a in doc("reviews.xml")//entry
    where $b/title = $a/title
    return
      <book-with-prices>
        { $b/title }
```

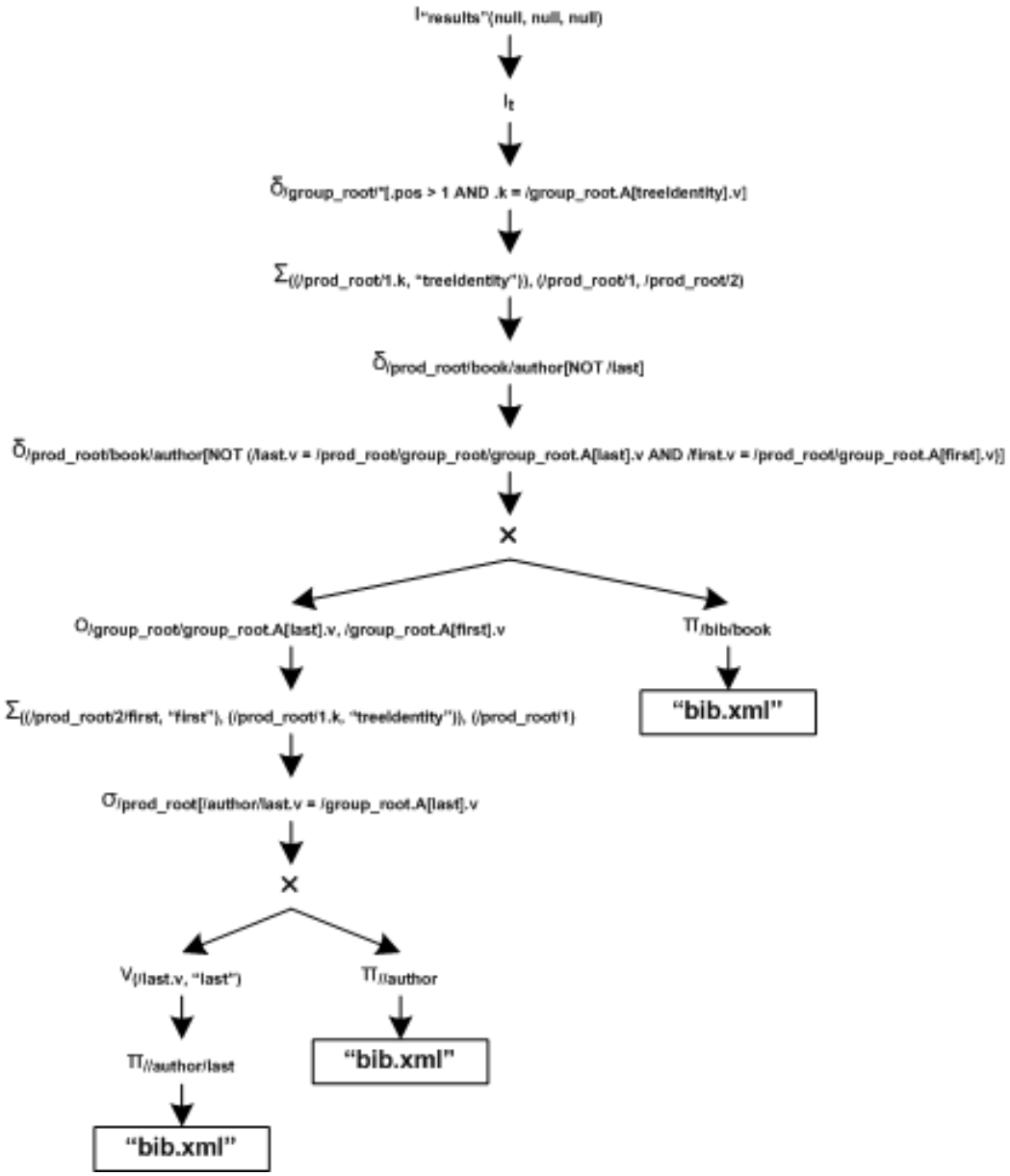


Figure 4.3: Graphical representation of the AFTX expression of Example 4.20.

```
        <price-bstore2>{ $a/price/text() }</price-bstore2>
        <price-bstore1>{ $b/price/text() }</price-bstore1>
      </book-with-prices>
  }
</books-with-prices>
```

AFTX translation:

$\iota_{\text{“books-with-prices”}(\texttt{null},\texttt{null},\texttt{null})}(\iota_t(\sigma_{/\texttt{prod\_root}[/\texttt{book/title.v}=/\texttt{entry/title.v}]}($
$\pi_{//\texttt{entry}}(\text{“}\texttt{reviews.xml}\text{”}) \times \pi_{//\texttt{book}}(\text{“}\texttt{bib.xml}\text{”}))))$
where $t =$ “book-with-prices”$(\texttt{null}, \texttt{null}, (/\texttt{prod\_root}/\texttt{book}/\texttt{title},$
“price-bstore2”$(/\texttt{prod\_root}/\texttt{entry}/\texttt{price.v}, \texttt{null}, \texttt{null}),$
“price-bstore1”$(/\texttt{prod\_root}/\texttt{book}/\texttt{price.v}, \texttt{null}, \texttt{null})))$

Example 4.22 [Use Case “XMP” Q6]

For each book that has at least one author, list the title and first two authors.

XQuery solution:

```
<bib>
  {
    for $b in doc("bib.xml")//book
    where count($b/author) > 0
    return
      <book>
        { $b/title }
        {
          for $a in $b/author[position()<=2]
          return $a
        }
      </book>
  }
</bib>
```

This query has been modified by removing the `if-then-else` construct, that is not expressible in AFTX.

AFTX translation:

$$\iota_{\text{“bib”}(\texttt{null},\texttt{null},\texttt{null})}\big(\iota_{\text{“book”}(\texttt{null},\texttt{null},(/\texttt{book/title},/\texttt{book/author}))}\big(\\ \delta_{/\texttt{book/author}[\texttt{NOT .pos<=2}]}\big(\sigma_{/\texttt{book/author}[\texttt{.count>0}]}\big(\pi_{//\texttt{book}}(\text{“}\texttt{bib.xml}\text{”})\big)\big)\big)\big)$$

Example 4.23 [Use Case “XMP” Q7] List the titles and years of all books published by Addison-Wesley after 1991, in alphabetic order.

XQuery solution:

```
<bib>
  {
    for $b in doc("bib.xml")//book
    where $b/publisher = "Addison-Wesley"
      and $b/@year > 1991
    order by $b/title
    return
      <book year={ $b/@year }>
        { $b/title }
      </book>
  }
</bib>
```

AFTX translation:

$$\iota_{\text{“bib”}(\texttt{null},\texttt{null},\texttt{null})}\big(\iota_{\text{“book”}(\texttt{null},((\text{“}\texttt{year}\text{”},/\texttt{book.A[year].v})),(/\texttt{book/title}))}\big(\\ o_{/\texttt{book/title.v ASC}}\big(\\ \sigma_{/\texttt{book}[\texttt{.A[year].v>1991}]}\big(\sigma_{/\texttt{book/publisher}[\texttt{.v=}\text{“}\texttt{Addison}-\texttt{Wesley}\text{”}]}\big(\\ \pi_{//\texttt{book}}(\text{“}\texttt{bib.xml}\text{”})\big)\big)\big)\big)\big)$$

Example 4.24 [Use Case “XMP” Q11] For each book with an author, return the book with its title and authors. For each book with an editor, return a reference with the book title and the editor’s affiliation.

XQuery solution:

```
<bib>
  {
    for $b in doc("bib.xml")//book[author]
    return
      <book>
        { $b/title }
        { $b/author }
      </book>
  }
  {
    for $b in doc("bib.xml")//book[editor]
    return
      <reference>
        { $b/title }
        {$b/editor/affiliation}
      </reference>
  }
</bib>
```

AFTX translation:

$\iota_{\text{“bib”}(\texttt{null},\texttt{null},\texttt{null})}($

$\quad \iota_{\text{“book”}(\texttt{null},\texttt{null},(/\texttt{book}/\texttt{title},/\texttt{book}/\texttt{author}))}($

$\quad\quad \sigma_{/\texttt{book}[/\texttt{author}]}(\pi_{//\texttt{book}}(\text{“bib.xml”}))) \cup$

$\quad \iota_{\text{“reference”}(\texttt{null},\texttt{null},(/\texttt{book}/\texttt{title},/\texttt{book}/\texttt{editor}/\texttt{affiliation}))}($

$\quad\quad \sigma_{/\texttt{book}[/\texttt{editor}]}(\pi_{//\texttt{book}}(\text{“bib.xml”}))))$

This example shows the translation process of a query consisting of a constructor with two inner FLWOR expressions. Each internal FLWOR expression is translated independently, and the resulting AFTX expressions are fed to the union operator. Finally a tree construction operator is applied as usual, in order to build the outer `bib` element.

Example 4.25 [Use Case “R” Q3] Find cases where a user with a rating worse (alpha-

betically, greater) than ”C” is offering an item with a reserve price of more than 1000.

XQuery solution:

```
<result>
  {
    for $u in doc("users.xml")//user,
      $i in doc("items.xml")//item
    where $u/rating > "C"
      and $i/reserve_price > 1000
      and $i/offered_by = $u/userid
    return
      <warning>
        { $u/name }
        { $u/rating }
        { $i/description }
        { $i/reserve_price }
      </warning>
  }
</result>
```

This query has been slightly modified in order to be consistent with our accepted grammar. The original query had two `for` clauses instead that a single clause with two variable bindings. Moreover, in order to improve expression readability, the tag names `user_tuple` and `item_tuple` have been substituted with the shorter ones `user` and `item`.

AFTX translation:

$$\iota_{\text{“result”}(\texttt{null},\texttt{null},\texttt{null})}\Big($$
$$\iota_{\text{“warning”}(\texttt{null},\texttt{null},(/1/\texttt{user}/\texttt{name},/1/\texttt{user}/\texttt{rating},/1/\texttt{item}/\texttt{description},/1/\texttt{iteme}/\texttt{reserve\_price}))}\Big($$
$$\sigma_{/\texttt{prod\_root}[/\texttt{item}/\texttt{offered\_by}.\texttt{v}=/\texttt{user}/\texttt{userid}.\texttt{v}]}\Big($$
$$\sigma_{/\texttt{prod\_root}/\texttt{item}/\texttt{reserve\_price}[.\texttt{v}>1000]}\Big($$
$$\sigma_{/\texttt{prod\_root}/\texttt{user}/\texttt{rating}[.\texttt{v}>\text{“C”}]}\Big($$

$\pi_{//\texttt{user}}(\text{“}\texttt{users.xml}\text{”}) \times$
$\pi_{//\texttt{item}}(\text{“}\texttt{items.xml}\text{”}))))))$

4.3.2 XQuery Full-Text Expressions

In this section we present a series of examples of translation of complex XQuery Full-Text expressions into AFTX expressions. These examples are taken from W3C XQuery Full-Text Use Cases [Con06e] and demonstrate that almost any XQuery Full-Text expression can be translated into AFTX.

For each example, we present the query requirements (expressed in natural language), the solution in XQuery Full-Text and the solution in AFTX.

Example 4.26 [Use Case “ELEMENT” Q1] Find all book titles containing the word “usability”.

XQuery Full-Text solution:

```
for $t in doc("full-text.xml")/books/book/metadata/title
    [. ftcontains "usability"]
return {$t}
```

This query, like many others following, has been modified in order to be accepted by our grammar: it has been transformed into a FLWOR expression.

AFTX translation:

$\iota_{/\texttt{title}}($
$\varsigma_{/\texttt{title}[\text{“}\texttt{usability}\text{”}]}($
$\pi_{/\texttt{books/book/metadata/title}}(\text{“}\texttt{full-text.xml}\text{”})))$

Example 4.27 [Use Case “ELEMENT” Q2] Find all book subjects containing the phrase “usability testing”.

XQuery Full-Text solution:

```
for $s in doc("full-text.xml")/books/book/metadata
    /subjects/subject [. ftcontains "usability testing"]
return {$s}
```

AFTX translation:

$\iota_{/\texttt{subject}}($

$\varsigma_{/\texttt{subject}[\text{“usability testing”}]}($

$\pi_{/\texttt{books/book/metadata/subjects/subject}}(\text{“}\texttt{full-text.xml}\text{”})))$

Example 4.28 [Use Case “ELEMENT” Q4] Find all books with “usability tests”in book or chapter titles.

XQuery Full-Text solution:

```
for $book in doc("full-text.xml")
    /books/book
let $title := $book/metadata/title
    [. ftcontains "usability tests"]
    or $book/content/part/chapter/title
    [. ftcontains "usability tests"]
where count($title) > 0
return $book
```

This query cannot be automatically translated into AFTX using the presented translation algorithms, because they do not support the `or` keyword inside a `let` clause. However the query is expressible in AFTX, as shown by the following translation.

AFTX translation:

$\pi_{/\texttt{group\_root/book}}($

$\sigma_{/\texttt{group\_root}[/\texttt{title.count}>0]}($

$\Sigma_{(/\texttt{prod\_root/book.k}),(/\texttt{prod\_root/book},/\texttt{prod\_root/title})}($

$\sigma_{/\texttt{prod\_root}[/\texttt{metadata/title}\equiv/\texttt{title OR /content/part/chapter/title}\equiv/\texttt{title}}($

$\pi_{/\texttt{books/book}}(\text{“}\texttt{full-text.xml}\text{”}) \times ($

$\varsigma_{/1[\text{“usability tests”}]}(\pi_{/1/\texttt{metadata/title}}(\pi_{/\texttt{books/book}}(\text{“}\texttt{full-text.xml}\text{”}))) \cup$

$\varsigma_{/1[\text{“usability tests”}]}(\pi_{/1/\texttt{content/part/chapter/title}}(\pi_{/\texttt{books/book}}($

$\text{“}\texttt{full-text.xml}\text{”})))))))$

Example 4.29 [Use Case "ELEMENT" Q6] Find all book titles which start with "improving" followed within 2 words by "usability".

XQuery solution:

```
for $book in doc("full-text.xml")/books/book
where $book/metadata/title ftcontains "improving"
    && "usability" distance at most 2 words
return $title
```

This query has been modified by removing the clauses `ordered` and `at start` from the `ftcontains` expression. It should be noted that, even if AFTX full-text operators does not provide such options, they should be easily introduced, because the data model already provides the necessary information for answering a query with those match options. Recall that the value of an element is a list of pairs (*word*, *position*); therefore, it would be possible to check if the searched words are found in the same order as in the query (by checking if $pos(\text{“}\texttt{improving}\text{”}) < pos(\text{“}\texttt{usability}\text{”})$) and if the title starts with "improving" (by checking if $pos(\text{“}\texttt{improving}\text{”}) = \mathit{firsttoken}$; as stated in Definition 3.5 of element full-text value, $\mathit{firsttoken}$ is the position of the first token in the full-text value of an element).

AFTX translation:

$$\iota_{/\texttt{book/title}}($$
$$\varsigma_{/\texttt{book/metadata/title}[\text{“}\texttt{improving}\text{”}\ \texttt{AND}\ \text{“}\texttt{usability}\text{”},2]}($$
$$\pi_{/\texttt{books/book}}(\text{“}\texttt{full-text.xml}\text{”})$$

Example 4.30 [Use Case "ACROSS" Q1] Find all book chapters containing the phrase "one of the best known lists of heuristics is Ten Usability Heuristics".

XQuery Full-Text solution:

```
for $book in doc("full-text.xml")/books/book
where $book//chapter ftcontains "one of the best known
    lists of heuristics is Ten Usability Heuristics"
return $book
```

This query has been slightly modified in order to be consistent with our grammar.

AFTX translation:

$\varsigma_{/1//\texttt{chapter}[\text{“one of the best known lists of heuristics is Ten Usability Heuristics”}]}($
$\pi_{/\texttt{books/book}}(\text{“}\texttt{full-text.xml}\text{”})))$

Example 4.31 [Use Case “OTHER” Q1] Find all books with “improve” “web” “usability” in the short title.

XQuery Full-Text solution:

```
for $book in doc("full-text.xml")/books/book
where $book/metadata/title/@shortTitle ftcontains
    "improve" && "web" && "usability" with stemming
    distance at most 2 words
return $book/metadata/title
```

AFTX translation:

$\pi_{/1/\texttt{metadata/title}}($
$\varsigma_{/1/\texttt{metadata/title.A}[\text{“}\texttt{shortTitle}\text{”}][\text{“}\texttt{improve}\text{” AND “}\texttt{web}\text{” AND “}\texttt{usability}\text{”},2,\texttt{stem}]}($
$\pi_{/\texttt{books/book}}(\text{“}\texttt{full-text.xml}\text{”})))$

Example 4.32 [Use Case “OTHER” Q2] Find all books with the phrase “manuscript guides” in the short title and the phrase “user profiling” in a component title.

XQuery Full-Text solution:

```
for $book in doc("full-text.xml")/books/book
where $book/metadata/title/@shortTitle ftcontains
    "manuscript guides" with stemming
  and $book//componentTitle ftcontains
    "user profiling" with stemming
return $book/metadata/title
```

AFTX translation:

$\pi_{/\texttt{book/metadata/title}}($

$\varsigma_{/\texttt{book//componentTitle}[\text{“}\texttt{user profiling}\text{”},\texttt{stem}]}($

$\varsigma_{/\texttt{book/metadata/title.A}[\text{“}\texttt{shortTitle}\text{”}][\text{“}\texttt{manuscript guides}\text{”},\texttt{stem}]}($

$\pi_{/\texttt{books/book}}(\text{“}\texttt{full-text.xml}\text{”}))))$

Example 4.33 [Use Case “THESAURUS” Q1] Find all introductions which quote someone.

XQuery solution:

```
for $book in doc("full-text.xml")/books/book
where $book//introduction ftcontains "quote"
   with thesaurus default
return $book
```

This query has been slightly modified by using the default thesaurus and the default relationship between words.

AFTX translation:

$\iota_{/\texttt{book}}($

$\varsigma_{/\texttt{book//introduction}[\text{“}\texttt{quote}\text{”},\texttt{thes}]}($

$\pi_{/\texttt{books/book}}(\text{“}\texttt{full-text}\text{”})))$

Example 4.34 [Use Case “STOP-WORD” Q1] Find all books with the phrase “planning then conducting” in the text where “then” is treated as a stop word.

XQuery solution:

```
for $book in doc("full-text.xml")
    /books/book
where $book//content ftcontains "planning then conducting"
    with default stop words
return $book
```

AFTX translation:

$\iota_{/\texttt{book}}($

$\varsigma_{/\texttt{book//content}[\text{“}\texttt{planning then conducting}\text{”},\texttt{stop}]}($

$\pi_{/\texttt{books/book}}(\text{“}\texttt{full-text}\text{”})))$

Example 4.35 [Use Case “LOGICAL” Q1] Find all books with the words “web” or “software” in the text.

XQuery solution:

```
for $book in doc("full-text.xml")/books/book
where $book//content ftcontains "web" || "software"
return $book
```

AFTX translation:

$\iota_{/\texttt{book}}($

$\varsigma_{/\texttt{book//content}[\text{“}\texttt{web}\text{”}\ \texttt{OR}\ \text{“}\texttt{software}\text{”}]}($

$\pi_{/\texttt{books/book}}(\text{“}\texttt{full-text}\text{”})))$

Example 4.36 [Use Case “PROXIMITY” Q1] Find all books with information on “software developers”. The query must find multiple words in any order allowing up to three intervening words.

XQuery Full-Text solution:

```
for $book in doc("full-text.xml")/books/book
where $book//content ftcontains
    "software" && "developer" with stemming
    distance at most 3 words
return $book
```

AFTX translation:

$\iota_{/\texttt{book}}($

$\varsigma_{/\texttt{book//content}[\text{“}\texttt{sofware}\text{”}\ \texttt{AND}\ \text{“}\texttt{developers}\text{”},\texttt{3},\texttt{stem}]}($

$\pi_{/\texttt{books/book}}(\text{“}\texttt{full-text.xml}\text{”})))$

Example 4.37 [Use Case "AXES" Q1] Find all books with paragraphs containing the phrase "computer workstation" and footnotes within those paragraphs containing the word "comfortable".

XQuery solution:

```
for $book in doc("full-text.xml")/books/book

let $para := $book//p[. ftcontains "computer workstation"],
    $fn := $para/footnote[. ftcontains "comfortable"]
where count($fn)>0
return $book/metadata/title, $para
```

AFTX translation:

$$\iota_{/\texttt{prod\_root/prod\_root/book},/\texttt{prod\_root/prod\_root/let\_root/p}}($$
$$\sigma_{/\texttt{prod\_root/footnote}[.\texttt{count}>0]}($$
$$(\pi_{/\texttt{books/book}}(\text{“}\texttt{full-text.xml}\text{”}) \bowtie_{/\texttt{book}[//\texttt{p}\equiv/\texttt{let\_root/p}]}$$
$$\iota_{\text{“}\texttt{let\_root}\text{”}(\texttt{null},\texttt{null},\texttt{null})}(\varsigma_{/\texttt{p}[\text{“}\texttt{computer workstation}\text{”}]}(\pi_{/\texttt{book//p}}(\pi_{/\texttt{books/book}}($$
$$\text{“}\texttt{full-text.xml}\text{”})))))) \bowtie_{/\texttt{prod\_root/let\_root/p}[/\texttt{footnote}\equiv/\texttt{let\_root/footnote}]}$$
$$\iota_{\text{“}\texttt{let\_root}\text{”}(\texttt{null},\texttt{null},\texttt{null})}(\varsigma_{/\texttt{footnote}[\text{“}\texttt{comfortable}\text{”}]}(\pi_{/\texttt{p/footnote}}($$
$$\iota_{\text{“}\texttt{let\_root}\text{”}(\texttt{null},\texttt{null},\texttt{null})}(\varsigma_{/\texttt{p}[\text{“}\texttt{computer workstation}\text{”}]}(\pi_{/\texttt{book//p}}($$
$$\pi_{/\texttt{books/book}}(\text{“}\texttt{full-text.xml}\text{”}))))))))$$

This quite complex query is shown graphically in Figure 4.4. The figure shows that some partial results can be built on the basis of previously calculated partial results.

4.4 About XML Updates

4.4.1 XQuery Update Facility

Recently W3C has published a working draft, called XQuery Update Facility [Con06h], for extending XQuery with update capabilities. In particular, the XQuery Update Facility provides facilities to perform any or all of the following operations on an instance of the XQuery Data Model:

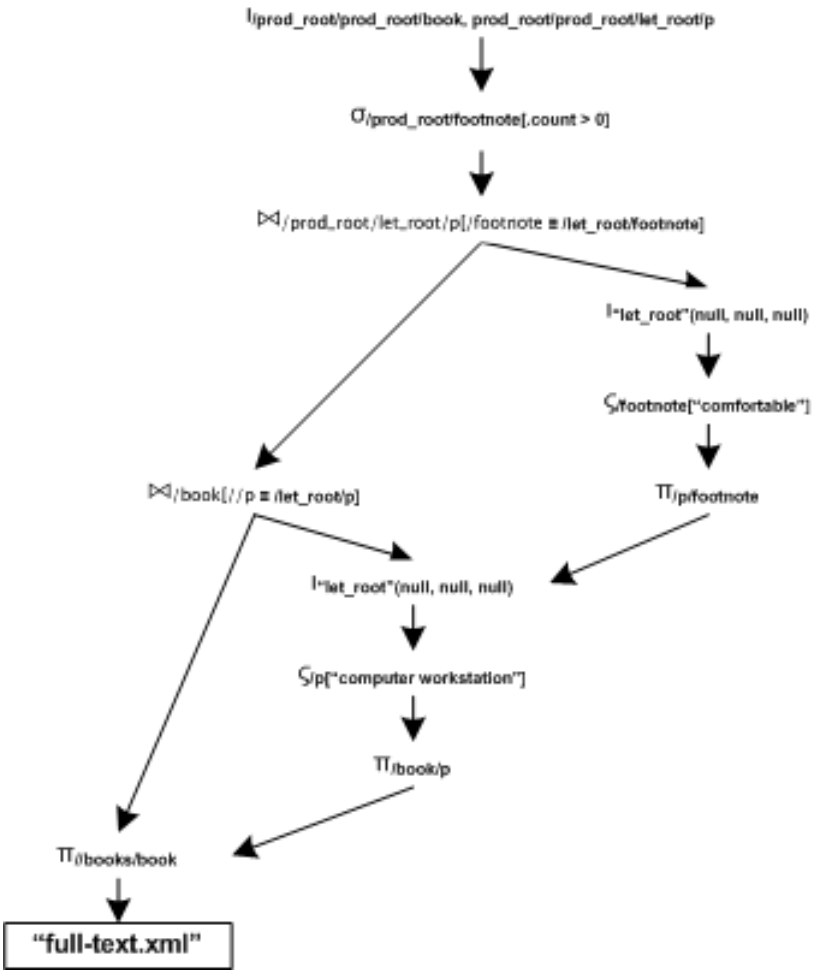


Figure 4.4: Graphical representation of the AFTX expression of Example 4.37.

- insertion of a node;
- deletion of a node;
- modification of a node by changing some of its properties while preserving its identity;
- creation of a modified copy of a node with a new identity.

Insertion of a node is performed through the expression

$$\texttt{do insert}\ \mathit{NewNodes}\ \mathit{Where}\ \mathit{OldNode}\ .$$

Its result is the insertion of the result of the XQuery expression *NewNodes* in a position specified by *Where* with respect to the node resulting from the XQuery expression *OldNode*. For example, the expression

```
do insert <year>2005</year>
    after fn:doc("bib.xml")/books/book[1]/publisher
```

inserts a new element named `year`, with value `2005`, as the following sibling of the `publisher` sub-element of the first `book` sub-element of the root element `books`.

Deletion of a node is performed through the expression

`do delete` $OldNodes$.

Its result is the deletion of the nodes resulting from the XQuery expression *OldNodes*. For example, the expression

```
do delete fn:doc("bib.xml")/books/book[1]/author[last()]
```

deletes the last author of the first book in a given bibliography.

Replacement of a node with a new sequence of zero or more nodes is performed through the expression

`do replace` $OldNode$ `with` $NewNodes$.

Its result is the replacement of the node resulting from the XQuery expression *OldNode* with the sequence resulting from the XQuery expression *NewNodes*. For example, the expression

```
do replace fn:doc("bib.xml")/books/book[1]/publisher
    with fn:doc("bib.xml")/books/book[2]/publisher
```

replaces the publisher of the first book with the publisher of the second book. Using the optional clause `value of`, the value of the node is modified while preserving its node identity. For example, the expression

```
do replace value of fn:doc("bib.xml")/books/book[1]/price
    with fn:doc("bib.xml")/books/book[1]/price * 1.1
```

increases the price of the first book by ten percent.

It is also possible to rename a node, using the expression

do `rename` $OldNode$ `as` $NewName$.

Its result is the renaming of the node resulting from the XQuery expression *OldNode* with the name resulting from the XQuery expression *NewName*. For example, the expression

```
do rename fn:doc("bib.xml")/books/book[1]/author[1]
    as $newname
```

renames the first `author` element of the first book to the QName that is the value of the variable `$newname`.

Finally, a transform expression can be used to create modified copies of existing nodes in an XDM instance. The expression

`transform copy` $VarName := OldNodes$ `modify` $UpdateExpr$ `return` $Expr$

creates a copy (bound to the variable *VarName*) of the nodes resulting from the XQuery expression *OldNodes*, modifies the copy according to the update expression *UpdateExpr* and returns the result of *Expr*. For example the expression

