University of Milano–Bicocca

Department of Informatics, Systems and Communication (DISCo)

Enhanced XML Retrieval with Flexible Constraints Evaluation

Ph.D dissertation of

Emanuele Panzeri

Supervisor: Prof. Gabriella Pasi

Tutor: Prof. Carla Simone

AA 2012/2013

Tell me and I forget,
Teach me and I may remember,
Involve me and I learn.

Benjamin Franklin

Abstract

Since its standardization by the World Wide Web Consortium (W3C) in 1998, the XML (acronym for *eXtensible Markup Language*) has been acknowledged as the *de-facto* standard format for data, besides being a data format employed by a wide and increasing number of application domains. XML allows data and textual contents to be structured; the structural elements are specified in plain text using strings of characters that can be easily read by computer programs, while maintaining human-readability.

XPath and XQuery represent the two main standard languages that have been defined to inquire XML data; the two languages allow to select a subset of elements from an XML document, and to further manipulate its contents and to restructure the document tree form. Both XPath and XQuery are based on a Database perspective of XML documents, where the evaluation of the query clauses is performed like in the database query language SQL, from which both the XML languages took inspiration. The data-centric perspective adopted by the XQuery and XPath languages has been recently extended by an Information Retrieval oriented approach, where a new set of content-based constraints have been defined that allow a full-text search in an IR-style, with an element relevance scoring computation. This extension is called XQuery/XPath Full-Text and has been standardized by the W3C.

In the Information Retrieval community other approaches have appeared that take into account the document structure and propose a set of approximate structural matching techniques, where the standard XQuery and XPath structural constraints are evaluated by path relaxation algorithms. Such approaches, however, do not offer the user the possibility to express vague structural constraints the approximate evaluation of which produces a set of weighted fragments, where the weight express the relevance of the fragment with respect to the structural constraints.

This thesis describes the definition and the implementation of a formal XQuery Full-Text extension named FleXy, aimed at taking into account the user perspective in the formulation of structure-based constraints, where vagueness can be associated to the specification of such constraints. FleXy has been designed as an extension of the XQuery Full-Text language to inherit both the full-text search features from the Full-Text extension, and the standard element selection provided by XQuery.

The evaluation of two new vague structural constraints defined in the FleXy language, named `below` and `near`, produces a set of weighted elements, where a structural-score is computed by taking into account the *distance* from the user required target element and the actually retrieved one. Thresholds variants of the `below` and `near` constraints have also been defined which allow to specify the extent of the application of the vague structural constraints.

The formal definition of the FleXy language is here provided through its syntax, its semantics, and the algorithms that define the `below` and the `near` axes.

The language implementation has been performed on top of an Open Source XQuery engine named BaseX, a fully featured XQuery and XPath engine with a complete adherence to the Full-Text language specification. Performance evaluations have been subsequently provided to compare the FleXy constraints with the standard XQuery counterparts, when available.

Finally, a patent search application has been developed by leveraging the FleXy implementation provided on top of the BaseX engine: the XML structure of the US Patent Collection (USPTO) has been exploited in conjunction with the textual contents of the patents to help non-expert users to effectively retrieve relevant patents by also offering a result categorization strategy.

Contents

1 Introduction

1.1 Motivations

Since its first introduction, and its subsequent definition as a W3C (World Wide Web Consortium) standard, the *eXtensible Markup Language* (XML) [1] has been acknowledged as the *de-facto* standard format for industrial and scientific data exchange. The semi-structured data representation defined by the XML format has been adopted from a wide number of domains, ranging from business to digital libraries, including word processing applications and communication protocols.

Standard XML query languages have been defined by the W3C for searching and manipulating the hierarchical structure of XML documents; the XPath [2], and the XQuery [3] languages have been explicitly defined to this aim. Both the XQuery and XPath languages, mainly based on a data-centric perspective of XML documents, provide a Database oriented and SQL-like specification of constraints: the evaluation of structure-based and content-based constraints is performed by a strict matching.

The availability of large XML document-centric collections such as the MedLine[1] medical data collection or the Wikipedia corpus [4] raised the Information Retrieval (IR) community interest for XML documents, where full-text search and approximate constraints evaluation approaches were introduced.

An important result obtained from a joint work between the Information Retrieval and the Database communities is the definition of the XQuery/XPath Full-Text extension; it has been recently defined and standardized by the W3C. The XQuery/XPath extension provides an extensive set of features for defining and evaluating full-text search over the textual contents of XML documents, such as the computation of relevance scores and the definition of IR-style text handling and matching functions.

Besides the successful introduction of Information Retrieval search techniques as the XQuery Full-Text extension, an important issue still remains open regarding the possibilities offered by standard XML query languages in querying XML documents. The content based query formulation provided by the Full-Text language extension is not coupled to the equally important possibility for the user to express vagueness in formulating structural-based constraints. Some approximate evaluations of standard structural constraints have been proposed in the literature, such as tree pattern relaxation [5, 6],

[1] The MedLine database, available at `http://www.ncbi.nlm.nih.gov/pubmed/` is constituted of more then 22 millions of publication records regarding biomedicine and health.

and approximate XML query tools like Juru [7] and FleXPath [8] have been defined By leveraging the standard W3C languages, most of these approaches provide an approximate evaluation of standard structure-based constraints by means of a set of constraint rewriting or other query modifications. The user query is thus taken into account as a mere template during the query evaluation process. None of the approaches in the literature, however, takes into account the approximate structural evaluation from a user perspective: any of the proposals provides the user the ability to explicitly state which structural constraint should be evaluated by means of a strict matching and which ones with an approximate technique providing a structural relevance estimation. Vague constraint specifications and their related approximate evaluation would allow the users to formulate queries where the structure-based constraints evaluation can produce a score expressing the extent to which a retrieved fragment satisfies the structural constraint. The research work undertaken during the Phd and reported in this thesis addressed this specific issue: providing users the ability to express vagueness while formulating structural constraints.

1.2 Contributions

In this thesis, a new proposal for querying XML documents by the formulation of vague structural constraints is presented. To this aim the formal definition of the FleXy language is here provided; this language, as outlined in the previous section, is aimed at addressing the aspect of querying XML documents by allowing the user to express vagueness while formulating structural constraints. The novelty of the FleXy language, defined as an extension of the standard XQuery/XPath Full-Text, is the definition of two new structural constraints, named `below` and `near`, that allow the users to express vague structural requirements in a query. Differently from all the approximate techniques proposed in the literature, the majority of which leverages tree pattern relaxation algorithms, the approach defined in this work presents a user-centered structural constraint formulation, with an associated approximate evaluation.

A relevance score for each retrieved element is also computed during the evaluation of the `below` and the `near` constraints: such score represents the structural relevance of the matched element given the vague constraints specified in the query. Furthermore, the user may explicitly express a tolerance to the approximate evaluation of such new constraints: parametric variants of the new FleXy axes have been defined to extend the possibilities offered by the `below` and `near` constraints.

Finally, by extending the XQuery Full-Text language, the set of FleXy constraints, and in particular the structural relevance scores, can be used to provide customized filtering and ranking functions. Furthermore FleXy provides a clean and complete integration with the set of FLWOR clauses of XQuery, thus allowing users to combine the full-text relevance score and the structural score and to obtain query results ranked by an aggregation of the two scores.

The main contributions of this thesis can be summarized as follows:

- The FleXy language has been formally defined as an extension of the XQuery Full-Text language: the language syntax and the axes semantics have been defined following the XQuery [3] standard, and the Core XPath [9] semantics. The scoring algorithms and the integration of the structural relevance score with the textual relevance score are also provided.
- A first implementation of the FleXy language and the scoring algorithms has been performed on top of a state of the art XQuery engine named `minPID`. The work has been aimed at evaluating the feasibility of the FleXy language by extending the indexing data structure defined in the engine to verify its support to the FleXy language requirements.
- The full implementation of the FleXy language on top of a fully featured XQuery Full-Text engine named BaseX has been subsequently performed. After undertaking an analysis of the state-of-the-art XQuery Full-Text engines and their characteristics, the BaseX engine has been selected to guarantee an efficient evaluation of the FleXy constraints. The FleXy language has been implemented on top of this XQuery engine, by leveraging its efficient data structure and by integrating the FleXy constraints in its XQuery parser.
- The efficiency of the FleXy language has been evaluated by its implementation on top of the BaseX engine: time comparison tests have been performed to measure the overhead introduced by the flexible axes, and by the evaluation of the structural relevance score computation.
- Finally, the Flex-BaseX engine has been used to build a Patent Search application: the structure-based evaluation of the FleXy language, combined with the standard XQuery Full-Text search capabilities are leveraged to help users to query XML patent documents, where result classification is also provided.