```
for $e in //employee[skill = "Java"]
    return
        transform
            copy $je := $e
            modify do delete $je/salary
            return $je
```

returns a sequence consisting of all `employee` elements that have Java as a skill, excluding their `salary` child-elements.

4.4.2 Expressing updates in AFTX

The definition of operators and translation techniques for expressing XQuery update operations into the AFTX algebra is beyond the scope of this thesis, and represents a valuable future research direction. However, we want to present in this section some informal ideas on how such a process could be carried out.

The first thing to notice is that a large part of the semantics of the update operations can be expressed using the algebraic operators already defined. In particular:

- *OldNode* in the `do insert` expression is a query, therefore it can be translated into an algebraic expression as explained in the previous sections of this chapter; *NewNodes* is also a query, possibly including some element construction specification, therefore it can be translated into an algebraic expression.
- *OldNodes* in the `do delete` expression is a query, therefore it can be translated into an algebraic expression.
- *OldNode* and *NewNodes* in the `do replace` expression are queries, therefore they can be translated into two algebraic expressions.
- *OldNode* in the `do rename` expression is a query, therefore it can be translated into an algebraic expression.
- *OldNodes* in the `do transform` expression is a query, therefore it can be translated into an algebraic expression.

Consequently, what should be done in order to express updates in AFTX is the definition of:

- an *insert* operator, which takes as input a node (i.e. a forest formed by a single tree including a single element) corresponding to *OldNode* and a forest corresponding to *NewNodes*; its predicate should indicate where to insert the new nodes with respect to the old node;

- a *replace* operator, which again takes as input a node corresponding to *OldNode* and a forest corresponding to *NewNodes*;
- a *transform* operator, which takes as input a forest corresponding to *OldNodes*, an AFTX expression corresponding to *UpdateExpr* and an AFTX expression corresponding to *Expr*.

For what concerns `do delete` expressions, it seems evident that its semantics is identic to that of the deletion operator defined in Chapter 3. Therefore its translation into an AFTX expression should be quite straightforward.

Chapter 5

Query Optimization

One of the main motivations for the definition of an algebra is the possibility to study optimization techniques that rely on some properties of the proposed algebra. In this section we show the most important properties of our operators.

It is worth specifying that the kind of optimization we study in this chapter is a *logical* optimization; the definition of performing algorithms that implement the algebraic operators, possibly using some access support structures, is beyond the scope of our doctoral work. Anyway such a *physical* optimization is unquestionably one the main interesting challenges in the development of a working XML database system, and is therefore one of the possible future research areas, as discussed later in Chapter 7.

We start in Section 5.1 by defining the kind of relations between algebraic expressions we are interested in. In Section 5.2 we present the first block of rules, which resemble similar well-known rules holding in relational algebra. In Section 5.3 we present rules which are instead intended to optimize expressions resulting from the translation of XQuery nested expressions.

5.1 Algebraic Properties of Interest

The goal of this chapter is to establish a set of rewriting rules which permit to substitute an algebraic expression A with an (hopefully more performing) algebraic expression A'.

Some sort of relationship must exist between A and A' for the rewriting to be worth. We define three kinds of relationships: *equivalence*, *containment*, and *similarity*.

Equivalence rules (indicated with $\equiv$) state that the two involved algebraic expressions always return two strictly equal forests. Equivalence is clearly the most attracting relationship, because the first expression can be safely transformed into the second one, in order to improve performance.

Definition 5.1 (Expression Equivalence) *Let x be a forest and let $A(x)$ and $B(x)$ be two AFTX expressions. A and B are equivalent (denoted $A \equiv B$) if, for any input forest x, they return two forests $F_x = (T_1, T_2, \ldots, T_n)$ and $F'_x = (T'_1, T'_2, \ldots, T'_n)$ such that $F_x \equiv F'_x$.*

Note that the kind of equivalence we consider is *absolute* equivalence: the result of $A(x)$ is equivalent to that $B(x)$, regardless of the input forest x; the same clarification holds for containment and similarity rules.

Containment rules (indicated with $\subset$) state that the first algebraic expression always returns a subforest of the forest returned by the second algebraic expression. Even if the two expressions are not equivalent, it can be sometimes worth to substitute the first expression with the second one, if it can be answered more quickly; while doing such substitution, however, it should be taken into account the fact that a subsequent selection is needed in order to eliminate false positives.

Definition 5.2 (Expression Containment) *Let x be a forest and let $A(x)$ and $B(x)$ be two AFTX expressions. A is contained into B (denoted $A \subset B$) if, for any input forest x, it returns a forest $F_x = (T_1, T_2, \ldots, T_n)$ such that $F_x \subset F'_x$, where $F'_x = (T'_1, T'_2, \ldots, T'_m)$ is the forest returned by $B(x)$.*

The equivalence relationship previously defined states that two expressions return two strictly equal forests. Remember that forest strict equality means that:

- the forests contain the same trees;
- the trees are in the same order.

Similarity rules (indicated with $\cong$), instead, state that the two involved algebraic expressions return two forests containing the same trees, but in a (possibly) different order. In general order is significative in the semi-structured world, therefore such transformations can be done only if ordering is not a matter, for example if a subsequent ordering operation must be performed.

Definition 5.3 (Expression Similarity) *Let x be a forest and let $A(x)$ and $B(x)$ be two AFTX expressions. A and B are similar (denoted $A \cong B$) if, for any input forest x, they return two forests $F_x = (T_1, T_2, \ldots, T_n)$ and $F'_x = (T'_1, T'_2, \ldots, T'_n)$ such that:*

- $\forall T_i \in F_x$, $\exists T'_j \in F'_x$ *such that* $T_i \equiv T'_j$;
- $\forall T'_j \in F'_x$, $\exists T_i \in F_x$ *such that* $T'_j \equiv T_i$.

Note that, for two expressions to be similar, order of trees can be different but order of elements must be the same, otherwise tree would not be strictly equal.

Example 5.1 Consider the forest F in Figure 5.1. It could be obtained using the following algebraic expression A:

$$\iota_{\text{“book”}(\texttt{null},\texttt{null},(/\texttt{book}/\texttt{title},/\texttt{book}/\texttt{price}))}($$
$$\sigma_{/\texttt{book}[/\texttt{publisher}=\text{“Addison-Wesley”}]}(\pi_{/\texttt{bib}/\texttt{book}}(\text{“}\texttt{bib.xml}\text{”}))) \ ,$$

where `bib.xml` is the XML document in Figure 2.3. Consider now the forests F', F'', and F''', also shown in Figure 5.1, and suppose they are obtained by three algebraic expressions A', A'', and A'''. Then:

- $A' \subset A$: F' contains part of the trees contained in F, in the same order;
- $A'' \cong A$, but $A'' \not\equiv A$: F'' contains the same trees of F, but in a different order;
- $A''' \ncong A$, because the first tree in F''' has not a corresponding strictly equal tree in F.

Note that, for the inclusion and similarity properties to hold, the containment/similarity relationship between resulting forests must be valid for any input forest.

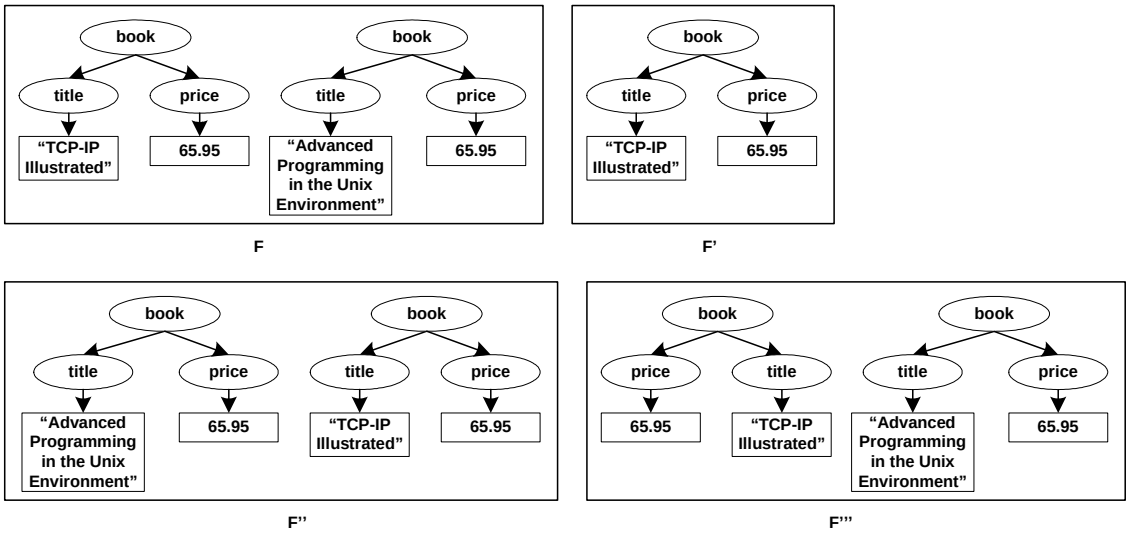


Figure 5.1: A sample forest (F), a contained forest (F'), a similar forest (F''), and a non-similar forest (F''').

5.2 Relational-like Rules

The rules presented in this section and summarized in Table 5.1 are inspired by similar rules holding in relational algebra. Such a similarity is one of the advantages of having defined an algebra whose operators are inspired by relational algebra operators. The rules can be used to leverage performances, either reducing the size of partial results or permitting the usage of available auxiliary data structures, like indexes.

In what follows we analyze the relational-like rules. For each of them we present an informal overview, the formal theorem (Theorem 5.x corresponds to Rule x), and the proof of the theorem. Rewriting examples complete the treatment of the subject.

5.2.1 Idempotency

Rule 1 states that a projection involving a path expression composed by a single step of the form `/1` or `/*` can be safely removed. In fact both path expressions retrieve the root element of each input tree, regardless its name, thus returning the input forest without any changes.

Table 5.1: Relational-like optimization rules.

1. Projection Idempotency	$\pi_{/1}(F) \equiv F \ , \ \pi_{/*}(F) \equiv F$
2. Projection Decomposition	$\pi_{\lambda_1\lambda_2}(F) \equiv \pi_{/1\lambda_2}(\pi_{\lambda_1}(F))$
3. Selection Decomposition	$\sigma_{\lambda[\gamma_1 \texttt{ AND } \gamma_2]}(F) \subset \sigma_{\lambda[\gamma_2]}(\sigma_{\lambda[\gamma_1]}(F))$
4. Full-Text (FT) Selection Decomposition	$\varsigma_{\lambda[\gamma_1 \texttt{ AND } \gamma_2,\texttt{stem,thes,stop}]}(F) \subset$ $\sigma_{\lambda[\gamma_2,\texttt{stem,thes,stop}]}(\sigma_{\lambda[\gamma_1,\texttt{stem,thes,stop}]}(F))$
5. Selection Disjunction	$\sigma_{\lambda[\gamma_1 \texttt{ OR } \gamma_2]}(F) \cong \sigma_{\lambda[\gamma_1]}(F) \cup (\sigma_{\lambda[\gamma_2]}(F) - \sigma_{\lambda[\gamma_1]}(F))$
6. FT Selection Disjunction	$\varsigma_{\lambda a[\gamma_1 \texttt{ OR } \gamma_2]}(F) \cong \varsigma_{\lambda a[\gamma_1]}(F) \cup (\varsigma_{\lambda a[\gamma_2]}(F) - \varsigma_{\lambda a[\gamma_1]}(F))$
7. Selection Push-Down	$\sigma_{/\texttt{prod\_root}\lambda[\gamma]}(F \times G) \equiv F \times (\sigma_{\lambda[\gamma]}(G)) \ ,$ $\sigma_{/\texttt{prod\_root}\lambda[\gamma]}(F \bowtie_P G) \equiv F \bowtie_P (\sigma_{\lambda[\gamma]}(G))$
8. FT Selection Push-Down	$\varsigma_{/\texttt{prod\_root}\lambda a[\gamma,x]}(F \times G) \equiv F \times (\varsigma_{\lambda a[\gamma,x]}(G)) \ ,$ $\varsigma_{/\texttt{prod\_root}\lambda a[\gamma,x]}(F \bowtie_P G) \equiv F \bowtie_P (\varsigma_{\lambda a[\gamma,x]}(G))$
9. FT Score Assignment Push-Down	$\xi_{/\texttt{prod\_root}/\lambda a[\gamma,x]f}(F \times G) \equiv F \times (\xi_{\lambda a[\gamma,x]f}(G)) \ ,$ $\xi_{/\texttt{prod\_root}/\lambda a[\gamma,x]f}(F \bowtie_P G) \equiv F \bowtie_P (\xi_{\lambda a[\gamma,x]f}(G))$
10. Selection Distributivity	$\sigma_P(F \cup G) \equiv \sigma_P(F) \cup \sigma_P(G) \ ,$ $\sigma_P(F - G) \equiv \sigma_P(F) - \sigma_P(G)$
11. FT Selection Distributivity	$\varsigma_P(F \cup G) \equiv \varsigma_P(F) \cup \varsigma_P(G) \ ,$ $\varsigma_P(F - G) \equiv \varsigma_P(F) - \varsigma_P(G)$
12. Projection Distributivity	$\pi_P(F \cup G) \equiv \pi_P(F) \cup \pi_P(G)$
13. Deletion Distributivity	$\delta_P(F \cup G) \equiv \delta_P(F) \cup \delta_P(G) \ ,$ $\delta_P(F - G) \equiv \delta_P(F) - \delta_P(G)$
14. Product and Join Distributivity	$F \times (G_1 \cup G_2) \cong (F \times G_1) \cup (F \times G_2) \ ,$ $F \bowtie_P (G_1 \cup G_2) \cong (F \bowtie_P G_1) \cup (F \bowtie_P G_2)$
15. Union Associativity	$(F_1 \cup F_2) \cup F_3 \equiv F_1 \cup (F_2 \cup F_3)$
16. Union Commutativity	$F \cup G \cong G \cup F$
17. Product and Join Commutativity	$F \times G \cong \iota_P(G \times F) \ ,$ $F \bowtie_P G \cong \iota_{P_2}(G \bowtie'_P F)$
18. Product and Join Associativity	$(F_1 \times F_2) \times F_3 \cong \iota_P(F_1 \times (F_2 \times F_3)) \ ,$ $(F_1 \bowtie_{P_1} F_2) \bowtie_{P_2} F_3 \cong \iota_P(F_1 \bowtie_{P'_1} (F_2 \bowtie_{P'_2} F_3))$
19. Selection Commutativity	$\sigma_{P_1}(\sigma_{P_2}(F)) \equiv \sigma_{P_2}(\sigma_{P_1}(F))$
20. FT Selection Commutativity	$\varsigma_{P_1}(\varsigma_{P_2}(F)) \equiv \varsigma_{P_2}(\varsigma_{P_1}(F))$

Theorem 5.1 (Projection Idempotency) *Let F be a forest. Then the following equivalence relation holds:*

$$\pi_{/1}(F) \equiv F \ , \ \pi_{/*}(F) \equiv F \tag{5.1}$$

Proof: The proof comes directly from Definition 3.17 of path expression and Definition 3.19 of projection. □

5.2.2 Decomposition

Rule 2 considers the application of the projection operator to a forest. The projection predicate is a path expression that can be composed by multiple steps; that path expression can be *decomposed*, thus transforming a single projection operation in multiple projection operations. Projection decomposition can speed up the evaluation of the query, because it can permit to use access support structures such as path indexes.

Example 5.2 Consider the XML document in Figure 2.3 and suppose we want to extract the last name of each author. The following AFTX expression answers the query:

$$\pi_{\texttt{/bib/book/author/last}}(\text{“}\texttt{bib.xml}\text{”}) \ .$$

Suppose now an index structure is available, which permits a fast recovery of the elements reachable following the path expression `/bib/book/author`. Then the previous AFTX expression can be optimized by using Rule 2, thus obtaining the following expression:

$$\pi_{\texttt{/1/last}}(\pi_{\texttt{/bib/book/author}}(\text{“}\texttt{bib.xml}\text{”})) \ .$$

Attention must be posed to the fact that, when splitting a path expression, the final part must be preceded by a `/1` step. In this example, the inner projection returns `author` elements; therefore the last part of the path expression (`/last`) must be headed by a `/author` (or, equivalently, `/1`) step.

Theorem 5.2 (Projection Decomposition) *Let F be a forest and let $\lambda_1\lambda_2$ be a path expression. Then the following equivalence relation holds:*

$$\pi_{\lambda_1\lambda_2}(F) \equiv \pi_{/1\lambda_2}(\pi_{\lambda_1}(F)) \ . \tag{5.2}$$

Proof: We demonstrate the theorem by induction on the number of steps of the two path expressions. As base case, let $\lambda_1 = \alpha_1^1\beta_1^1$ and $\lambda_2 = \alpha_2^1\beta_2^1$ be composed by a single step. Depending on the kind of α_i^1 and β_i^1, the following cases are possible:

1. $\lambda_1 = /1$ and $\lambda_2 = /\beta_2^1$. By Theorem 5.1 $\pi_{/1}(F) = F$; then $\pi_{/1\lambda_2}(\pi_{\lambda_1}(F)) = \pi_{/1\lambda_2}(F) = \pi_{\lambda_1\lambda_2}(F)$.

2. $\lambda_1 = /*$ and $\lambda_2 = /\beta_2^1$. The proof is identical to that of case 1.

3. $\lambda_1 = /s_1$ and $\lambda_2 = /x$, where s_1 is a string and x is an integer. By Definition 3.17 $\pi_{/s_1}(F) = \{T \in F \mid root(T).n = s_1\}$ and $\pi_{/1/x}(F) = \{T' \subset T \mid T \in F \wedge root(T').o = x \wedge root(T').p = root(T)\}$. Then $\pi_{/1\lambda_2}(\pi_{\lambda_1}(F)) = \pi_{/1/x}(\pi_{/s_1}(F)) = \{T' \subset T \mid T \in F \wedge root(T).n = s_1 \wedge root(T').o = x \wedge root(T').p = root(T)\} \stackrel{d}{=} \pi_{/s_1/x}(F) = \pi_{\lambda_1\lambda_2}$.

4. $\lambda_1 = /s_1$ and $\lambda_2 = /*$, where s_1 is a string. By Definition 3.17 $\pi_{/s_1}(F) = \{T \in F \mid root(T).n = s_1\}$ and $\pi_{/1/*}(F) = \{T' \subset T \mid T \in F \wedge root(T').p = root(T)\}$. Then $\pi_{/1\lambda_2}(\pi_{\lambda_1}(F)) = \pi_{/1/*}(\pi_{/s_1}(F)) = \{T' \subset T \mid T \in F \wedge root(T).n = s_1 \wedge root(T').p = root(T)\} \stackrel{d}{=} \pi_{/s_1/*}(F) = \pi_{\lambda_1\lambda_2}$.

5. $\lambda_1 = /s_1$ and $\lambda_2 = /s_2$, where s_1 and s_2 are strings. By Definition 3.17 $\pi_{/s_1}(F) = \{T \in F \mid root(T).n = s_1\}$ and $\pi_{/1/s_2}(F) = \{T' \subset T \mid T \in F \wedge root(T').n = s_2 \wedge root(T').p = root(T)\}$. Then $\pi_{/1\lambda_2}(\pi_{\lambda_1}(F)) = \pi_{/1/s_2}(\pi_{/s_1}(F)) = \{T' \subset T \mid T \in F \wedge root(T).n = s_1 \wedge root(T').n = s_2 \wedge root(T').p = root(T)\} \stackrel{d}{=} \pi_{/s_1/s_2}(F) = \pi_{\lambda_1\lambda_2}$.

6. $\lambda_1 = //x$ and $\lambda_2 = /\beta_2^1$, where x is an integer. By Definition 3.17 $\pi_{//x}(F) = \{T' \subset T \mid T \in F \wedge root(T')$ is the x-th element (in pre-order enumeration) of $T\}$. Then $\pi_{/1\lambda_2}(\pi_{\lambda_1}(F)) = \pi_{/1/\beta_2^1}(\pi_{//x}(F)) = \{T'' \subset T' \subset T \mid T \in F \wedge root(T')$ is the x-th element (in pre-order enumeration) of $T \wedge root(T'').p = root(T') \wedge$ the condition imposed by β_2^1 is satisfied$\} \stackrel{d}{=} \pi_{//x/\beta_2^1} = \pi_{\lambda_1\lambda_2}(F)$.

7. $\lambda_1 = //*$ and $\lambda_2 = /\beta_2^1$. By Definition 3.17 $\pi_{//*}(F) = \{T' \subset T \mid T \in F\}$. Then $\pi_{/1\lambda_2}(\pi_{\lambda_1}(F)) = \pi_{/1/\beta_2^1}(\pi_{//*}(F)) = \{T'' \subset T' \subset T \mid T \in F \wedge root(T'').p =$

$root(T') \wedge$ the condition imposed by β_2^1 is satisfied$\} \stackrel{d}{=} \pi_{//*/\beta_2^1} = \pi_{\lambda_1\lambda_2}(F)$.

8. $\lambda_1 = //s_1$ and $\lambda_2 = /\beta_2^1$, where s_1 is a string. By Definition 3.17 $\pi_{//s_1}(F) = \{T' \subset T \mid T \in F \wedge root(T').n = s_1\}$. Then $\pi_{/1\lambda_2}(\pi_{\lambda_1}(F)) = \pi_{/1/\beta_2^1}(\pi_{//s_1}(F)) = \{T'' \subset T' \subset T \mid T \in F \wedge root(T').n = s_1 \wedge root(T'').p = root(T') \wedge$ the condition imposed by β_2^1 is satisfied$\} \stackrel{d}{=} \pi_{//s_1/\beta_2^1} = \pi_{\lambda_1\lambda_2}(F)$.

9. $\lambda_1 = //\beta_1^1$ and $\lambda_2 = /\beta_2^1$. The proof is similar to that of cases 6–8, except that the condition $root(T'').p = root(T')$ is removed.

Now let $\lambda_1 = \alpha_1^1\beta_1^1$ be composed by a single step and let $\lambda_2 = \alpha_2^1\beta_2^1\alpha_2^2\beta_2^2 \ldots \alpha_2^n\beta_2^n$ be composed by n steps. By inductive hypothesis

$$\pi_{\alpha_1^1\beta_1^1\alpha_2^1\beta_2^1\alpha_2^2\beta_2^2 \ldots \alpha_2^{n-1}\beta_2^{n-1}}(F) = \pi_{/1\alpha_2^1\beta_2^1\alpha_2^2\beta_2^2 \ldots \alpha_2^{n-1}\beta_2^{n-1}}(\pi_{\alpha_1^1\beta_1^1}(F)) \ .$$

Depending on the kind of α_2^n the following cases are possible:

- $\alpha_2^n =$ "/". By Definition 3.17 $\pi_{\lambda_1\lambda_2}(F) = \{T' \subset T \mid T \in \pi_{\alpha_1^1\beta_1^1\alpha_2^1\beta_2^1\alpha_2^2\beta_2^2 \ldots \alpha_2^{n-1}\beta_2^{n-1}}(F) \wedge root(T').p = root(T) \wedge$ the condition imposed by β_2^n is satisfied$\} = \{T' \subset T \mid T \in \pi_{/1\alpha_2^1\beta_2^1\alpha_2^2\beta_2^2 \ldots \alpha_2^{n-1}\beta_2^{n-1}}(\pi_{\alpha_1^1\beta_1^1}(F)) \wedge root(T').p = root(T) \wedge$ the condition imposed by β_2^n is satisfied$\} \stackrel{d}{=} \pi_{/1\alpha_2^1\beta_2^1\alpha_2^2\beta_2^2 \ldots \alpha_2^{n-1}\beta_2^{n-1}/\beta_2^n}(\pi_{\alpha_1^1\beta_1^1}(F)) = \pi_{/1\lambda_2}(\pi_{\lambda_1}(F))$.

- $\alpha_2^n =$ "//". By Definition 3.17 $\pi_{\lambda_1\lambda_2}(F) = \{T' \subset T \mid T \in \pi_{\alpha_1^1\beta_1^1\alpha_2^1\beta_2^1\alpha_2^2\beta_2^2 \ldots \alpha_2^{n-1}\beta_2^{n-1}}(F) \wedge$ the condition imposed by β_2^n is satisfied by $root(T')\} = \{T' \subset T \mid T \in \pi_{/1\alpha_2^1\beta_2^1\alpha_2^2\beta_2^2 \ldots \alpha_2^{n-1}\beta_2^{n-1}}(\pi_{\alpha_1^1\beta_1^1}(F)) \wedge$ the condition imposed by β_2^n is satisfied by $root(T')\} \stackrel{d}{=} \pi_{/1\alpha_2^1\beta_2^1\alpha_2^2\beta_2^2 \ldots \alpha_2^{n-1}\beta_2^{n-1}//\beta_2^n}(\pi_{\alpha_1^1\beta_1^1}(F)) = \pi_{/1\lambda_2}(\pi_{\lambda_1}(F))$.

□

Rules 3 and 4 state that an equivalence rule similar to that of Rule 2 does not hold for selection and full-text selection; while in relational algebra $\sigma_{F_1 \wedge F_2}(E) \equiv \sigma_{F_1}(\sigma_{F_2}(E))$, in AFTX the following more general containment rules hold:

$$\sigma_{\lambda[\gamma_1 \texttt{ AND } \gamma_2]}(F) \subset \sigma_{\lambda[\gamma_2]}(\sigma_{\lambda[\gamma_1]}(F)) \ ;$$

$$\varsigma_{\lambda[\gamma_1 \texttt{ AND } \gamma_2,\texttt{stem},\texttt{thes},\texttt{stop}]}(F) \subset \sigma_{\lambda[\gamma_2,\texttt{stem},\texttt{thes},\texttt{stop}]}(\sigma_{\lambda[\gamma_1,\texttt{stem},\texttt{thes},\texttt{stop}]}(F)) \ .$$

This is due to the fact that, while in relational algebra every tuple resulting from $\sigma_{F_2}(E)$ is guaranteed to satisfy the selection condition F_2, in AFTX a subtree $T' \in \pi_\lambda(\sigma_{\lambda[\gamma_1]}(F))$ (respectively $T' \in \pi_\lambda(\varsigma_{\lambda[\gamma_1,\texttt{stem},\texttt{thes},\texttt{stop}]}(F))$) is not guaranteed to satisfy the selection condition $[\gamma_1]$ (respectively $[\gamma_1, \texttt{stem}, \texttt{thes}, \texttt{stop}]$). The key point is that AFTX selection and full-text selection have an existential semantic: a tree satisfies a (full-text) selection condition if at least one of its subtrees satisfies it. For example, given an XML document named `"books2.xml"`, the book shown in Figure 5.2 would be contained in the result of the query

$$\sigma_{/\texttt{book}/\texttt{author}[/\texttt{first.v}=\text{“}\texttt{Serge}\text{”}]}(\sigma_{/\texttt{book}/\texttt{author}[/\texttt{last.v}=\text{“}\texttt{Abiteboul}\text{”}]}(\pi_{/\texttt{bib}/\texttt{book}}(\text{“}\texttt{books2.xml}\text{”}))))$$

because there is an author (the first one) whose last name is *Abiteboul* and there is an author (the second one) whose first name is *Serge*; contrariwise, such a book would not be contained in the result of the query

$$\sigma_{/\texttt{book}/\texttt{author}[/\texttt{last.v}=\text{“}\texttt{Abiteboul}\text{”}\ \texttt{AND}\ /\texttt{first.v}=\text{“}\texttt{Serge}\text{”}]}(\pi_{/\texttt{bib}/\texttt{book}}(\text{“}\texttt{books2.xml}\text{”}))$$

because there is no author whose last and first name are respectively *Abiteboul* and *Serge*.

```
<book year="2000">
  <title>Data on the Web</title>
  <author><last>Abiteboul</last><first>Peter</first></author>
  <author><last>Buneman</last><first>Serge</first></author>
  <author><last>Suciu</last><first>Dan</first></author>
  <publisher>Morgan Kaufmann Publishers</publisher>
  <price>65.95</price>
</book>
```

Figure 5.2: An XML document showing why selection decomposition is a containment rule.

The selection decomposition containment rule is not valid in general if some basic selection condition are of the form `.pos=`c or `.count=`c. In this case, in fact, the evaluation of the selection condition depends on the entire input forest: changing the input

forest (e.g. applying a selection predicate), the evaluation of such a selection condition changes. For example, consider the XML document in Figure 2.3; the algebraic expression

$$\sigma_{[.\texttt{pos}=1 \texttt{ AND } .\texttt{count}=4]}(\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”}))$$

would return the book “TCP/IP Illustrated”, because it is the first book in a forest containing four books. That book, however, would not be returned by the algebraic expression

$$\sigma_{[.\texttt{count}=4]}(\sigma_{[.\texttt{pos}=1]}(\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”}))))$$

because the forest resulting from $\sigma_{[.\texttt{pos}=1]}(\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”}))$ contains just one book.

For what concerns the full-text decomposition, we have omitted the optional window parameter which, if present, forces the searched words to be at a distance between one and another not greater than x; actually the rule is still valid even if such a parameter is present, but it is discarded when atomizing the full-text selection. In fact the decomposition transforms searching for (say) two words into searching for one word then searching for another word; clearly, the window parameter does not make sense when searching for just one word.

Theorem 5.3 (Selection Decomposition) *Let F be a forest, λ be a path expression, γ_1 and γ_2 be two selection conditions not using the element properties* `.count` *and* `.pos`. *Then the following containment relation holds:*

$$\sigma_{\lambda[\gamma_1 \texttt{ AND } \gamma_2]}(F) \subset \sigma_{\lambda[\gamma_2]}(\sigma_{\lambda[\gamma_1]}(F)) \ . \tag{5.3}$$

Proof: Let $T \in \sigma_{\lambda[\gamma_1 \texttt{ AND } \gamma_2]}(F)$; by Definition 3.22 of selection, $\exists T' \in \pi_\lambda(T)$ such that:

- T' satisfies the selection condition γ_1: this means that $T \in \sigma_{\lambda[\gamma_1]}(F)$;
- T' satisfies the selection condition γ_2: this means that $T \in \sigma_{\lambda[\gamma_2]}(F)$;

Therefore, $T \in \sigma_{\lambda[\gamma_2]}(\sigma_{\lambda[\gamma_1]}(F))$. □

Theorem 5.4 (Full-Text Selection Decomposition) *Let F be a forest, λ be a path expression, γ_1 and γ_2 be two full-text basic selection conditions. Then the following containment relation holds:*

$$\varsigma_{\lambda[\gamma_1 \texttt{ AND } \gamma_2,\texttt{stem},\texttt{thes},\texttt{stop}]}(F) \subset \sigma_{\lambda[\gamma_2,\texttt{stem},\texttt{thes},\texttt{stop}]}(\sigma_{\lambda[\gamma_1,\texttt{stem},\texttt{thes},\texttt{stop}]}(F)) \ . \tag{5.4}$$

Proof: The proof is identical to that of Theorem 5.3. □

Example 5.3 Consider the XML document in Figure 2.3 and suppose we want to retrieve all the books written after 1995 whose price is not greater than 100. The following AFTX expression answers the query:

$$\sigma_{/\texttt{book}[.\texttt{A}[\text{“}\texttt{year}\text{”}].\texttt{v}>1995 \texttt{ AND } /\texttt{price}.\texttt{v}\leq 100]}(\pi_{/\texttt{bib}/\texttt{book}}(\text{“}\texttt{bib.xml}\text{”})) \ .$$

Using Rule 3, the previous expression can be rewritten into the following:

$$\sigma_{/\texttt{book}[/\texttt{price}.\texttt{v}\leq 100]}(\sigma_{/\texttt{book}[.\texttt{A}[\text{“}\texttt{year}\text{”}].\texttt{v}>1995]}(\pi_{/\texttt{bib}/\texttt{book}}(\text{“}\texttt{bib.xml}\text{”}))) \ .$$

In this special case, the two expressions are equivalent: they both return the third book (*Data on the Web*). Why such an equivalence, which does not hold in general, is guaranteed? We should recall again the definition of selection: *“a tree satisfies the selection predicate $\lambda[\gamma_1 \texttt{ AND } \gamma_2 \texttt{ AND } \ldots \texttt{ AND } \gamma_\texttt{n}]$ if exists at least one subtree reachable following the path λ that satisfies each base condition γ_i”*. What happens in general is that, even if a tree T does not satisfy a composed selection predicate, that tree satisfies the (say) two selection conditions obtained by *splitting* the original composed selection because:

- a subtree T' satisfies the first selection condition;
- a different subtree T'' satisfies the second selection condition.

Consequently, we can say that the equivalence rule

$$\sigma_{\lambda[\gamma_1 \texttt{ AND } \gamma_2]}(F) \equiv \sigma_{\lambda[\gamma_2]}(\sigma_{\lambda[\gamma_1]}(F))$$

holds if, for each input tree, there exists only one subtree over which the selection condition can be tested. Formally, the condition for the equivalence to hold is:

$$\forall T \in F,\ F' = \pi_\lambda(T) \text{ contains at most one tree.}$$

This condition is obviously satisfied when, as in the previous example, λ is composed by the single step "$/\beta$", where β is the root element name (or, equivalently, $\beta = 1$ or $\beta = *$). In general, if the input XML document is conforming to an XML Schema [Con01] and λ is such that the schema guarantees that at most one element can be reached following such path, then the equivalence rule is guaranteed to hold.

Example 5.4 Consider a bibliographic XML document similar to that of Figure 2.3, with the difference that each book has exactly one author. For such document, the following equivalence holds:

$\sigma_{\texttt{/book/author[/first.v="Serge"]}}(\sigma_{\texttt{/book/author[/last.v="Abiteboul"]}}(\pi_{\texttt{/bib/book}}(\texttt{"bib2.xml"}))) \equiv$
$\sigma_{\texttt{/book/author[/last.v="Abiteboul" AND /first.v="Serge"]}}(\pi_{\texttt{/bib/book}}(\texttt{"bib2.xml"}))$

Rules 5 and 6 state that a (full text) selection predicate containing two basic conditions connected with the `OR` operator can be transformed into the union of two (full text) selections.

Theorem 5.5 (Selection Disjunction) *Let F be a forest, λ be a path expression, γ_1 and γ_2 be two selection conditions. Then the following similarity relation holds:*

$$\sigma_{\lambda[\gamma_1 \texttt{ OR } \gamma_2]}(F) \cong \sigma_{\lambda[\gamma_1]}(F) \cup (\sigma_{\lambda[\gamma_2]}(F) - \sigma_{\lambda[\gamma_1]}(F)) \ . \tag{5.5}$$

Here we used the similarity relation $\cong$ instead of the equivalence relation $\equiv$. This indicates that the forest resulting from the left hand side expression is not exactly equal to the forest resulting from the right hand side expression: the two forests contain the same trees, but in different order. Recall that the union operator creates a new forest containing the trees of the first forest, followed by the trees of the second forest; consequently a tree which satisfies the selection condition $\lambda[\gamma_1]$ always precedes (in the forest resulting from the right hand side expression) a tree which instead satisfies the selection condition $\lambda[\gamma_2]$, even if the two trees were in reverse order in the input forest.

Example 5.5 Consider the XML document in Figure 2.3; the expression

$$\sigma_{\texttt{/book[/author/last.v=“Abiteboul” OR .A[“year”].v=1994]}}(\pi_{\texttt{/bib/book}}(\texttt{“bib.xml”}))$$

returns the books *TCP/IP Illustrated* (because it satisfies the second condition) and *Data on the Web* (because it satisfies the first condition), in that order, i.e. the order in which they are found in the input XML document. The expression

$$\begin{aligned}&\sigma_{\texttt{/book[/author/last.v=“Abiteboul”]}}(\pi_{\texttt{/bib/book}}(\texttt{“bib.xml”})) \cup \\ &\quad(\sigma_{\texttt{/book[.A[“year”].v=1994]}}(\pi_{\texttt{/bib/book}}(\texttt{“bib.xml”})) - \\ &\quad\sigma_{\texttt{/book[/author/last.v=“Abiteboul”]}}(\pi_{\texttt{/bib/book}}(\texttt{“bib.xml”})))\end{aligned}$$

would instead return the same two books, but in reverse order. In fact *Data on the Web* would be included in the result of the first selection, thus it would be included in the result of the union before any result of the second selection.

Proof: Let $T \in \sigma_{\lambda[\gamma_1 \texttt{ OR } \gamma_2]}(F)$. By Definition 3.22 of selection, either $T \in \sigma_{\lambda[\gamma_1]}(F)$ or $T \in \sigma_{\lambda[\gamma_2]}(F)$. Then, by Definition 3.15 of union and Definition 3.16 of difference, $T \in \sigma_{\lambda[\gamma_1]}(F) \cup (\sigma_{\lambda[\gamma_2]}(F) - \sigma_{\lambda[\gamma_1]}(F))$.

Now let $T \in \sigma_{\lambda[\gamma_1]}(F) \cup (\sigma_{\lambda[\gamma_2]}(F) - \sigma_{\lambda[\gamma_1]}(F))$. By definitions of union and difference either $T \in \sigma_{\lambda[\gamma_1]}(F)$ or $T \in \sigma_{\lambda[\gamma_2]}(F)$. Then, by definition of selection, $T \in \sigma_{\lambda[\gamma_1 \texttt{ OR } \gamma_2]}(F)$. □

Theorem 5.6 (Full-Text Selection Disjunction) *Let F be a forest, λ be a path expression, a (if present) be an attribute name, γ_1 and γ_2 be two basic full-text selection conditions. Then the following similarity relation holds:*

$$\varsigma_{\lambda a[\gamma_1 \texttt{ OR } \gamma_2]}(F) \cong \varsigma_{\lambda a[\gamma_1]}(F) \cup (\varsigma_{\lambda a[\gamma_2]}(F) - \varsigma_{\lambda a[\gamma_1]}(F)) \ . \tag{5.6}$$

Note that, in this theorem and in all the following theorems regarding full-text operators, we omitted the optional parameters `stem`, `thes`, and `stop`. This is just for the sake of simplicity; the theorem is still valid if one or more of such attributes are used.

Proof: The proof is identical to that of Theorem 5.5. □

5.2.3 Pushing Down

A common transformation in relational algebra is the push down of the selection operator with respect to product or join. The same equivalence rules hold in AFTX for selection (Rule 7) and full-text selection (Rule 8), provided that λ is a path expression referred to the forest G. This transformation can have a great impact on the performance, because it reduces the size of forests passed as input to the product/join operator.

Theorem 5.7 (Selection Push-Down) *Let F and G be two forests and γ be a selection condition. Let λ be a path expression such that $\forall T' \in \pi_{/\texttt{prod\_root}\lambda}(F \times G)\ \exists T \in G$ such that $T' \in \pi_\lambda(T)$. Then the two following equivalence relations hold:*

$$\sigma_{/\texttt{prod\_root}\lambda[\gamma]}(F \times G) \equiv F \times (\sigma_{\lambda[\gamma]}(G)) \ , \tag{5.7}$$

$$\sigma_{/\texttt{prod\_root}\lambda[\gamma]}(F \bowtie_P G) \equiv F \bowtie_P (\sigma_{\lambda[\gamma]}(G)) \ . \tag{5.8}$$

Proof: Let us consider the product case. Let $T \in \sigma_{/\texttt{prod\_root}\lambda[\gamma]}(F \times G)$ and let $T' = \pi_{\texttt{prod\_root}/2}(T)$ be the right subtree of its root. By Definition 3.22 of selection, $\exists T'' \in \pi_{/\texttt{prod\_root}\lambda}(T)$ such that $root(T'')$ satisfies the selection condition γ. By hypothesis, $T'' \in \pi_\lambda(T')$. Then, again by definition of selection, $T'' \in \sigma_{\lambda[\gamma]}(G)$. Then, by Definition 3.23 of product, $T \in (F \times (\sigma_{\lambda[\gamma]}(G)))$.

Now let $T \in (F \times (\sigma_{\lambda[\gamma]}(G)))$ and let $T' = \pi_{\texttt{prod\_root}/2}(T)$ be the right subtree of its root. By definition of selection $\exists T'' \in \pi_\lambda(T')$ such that $root(T'')$ satisfies the selection condition γ. By definition of product $T'' \in \pi_{/\texttt{prod\_root}\lambda}(F \times G)$. Then, again by definition of selection, $T \in \sigma_{/\texttt{prod\_root}\lambda[\gamma]}(F \times G)$.

A similar proof can be used to demostrate the join case. □

Theorem 5.8 (Full-Text Selection Push-Down) *Let F and G be two forests, a (if present) be an attribute name, x (if present) be a window option, and γ be a full-text selection condition. Let λ be a path expression such that $\forall T' \in \pi_{/\texttt{prod\_root}\lambda}(F \times G)\ \exists T \in G$ such that $T' \in \pi_\lambda(T)$. Then the two following equivalence relations hold:*

$$\varsigma_{/\texttt{prod\_root}\lambda a[\gamma,x]}(F \times G) \equiv F \times (\varsigma_{\lambda a[\gamma,x]}(G)) \ , \tag{5.9}$$

$$\varsigma_{/\texttt{prod\_root}\lambda a[\gamma,x]}(F \bowtie_P G) \equiv F \bowtie_P (\varsigma_{\lambda a[\gamma,x]}(G)) \ . \tag{5.10}$$

Proof: The proof is identical to that of Theorem 5.7. □

Example 5.6 Consider the algebraic expression of Example 4.25:

$$\iota_{\text{“result”}(\texttt{null},\texttt{null},\texttt{null})}($$
$$\iota_{\text{“warning”}(\texttt{null},\texttt{null},(/1/\texttt{user/name},/1/\texttt{user/rating},/1/\texttt{item/description},/1/\texttt{iteme/reserve\_price}))}($$
$$\sigma_{/\texttt{prod\_root}[/\texttt{item/offered\_by.v}=/\texttt{user/userid.v}]}($$
$$\sigma_{/\texttt{prod\_root/item/reserve\_price}[.\texttt{v}>1000]}($$
$$\sigma_{/\texttt{prod\_root/user/rating}[.\texttt{v}>\text{“C”}]}($$
$$\pi_{//\texttt{user}}(\text{“users.xml”}) \times$$
$$\pi_{//\texttt{item}}(\text{“items.xml”})))))$$

The outer selection compares two properties of elements found in the trees resulting from the product; the first property is referred to an element of the left subtree, while the second one is referred to an element of the right subtree. This is a typical join operation, thus the first transformation we can do is the substitution of the product with a join:

$$\iota_{\text{“result”}(\texttt{null},\texttt{null},\texttt{null})}($$
$$\iota_{\text{“warning”}(\texttt{null},\texttt{null},(/1/\texttt{user/name},/1/\texttt{user/rating},/1/\texttt{item/description},/1/\texttt{item/reserve\_price}))}($$
$$\sigma_{/\texttt{prod\_root/item/reserve\_price}[.\texttt{v}>1000]}($$
$$\sigma_{/\texttt{prod\_root/user/rating}[.\texttt{v}>\text{“C”}]}($$
$$\pi_{//\texttt{user}}(\text{“users.xml”}) \bowtie_{/\texttt{user/userid}[.\texttt{v}=/\texttt{item/offered\_by.v}]}$$
$$\pi_{//\texttt{item}}(\text{“items.xml”}))))$$

The two remaining selections refer to, respectively, the left and right subtrees of the trees resulting from the join. Then, using two times Rule 7, we obtain the following optimized algebraic expression:

$$\iota_{\text{“result”}(\texttt{null},\texttt{null},\texttt{null})}($$
$$\iota_{\text{“warning”}(\texttt{null},\texttt{null},(/1/\texttt{user/name},/1/\texttt{user/rating},/1/\texttt{item/description},/1/\texttt{item/reserve\_price}))}($$
$$\sigma_{/\texttt{user/rating}[.\texttt{v}>\text{“C”}]}(\pi_{//\texttt{user}}(\text{“users.xml”})) \bowtie_{/\texttt{user/userid}[.\texttt{v}=\texttt{item/offered\_by.v}]}$$
$$\sigma_{/\texttt{item/reserve\_price}[.\texttt{v}>1000]}(\pi_{//\texttt{item}}(\text{“items.xml”}))))$$

A similar pushing down optimization can be used also for full-text score assignment (Rule 9). In this case the advantage of the transformation consists in the fact that, for each tree in G, the score is calculated just once.

Theorem 5.9 (Full-Text Score Assignment Push-Down) *Let F and G be two forests, a (if present) be an attribute name, x (if present) be a window option, γ be a full-text selection condition, and f (if present) be a function pointer. Let λ be a path expression such that $\forall T' \in \pi_{/\texttt{prod\_root}\lambda}(F \times G)\ \exists T \in G$ such that $T' \in \pi_\lambda(T)$. Then the two following equivalence relations hold:*

$$\xi_{/\texttt{prod\_root}\lambda a[\gamma,x]f}(F \times G) \equiv F \times (\xi_{\lambda a[\gamma,x]f}(G)) \ , \tag{5.11}$$

$$\xi_{/\texttt{prod\_root}\lambda a[\gamma,x]f}(F \bowtie_P G) \equiv F \bowtie_P (\xi_{\lambda a[\gamma,x]f}(G)) \ . \tag{5.12}$$

Proof: The proof is identical to that of Theorem 5.7. □

It is worth considering the full-text score assignment push-down an equivalence rule; however it must be noted that the two expressions are not exactly equivalent, because, in the right hand side expression, the score property is set for the root element of the right subtree of the tree root element. This slight difference between the first and the second forest must be considered when applying such a transformation, modifying the outer operators' predicate as needed.

Example 5.7 Consider the following AFTX expression:

$\iota_P(\xi_{/\texttt{prod\_root/book}[\text{“}\texttt{XML}\text{”}],\texttt{f}}($
$\quad \pi_{//\texttt{author}}(\text{“}\texttt{authors.xml}\text{”}) \bowtie_{/\texttt{author}[.\texttt{A}[\text{“}\texttt{id}\text{”}].\texttt{v}=/\texttt{book/author.v}]} \pi_{//\texttt{book}}(\text{“}\texttt{books.xml}\text{”})))$
where $P =$ “result”(null, null, (
$\quad$ “author”(/prod_root/author/last, null, null),
$\quad$ “book”(/prod_root/book/title, null, null),
$\quad$ “relevance”(/prod_root.score, null, null)).

There are two input XML documents. The first (*authors.xml*) contains information about authors, the second (*books.xml*) contains information about books. First two projections are executed, obtaining a forest of author (respectively book) trees. Then a join combines each author with each book written by him. Then a full-text score is assigned by searching for the word *XML* into each book. Finally each pair (author, book) is returned, including in the output: the author last name, the book title, the book score.

Suppose now to apply the full-text score assignment push-down rule. The expression is rewritten as follows:

$\iota_{P'}(\pi_{//\texttt{author}}(\text{“}\texttt{authors.xml}\text{”}) \bowtie_{/\texttt{author}[.\texttt{A}[\text{“}\texttt{id}\text{”}].\texttt{v}=/\texttt{book/author.v}]}$
$\xi_{/\texttt{prod\_root/book}[\text{“}\texttt{XML}\text{”}],\texttt{f}}(\pi_{//\texttt{book}}(\text{“}\texttt{books.xml}\text{”})))$

The score of each book, which was read by the tree construction operator using the expression `/prod_root.score`, is now reachable using the expression `prod_root /book.score`. The tree construction predicate P must be therefore changed into P' in order to obtain an equivalent result:

$P' =$ “result”(null, null, (
“author”(/prod_root/author/last, null, null),
“book”(/prod_root/book/title, null, null),
“relevance”(/prod_root/book.score, null, null)).

5.2.4 Distributivity

In relational algebra, the selection operator is distributive with respect to union and difference. Rules 10 and 11 state that the same holds in AFTX for selection and full-text selection, provided that (in the case of basic selection) the selection predicate does not use the element properties `.pos` and `.count`. Also in this case, the goal of the transformation is to reduce the size of the partial results.

Theorem 5.10 (Selection Distributivity) *Let F and G be two forests and let P be a selection predicate not involving the element properties* `.pos` *and* `.count`. *Then the following equivalence relations hold:*

$$\sigma_P(F \cup G) \equiv \sigma_P(F) \cup \sigma_P(G) \ ; \tag{5.13}$$

$$\sigma_P(F - G) \equiv \sigma_P(F) - \sigma_P(G) \ . \tag{5.14}$$

Proof: Let us demonstrate the union case; a similar proof can be used to demonstrate the difference case.

Let $T \in \sigma_P(F \cup G)$. By Definition 3.22 of selection, $T \in (F \cup G)$ and satisfies the selection predicate P. By Definition 3.15 of union, either $T \in F$ or $T \in G$. If $T \in F$, then $T \in \sigma_P(F)$; if $T \in G$, then $T \in \sigma_P(G)$. Then, again by definition of union, $T \in \sigma_P(F) \cup \sigma_P(G)$.

Now let $T \in \sigma_P(F) \cup \sigma_P(G)$. By definition of union either $T \in \sigma_P(F)$ or $T \in \sigma_P(G)$. If $T \in \sigma_P(F)$ (respectively $T \in \sigma_P(G)$), then T satisfies the selection condition P and $T \in F$ (respectively $T \in G$); consequently $T \in (F \cup G)$ and $T \in \sigma_P(F \cup G)$.

We have demonstrated that a similarity relation holds between the two expressions; in order to demonstrate an equivalence relation, we must show that order between trees is respected.

Let $T \in \sigma_P(F \cup G)$. By definition of selection, $T \in (F \cup G)$, then by definition of union either $T \in F$ or $T \in G$; suppose that $T \in F$. Let $T' \in \sigma_P(F \cup G)$; two cases are possible:

- $T' \in F$; suppose that T precedes T' in F. By definition of union, T precedes T' in $F \cup G$. By definition of selection, T precedes T' in $\sigma_P(F \cup G)$ and T precedes T' in $\sigma_P(F)$. Then, again by definition of union, T precedes T' in $\sigma_P(F) \cup \sigma_P(G)$.
- $T' \in G$; then $T \in \sigma_P(F)$ and $T' \in \sigma_P(G)$. By definition of union T precedes T' in $F \cup G$; then, by definition of selection, T precedes T' in $\sigma_P(F \cup G)$. Again by definition of union T precedes T' in $\sigma_P(F) \cup \sigma_P(G)$.

□

Theorem 5.11 (Full-Text Selection Distributivity) *Let F and G be two forests and let P be a full-text selection predicate. Then the following equivalence relations hold:*

$$\varsigma_P(F \cup G) \equiv \varsigma_P(F) \cup \sigma_P(G) \ ; \tag{5.15}$$

$$\varsigma_P(F - G) \equiv \varsigma_P(F) - \varsigma_P(G) \ . \tag{5.16}$$

Proof: The proof is identical to that of Theorem 5.10. □

For what concerns the projection operator, Rule 12 states that it is distributive with respect to union. Like in relational algebra, projection is not distributive with respect to difference.

Theorem 5.12 (Projection Distributivity) *Let F be a forest and let P be a projection predicate. Then the following equivalence relation holds:*

$$\pi_P(F \cup G) \equiv \pi_P(F) \cup \pi_P(G) \ . \tag{5.17}$$

Proof: Let $T \in \pi_P(F \cup G)$. By Definition 3.19 of projection either $T \in \pi_P(F)$ or $T \in \pi_P(G)$. Then by Definition 3.15 of union $T \in (\pi_P(F) \cup \pi_P(G))$.

Now let $T \in (\pi_P(F) \cup \pi_P(G))$. By definition of union either $T \in \pi_P(F)$ or $T \in \pi_P(G)$. Then by definition of projection $T \in (\pi_P(F) \cup \pi_P(G))$.

We have demonstrated that a similarity relation holds between the two expressions; in order to demonstrate an equivalence relation, we must show that order between trees is respected. This can be done using a proof similar to that used in Theorem 5.10. □

Example 5.8 Consider the following AFTX expression:

$$\sigma_{/\texttt{book/price}[.\texttt{v}<100]}(\pi_{/\texttt{bib/book}}(\text{“}\texttt{csbooks.xml}\text{”} \cup \text{“}\texttt{mathbooks.xml}\text{”})) \ .$$

Using Rule 12, we obtain the following equivalent expression:

$$\sigma_{/\texttt{book/price}[.\texttt{v}<100]}(\pi_{/\texttt{bib/book}}(\text{“}\texttt{csbooks.xml}\text{”}) \cup \pi_{/\texttt{bib/book}}(\text{“}\texttt{mathbooks.xml}\text{”})) \ .$$

Then, applying Rule 10, we obtain the final optimized expression:

$$\sigma_{/\texttt{book/price}[.\texttt{v}<100]}(\pi_{/\texttt{bib/book}}(\text{“}\texttt{csbooks.xml}\text{”})) \cup$$
$$\sigma_{/\texttt{book/price}[.\texttt{v}<100]}(\pi_{/\texttt{bib/book}}(\text{“}\texttt{mathbooks.xml}\text{”})) \ .$$

Rule 13 states that the deletion operator is distributive with respect to union and difference.

Theorem 5.13 (Deletion Distributivity) *Let F and G be two forests and let P be a deletion predicate. Then the following equivalence relations holds:*

$$\delta_P(F \cup G) \equiv \delta_P(F) \cup \delta_P(G) \tag{5.18}$$

$$\delta_P(F - G) \equiv \delta_P(F) - \delta_P(G) \ . \tag{5.19}$$

Proof: Let us demonstrate the union case; a similar proof can be used to demonstrate the difference case.

Let $T \in \delta_P(F \cup G)$. By Definition 3.26 of deletion $\exists T' \in (F \cup G)$ such that $T \subset T'$. By Definition 3.15 of union either $T' \in F$ or $T' \in G$. Then either $T \in \delta_P(F)$ or $T \in \delta_P(G)$. Then, again by definition of union, $T \in (\delta_P(F) \cup \delta_P(G))$.

Now let $T \in (\delta_P(F) \cup \delta_P(G))$. By definition of union either $T \in \delta_P(F)$ or $T \in \delta_P(G)$. Then by definition of deletion $T \in \delta_P(F \cup G)$.

We have demonstrated that a similarity relation holds between the two expressions; in order to demonstrate an equivalence relation, we must show that order between trees is respected. This can be done using a proof similar to that used in Theorem 5.10. □

Example 5.9 Consider the following AFTX expression:

$$\delta_{\texttt{/book/author[NOT /country.v="Italy"]}}(\pi_{\texttt{/bib/book}}(\text{“}\texttt{csbooks.xml}\text{”} \cup \text{“}\texttt{mathbooks.xml}\text{”}))\ .$$

Using Rule 12 (in the same way as in Example 5.8) and then Rule 13 we obtain the following equivalent expression:

$$\delta_{\texttt{/book/author[NOT /country.v="Italy"]}}(\pi_{\texttt{/bib/book}}(\text{“}\texttt{csbooks.xml}\text{”}))\cup$$
$$\delta_{\texttt{/book/author[NOT /country.v="Italy"]}}(\pi_{\texttt{/bib/book}}(\text{“}\texttt{mathbooks.xml}\text{”}))\ .$$

Rule 14 states that the product and join operators are distributive with respect to union, up to the order of trees. In order to understand why order is not respected, we must remember how product (and join, which derives from it) combines trees from the two input forests: it first combines the first tree from the first input forest with all the trees (in the order in which they appear) from the second input forest, then the second tree of the first input forest with all the trees from the second input forest, and so on. Figure 5.3(a) shows three sample input forests; it can be noted that order between trees resulting from $F \times (G_1 \cup G_2)$ (Figure 5.3(b)) is different from that of $(F \times G_1) \cup (F \times G_2)$ (Figure 5.3(c)).

Example 5.10 Consider the following AFTX expression:

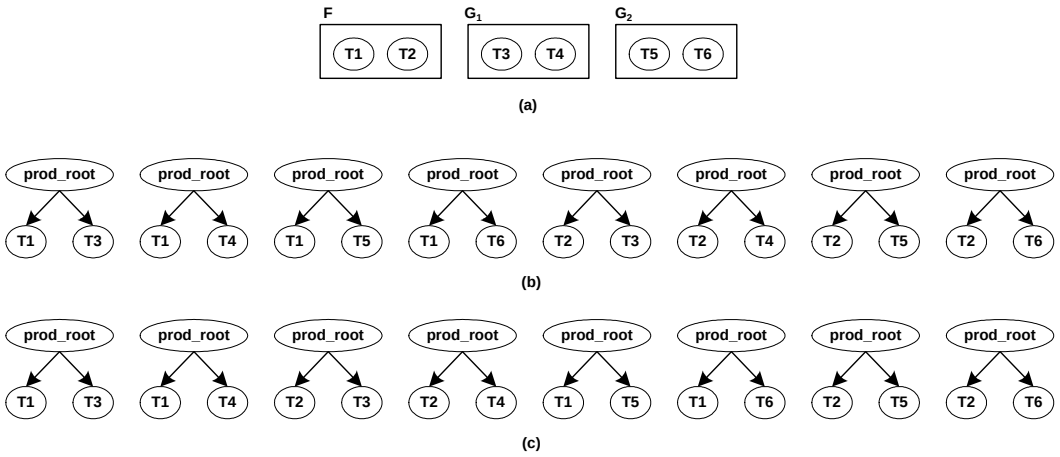


Figure 5.3: Three sample input forests (a), the forest resulting from $F \times (G_1 \cup G_2)$ (b), and the forest resulting from $(F \times G_1) \cup (F \times G_2)$ (c).

$o_{/\texttt{prod\_root/author/last.v ASC,/prod\_root/book/title.v ASC}}($
$\pi_{//\texttt{author}}(\text{“}\texttt{authors.xml}\text{”}) \bowtie_{/\texttt{author[.A[“id”].v=/book/author.v]}} ($
$\pi_{//\texttt{book}}(\text{“}\texttt{csbooks.xml}) \cup \pi_{//\texttt{book}}(\text{“}\texttt{mathbooks.xml}))) \ .$

Using Rule 14, the expression can be rewritten as follows:

$o_{/\texttt{prod\_root/author/last.v ASC,/prod\_root/book/title.v ASC}}($
$(\pi_{//\texttt{author}}(\text{“}\texttt{authors.xml}\text{”}) \bowtie_{/\texttt{author[.A[“id”].v=/book/author.v]}} \pi_{//\texttt{book}}(\text{“}\texttt{csbooks.xml})) \cup$
$(\pi_{//\texttt{author}}(\text{“}\texttt{authors.xml}\text{”}) \bowtie_{/\texttt{author[.A[“id”].v=/book/author.v]}} \pi_{//\texttt{book}}(\text{“}\texttt{mathbooks.xml}))) \ .$

Rule 14 is a similarity rules, not an equivalence; in fact the order of trees resulting from join in the first expression is different from the order of trees resulting from union of joins in the second expression. However we can safely apply such transformation, because there is an outer ordering operator which makes unimportant order of its input forest.

Theorem 5.14 (Product and Join Distributivity) *Let F, G_1 and G_2 be three forests. Then the following similarity relations hold:*

$$F \times (G_1 \cup G_2) \cong (F \times G_1) \cup (F \times G_2) \ ; \tag{5.20}$$

$$F \bowtie_P (G_1 \cup G_2) \cong (F \bowtie_P G_1) \cup (F \bowtie_P G_2) \ . \tag{5.21}$$

Proof: Let us demonstrate the product case; a similar proof can be used to demonstrate the join case.

Let $T \in F \times (G_1 \cup G_2)$. By Definition 3.23 of product:

- $\exists T' \in F$ such that $\pi_{/\texttt{prod\_root}/1}(T) \equiv T'$;
- $\exists T'' \in (G_1 \cup G_2)$ such that $\pi_{/\texttt{prod\_root}/2}(T) \equiv T''$.

By Definition 3.15 of union either $T'' \in G_1$ or $T'' \in G_2$. Then, again by definition of product, either $T \in (F \times G_1)$ or $T \in (F \times G_2)$. Then, again by definition of union, $T \in ((F \times G_1) \cup (F \times G_2))$.

Now let $T \in ((F \times G_1) \cup (F \times G_2))$. By definition of union either $T \in ((F \times G_1)$ or $T \in ((F \times G_2)$; suppose that $T \in ((F \times G_1)$. By definition of product:

- $\exists T' \in F$ such that $\pi_{/\texttt{prod\_root}/1}(T) \equiv T'$;
- $\exists T'' \in G_1$ such that $\pi_{/\texttt{prod\_root}/2}(T) \equiv T''$.

If $\exists T'' \in G_1$, by definition of union $T'' \in (G_1 \cup G_2)$. Then $T \in F \times (G_1 \cup G_2)$. □

5.2.5 Associativity and Commutativity

Rule 15 states that, like in relational algebra, the union operator is associative. Therefore we can safely write a union expression involving three or more input forests, like $F_1 \cup F_2 \cup F_3$.

Theorem 5.15 (Union Associativity) *Let F_1 F_2 and F_3 be three forests. Then the following equivalence relation holds:*

$$(F_1 \cup F_2) \cup F_3 \equiv F_1 \cup (F_2 \cup F_3) \ . \tag{5.22}$$

Proof: The demonstration comes directly from Definition 3.15 of union. □

The relational union operator is also commutative; Rule 16 states that in our algebra the union operator is commutative up to the order of trees. It should be clear why a similarity relation holds instead of an equivalence relation. For example, suppose that $T \in F$ and $T' \in G$; then T precedes T' in $F \cup G$, while T follows T' in $G \cup F$.

Theorem 5.16 *Let F and G be two forests. Then the following equivalence relation holds:*

$$A \cup B \cong B \cup A \ . \tag{5.23}$$

Proof: The demonstration comes directly from Definition 3.15 of union. □

Example 5.11 Consider the following AFTX expression:

$$(\pi_{//\texttt{book}}(\text{“}\texttt{csbooks.xml}\text{”}) \cup \pi_{//\texttt{book}}(\text{“}\texttt{mathbooks.xml}\text{”})) \cup \pi_{//\texttt{book}}(\text{“}\texttt{physicsbooks.xml}\text{”}) \ .$$

Using Rule 15 we can rewrite the expression into the following equivalent one:

$$\pi_{//\texttt{book}}(\text{“}\texttt{csbooks.xml}\text{”}) \cup (\pi_{//\texttt{book}}(\text{“}\texttt{mathbooks.xml}\text{”}) \cup \pi_{//\texttt{book}}(\text{“}\texttt{physicsbooks.xml}\text{”})) \ .$$

Therefore we can safely write the expression as follows:

$$\pi_{//\texttt{book}}(\text{“}\texttt{csbooks.xml}\text{”}) \cup \pi_{//\texttt{book}}(\text{“}\texttt{mathbooks.xml}\text{”}) \cup \pi_{//\texttt{book}}(\text{“}\texttt{physicsbooks.xml}\text{”}) \ .$$

If we now rewrite the last expression, using Rule 16, as follows:

$$\pi_{//\texttt{book}}(\text{“}\texttt{mathbooks.xml}\text{”}) \cup \pi_{//\texttt{book}}(\text{“}\texttt{csbooks.xml}\text{”}) \cup \pi_{//\texttt{book}}(\text{“}\texttt{physicsbooks.xml}\text{”}) \ .$$

we obtain a forest containing the same trees, but in different order.

The product and join operators are not commutative; this difference with respect to the relational algebra is due to the fact that ordering of columns in a relation is not relevant, while ordering of children of a node in an XML document is relevant. However, Rule 17 states that it is possible to obtain the same trees resulting from a product/join operation between two forests by applying the tree construction operator to the result of the opposite product/join operation.

Theorem 5.17 (Product and Join Commutativity) *Let F and G be two forests. Then the following similarity relations hold:*

$$F \times G \cong \iota_{\text{“}\texttt{prod\_root}\text{”}(\texttt{null},\texttt{null},(/\texttt{prod\_root}/2,/\texttt{prod\_root}/1))}(G \times F) \ ; \tag{5.24}$$

$$F \bowtie_P G \cong \iota_{\text{“}\texttt{prod\_root}\text{”}(\texttt{null},\texttt{null},(/\texttt{prod\_root}/2,/\texttt{prod\_root}/1))}(G \bowtie_{P'} F) \ . \tag{5.25}$$

Here P' is the opposite of predicate P, i.e. if $P = \lambda_1[p_1 \theta \lambda_2 p_2]$, then $P' = \lambda_2[p_2 \theta \lambda_1 p_1]$.

Example 5.12 Consider the following AFTX expression:

$$\pi_{//\texttt{author}}(\text{“}\texttt{authors.xml}\text{”}) \bowtie_{/\texttt{author/last}[\texttt{.v=/book/author.v}]} \pi_{//\texttt{book}}(\text{“}\texttt{books.xml}\text{”}) \ .$$

If we apply Rule 17, we obtain the following expression:

$$\iota_{\text{“}\texttt{prod\_root}\text{”}(\texttt{null},\texttt{null},(/\texttt{prod\_root/2},/\texttt{prod\_root/1}))}($$
$$\pi_{//\texttt{book}}(\text{“}\texttt{books.xml}\text{”}) \bowtie_{/\texttt{book/author}[\texttt{.v=/author/last.v}]} \pi_{//\texttt{author}}(\text{“}\texttt{authors.xml}\text{”}))$$

The second expression returns a forest containing the same trees contained in the output of the first expression, but in a different order.

Proof: Let us demonstrate the product case; a similar proof can be used to demonstrate the join case.

Let $T \in (F \times G)$. By Definition 3.23 of product:

- $\exists T' \in F$ such that $\pi_{/\texttt{prod\_root/1}}(T) \equiv T'$;
- $\exists T'' \in G$ such that $\pi_{/\texttt{prod\_root/2}}(T) \equiv T''$.

Then $\exists T_2 \in (G \times F)$ such that:

- $\pi_{/\texttt{prod\_root/1}}(T_2) \equiv T''$;
- $\pi_{/\texttt{prod\_root/2}}(T_2) \equiv T'$.

By Definition 3.34 of tree construction $\iota_{\text{“}\texttt{prod\_root}\text{”}(\texttt{null},\texttt{null},(/\texttt{prod\_root/2},/\texttt{prod\_root/1}))}($ $T_2) \equiv T$. Therefore $T \in \iota_{\text{“}\texttt{prod\_root}\text{”}(\texttt{null},\texttt{null},(/\texttt{prod\_root/2},/\texttt{prod\_root/1}))}(G \times F)$.

Now let $T \in (G \times F)$. By definition of product:

- $\exists T' \in G$ such that $\pi_{/\texttt{prod\_root/1}}(T) \equiv T'$;
- $\exists T'' \in F$ such that $\pi_{/\texttt{prod\_root/2}}(T) \equiv T''$.

Then $\exists T_2 \in (F \times G)$ such that:

- $\pi_{/\texttt{prod\_root/1}}(T_2) \equiv T''$;
- $\pi_{/\texttt{prod\_root/2}}(T_2) \equiv T'$.

By definition of tree construction $\iota_{\text{“prod\_root”}(\texttt{null},\texttt{null},(/\texttt{prod\_root}/2,/\texttt{prod\_root}/1))}(T) \equiv T_2$. Therefore $T_2 \in \iota_{\text{“prod\_root”}(\texttt{null},\texttt{null},(/\texttt{prod\_root}/2,/\texttt{prod\_root}/1))}(G \times F)$. □

It is important to stress on the fact that, in both cases, the first algebraic expression is not equivalent to the second one: the ordering of trees is different in the two cases, as shown in Fig. 5.4.

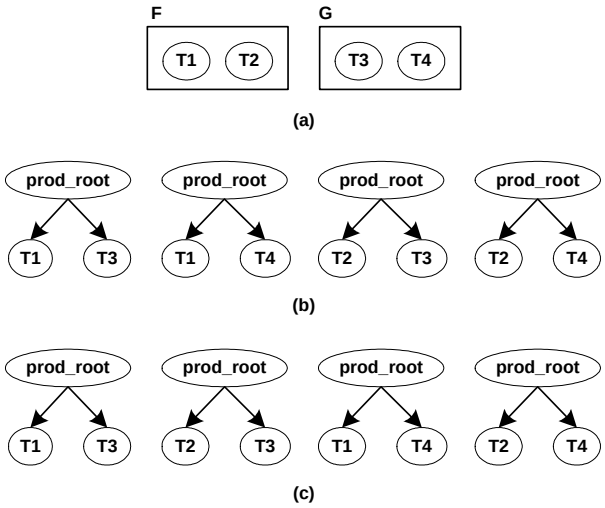


Figure 5.4: Two sample input forests (a), the forests resulting from $F \times G$ (b), and the forest resulting from $\iota_{\text{“prod\_root”}(\texttt{null},\texttt{null},(/\texttt{prod\_root}/2,/\texttt{prod\_root}/1))}(G \times F)$ (c).

The product and join operators are not associative, either. Again, this is due to the relevance of ordering of children of a node in an XML document; moreover, each product operation introduces a new root node called `prod_root`, leading to trees resulting from $(A \times B) \times C$ having a different hierarchical structure to that of trees resulting from $A \times (B \times C)$, as shown in Fig. 5.5. As previously seen for the commutative property, Rule 18 states that it is possible to obtain $(A \times B) \times C$ (up to the ordering of trees) by applying the tree construction operator to the result of $A \times (B \times C)$.

Theorem 5.18 (Product and Join Associativity) *Let F_1, F_2 and F_3 be three forests; let P_1 and P_2 be two join predicates. Then the following similarity relations hold:*

$$(F_1 \times F_2) \times F_3 \cong \iota_P(F_1 \times (F_2 \times F_3)) \tag{5.26}$$

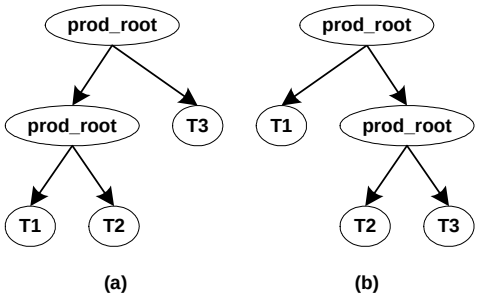


Figure 5.5: The tree resulting from (a) $(T_1 \times T_2) \times T_3$ and (b) $T_1 \times (T_2 \times T_3)$.

$$(F_1 \bowtie_{P_1} F_2) \bowtie_{P_2} F_3 \cong \iota_P(F_1 \bowtie_{P_1'} (F_2 \bowtie_{P_2'} F_3)) \tag{5.27}$$

In both relations $P =$ "`prod_root`"(`null`, `null`, ("`prod_root`"(`null`, `null`, (/`prod_root`/`1`, /`prod_root`/`prod_root`/`1`)), /`prod_root`/`prod_root`/`2`)). *In the second relation, if* $P_1 = \lambda_1[p_1\theta\lambda_2 p_2]$ *and* $P_2 =$ /`prod_root`/$\lambda_3[p_3 = \lambda_4 p_4]$, *then* $P_1' = \lambda_1[p_1\theta$/`prod_root`/$\lambda_2 p_2]$ *and* $P_2' = \lambda_3[p_3\theta\lambda_4 p_4]$.

Proof: Let us demonstrate the product case; a similar proof can be used to demonstrate the join case.

Let $T \in ((F_1 \times F_2) \times F_3)$. By Definition 3.23 of product:

- $\exists T' \in F_1$ such that $\pi_{/\texttt{prod\_root}/\texttt{prod\_root}/\texttt{1}}(T) \equiv T'$;
- $\exists T'' \in F_2$ such that $\pi_{/\texttt{prod\_root}/\texttt{prod\_root}/\texttt{2}}(T) \equiv T''$;
- $\exists T''' \in F_3$ such that $\pi_{/\texttt{prod\_root}/\texttt{2}}(T) \equiv T'''$.

Then $\exists T_2 \in (F_1 \times (F_2 \times F_3))$ such that:

- $\pi_{/\texttt{prod\_root}/\texttt{1}}(T_2) \equiv T'$;
- $\pi_{/\texttt{prod\_root}/\texttt{prod\_root}/\texttt{1}}(T_2) \equiv T''$;
- $\pi_{/\texttt{prod\_root}/\texttt{prod\_root}/\texttt{2}}(T_2) \equiv T'''$.

By Definition 3.34 of tree construction $\iota_P(T_2) \equiv T$, where P is the tree construction predicate defined in the theorem. Therefore $T \in \iota_P(F_1 \times (F_2 \times F_3))$.

Now let $T \in (F_1 \times (F_2 \times F_3))$. By definition of product:

- $\exists T' \in F_1$ such that $\pi_{/\texttt{prod\_root}/\texttt{1}}(T) \equiv T'$;
- $\exists T'' \in F_2$ such that $\pi_{/\texttt{prod\_root}/\texttt{prod\_root}/\texttt{1}}(T) \equiv T''$;
- $\exists T''' \in F_3$ such that $\pi_{/\texttt{prod\_root}/\texttt{prod\_root}/\texttt{2}}(T) \equiv T'''$.

Then $\exists T_2 \in ((F_1 \times F_2) \times F_3)$ such that:

- $\pi_{/\texttt{prod\_root}/\texttt{prod\_root}/\texttt{1}}(T_2) \equiv T'$;
- $\pi_{/\texttt{prod\_root}/\texttt{prod\_root}/\texttt{2}}(T_2) \equiv T''$;
- $\pi_{/\texttt{prod\_root}/\texttt{2}}(T_2) \equiv T'''$.

By definition of tree construction $\iota_P(T) \equiv T_2$. Therefore $T_2 \in \iota_P(F_1 \times (F_2 \times F_3))$. □

Finally, Rules 19 and 20 state that the commutative property is also valid for selection and full-text selection. As in the case of Theorem 5.3, the selection predicate must not make use of the element properties `.count` and `.pos` for the equivalence to be always valid. For example, consider the algebraic expression

$$\sigma_{[\texttt{.pos=3}]}(\sigma_{[\texttt{.count=4}]}(\pi_{/\texttt{bib}/\texttt{book}}(\text{“}books.xml\text{”}))) \ .$$

This expression retrieves the third book out of a forest containing exactly four books, and no book if the forest contains a number of books different from four. If we change the order of selections

$$\sigma_{[\texttt{.count=4}]}(\sigma_{[\texttt{.pos=3}]}(\pi_{/\texttt{bib}/\texttt{book}}(\text{“}books.xml\text{”}))) \ .$$

we first select the third book of the input forest, thus obtaining a forest including just one tree, then we check if the obtained forest contains exactly four books; it should be clear that this expression always returns an empty forest.

Theorem 5.19 (Selection Commutativity) *Let P_1 and P_2 be two selection predicates not using the element properties* `.count` *and* `.pos`*. Then the following equivalence relation holds:*

$$\sigma_{P_1}(\sigma_{P_2}(F)) \equiv \sigma_{P_2}(\sigma_{P_1}(F)) \tag{5.28}$$

Proof: Let $T \in \sigma_{P_1}(\sigma_{P_2}(F))$. By Definition 3.22 of selection, T satisfies both the selection conditions P_1 and P_2. Therefore $T \in \sigma_{P_2}(\sigma_{P_1}(F))$.

Using the same proof we can also demonstrate the inverse containment relationship, thus proving the similarity relationship. □

Theorem 5.20 (Full-Text Selection Commutativity) *Let P_1 and P_2 be two full-text selection predicates. Then the following equivalence relation holds:*

$$\varsigma_{P_1}(\varsigma_{P_2}(F)) \equiv \varsigma_{P_2}(\varsigma_{P_1}(F)) \tag{5.29}$$

Proof: The proof is identical to that of Theorem 5.19. □

Example 5.13 Consider the XML document in Figure 2.3 and the following AFTX expression:

$$\sigma_{\texttt{/book[.A[“year”].v>1995]}}(\sigma_{\texttt{/book/price[.v<100]}}(\pi_{\texttt{//book}}(\texttt{“books.xml”}))) \ .$$

Using Rule 14, the expression can be rewritten into the following:

$$\sigma_{\texttt{/book/price[.v<100]}}(\sigma_{\texttt{/book[.A[“year”].v>1995]}}(\pi_{\texttt{//book}}(\texttt{“books.xml”}))) \ .$$

The two expressions are equivalent. In fact, they both return the book *Data on the Web*.

5.2.6 Derived Full-Text Operators Usage

In Section 3.3 we have defined two useful derived operators, top-K and threshold full-text selection. These operators have no equivalent XQuery Full-Text constructs; it is therefore not surprising that they are never included in the AFTX expressions resulting from XQuery automatic translation (see Chapter 4). However, if the system is able to include, when appropriate, such operators in an algebraic expression, performances could be leverages, because they can be implemented using specialized algorithms.

It is therefore important to understand when these operators can be used, i.e. we must identify special *algebraic patterns* which amount to a top-K (or threshold) operation. In particular, consider an algebraic expression like

$$\sigma_{[.\texttt{pos}\leq\texttt{k}]}(o_{/*.\texttt{score DESC}}(\delta_*(\sigma_*(\xi_P(A)))))$$

where:

- A is any algebraic expression;
- σ_* is a sequence zero or more selection (or full-text selections) operations;
- δ_* is a sequence of zero or more deletion operations.

First of all, the selection (or full-text selection) operations can be pushed down with respect to score assignment; deletion operations, instead, can be pushed up, because they do not affect the outer ordering and selection operations. The expression can thus be transformed into the following one:

$$\delta_*(\sigma_{[.\texttt{pos}\leq\texttt{k}]}(o_{/*.\texttt{score DESC}}(\xi_P(\sigma_*(A))))) \ .$$

It is now evident the pattern of a top-k operation: a full-text score assignment, followed by an ordering by score value, followed by a selection by position. We can therefore introduce the ad-hoc operator and obtain the following final expression:

$$\delta_*(\top_{P,\texttt{k}}(\sigma_*(A))) \ .$$

A similar transformation can also be done in order to introduce the threshold operator. Formally, we can state that the following equivalence rules hold:

$$\sigma_{[.\texttt{pos}\leq\texttt{k}]}(o_{/*.\texttt{score DESC}}(\delta^*(S^*(\xi_P(A))))) \equiv \delta^*(\top_{P,\texttt{k}}(S^*(A))) \ , \tag{5.30}$$

$$o_{/*.\texttt{score DESC}}(\sigma_{/*[.\texttt{score}\geq\tau]}(\delta^*(S^*(\xi_P(A))))) \equiv \delta^*(\omega_{P,\tau}(S^*(A))) \ . \tag{5.31}$$

Example 5.14 Consider the following XQuery Full-Text expression, which is a slightly modified example taken from [Con06e]:

```
for $book in doc("full-text.xml")/books/book
  let score $s := $book ftcontains "usability"
  where $s >= 0.1
  order by $s descending
  return <focusedBook relevance="{$s}">
           {$book/metadata/title/text()}
         </focusedBook>
```

Using the translation algorithms presented in Chapter 4, it is translated into the following AFTX expression:

$$\iota_{\text{“focusedBook”}(/\texttt{book/metadata/title.v},((\text{“relevance”},/\texttt{book.score})),\texttt{null})}($$
$$o_{/\texttt{book.score DESC}}($$
$$\sigma_{/\texttt{book}[.\texttt{score}\geq 0.1]}($$
$$\xi_{/\texttt{book}[\text{“usability”}]}($$
$$\pi_{/\texttt{books/book}}(\text{“}\texttt{full-text}\text{”}))))) \ .$$

In this algebraic expression we can find the algebraic pattern depicted by the left hand side part of Expression 5.31:

- A is $\pi_{/\texttt{books/book}}(\text{“}\texttt{full-text}\text{”})$;
- P is $/\texttt{book}[\text{“}\texttt{usability}\text{”}]$;
- S^* and δ^* are the empty string;
- τ is 0.1.

By using the threshold operator the previous expression can therefore be rewritten into the following one:

$$\iota_{\text{“focusedBook”}(/\texttt{book/metadata/title.v},((\text{“relevance”},/\texttt{book.score})),\texttt{null})}($$
$$\omega_{/\texttt{book}[\text{“usability”}],\tau}($$
$$\pi_{/\texttt{books/book}}(\text{“}\texttt{full-text}\text{”}))) \ .$$

5.3 Nested Queries Rules

XQuery permits the nesting of a FLWOR expressions; in Chapter 4 we have seen how such nested queries can be translated into AFTX expressions. There are however cases in which interesting optimizations can be performed over the expression built using the standard translation rules.

5.3.1 Product Elimination

A first case to consider is the presence in the expression of a product operation whose right input forest contains trees which are subtrees of those contained in the left input forest. Consider the following XQuery expression:

```
for $i in doc("books.xml")/bib/book
  return <book title={$i/title/text()}>
    {
      for $j in $i/author
      where $j/first="John"
      return <author>{$j/last/text()}</author>
    }
    </book>
```

This query returns, for each book, the title and the last name of each author whose first name is *John*. Following what already seen in Chapter 4, the `for` clause in the inner FLWOR expression, which refers to a variable defined in the outer FLWOR expressions, must be translated using product and deletion in order to perform a left outer join. The XQuery expression is then translated into the following algebraic expression:

$$\iota_{\text{“book”}(\texttt{null},((\text{“title”},/\texttt{group\_root/book/title.v})),(\text{“author”}(/\texttt{group\_root/author/last.v},\texttt{null},\texttt{null})))}($$
$$\delta_{/\texttt{group\_root}/*[\texttt{.k}=/\texttt{group\_root.A}[\text{“treeIdentity”}]\texttt{.v AND .pos}>1]}($$
$$\Sigma_{((/\texttt{prod\_root/1.k},\text{“treeIdentity”})),(/\texttt{prod\_root/1},/\texttt{prod\_root/2})}($$
$$\delta_{/\texttt{prod\_root/author}[\texttt{NOT} \equiv /\texttt{prod\_root/book/author}]}($$

$$\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”}) \times$$
$$\sigma_{/\texttt{author/first}[.\texttt{v}=\text{“}\texttt{John}\text{”}]}(\pi_{/\texttt{bib/book/author}}(\text{“}\texttt{books.xml}\text{”}))))))\ .$$

It is easy to notice that the right hand side expression of the product operation

$$\sigma_{/\texttt{author/first}[.\texttt{v}=\text{“}\texttt{John}\text{”}]}(\pi_{/\texttt{bib/book/author}}(\text{“}\texttt{books.xml}\text{”}))$$

results in a forest whose trees are subtrees of those contained in the forest resulting from the left hand side expression

$$\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”})\ .$$

In fact:

- the left projection predicate (`/bib/book`) is a subexpression of the right one (`/bib/book/author`);
- no further operators are applied to the left expression, while only a selection operator is applied to the right one.

Such conditions let us apply a very impacting optimization: the complete elimination of the product operation, along with its right side expression. In more details, the optimization is done as follows:

- the selection on authors is substituted with a deletion of *author* subtrees of the trees resulting from the projection on books; the deletion predicate is the opposite of the original selection predicate;
- the inner deletion is no more necessary, because there is no author coupled with books not written by him;
- the grouping was done in order to group each book with its authors; it is therefore no more necessary, because authors are already grouped (they are sub-elements of the respective *book* element);
- the outer deletion was done in order to eliminate duplicate book subtrees in the trees resulting from the previous grouping; also this operation is clearly no more necessary;

- the tree construction predicate is rewritten in order to be consistent with the new structure of the input trees.

The previous algebraic expression is then rewritten as follows:

$$\iota_{\text{“book”}(\texttt{null},((\text{“title”},/\texttt{book/title.v})),(\text{“author”}(/\texttt{book/author/last.v},\texttt{null},\texttt{null})))}\Big($$
$$\delta_{/\texttt{book/author}[\texttt{NOT }/\texttt{first.v}=\text{“John”}]}\Big($$
$$\pi_{/\texttt{bib/book}}(\text{“books.xml”}))\ .$$

5.3.2 Inner Join vs Outer Join

Another interesting optimization involves expressions whose purpose is to invert hierarchy of elements. Consider the following XQuery expression:

```
for $i in distinct-values(doc("books.xml")/bib/book
    /author/last)
return <author name={$i}>
  {
    for $j in doc("books.xml")/bib/book
    where $i=$j/author/last
    return <book>{$j/title/text()}</book>
  }
  </author>
```

Its purpose is to return, for each distinct author, the last name and the list of books written by him. Using the presented translation algorithms, the following AFTX expression is built:

$$\iota_{\text{“author”}(\texttt{null},((\text{“name”},\texttt{group\_root/group\_root.A}[\text{“last”}].\texttt{v})),(\text{“book”}(/\texttt{group\_root/book/title.v},\texttt{null},\texttt{null})))}\Big($$
$$\delta_{/\texttt{group\_root}/*[.\texttt{k}=/\texttt{group\_root.A}[\text{“treeIdentity”}].\texttt{v AND .pos}>1]}\Big($$
$$\Sigma_{((/\texttt{prod\_root/1.k},\text{“treeIdentity”})),(/\texttt{prod\_root/1},/\texttt{prod\_root/2})}\Big($$
$$\delta_{/\texttt{prod\_root/book}[\texttt{NOT }/\texttt{author/last.v}=/\texttt{prod\_root/group\_root.A}[\text{“last”}].\texttt{v}]}\Big($$
$$\nu_{(/\texttt{last.v},\text{“last”}}(\pi_{/\texttt{bib/book/author/last}}(\text{“books.xml”}))\ \times$$
$$\pi_{/\texttt{bib/book}}(\text{“books.xml”}))))))\ .$$

In this case we note that the left hand side expression of the product

$$\nu_{(/\texttt{last.v},\text{“}\texttt{last}\text{”})}(\pi_{/\texttt{bib/book/author/last}}(\text{“}\texttt{books.xml}\text{”}))$$

results in a forest whose trees are built using data (authors' last name) contained in the forest resulting from the right hand side expression

$$\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”}) \ .$$

It is not possible to completely eliminate the product operation, because it is needed in order to revert the hierarchy of the document, but it is possible to substitute it and the subsequent deletion with a join, in order to reduce the size of partial results. The expression can thus be rewritten as follows:

$\iota_{\text{“}\texttt{author}\text{”}(\texttt{null},((\text{“}\texttt{name}\text{”},/\texttt{prod\_root/group\_root.A}[\text{“}\texttt{last}\text{”}].\texttt{v})),(\text{“}\texttt{book}\text{”}(/\texttt{prod\_root/book/title.v},\texttt{null},\texttt{null})))}($
$\nu_{/\texttt{last.v}}(\pi_{/\texttt{bib/book/author/last}}(\text{“}\texttt{books.xml}\text{”})) \bowtie_{P'}$
$\pi_{/\texttt{bib/book}}(\text{“}\texttt{books.xml}\text{”}))))$,

where $P' = /\texttt{group\_root}[.\texttt{A}[\text{“}\texttt{last}\text{”}].\texttt{v} = \texttt{book}/\texttt{author}/\texttt{last.v}]$. In practice, the outer join has been substituted by an inner join. This optimization has been made possible by the consideration that the algebraic expression to the left of the join returns a forest that cannot contain a tree not having a corresponding tree in the forest resulting from the algebraic expression to the right of the join. Suppose now that the outer `for` clause is changed into the following:

```
for $i in distinct-values(doc("books.xml")//author/last)
```

Can we still optimize the algebraic expression as before? Unfortunately the answer is not, because now it is not guaranteed that the left hand side path expression (`//author/last`) is a subexpression of the right one (`/bib/book`). However, if we have an XML Schema [Con01] specification stating that (1) an *author* element can appear only as child of a *book* element and (2) a *book* element can appear only as child of a *bib* element, we can safely rewrite `//author/last` as `/bib/book/author/last`; consequently the rewriting rule is still valid and can be executed.

Part III

Conclusions

Chapter 6

Final Remarks

The integration of semi-structured data management and Information Retrieval techniques poses serious challenges to database system developers. Nevertheless, such an integration is recognized as a need from the scientific community, as testified by the definition of a full-text extension of XQuery, the W3C candidate standard query language for XML documents.

The definition of such a query language would be useless, if it is not coupled with a formal algebraic framework underlying it. In fact, the implementation of the algorithms needed for executing a query should be based on the definition of algebraic operators which, carefully combined, are able to represent each query expressible in the query language. Moreover, the availability of such an algebra facilitates the task of finding an optimized query execution plan, by exploiting equivalence and containment properties of algebraic expressions in order to formalize a set of query rewriting rules.

These considerations, along with the convincement that existing proposals on this subject cannot be considered totally adequate, led us to the definition of a formal model for representing XML documents and an algebra for querying instances of that model.

The data model represents XML databases through ordered trees contained into forests. Special care is dedicated to the textual content: it is tokenized, and an ordinal value is assigned to each token. This permits to precisely represent either data-centric or document-centric repositories. The data model we have presented has to be intended as a *formal* model; an implementation of our framework could obviously choose a different internal

representation of trees and forests, provided that all the necessary properties of elements and attributes are equally available.

AFTX, the proposed algebra, performs either standard queries and full-text queries; it is able to represent many XQuery Full-Text expression. To our knowledge, AFTX is one of the very few proposed algebras for XML covering the issue of full-text retrieval. Moreover, in our opinion, it does not suffer of some limitations found in other proposals; in fact:

- its data model is based on trees, the natural way to represent XML documents; no transformation towards classical relational model is needed;
- its query capabilities are not limited to simple XPath-like constructs; it can represent complex nested expressions and it can freely restructure the content of input trees;
- its full-text capabilities are comparable to that of XQuery Full-Text, rather, AFTX provides a fine grained control over score evaluation through the availability of a parameter that defines the function to use when calculating the full-text score of a tree;
- its operators, being for the most part similar to classical relational operators, have a precise and easy to understand semantics.

Another important contribution of our work is the definition of a series of equivalence, containment and similarity rules. Some of them are adaptation of rules used in different contexts, namely in relational algebra. Other rules deal with special characteristics of queries over XML, like the possibility to nest expressions inside other expressions. Full-text operators are also analyzed and a set of rules are targeted to those operators. In the complex, the set of presented rewriting rules makes our framework a valuable starting point for studying query optimization strategies.

Finally, a formal algorithm for the automatic translation of XQuery Full-Text expression into AFTX expressions has been developed. There are some limitations on the kind of expressions that are recognized by the translation algorithm, but the accepted XQuery fragment should be considered, in our opinion, quite expressive.

Chapter 7

What is next

This thesis poses the formal basis for the implementation of an efficient database system for XML documents with Information Retrieval capabilities. The first future activity is therefore the development of such a working system. To this purpose, what should be realized is:

- an implementative model for storing XML documents; it must not be necessarily equal to the presented formal model, but it should in any case expose all the properties of elements, trees, and forests defined in the formal model;
- efficient algorithms implementing the defined algebraic operators.

The rewriting rules presented in this thesis should then be used for finding efficient query execution plans. This process presupposes the availability of:

- access support structures, like path indexes and value indexes, which should be developed;
- statistics on the content of the XML repository.

Besides this evident future activity, we envisage two possible interesting extensions to the algebra. The first deals with relaxed queries, a valid way to manage the structural heterogeneity typical of XML repositories. The second is an attempt to consider data mining tasks over XML documents just like one of the various manipulation tasks that can be represented by an algebra.

One of the main differences between structured and semi-structured paradigm is the *vagueness* of the schema. When dealing with structured data, the exact schema of relations involved in a query is known in advance; this is not always the case for semi-structured data. In fact the schema specifications for XML documents can leave an high level of flexibility to document producers, for example defining some elements as *optional*; moreover, it is perfectly legal for an XML document not to have an associated schema at all. From a point of view, this flexibility is a great advantage; for example, it facilitates the integration of heterogeneous data sources. On the other side, it poses some problems for what concerns answering to a query that imposes constraints on the structure of XML fragments to retrieve; it could be the case that such constraints are satisfied only by a small part of source documents, thus leading to almost empty answers. Nevertheless, there could be documents that are relevant to users, even if they do not closely respect structure constraints expressed using XPath constructs.

A possible future research area is therefore aimed at incorporating into AFTX the notion of *query relaxation*, i.e. the transformation of an algebraic expression into a less restrictive one, following the ideas presented in [AYLP04, AYKM+05, MAYKS05]. A relaxed version of some algebraic operators (selection, full-text selection and projection are the main candidates) should be defined; such relaxed operators should be based on the concept of *score*, i.e. a relaxed answer is assigned a score which reflects how exact is the query that returns such an answer. In a certain way exact and approximate queries should play the same role of boolean and ranked retrieval in classical Information Retrieval (and of course in XQuery Full-Text): while exact queries classify each document fragment as either relevant (i.e. fulfilling constraints imposed on the structure of a document and the value of elements or attributes) or not relevant, relaxed queries should establish how relevant such a fragment is.

For what concerns data mining, many works have been presented in the last few years [BCC+02, WL00, AAK+02, Zak02, TRS02, ZA03, TSW03] trying to adapt concepts from data mining over structured data to semi-structured repositories. However they typically consider data mining as a stand-alone subject, with poor connections with standard manipulation operations. We believe instead that XML data mining tasks could be seen

as the composition of basic manipulation tasks, which can be expressed by algebraic operators working on forest of trees, like the ones present in AFTX. Clearly standard and data mining operators would work of documents containing data at a different level of abstraction, but they would share the same formal model. This extensions, along with the support for approximate queries, would transform AFTX into a complete framework for the management of semi-structured data.

References

[AAK+02] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficient Substructure Discovery from Large Semi-Structured Data. In *Proceedings of the 2nd SIAM International Conference on Data Mining*, pages 158–174, 2002.

[Abi97] Serge Abiteboul. Querying Semi-Structured Data. In *Proceedings of the International Conference of Data Base Theory (ICDT)*, pages 1–18, Delphi, Greece, 1997.

[AKYJ03] Shurug Al-Khalifa, Cong Yu, and H. V. Jagadish. Querying Structured Text in an XML Database. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 4–15, San Diego, California, 2003. ACM Press.

[AQM+97] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[ASB99] Serge Abiteboul, Dan Suciu, and Peter Buneman. *Data on the Web : From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[AYBS04] Sihem Amer-Yahia, Chavdar Botev, and Jayavel Shanmugasundaram. TeXQuery: A Full-Text Search Extension to XQuery. In *Proceedings of the 13th Conference on World Wide Web*, pages 583–594, New York, NY, USA, May 2004.

[AYCD06] Sihem Amer-Yahia, Emiran Curtmola, and Alin Deutsch. Flexible and Efficient XML Search with Complex Full-Text Predicates. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 575–586, Chicago, IL, USA, 2006.

[AYKM+05] Sihem Amer-Yahia, Nick Koudas, Amélie Marian, Divesh Srivastava, and David Toman. Structure and Content Scoring for XML. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB 2005)*, pages 361–372, Trondheim, Norway, August 2005.

[AYLP04] Sihem Amer-Yahia, Laks V. S. Lakshmanan, and Shashank Pandit. FleXPath: Flexible Structure and Full-Text Querying for XML. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 83–94, Paris, France, June 2004.

[BAYS06] Chavdar Botev, Sihem Amer-Yahia, and Jayavel Shanmugasundaram. Expressiveness and Performance of Full-Text Search Languages. In *Proceedings of the 10th International Conference on Extending Database Technology (EDBT 2006)*, pages 349–367, Munich, Germay, March 2006.

[BCC+02] D. Braga, A. Campi, S. Ceri, M. Klemettinen, and P. L. Lanzi. A Tool for Extracting XML Association Rules. In *Proceedings of IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 57–65, 2002.

[BDHS96] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 505–516, 1996.

[BG02] Jan-Marco Bremer and Michael Gertz. XQuery/IR: Integrating XML Document and Data Retrieval. In *Proceedings of the 5th International Workshop on the Web and Databases (WebDB)*, pages 1–6, June 2002.

[BHN+02] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. In *Proceedings of the 18th International Conference on Data Engineering (ICDE 2002)*, pages 431–440, San Jose, California, USA, February 2002.

[BM06a] Giacomo Buratti and Danilo Montesi. A Data Model and an Algebra for Querying XML Documents. In *Proceedings of the 17th International Workshop on Database and Expert Systems Applications (DEXA 2006)*, pages 482–286, Krakow, Poland, September 2006. IEEE Computer Society.

[BM06b] Giacomo Buratti and Danilo Montesi. Equivalence and Containment of XQuery Full-Text Expressions. *WSEAS Transactions on Information Science Applications*, 3(10):1818–1825, October 2006.

[BM06c] Giacomo Buratti and Danilo Montesi. Full-Text Capabilities for Querying XML Repositories: a Formal Model. In *Proceedings of the 10th WSEAS International Conference on Computers*, pages 738–743, Athens, Greece, July 2006.

[BM06d] Giacomo Buratti and Danilo Montesi. XQuery Full-Text Optimization through a Formal Algebra. In *Proceedings of the 2nd International Advanced Database Conference (IADC-2006)*, San Diego, California, June 2006.

[BMBdlI05] G. Buratti, D. Montesi, A. Bangham, and B. de la Iglesia. Data Model and Query Languages for Biological Databases. *IEEE Multimedia Communications Technical Committee e-Newsletters*, 2(2):7–10, August 2005.

[BT99] Catriel Beeri and Yariv Tzaban. SAL: An Algebra for Semistructured Data and XML. In *Proceedings of the International Workshop on The Web and Databases (WebDB)*, pages 37–42, Philadelphia, Pennsylvania, June 1999.

[Bun97] Peter Buneman. Semistructured Data. In *Proceedings of the 16th ACM Symposium on Principles of Database Systems (PODS)*, pages 117–121, Tucson, Arizona, 1997.

[BYRN99] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[CCD+99] Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. XML-GL: A Graphical Language for Querying and Restructuring XML Documents. In *Sistemi Evoluti per Basi di Dati*, pages 151–165, 1999.

[CCS00] Vassilis Christophides, Sophie Cluet, and Jérŏme Simèon. On wrapping Query Languages and Efficient XML Integration. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 141–152, 2000.

[CM98] Mariano P. Consens and Tova Milo. Algebras for Querying Text Regions: Expressive Power and Optimization. *Journal of Computer and System Sciences (JCSS)*, 57(3):272–288, 1998.

[CMKS03] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A Semantic Search Engine for XML. In *Proceedings of the International Conference on Very Large Databases*, pages 45–56, 2003.

[Cod70] Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM (CACM)*, 13(6):377–387, 1970.

[Con01] World Wide Web Consortium. XML Schema, W3C Recommendation. `http://www.w3.org/XML/Schema/`, May 2001.

[Con04] World Wide Web Consortium. Extensible Markup Language (XML) 1.1, W3C Recommendation. `http://www.w3.org/TR/xml11/`, February 2004.

[Con06a] World Wide Web Consortium. XML Path Language (XPath) 2.0, W3C Candidate Recommendation. http://www.w3.org/TR/xpath20/, June 2006.

[Con06b] World Wide Web Consortium. XML Query Use Cases, W3C Working Draft. http://www.w3.org/TR/xquery-use-cases/, June 2006.

[Con06c] World Wide Web Consortium. XQuery 1.0: An XML Query Language, W3C Candidate Recommendation. http://www.w3.org/TR/xquery/, June 2006.

[Con06d] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Data Model (XDM), W3C Candidate Recommendation. http://www.w3.org/TR/xpath-datamodel/, July 2006.

[Con06e] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Full-Text Use Cases, W3C Working Draft. http://www.w3.org/TR/xmlquery-full-text-use-cases/, May 2006.

[Con06f] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Full-Text, W3C Working Draft. http://www.w3.org/TR/xquery-full-text/, May 2006.

[Con06g] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Functions and Operators, W3C Candidate Recommendation. http://www.w3.org/TR/xpath-functions/, November 2006.

[Con06h] World Wide Web Consortium. XQuery Update Facility, W3C Working Draft. http://www.w3.org/TR/xqupdate/, July 2006.

[CRF00] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: an XML Query Language for Heterogeneous Data Sources. In *Proceedings of the Third International Workshop on the Web and Databases (WebDB 2000)*, pages 53–62, Dallas, Texas, USA, 2000.

[DFF+99] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A Query Language for XML. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(11–16):1155–1169, 1999.

[DSR] DSRG. Rainbow: XQuery Processing System Using Relational Technology. `http://davis.wpi.edu/~dsrg/rainbow/`.

[FG01] Norbert Fuhr and Kai Grosjohann. XIRQL: A Query Language for Information Retrieval in XML Documents. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 172–180, New Orleans, USA, September 2001.

[FHP02] Flavius Frasincar, Geert-Jan Houben, and Cristian Pau. XAL: an Algebra for XML Query Optimization. In *Proceedings of the thirteenth Australasian Conference on Database Technologies*, pages 49–56, 2002.

[gal] Galax: An Implementation of XQuery. `http://www.galaxquery.org/`.

[GSBS03] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *Proceedings of the 2003 ACM SIGMOD Conference*, pages 16–27, 2003.

[HP02] Vagelis Hristidis and Yannis Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB 2002)*, pages 670–681, Hong Kong, China, August 2002.

[INE] INEX. INitiative for the Evaluation of XML Retrieval. `http://inex.is.informatik.uni-duisburg.de/2006/`.

[JLST01] H. Jagadish, L. Lakshmanan, D. Srivastava, and K. Thompson. TAX: a Tree Algebra for XML. In *Proceedings of the International Workshop on Data Bases and Programming Languages (DBPL'01)*, pages 149–164, Frascati, Rome, Italy, September 2001.

[MAYKS05] Amélie Marian, Sihem Amer-Yahia, Nick Koudas, and Divesh Srivastava. Adaptive Processing of Top-K Queries in XML. In *Proceedings of the 21st International Conference on Data Engineering (ICDE 2005)*, pages 162–173, Tokyo, Japan, April 2005.

[MHBA04] Vojkan Mihajlović, Djoerd Hiemstra, Henk Ernst Blok, and Peter M. G. Apers. An XML-IR-DB Sandwich: Is it Better With an Algebra in Between? In *The First Workshop on the Integration of Information Retrieval and Databases (WIRD04)*, 2004.

[MM06] Matteo Magnani and Danilo Montesi. A Unified Approach to Structured and XML Data Modeling and Manipulation. *Data & Knowledge Engineering*, 59(1):25–62, October 2006.

[NDM+01] Jeffrey F. Naughton, David J. DeWitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayavel Shanmugasundaram, Feng Tian, Kristin Tufte, Stratis Viglas, Yuan Wang, Chun Zhang, Bruce Jackson, Anurag Gupta, and Rushan Chen. The Niagara Internet Query System. *IEEE Data Engineering Bulletin*, 24(2):27–33, 2001.

[PG04] Benjamin Piwowarski and Patrick Gallinari. An Algebra for Probabilistic XML Retrieval. In *The First Twente Data Management Workshop*, pages 59–66, Enschede, The Netherlands, June 2004. SIKS.

[RSF06] Christopher Ré, Jérôme Siméon, and Mary Fernández. A Complete and Efficient Algebraic Compiler for XQuery. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE 2006)*, page 14, Atlanta, Georgia, USA, April 2006.

[SA02] Carlo Sartiani and Antonio Albano. Yet Another Query Algebra For XML Data. In *Proceedings of the 2002 International Database Engineering and Applications Symposium (IDEAS'02)*, pages 106–115, 2002.

[Suc98] Dan Suciu. Semistructured Data and XML. In *Proceedings of the International Conference on Foundations of Data Organization (FODO)*, pages 1–12, Kobe, Japan, 1998.

[TG02] A. Theobald and G.Weikum. The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking. In *Proceedings of the 8th International Conference on Extending Database Technology*, pages 477–495, 2002.

[TRS02] A. Termier, M.C. Rousset, and M. Sebag. TreeFinder: a First Step towards XML Data Mining. In *Proceedings of International Conference on Data Mining (ICDM)*, pages 450–457, 2002.

[TS04] A. Trotman and B. Sigurbjrnsson. NEXI: Now and Next. In *INEX 2004 Proceedings*, 2004.

[TSW03] M. Theobald, R. Schenkel, and G. Weikum. Exploiting Structure, Annotation, and Ontological Knowledge for Automatic Classification of XML Data. In *Proceedings of the 6th International Workshop on the Web and Databases (WebDB)*, pages 1–6, 2003.

[WL00] K. Wang and H. Liu. Discovering Structural Association of Semistructured Data. *IEEE Transaction of Knowledge and Data Engineering*, 12(2):353–371, 2000.

[ZA03] M. J. Zaki and C. C. Aggarwal. XRules: An Effective Structural Classifier for XML Data. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 316–325, 2003.

[Zak02] M. J. Zaki. Efficiently Mining Frequent Trees in a Forest. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 71–80, 2002.

[ZPR02] Xin Zhang, Bradford Pielech, and Elke A. Rundesnteiner. Honey, I Shrunk the XQuery! — An XML Algebra Optimization Approach. In *Proceedings*

of the 4th international workshop on Web information and data management (WIDM'02), pages 15–22, 2002.