1.3 Structure of the Thesis

Besides the Introduction provided in this Chapter, the rest of the thesis is organized as follows:

- **Chapter 2** introduces the eXtensible Markup Language (XML), the standard structured document format representation that has been the basic object of the research reported in this thesis. It briefly presents the main characteristics of this language and describes the most important, database-oriented, languages defined to process documents written in XML format: XPath and XQuery. The NEXI query language is also described as it proposes, from an Information Retrieval perspective, a full-text search functionality on the textual contents of an XML document. Finally, given the high relevance to the work undertaken in this thesis, the XQuery/XPath Full-Text extensions, as recently defined by the W3C, are discussed: in particular their features and relevance estimate computation are detailed.

- **Chapter 3** deals with the different perspectives of querying XML documents as addressed by the Database community and the Information Retrieval community: since they have different views on how an XML document is conceived, they accordingly proposed different query paradigms. The chapter presents both the strict querying paradigm proposed by the Database community, and the approximate textual content matching as defined by the Information Retrieval one. More attention, given the subject of this thesis, is paid to the IR querying perspective regarding the structural constraints evaluation, including the evaluation proposed by the INEX[10] initiative.

- **Chapter 4** presents the main contribution of the work undertaken in this thesis: the FleXy language is presented and described by outlining its motivations, syntax, semantics and structure-based constraints evaluation. The new constraints `below` and `near` are here formally presented alongside their syntax and their integration within the XQuery FLWOR clauses; the structural relevance score integration and its coupling with the relevance score computed by the Full-Text extension are also addressed. Finally, besides the examples of the FleXy usage, the semantics of the vague constraints are defined in Chapter 4 by leveraging the Core XPath semantics as introduced by Gottlob et al. in [9].

- In **Chapter 5** the main activities performed in relation to the implementation of the FleXy language are described: a first task aimed at identifying a data structure able to offering an efficient evaluation strategy on top of which to provide a first implementation and a feasibility evaluation of the FleXy language is here described. A second activity aimed at providing a full integration of the FleXy language on top of an existing, and fully compliant to the Full-Text extension, XQuery engine is described in the second part of Chapter 5. In particular, the analysis and the comparison of different XQuery engine alternatives are reported before the description of the final implementation of the FleXy language performed on top of the BaseX XQuery engine. The data structures, the algorithms defined to efficiently evaluate the FleXy constraints, and the integration performed with the BaseX XQuery engine are here detailed.

- **Chapter 6** presents the efficiency evaluations conducted to provide a first comparison of the FleXy constraints implemented on top of the BaseX engine with their standard XQuery counterparts, when available. The `below` axis has been compared with the related `descendant` axis, while for the `near` axis, for which there is no similar axis in the xtandard XQuery language, an alternative comparison solution is provided. Chapter 6 also presents a use case of the FleXy language usage: the implementation performed on top of the BaseX engine is leveraged to provide a flexible patent search application named PatentLigth.

- **Chapter 7** summarizes the main contributions of this thesis.

2 XML and XML Querying

In this chapter the basic concepts of XML and the query languages for inquiring XML document repositories are presented: in Section 2.1 the *eXtensible Markup Language* is introduced and its specification is detailed. In Section 2.2 the most important languages for querying and processing XML documents are described with particular attention to the XPath and XQuery languages. Finally Section 2.3 presents the Full-Text extension, defined for both the XQuery and the XPath languages, which allows querying of XML documents in an Information Retrieval style, with keyword based constraint evaluation (with scoring and ranking).

Particular details are provided for the XQuery Full-Text extension given its relevance to the research activity reported in this manuscript.

2.1 XML

XML [1] is the acronym of *eXtensible Markup Language*; it is, in fact, a markup language defined by the World Wide Web Consortium (W3C [14]) in 1998. The aim of W3C for the XML design was to "*meet the challenges of large-scale electronic publishing*" and to "*play an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere.*" [15]. XML is a text format derived from a previous markup language named SGML [16] (Standard Generalized Markup Language) defined in 1986. Both XML and SGML specify a set of rules for describing documents where their data content can be structured in a tree form; both the document structure and the contents are described in plain text using strings of characters that can be easily read by computer programs, while maintaining human-readability. These aspects, among others such as the highly adaptability of the language, motivated the wide adoption of the XML specification on an increasing number of domains. The XML data model has been used for representing general semi-structured data and employed also for data-exchange between applications. Web-Services, Content Managements Systems, smart-phones, ePublishing systems, biology and chemistry are a brief list of domains where XML came to play an important role for data representation and exchange.

2.1.1 XML Elements and Comments

As previously outlined, in an XML document the contents are represented in a tree form using a conventional markup notation. The basic markup unit is called *element*, the contents of which are enclosed in a pair of *tag*s named *start-tag* and *end-tag* respectively. Each element is identified by a *label*, that is generally related to the semantics of the element itself; both the *start-tag* and the *end-tag* of an element are built with the following syntax: '<' *label* '>' for the start-tag, and '</' *label* '>' for the end-tag.

As an example, the following code describes an element labeled *name* the content of which is the text *Andromeda*[1]:

```
<name>Andromeda</name>
```

In XML documents there is a special element, called *root-element*, that defines the root node of the document tree structure: it is usually the first node opening the document; the XML specification allows only one *root-element* per document.

The XML definition allows elements to have *attributes*: they can be sets of *attribute-value* pairs defined in the start-tag of an element in the form: `attribute="value"`. In the following example the element `object` includes two attributes: one named *catalog* with the value `"M31"` and one named *type* with a text value `"galaxy"` associated with it.

```
<object catalog="M31" type="galaxy">
```

In an XML document the set of elements can be nested to create a tree-like structure for the contained data; there is no limit to the nesting level that a document can have. In the following Listing 2.1 an example of an XML document is shown: elements indentation has been added to improve the readability.

```
<collection>
  <object catalog="M31" type="galaxy">
    <name>Andromeda</name>
    <position>
      <latitude unit="degrees">80.893</latitude>
      <longitude unit="degrees">69.756</longitude>
    </position>
    <distance unit="meters">7.92 E+23</distance>
  </object>
</collection>
```

Listing 2.1: Example of element nesting in XML documents.

[1]In the following sections, the XML document examples are taken from the NASA Extragalactic database http://ned.ipac.caltech.edu

In Fig. 2.1 a tree-representation of the previous XML document is provided: XML elements are represented as circles paired with the corresponding label. Rectangles represent the textual contents of elements, while attributes are represented as plain text connected to the corresponding XML element with a circle-ended line.

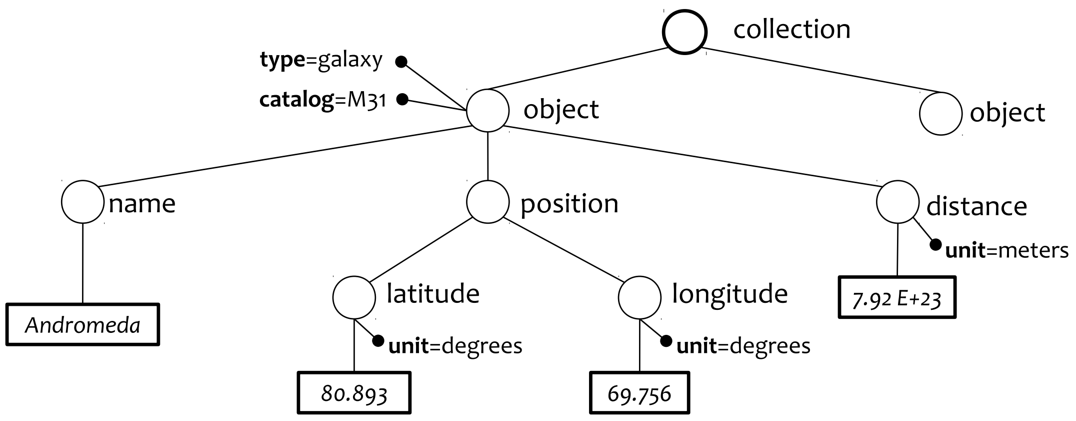


Figure 2.1: XML Tree example

In an XML document not only `tags` or attributes can be defined: another important markup is represented by the `comment-tags`. Comments can appear in any location of the XML document outside other markup; they are enclosed in the two tags: '`<!--`' and '`-->`'. Any textual content that is surrounded by the `comment-tags` is intended to be read by humans, and thus it should not be taken into account by any XML processor. An example of an XML comment is presented here below:

```
<!-- Data taken from NASA Extragalactic Database -->
```

When dealing with markup languages, as XML is, an important note should be made: markup languages use particular symbols to special purposes, like the previous mentioned '<' and '>' characters, aimed at representing starting and ending tags.

It is thus required, for the markup language, to provide an *escaping* technique to allow the specification of textual content that includes the special characters as simple characters that will not be interpreted during the XML processing (for example, to define textual contents to contain the '<' and '>' symbols). To this purpose XML provides a set of *entity references*:

- `&lt;` represents the '<' (lower-than) character;
- `&gt;` represents the '>' (greater-than) character;
- `&amp;` represents the '&' (ampersand) character;
- `&quot;` represents the '"' (double quote) character;

- `&apos;` represents the ''' (single quote) character;

During the document processing the XML parser will take care of automatically changing every instance of an *entity reference* back to its original representation. Taking as an example an element labeled "`text`" with a textual content of "*Is true that $5 < 6$?*", it has to be written, following the XML format, as:

```
<text>Is true that 5 &lt; 6?</text>
```

A second option provided by XML to avoid conflicts in special symbol usage are the so called *CDATA blocks*: such block of text, enclosed by the tags "`<![CDATA[`" and "`]]>`", provides the ability to write long chunks of text where no special symbols are identified as markup data. The same example provided before, but with the use of *CDATA* block is:

```
<text><![CDATA[Is true that 5 < 6?]]></text>
```

2.1.2 XML prolog, Declaration and Processing Instructions

XML documents are usually introduced by an *XML prolog*: it is the information written in an XML document the precedes the *root-element*; usually the XML prolog includes processing instructions (PI) or the *XML declaration.* Processing instructions are elements, enclosed by the special start-tag '`<?`' and the end-tag '`?>`', that are intended to provide information or commands to an application that is processing the XML document. For such PI the element label represents the target application, while the element's attributes are set of instructions or commands.

An example of Processing Instructions is provided in the following code, where a target application, called `xml-stylesheet`, is instructed on how to apply a particular stylesheet[2] to the current document.

```
<?xml-stylesheet type="text/css" href="docstyle.css"?>
```

Differently from a Processing Instruction element, that can be placed in any part of the XML document, an XML Declaration must be, if present, placed in the XML prolog; such declaration follows the same syntax of a PI element, but it does not target any application. An XML Declaration encodes information about the subsequent XML contents; its start-tag is `<?xml`, while the end-tag is `?>`. In the following code the XML Declaration defines the XML *version* (`1.0`) and the document charset *encoding* (`UTF-8`).

```
<?xml version="1.0" encoding="UTF-8"?>
```

[2]A stylesheet represents a set of instructions on how the XML document contents should be transformed or rendered; for further details refer to CSS Specification [17] and XSL Transformation [18].

The information present in an XML Declaration is used by XML applications to correctly handle the document format, structure and content.

2.1.3 Well-formed and Valid XML Documents

In addition to the formal syntax that has to be used for writing a correct XML document (as described in Section 2.1.1), the XML format also defines a formal validation of XML documents: documents that respect the allowed entities references names, the element attributes positioning, and other specific aspects, are called *well-written documents*. The XML language defines a document as *well-formed* when such document is both *well-written* and conform to the following rules:

- the document has a unique root element;
- every start-tag has its corresponding end-tag;
- every element's attribute value is quoted;
- element attributes can not be duplicated;
- the document does not contain overlapping elements: an element's closing tag must be placed after that all of its contained elements have been closed.

Furthermore a document can be labeled as *valid* if besides being *well-written* and *well-formed*, its contents and structure adhere to a specific *schema* that defines rules and guidelines regarding both the allowed markup (both for the elements and attributes labels), and their correct nesting. An XML document can then be considered *valid* with respect to a specific *schema* definition.

The *Document Type Definitions* (DTDs), as defined by the W3C, allow to formally define an XML document structure by specifying the elements and attributes labels and a basic content specification. The DTD allows to express, for each element and attribute, the required cardinality in the document markup: an element (or attribute) can be defined to be required or optional or to appear only in a certain position in the document markup. Furthermore, for XML element contents, a DTD can specify the contained data as composed of: a predefined subset of elements, plain text, a mixed content (text mixed with other markup elements) or force an element not to have any content at all. Documents that match against a given schema are said to be *valid*, or *invalid* in the opposite case. Documents are not required to specify the schema they adhere to, in many cases the applications that handle XML documents only require documents to be *well-formed*.

An XML document can specify which Document-Type is associated with its markup by using the reserved element named `<!DOCTYPE` provided in the document prolog; in the following code, an example of XML document that specifies the associated DTD is shown:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE collection SYSTEM "http://exampledtd.com/nasa/ned.dtd" >
```

The XML `<!DOCTYPE` markup (also described as *Document Type Declaration* [19]) in the example above defines:

collection as the Root Element Type; this first parameter must match the document root label in the current document;

SYSTEM specifies that the DTD is external to the document, the DTD is subsequently specified in the following parameter;

http://../ned.dtd this last parameter explicitly states the URI location of the DTD that matches the current document, the URI can specify a resource available online (such as in the example) or available elsewhere ("file://" and other protocols can be used).

In the following section a short description of the DTD specification is provided, along with a discussion about other schema definition languages proposed in the literature to solve some DTD limitations.

2.1.4 Document Type Definition (DTDs)

The *Document Type Definition* language defines the set of XML markup (such as elements, attributes, notations and entities) that a document may use; furthermore the DTD specifies the basic data validation, the element nesting and their composition.

As an example the following code shows the Document Type Definition for the XML document introduced in Section 2.1.1:

```
1  <!ELEMENT collection (object+)>
2  <!ELEMENT object (name, position, distance?, description*)>
3  <!ELEMENT name (#PCDATA)>
4  <!ELEMENT position (latitude, longitude)>>
5  <!ELEMENT distance (#PCDATA)>
6  <!ATTLIST distance unit (meters | centimeters | lightyear) #REQUIRED>
```

In the example the XML elements are defined by the `<!ELEMENT` symbol followed by the element label and, enclosed by rounded brackets, the list of child element names or the type of the element content. In case a list of elements is provided, such list can include specific cardinality modifiers that define if an element can appear "one or more" times (modifier '`+`'), "zero or more" (modifier '`*`') or "zero or one" (modifier '`?`'); if no modifier is specified the element is assumed to appear exactly once.

For example the element `object` (defined in line 2) can appear one or more times as a child of the `collection` element; the element `distance` is optional for the element object (but can appear at most one time), while the element description can appear zero or multiple times as a child of object.

Taking as an example the definition of the element distance in line 6, the DTD specifies that its contents must be a *Parsed Character Data* (#PCDATA) that is a plain text without any other child XML element.

The <!ATTLIST markup allows to define XML element's attributes: the element name is specified as the first parameter, followed by the list of the defined attributes. Attributes are specified with a triple that corresponds to: attribute name, attribute data type and the *attribute value declaration* (i.e. if the attribute is required, optional, fixed or has a default value). The most used data types, as defined by DTD, are the following:

CDATA Character data, similar to the #PCDATA defined for elements contents;

ID the value represents a unique identifier for the containing element;

IDREF the value is a reference to a unique value ID that identifies an element;

IDREFS the value is a list of referenced IDs, separated by spaces;

Enumeration list This data type defines a list of admissible values for the attribute.

Other data types defined by DTD are: NMTOKEN (a string of characters without spaces), NMTOKENS (a list of NMTOKENs separated by spaces), ENTITY (a space separated list of entity or *entity-reference* values).

Regarding the *attribute value declaration*, the DTD allows the following attribute definition:

#IMPLIED the attribute value is optional, and no default value is provided;

#FIXED the attribute value is fixed, this means that if the attribute is specified in an XML document its value must match the DTD one, if the attribute is not present in the document, it is automatically added;

#REQUIRED the attribute is required, but no default value is provided.

Taking as an example the DTD provided above, the unit attribute for the element distance is *required* and it can take one of the three provided values: meters, centimeters and lightyear.

As previously outlined, and as shown from the list of data types provided above, the DTD only allows a basic data type definition for element content and attribute values: only plain text (such as #PCDATA and CDATA) is actually used for schema evaluation thus limiting the DTD expressiveness. Such limitation does not allow, for example, to define numeric data types or values representing dates.

After DTD, other languages that allow a more powerful schema specification for XML documents have been defined such as *RELAX NG* (REgular LAnguage for XML Next

Generation) [20] or *Schematron* [21]. The W3C has defined the standard XML-Schema [22] language that allows the definition of XML Document Type Definition with a more detailed specification, thus allowing a more precise validation.

The main differences among the mentioned languages rely on the syntax for the schema definition and on the XML document validation: *RELAX NG* adopts a simple syntax inspired by *regular expressions*, *Schematron* is a rule-based validation language, while *XML-Schema* adopts XML as the definition language for XML documents.

2.1.5 XML Document Example

In this section, as a short conclusion of the synthetic introduction to XML, the source code of the XML document provided in the examples of the previous sections, is presented in its complete form in Listing 2.2.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE collection SYSTEM "http://exampledtd.com/nasa/ned.dtd" >
<?xml-stylesheet type="text/css" href="docstyle.css"?>

<!-- Data taken from NASA Extragalactic Database -->
<collection>
  <object catalog="M31" type="galaxy">
    <name>Andromeda</name>
    <position>
      <latitude unit="degrees">80.893</latitude>
      <longitude unit="degrees">69.756</longitude>
    </position>
    <distance unit="meters">7.92 E+23</distance>
    <description><![CDATA[
      NGC 224 - M 31: The M31 nucleus is highly compact, and is the
      prototype of the central luminous star clusters found in many
      galaxies. The bulk of its stellar population corresponds
      to an old (& >~10 Gyr) metal-rich component (Bica et al. 1990).
      This is also reproduced by our synthesis analysis.
    ]]></description>
  </object>
  <object catalog="LMC" type="galaxy">
    <name>Large Magellanic Cloud</name>
    <position>
      <latitude unit="degrees">21.847</latitude>
      <longitude unit="degrees">22.454</longitude>
    </position>
    <distance unit="meters">4.90 E+22</distance>
    <description><![CDATA[
      This is the Large Magellanic Cloud. The plate was taken
      by Dr. Karl Henize with the Mount Wilson 10-inch refractor
      in the light of H{alpha}. Although the form of this galaxy[..]
    ]]></description>
  </object>
  <object catalog="virgo-cluster" type="cluster">
```

```
    <name>Virgo Cluster</name>
    [...]
  </object>
</collection>
```

Listing 2.2: Example of XML document

In the above XML document code the XML *prolog*, *CDATA*, DTD, and the whole markup correspond to the previously described XML language features; only the XML element named `description` has been added to further illustrate the `CDATA` block usage when the element contains XML entities such as “`&`” and “`<`”.

2.1.6 Summary

In Section 2.1 the main features of XML have been described: XML Elements, Attributes, Schema and Entities provide the building blocks for defining XML document contents and their structure. An important aspect of XML is the implied user-readability of an XML document and the high flexibility degree offered by the language: ad-hoc structures can be defined to any purpose, from data exchange to document layout and presentation. Not surprisingly XML is the de-facto standard for Web documents (xHTML[23] and its variants), WebServices protocols (SOAP [24], WSDL [25]) and data interchange formats such as websites syndication formats like RSS and Atom. Not limited to the above list, the adoption of XML is increasing, an example is the document definition in wide used Office-Productivity tools (as Microsoft Office, LibreOffice) or a standard for instant-messaging and VoIP communications systems with the XMPP [26] protocol.

2.2 Querying XML documents

Processing XML documents contents and structure is another crucial part for the XML language definition: huge structured data, once defined and stored, require appropriate techniques to query and retrieve document contents and elements.

Several retrieval methods have been proposed in the literature to exploit the salient features of XML documents, where the semantic information of contents may be conveyed through the document structure and the provided element nesting. Such characteristic is different from other markup formats, such as HTML, where tags are used mainly for presentation and stylistic purposes.

The use of the structure associated with a document content is not always aimed at the same intent in different documents: XML documents can be classified into two different categories given their content organization. In the literature **data-oriented** documents refers to XML documents that contain a highly structured organization of their data contents, usually the XML components in such type of documents can be considered as database records. The second type of XML documents is called **text-oriented**, in these documents the structure is often irregular and contents are organized to be easily used and read by humans.

Given this classification, document querying and content access have been proposed in the literature from two different perspectives: the *Information Retrieval* and the *Database* point of view. The *Information Retrieval* point of view considers approximate matching of document fragments, and it focuses on ranking retrieved XML elements according to their relevance to the query. On the other side, the *Database* point of view deals with XML documents as database records, focusing on exact matching during query evaluation instead of ranking and relevance assessment of retrieved elements.

Since the definition of XML documents and the widespread usage of XML, querying highly structured documents has been considered by the database community as *querying semi-structured data*, as opposite to the Information Retrieval community that talks about *structured text retrieval*, by taking into consideration that the structure serves the purpose to organize the textual contents in XML documents. The differences between the two perspectives on querying XML documents are nowadays not sharply defined: Baeza et al. [27, p. 2867], for example, summarize the relationship between the two approaches as:

> "From a terminology point of view, structured text retrieval and querying semi-structured data, in terms of end goals, are the same, i.e., finding elements answering a given query. The difference comes from the fact that in information retrieval, the structure is added, and in database, the structure is loosened".

In the following sections the two main standard languages for querying XML documents, as defined by the World Wide Web Consortium (W3C), are described: XPath [2] (in Section 2.2.1) and XQuery [3] (in Section 2.2.2). The XPath language allow users to select and retrieve XML elements and attributes by the specification of structure-based and content-based constraints. The XQuery language, subsequently defined on top of the XPath language, provides techniques to manipulate XML documents by element selection, ordering, and re-arrangement. Both languages have been mainly inspired from a database perspective of XML documents and allow exact matching of query constraints.

Furthermore, in Section 2.2.3 the most important query language proposed by the IR community, named NEXI [28], is presented.

2.2.1 The XPath language

The XPath [2] language has been defined by the W3C to allow the selection of parts (or fragments) of an XML document: XML nodes, XML attributes and textual contents can be selected and retrieved using the XPath language.

The XPath constructs and expressions, and the XQuery ones described in Section 2.2.2, have been conceived based on a highly structured view of an XML document content. As further discussed in the XQuery language section, these languages adopt a database like style for querying and matching XML elements.

The main components of the XPath language are:

- *Expressions*: they represent the XPath query, its components and the set of predicates that constitute the query;
- *Nodes* and *Node Sets*: they specify the types of elements returned by the evaluation of an XPath expression;
- *Context*: it represents the environment in which an expression is evaluated, it can be the whole document tree or a sub-tree of the complete document;

When querying an XML document, and in particular when a set of nodes is retrieved, such set of items is usually called *Node-Set* as it actually represents a set of XML nodes of any type (text contents, attributes, comments, etc.). Every item in such set is also informally called *fragment* as it represents a section of the entire XML document from which it was selected; from now on the concept of *document fragment* is used with this meaning.

Before describing the semantics and the complete syntax of the XPath language, the main characteristics of the language are introduced in the following sub-sections.

2.2.1.1 The XPath Data Model

The XPath language handles both the structure and data of XML documents by using a Document Object Model (DOM) [29] that describes the document as a particular graph, in particular a tree of nodes, where a type is associated with each node. XPath DOM defines seven different types of nodes, also called XPath Node Types, which are: the *Root-Node*, *Element-Node*s, *Text-Nodes*, *Attribute-Node*s, *Comment-Node*s, *ProcessingInstruction-Node*s, and *Namespace-Node*s; such node types have a correspondence with the XML language definition as provided in Section 2.1.

In the XPath Document Object Model each XML document has a single and unique *Root-Node* that represents the root of the document elements hierarchy; it does not have a name and it admits a single child that is the *root element* of the XML document as described in Section 2.1.3. Subsequently, all the descending *Element-Node*s of the *Root-Node* represent, in turn, all the descending XML elements in the XML document. Each *Element-Node* may have, following the XPath DOM, children nodes that identify attributes, comments, descending XML elements or the contained text; such children are represented by *Attribute Node*s, *Comment Node*s, *Element-Node*s and *Text-Node* respectively. Furthermore, any *Attribute-Node*, *Comment-Node* or *Text-Node* must not have children as they are defined as leaf nodes in the XPath Document Object Model and no tags can be defined, nor nested, in such elements.

2.2.1.2 XPath Expressions

An *XPath Expression* constitutes the main component of an XPath query: the evaluation of an XPath expression results in a Node-Element or a Node-Set that matches the constraints specified in the query.

The most used XPath expression is the *Path Expression* (also known as *Location Path*) that allows to identify and retrieve an XML element, or a set of XML elements (or *Node-Set*), given their path from the root XML element or from a context. A Path Expression is composed of a sequence of one or more *Location Step*, that define the relation between two XML elements. Let us take into consideration the following XPath Expression:

```
/collection/object/name
```

The XPath expression would match all the XML elements `<name>` children of an `<object>` elements that are, in turn, children of the root XML element labeled `collection`. Following the XML example document in Section 2.1.5 (Listing 2.2), the result of the evaluation of such an expression would be the following set of elements:

```
<name>Andromeda</name>
<name>Large Magellanic Cloud</name>
<name>Virgo Cluster</name>
```

Listing 2.3: Node Set returned for the search `/collection/object/name`

The provided example is also called *absolute location path* given its starting '/' character that identifies the context of the expression the Root Node in the XPath data model. If the context of the evaluation is not specified, or if it is not the Root Node, the location-path is called *relative location path*, and it is evaluated by using the current context of the expression.

Path expressions are composed of a sequence of units called *location step*s that conform to the following pattern: `/axis::nodetest[predicate]`. Before describing the formal XPath language syntax, some further details are provided to better understand XPath query components:

- an **axis** allows to define the location path in terms of the relationship that must occur between two nodes;
- a **nodetest** allows to identify and match a set of XML nodes by their type (tag, attribute, etc..) or their label name;

- a **predicate** allows to define boolean expressions that are matched against a given context, such evaluation allows to filter nodes that do not match a given condition such as the presence of a given child node or the exact value of an attribute.

2.2.1.3 XPath Axes

In the XPath language there are 13 different *axes*, usually classified as *forward* and *reverse* axes that refer to the direction in which the nodes are visible relatively to the actual context node. In Table 2.1 the complete list of the available axes, as defined in the XPath language, is presented; for some axes the abbreviated form is also cited.

Regarding the implementation of the XPath language (recommendation 1.0) the W3C does not require the availability of the following axes: `following`, `following-sibling`, `ancestor`, `ancestor-or-self`, `preceding`, and `preceding-sibling`. Furthermore the axis `namespace` has been deprecated in the subsequent versions of the XPath language.

2.2.1.4 XPath Node Test

The Node Test part of an XPath location step indicates which nodes have to be selected by the axis evaluation: the selection can be by node name or by node type. A list of the main *node test* available in XPath is presented in Table 2.2.

2.2.1.5 XPath Predicates

As previously outlined, the XPath language allows to include in any XPath expression the predicates useful to filter the nodes based on specific constraints. A predicate is simply an expression in square brackets (`[` and `]`), where an expression can contain zero or more predicates.

If a predicate evaluation produces a *false* value (equivalent to the number *0*, the empty sequence, or a *Not-a-Number* value), the entire predicate is considered false, true otherwise. Instead, if the expression evaluation produces a *non-zero* number, this number is considered as the positional sequence of the item to be processed; in such case the predicate is called *positional predicate*. An example of such type of predicate is shown in the following expression, where to a path expression (`/collection/object`) a simple *positional predicate* (in the form `[2]`) is added:

```
/collection/object[2]
```

The expression will retrieve, if it exists, the second element labeled `object` from the set of children of the `collection` element; given the example XML document in Listing 2.2, the results of the previous query are presented in Listing 2.4.

Axis	**Description**	**Direction**
`child` (abbreviated as “`/`”)	Matches all the children Element Nodes of the context node (Attribute Nodes are excluded)	Forward
`parent` (abbreviated as “`..`”)	Matches the parent Element Node of the context node	Reverse
`descendant` (abbreviated as “`//`”)	Matches all the descending Element Nodes from the context node, including the children nodes	Forward
`ancestor`	Matches all the ancestors Element Nodes of the context node, from the parent element back to the root element	Reverse
`descendant-or-self`	Selects all the descendants Element Node of the context node, including the context node itself	Forward
`ancestor-or-self`	Like the `ancestor` axis, it matches all the ancestors Element Nodes, also including the context node itself	Reverse
`following`	Selects all Element Nodes that follow, in document order, the context node in the document, excluding the context node’s descendants	Forward
`preceding`	Matches all nodes that precede, in document order, the context node in the document, excluding the context node’s ancestors	Reverse
`following-sibling`	Selects all siblings Element Nodes of the context node that appear before it (in document order)	Forward
`preceding-sibling`	Selects all preceding Element Nodes of the context node that appear after it (in document order)	Reverse
`attribute` (abbreviated as “`@`”)	Matches all the attributes of the context node (if any)	Forward
`namespace`	Selects the namespace of the context node	Forward
`self` (abbreviated as “`.`”)	The context node itself	N/A

Table 2.1: List of XPath axes

Node Test	**Node type matched**
`node()`	Matches any node
`text()`	Matches a text node
`comment()`	Matches a comment node
`name`	Matches an Element Node labeled `name`
`@attr`	Matches an Attribute Node labeled `attr`
`*`	Matches any Element Node

Table 2.2: Main list of *Node Test*s in XPath language

```
<object catalog="LMC" type="galaxy">
  <name>Large Magellanic Cloud</name>
  <position>
    <latitude unit="degrees">21.847</latitude>
    <longitude unit="degrees">22.454</longitude>
  </position>
  <distance unit="meters">4.90 E+22</distance>
  <description><![CDATA[
    This is the Large Magellanic Cloud. The plate was taken
    by Dr. Karl Henize with the Mount Wilson 10-inch refractor
    in the light of H{alpha}. Although the form of this galaxy[..]
  ]]></description>
</object>
```

Listing 2.4: Node Set returned from the evaluation of the expression `/collection/object[2]`

An example of a simple XPath expression containing a predicate is the following, where the predicate `[@type="galaxy"]` evaluation produces a *true/false* value:

```
/collection/object[@type="galaxy"]/name
```

The evaluation of this expression retrieves, if querying the sample XML document in Listing 2.2, the elements labeled `name` child of an `object` element that must have an attribute named `type` the value of which must be "*galaxy*". The set of retrieved elements is shown in Listing 2.5.

```
<name>Andromeda</name>
<name>Large Magellanic Cloud</name>
```

Listing 2.5: The *Node Set* returned by evaluating the expression `/collection/object[@type="galaxy"]/name`

Predicates can be applied to any node, and multiple predicates can be logically combined by using the `and` and the `or` connectives. An example of an XPath expression with multiple predicates is the following:

```
/collection/object[@type="galaxy" and @catalog="M31"]/name
```

Like in the previous example, the expression selects only the elements labeled `name`, but it requires that the parent `object` element has two attributes named `type` and `catalog`, having the value of "*galaxy*" and "*M31*" respectively. The only retrieved element, in this case, is the XML element having *Andromeda* as the textual content of the tag `name`.

Another example is the following expression, where the `name` node of `object` elements that have a child element `position` are retrieved: this criterion only checks that the element `position` exists as a child of the current context node (the `object` node).

```
/collection/object[/position]/name
```

2.2.1.6 XPath Functions

The XPath language also includes a set of functions that may be used in predicates: the most common functions operate on strings (like node content or attribute values) or node sets. Here below, in Table 2.3 the most commonly used XPath functions are briefly described; for a complete list of the available built-in functions in XPath please refer to [2].

Function	**Description**
`contains(string1, string2)`	Returns true if the text in *string1* is contained in *string2*, false otherwise
`start-with(string1, string2)`	Returns true if the the text in *string1* starts with the value in *string2*, false otherwise
`count(nodeset)`	Returns the count of elements in the given *node-set*
`position()`	Returns the ordinal position of the current context node within the context sequence being processed

Table 2.3: Short list of built-in functions for the XPath language.

2.2.1.7 Summary

In Section 2.2.1 the XPath language has been introduced and its characteristics analyzed and described. XPath has been defined as a language for selecting elements and attributes form an XML document while traversing its hierarchy and filtering out unwanted content. Given its simple syntax, it has been adopted and leveraged in other languages that used XPath features for element selection; such languages, besides the previously mentioned XQuery, include: XSLT [18], XPointer [30], XForms [31] and XML Schema [22].

In December 2010 the W3C defined the first recommendation for a 2.0 version of the XPath language [32], which includes a wide variety of expressions and functions (like `for`, `let`, `if`, `some`, and `every` expressions, as well as type casts) shared with the XQuery language; thus the new XPath language allows to express not just path expressions. The new version of the XPath language has been defined to be compatible with the previous 1.0 version as possible: almost all the XPath 1.0 expressions are still valid in XPath 2.0, slight differences are in some value handling functions.

The XPath 2.0 and the XQuery language (described in Section 2.2.2), share the same underlying document object model called XQuery and XPath Object Model (XDM) Core 3 [33]: it is a slightly different, and more powerful model than the aforementioned DOM model. XDM extends the DOM by including the support for objects with type (described by the W3C XML Schema), and items such as floating-point numbers. The XQuery language is described in the next Section 2.2.2.

2.2.2 The XQuery language

The XQuery language [3], as initially defined by the W3C in 1999, allows to select, reorganize, transform and finally return a set of XML elements in a structured form. It is also indicated as an Information Processing Language [34] as it provides more features than a pure query language.

The XQuery language, leveraging the XPath language to select and filter XML elements, is a functional language that offer features such as:

- searching and joining information across multiple documents;
- sorting, grouping and aggregating data;
- transforming XML data into another XML structure;
- performing arithmetic computation on dates and numbers;
- updateing XML data in the source XML document[3].

As it can be noticed from the set of features of XQuery, this language has several characteristics in common with the SQL language (some designers of the XQuery language were also involved in the design of the SQL language) and for this reason XQuery was initially referred as the "SQL for XML". This correlation between XQuery and SQL is confirmed by the fact that the early adoption and implementation of the XQuery language were on Native-XML Database products, where the data handling processes, as

[3]This feature has been defined for a future version of the XQuery language under the name of *XQuery Update Facility*, its definition is available in [35]

opposite to traditional relational databases, are oriented towards XML and structured documents.

Starting from its first versions the XQuery language has been defined as a collaborative effort to integrate different XML query languages proposed in the literature into a standard language. Some languages from which XQuery was inspired were: Quilt [36], XQL [37], XML-QL [38] and Un-QL [39]. For a report of the main existing query languages proposed before the actual definition of the XQuery 1.0 language, a summary and comparison is presented in [40] and [41].

One of the main characteristics of the XQuery language is that it can be extended, similarly to object-oriented languages, with additional functions and modules; such extensions can be defined for the XQuery language to further reduce the query complexity and to make the query more reusable and readable.

2.2.2.1 XQuery 1.0 (and XPath 2.0) Data Model (XDM)

The XQuery language adopts a slightly enhanced data model than the previous DOM (also called Core 1) model; the XDM data model (*XQuery 1.0 and XPath 2.0 Data Model*) was defined by the W3C in [33]. The XDM has added support for sequences of elements and atomic values; such features were missing from the previous Document Object Model 1.0.

An important difference between DOM and XDM relies on how the *root element* or *root node* is handled by the two models: in XPath 1.0 the input of a query is expected to be a well-formed and complete XML document, and only a *root node* is expected. In the XDM (that allows the input to be a document fragment or a sequence of multiple elements) the meaning of *root node* for inputs is not maintained, and the concept of *document node* and *root element* have been introduced. Such differences between DOM and XDM are due to the enhanced ability of XQuery 1.0 and XPath 2.0 to work on *data model instances* (also called abstract trees) that could be constructed not only from XML documents, but also from relational databases, RDF triple store, and more.

2.2.2.2 XQuery Expressions

The XQuery language expressions are named FLWORs expressions, an acronym that stands for: "*For, Let, Where, Order-by, Return*". The acronym describes, in a compact way, all the features of the XQuery language. FLWOR expressions allow the query to select, manipulate and transform the set of XML elements into a desired result such as a new XML document or a set of XML elements.

The XQuery language leverages the XPath language to select XML elements using path expressions; such dependency limits the manipulation power of the language: the selected elements (XML nodes, content or attributes) can be only returned as matched by the path expression, but neither manipulated nor re-ordered. A first difference with respect to the XPath expressions is on the absence of the so called *default context item*: in XPath the initial context of any expression is the entire XML document, while in XQuery expressions such context must be always declared.

In Listing 2.6 a simple XQuery expression is presented; the result of its evaluation is the same of Listing 2.5; the first difference is the required specification of the *default context* of the expression, in this case it is returned by the evaluation of the function `doc()` that loads the XDM data model from the given XML file.

```
doc("nasa-ned-collection.xml")/collection/object/name[@type="galaxy"]
```

Listing 2.6: Simple XQuery expression with default context speficied.

A more extensive example of XQuery FLWOR expression is readable in Listing 2.7, where most features of the XQuery language are used to retrieve, manipulate and return a new set of XML elements.

```
1   for $obj in doc("nasa-ned-collection.xml")/collection/object
2   where $obj@type="galaxy"
3   order by fn:lower-case($obj/name/text()) descending
4   return
5     <galaxy catid="{$obj@catalog}">
6       <name>{$obj/name/text()}</name>
7       <data>{$obj/position}</data>
8     </galaxy>
```

Listing 2.7: Example of XQuery expression with FLWOR clauses.

The example expression allows to select, from a local XML document named *nasa-ned-collection.xml* (the content of which is shown in Listing 2.2), all the `objects` having a `type` attribute containing the value *galaxy*. The results are then sorted in a descending lexicographic order (from Z to A) avoiding uppercase/lowercase mismatch, and the elements are restructured as a new XML fragment having only a `name` element and all data related to the element `position`.

As shown in Listing 2.7 an XQuery FLWOR expression is composed by multiple parts:

For The `for` clause, differently from the typical meaning in procedural languages such as C or Java, generates a sequence of nodes retrieved from the evaluation of the path expression `doc("nasa-ned-collection.xml")/collection/object` (in line 1); the subsequent FLWOR expression is evaluated once for each of the items in the sequence, and its value is stored in the variable named `$obj`[4].

[4] In the XQuery language the dollar (`$`) sign is used to define variable names: all the variable's names must start with such symbol. XQuery variables may contain any sequence: nodes, numeric values, strings, or any *Atomic Value* as defined in the XQuery XDM. Variables values can not be changed: once the expression evaluation bounds a variable to a value, such value becomes unchangeable.

Let The `let` clause (not used in the example in Listing 2.7) allows to define variables and assign values to them; path expressions or XQuery functions may be used to obtain such value. The `Let` clause is often used to avoid repetitions in XQuery expressions and to provide, in some XQuery engine implementations, better performances during the query evaluation.

Where As in the SQL language from which XQuery has been defined, the `where` clause filters and selects only elements matching given criteria, in the example in Listing 2.7 (line 2) only the items with an attribute named `type` having a *galaxy* value are kept and not filtered out.

Order by This clause sorts the results; like the similar clause in the SQL language, an `ascending` or `descending` sorting may be specified. In the example query expression, at line 3, the `fn:lower-case()` function is applied to the contents of the element `name`, and it provides the values for the sorting process; further details about the XQuery functions are given in Section 2.2.2.3.

Return This clause allows to define which and how the elements retrieved by the expression should be returned from the evaluation: new XML elements with a customized structure may be returned with textual contents or attributes values taken from the current items and variables. In lines from 5 to 8 of Listing 2.7 the expression defines a new XML structure for the retrieved elements. The variable `$obj` can be accessed and queried by using a path expression to access the element attributes or values by using the XQuery technique called *enclosed expression*.

```
<galaxy>
  <name catid="LMC">Large Magellanic Cloud</name>
  <data>
    <position>
      <latitude unit="degrees">21.847</latitude>
      <longitude unit="degrees">22.454</longitude>
    </position>
  </data>
</galaxy>
<galaxy>
  <name catid="M31">Andromeda</name>
  <data>
    <position>
      <latitude unit="degrees">80.893</latitude>
      <longitude unit="degrees">69.756</longitude>
    </position>
  </data>
</galaxy>
```

Listing 2.8: Results of the evaluation of the XQuery expression in Listing 2.7.

Listing 2.8 shows the result of the evaluation of the expression in Listing 2.7; each matched element has a completely different structure than in the original XML document: some elements have been kept (for instance the `position` element and its descending nodes) and other have been reorganized (the new `catid` attribute have been added to the `name` element).

The XQuery language allows to combine a query expression with a so called *enclosed expression*: such an expression is enclosed in curly brackets “`{`” and “`}`” and, once evaluated by an XQuery engine, its value is returned as one or more atomic values. As an example, in Listing 2.7 at line 5 an enclosed expression is specified to access the item data: the evaluation of `{$obj@catalog}` returns the value of the attribute `catalog` of the item `$obj`, as shown in the obtained results.

2.2.2.3 XQuery functions

The XQuery 1.0 language defines more than 100 built-in functions, spanning from string manipulation to date/time computation and Regular Expressions matching. The functions have been defined by the W3C in [42], and they are shared by the XQuery 1.0, XPath 2.0 [32] and XSLT 2.0 [43] languages.

Besides the built-in functions, XQuery 1.0 permits user defined functions to be provided and used in any XQuery expression; this allows query fragments to be reused, and the development of libraries that can be shared and reused by other parties.

Differently from the XPath functions described in Section 2.2.1.6, the XQuery 1.0 functions are enclosed in a namespace named `fn:`; the user can invoke a function by calling the function, for example, as `fn:lower-case(//name/text())` to obtain a lowercase representation of the textual contents of the `//name` node.

In Table 2.4 a short example of XQuery functions is provided; the XQuery function signatures, as they employ the `fn:` namespace, differ from XPath also for the usage of typed elements: as an example the input parameter for the function `fn:round()` must be a *numeric* object, while the parameter for the `fn:data()` is a generic element.

For a complete list of the functions defined for XQuery 1.0 (and the related languages) refer to [44].

The XQuery 1.0 functions, as defined in [42], are allowed to have a behavior either *implementation dependent* or *implementation defined*: such characteristics allow the XQuery engines to behave differently given the same function. This point is of particular interest for the XQuery/XPath Full Text extension (further described in Section 2.3), where the computation of the score, related to a text relevance, is left to vendor implementations without the need to conform to any standard or common algorithm.

Function	Description
`fn:contains($arg1 as xs:string?, $arg2 as xs:string?)`	Returns a boolean indicating whether the value of *$arg1* contains the value of *$arg2*.
`fn:lower-case($arg as xs:string?)`	Returns the string representation of the given input where all characters have been converted to their lowercase representation, adhering to the Unicode Standard [45].
`fn:data($arg as item()*)`	Accepts a sequence of items and it returns their typed values: for atomic values it returns the value itself, for nodes it extracts the typed value of the node.
`fn:matches($input as xs:string?, $pattern as xs:string)`	Returns a boolean value if the given input string matches the regular expression provided in the `pattern` parameter.
`fn:id($arg as xs:string*)`	Returns the elements in the XML document with the given ID, or the given set of space separated list of IDs.

Table 2.4: Example list of built-in functions for the XQuery 1.0 language.

2.2.2.4 XQuery 1.0 (and XPath 2.0) extensions

Given the increasing areas of application of the XQuery language, from digital documents to Scalable Vector Graphics (SVG [46]), the standard built-in functions that are available in both XQuery 1.0 and XPath 2.0 have been devised to be not sufficient for some tasks. For that reason the W3C created the EXPath Community Group [47] with the aim of leading the definition of language extensions for all the XPath 2.0 related languages such as XQuery 1.0, XProc [48], XForms [31], XSLT [43] and XML Schema [22]. The EXPath defined a set of function libraries, also called *modules*, to extend the functions provided by XQuery with commonly adopted solutions to the aim of reusing standard techniques, and to provide a shared set of extensions. Such purpose is well described in the EXPath website:

> "EXPath exists to provide specifications for such missing features in a collaborative- and implementation-independent way. EXPath also provides facilities to help and deliver implementations to as many processors as possible, via extensibility mechanisms from the XPath 2.0 Recommendation itself".

The list of defined modules includes Filesystem I/O [49], Geospatial API [50], Crypt functions [51], compressed (ZIP) archives handling [52], HTTP protocol communication [53] and execution of SQL expressions on databases [54]. Please refer to the EXPath [47] website for the complete list and details of EXPath defined modules.

Apart from EXPath functionality extensions, the XQuery language is being under a review process from the W3C, and the definition of a 3.0 version [55] is, at the time of writing, at a *Last Call Working Draft* status[5]. The new XQuery version is planned to include:

- database-oriented clauses like the missing `group by` and `count` expressions and `allow empty` clause to simulate SQL outer-joins;
- enhanced function declaration with the inclusion of: private and inline functions;
- execution flow controls with `try/catch` and `switch` expressions;

2.2.2.5 Summary

The XQuery 1.0 language has been defined as a joint work of Database and Information Retrieval experts; the prominent influence from the database community is proven by the similarities between the XQuery expressions with the SQL language. The XQuery language has been integrated in many systems where the main aspect of manipulation and element reorganization of XML documents played an important role in data handling and data manipulation.

More and more companies are adopting the XML as a standard representation of their data; this pushed both the implementation of XQuery engines, and the definition of ad-hoc solutions for data handling when XML documents contain complex data (such as compressed archives or geo-spatial data). The goal of the W3C EXpath community is, in fact, to propose an extended set of functionalities to address such challenges by leveraging the XQuery and XPath extendibility characteristics.

For those reasons a large number of companies started to adopt XML as the underlying structure for their data management systems, and to integrate the XQuery language as the main query language. Such XQuery engines go from commercial products like Oracle 10g, IBM DB2 to open source project like Berkley DB XML, eXist and BaseX only to mention some of the available systems that implement the XQuery language specification[6].

2.2.3 The NEXI query language

As mentioned in Section 2.2, the issue of querying XML documents has been addressed by two communities: the Information Retrieval (IR) one and the Database one. The IR community envisioned a twofold XML querying approach: by a first approach the XML

[5] The XQuery 3.0 draft is available at `http://www.w3.org/TR/2013/WD-xquery-30-20130723/`, while, regarding the version numbering, the W3C never worked on a XQuery 2.0 version, but just went from the 1.0 to the 3.0 version release.

[6] A more complete and exhaustive list of XQuery 1.0 and XPath 2.0 implementations can be found at the W3C website: `http://www.w3.org/XML/Query/#implementations`

structure should be taken into account while evaluating a query, and the content part of a document (XML text nodes, as referring to the XPath and XQuery data model) should be evaluated with IR techniques for text matching.

The IR community outlined that in the XPath (and thus in the XQuery language) only exact matching for content based constraint were defined: all the XPath functions for selecting nodes given their content were based on exact matching of strings or substrings. As an example the `contains` function, as described in Table 2.3 only provides an exact containment matching for `test nodes`, without any score computation.

To allow a keyword based matching for XML documents Trotman and Sigurbjörnsson proposed the NEXI [28] language; NEXI leverages the XPath 1.0 element selection syntax (thus the set of XPAth axes, path-predicates and node-test) and adds the `about` function, specified to express content constraints in a keyword-based IR style. The NEXI language has been defined by its authors as the "*[..] simplest query language that could possibly work.*" [56] as compared to the complete, but hard-to-learn, XPath or XQuery languages.

An example of a NEXI query is presented in Listing 2.9, where all the target `object`s having a descendant node `description` about the words "*Magellan Cloud*" should be retrieved. The actual evaluation of such a query is highly dependent on the underlying query system, where different selection and matching strategies could be applied.

```
/collection/object[about(./description, "Magellan Cloud")]
```

Listing 2.9: Example of NEXI query, using the content constraint function `about`.

The NEXI language has been introduced during the campaign of the INitiative for the Evaluation of XML retrieval (INEX) in 2004. In such initiative, the NEXI language has been used to evaluate two different types of query: Content Only (CO) queries and Content and Structure (CAS) queries. Apart from the CO queries, that aim at querying structured documents in XML format using only a set of keywords, the CAS queries provide both structured-based and content-based constraints using the XPath element selection and the `about` constraint for content evaluation. The NEXI language has also been extended and used for question answering by Olgivie [57] and for querying multimedia documents [58] by the same authors of the NEXI language.

A more detailed description of the use of the NEXI language for evaluations performed by the Information Retrieval community for querying XML documents is presented in the next Section 3.

2.3 The XPath/XQuery Full-Text extensions

Given the increasing popularity of the XQuery and XPath languages, and also due to the huge adoption of XML as a standard format for documents, in particular text-oriented documents (as introduced in Section 2.2), the XML community required a standard definition for full text search in XML documents. Such features were required to effectively query XML documents that contain large texts, and where the exact string/substring matching functions are not enough.

The definition of a standard XQuery 1.0 and XPath 2.0 Full-Text [59] (XQFT) language extension by the W3C also represents a joint work from the Database community and the Information Retrieval community: standard IR techniques have been added to the XQuery 1.0 and the XPath 2.0 languages. The extensions include features such as element relevance computation (element scoring), and full-text search capabilities like tokenization, stop-word removal and stemming.

The W3C group also published a set of XQuery Full-Text Requirements [60] and Use Cases [61] that lead to the Full-Text 1.0 specification, other than the definition of conformance levels of XQuery Full-Text processors to the language. Finally a Test Suite [62] with more than 650 test cases has been defined to support XQuery Full-Text processors implementation.

In this section the Full-Text [59] extension of the XQuery 1.0 language is presented and detailed. Given the high relevance of the XQuery Full-Text language extension to the research activity undertaken during my doctoral period and described in this manuscript, the XQuery Full-Text extension will be described at a deeper level of details than the one adopted for presenting the other languages described in the previous sections.

The XQuery 1.0 Full-Text extension is introduced in the following Section 2.3.1, and its syntax is detailed in Section 2.3.2. A formal and a semantic description of the relevance score computation is finally given in Section 2.3.3.

2.3.1 Introduction

A Full-text search functionality was initially introduced as a language extension in different XML query engines with custom expressions or ad-hoc functions, thus lacking a standard definition in both the XQuery and XPath languages by the W3C. The same requests, and a subsequent standardization, already took place for the SQL language where the ISO provided full-text search features in the SQL/MM Full-Text [63] standard in 2003.

The introduction of full-text search for XML documents has been studied by the Information Retrieval community since the definition of text-oriented XML documents; among the language proposals provided in the literature we cite the ones by Florescu et al. [64], Chinenyanga et al. [65], Theobald et al. [66] and Fuhr et al. with the XIRQL [67] language.

The proposal adopted (and further extended by the W3C) as the Full-Text extension of the XQuery 1.0 and the XPath 2.0 languages was TexQuery [68] defined by Amer et al.:

the TexQuery language introduced a set of primitives for text search such as: boolean connectives, phrase matching, and proximity distance as an extension of XQuery 1.0.

The full-text search differ from standard XQuery textual content string matching in many ways; the first is that textual content and query input string are *tokenized* (split into atomic chunks of text, such as words). The Tokenization, as in IR, may include several operations before producing the actual set of tokens that will be handled during a Full-Text search; such operations include uppercase/lowercase conversion, stop-words removal, word stemming and diacritics handling.

An example of the differences between substring matching and the matching defined in the XQuery Full-Text language are shown in the example reported in Listing 2.10: the `contains` function, that allows substring matching, is compared to the Full-Text expression `contains text` (further described in Section 2.3.2); as the reader can notice, the Full-Text syntax is readable and intuitive even if its various matching options could led to long and articulated expressions.

```
XQuery:
  contains("release", "lease")         ==> true
XQuery Full-Text
  "release" contains text "lease"      ==> false

XQuery
  contains("Run", "running")           ==> false
XQuery Full-Text
  "Run" contains text "running" using
  stemming using case insensitive      ==> true
```

Listing 2.10: Differences between `contains` and the *Full-Text* `contains text`.

The *contains text* keyword[7] defines a *Full-Text contains expression* and may be used as a comparison expression in any XQuery or XPath query. The Full-Text extension also introduces a *score variable* in both `let` and `for` FLWOR clauses that express the relevance of the matched element to the search conditions.

2.3.1.1 Full-Text contains expression

The more relevant part of the extension is called *Full-Text contains expression*, and it is identified by the use of the `contains text` keyword, as shown in Listing 2.10. It may be composed of different parts, such as:

Connectives used to logically combine Full-Text selections by the connectives *and*, *or* and *not*;

[7]The XQuery Full-Text keyword is represented by the pair of the two words "`contains`" and "`text`"; the formal syntax is described in the following subsection.

Containment, Cardinality and Positional Conditions define how the set of tokens have to be searched and matched against textual content; to this purpose different search strategies can be defined.

Matching Options define the tokenization and matching behavior of the evaluation of the Full-Text expression, including stemming, stop-words removal, and language defined tokenization.

Weights allow to express, when more than one Full-Text selection is provided, the importance of each expression, thus affecting the final score computation. Weights can be defined as floating-point values in the interval $[0, 1000]$.

Ignore Options define the subset of elements that have to be omitted during the Full-Text evaluation: XPath expressions may be used to fine-tune the textual contents to be ignored.

More details about the XQuey Full-Text expressions syntax will be given in the subsequent section.

2.3.2 Syntax

In this section the main features of the Full-Text extension of both the XQuery 1.0 and he XPath 2.0 languages will be introduced with their syntax [8] as described in [59].

The grammar of a *Full-Text contains expression*, introduced in the previous section, is the following:

```
FTContainsExpr ::=  RangeExpr ( "contains" "text" FTSelection
    FTIgnoreOption? )?
```

A Full-Text contains expression returns a Boolean value: *true* is returned if some of the items in the `RangeExpr` match the selection specified in the `FTSelection` expression, *false* otherwise. The `RangeExpr` refers to the target element (direct text elements or descendant text elements) that has to be taken into account for the full-text search; the `FTIgnoreOption` selects, instead, the set of descendant elements that may be ignored. The `FTSelection` allows the specification of the actual full-text search where all the characteristics of the Full-Text extension come into play for the complete evaluation.

In the following the main options and expressions of the XQuery Full-Text extension are described; for sake of clarity they have been grouped as previously described in Section 2.3.1.

[8] More attention will be paid to the *Scoring* feature introduced by the extension, as the main focus of this thesis is leveraging such characteristic to provide flexible structural matching with score computation.

2.3.2.1 Connectives

The Full-Text selections may be combined with logical connectives to better express a full-text search; four connectives have been defined in the XQuery Full-Text language extension:

ftor the `ftor`, given two full-text selections, finds all matches that satisfy at least one of the selections.

ftand the `ftand`, given two full-text selections, finds all matches that simultaneously satisfy the selections.

ftnot the `ftnot`, placed before a full-text selection, finds all matches that do not satisfy the selection.

ftnotin the `ftnotin`, given two full-text selections, finds all matches that satisfy the first selection, but only when it is not a part of the second selection.

2.3.2.2 Containment, Cardinality and Positional conditions

This set of filters allows to specify how the token matching has to be performed against the textual contents. The XQuery Full-Text extension supports the expression of two different types of search: a first search allows users to specify a set of keywords, the second type of search is the *phrase* search where the user may specify an entire phrase (as a space separated list or words) that need to be matched against the textual contents.

Containment A *Containment* condition can define if all the given tokens (or set of phrases) must occur in the matched content or if the matching could be satisfied even if only some of the specified tokens or phrases are found in the textual content. The available options are: `any`, `all`, `any word`, `all words`, `phrase`.

Cardinality A *Cardinality* condition allows to specify the number of times a full-text expression must be satisfied in the evaluation; the grammar for the cardinality selection is the following `FTTimes` production:

```
FTTimes  ::=  "occurs" FTRange "times"
```

Where `FTRange` may assume one of the following values: "`at least` *N*", "`at most` *N*" or "`from` *N* `to` *M*".

Positional A *Positional* condition specifies the distance between each single token or phrase specified in the given set of full-text selections: (i) all selections have to match in the specified order as the query (`ordered`); (ii) it allows selections to appear within a specified boundaries (`window` of `words`, `phrases` or `paragraphs`); (iii) at a specified `distance` or (iv) within a given scope such as the `same paragraph` or `different sentence`.

2.3.2.3 Matching options

A *Matching Option* defines how the tokenization and the matching algorithm have to be handled; the XQuery Full-Text grammar production associated with the `MatchOptions` is the following:

```
FTMatchOptions ::= ("using" FTMatchOption)+
FTMatchOption  ::= FTLanguageOption| FTWildCardOption|
    FTThesaurusOption| FTStemOption| FTCaseOption| FTDiacriticsOption|
    FTStopWordOption| FTExtensionOption
```

Listing 2.11: XQuery Full-Text syntax for the Matching Options.

Where each `FTMatchOption` allows to specify a different aspect of the evaluation process, in particular:

FTLanguageOption specifies, by the keyword `language` followed by the standard language-code, the default language in which the full-text evaluation may be carried out: this option influences the tokenization, the stemming and the selection of a default set of stop-words/thesauri.

FTWildcardOption when the *wildchard* option is enabled (`using wildchard`), the string contents of a full-text selection are treated as a wildchard expression, thus containing a subset of a Regular Expression.

FTThesaurusOption The Thesaurus option specifies where the thesaurus is located, through a URI reference, or if the default one (if provided by the used engine) may be used.

FTStemOption If the stemming option is enabled (by specifying the option `using stemming`), the full-text expression is evaluated by applying a stemming procedure on the given set of tokens or phrases. By default the stemming is disabled (`using no stemming`).

FTCaseOption The Case Option modifies the matching of tokens and phrases by specifying how the uppercase and the lowercase characters are considered. The default value of this option is to ignore tokens and phrases capitalization (`using case insensitive`).

FTDiacriticsOption If the Diacritics option is enabled (`using diacritics sensitive`) tokens and phrases are matched only if they contain the diacritics as they are written in the query. By default this option is not enabled (`using diacritics insensitive`).

FTStopWordOption The keywords `stop words` allows the specification of the list of stop words either as a comma-separated list of strings or by a URI from which the stop-words are loaded.

FTExtensionOption This particular option allows to define extended options that will be recognized by some implementations and not by others. It allows to configure particular matching mechanisms or non-standard options offered by a particular XQuery Full-Text compliance engine.

2.3.3 The Relevance Score computation

Another important feature that the XQuery Full-Text extension adopts from the TexQuery language (besides the `contains text` expression and the set of matching options) is the introduction of a score computation during a *Full-Text contains expression* evaluation. A special variable, defined by the special keyword `score`, can be accessed after a full-text expression is evaluated, and its value represents the relevance estimate of the result to the full-text expression. The relevance score is computed for each element addressed by a Full-Text search, and then it is made available during the evaluation of the FLWOR expression. Both XQuery 1.0 `let` and `for` clauses have been extended to accept the new `score` variable definition. The grammar rules of the `for` and the `let` clauses, extended with the new variable, are presented in Listing 2.12.

```
ForClause      ::= "for" "$" VarName TypeDeclaration? PositionalVar?
    FTScoreVar? "in" ExprSingle ("," "$" VarName TypeDeclaration?
    PositionalVar? FTScoreVar? "in" ExprSingle)*
PositionalVar ::= "at" "$" VarName
FTScoreVar     ::= "score" "$" VarName
LetClause      ::= "let" (("$" VarName TypeDeclaration?) | FTScoreVar)
    ":=" ExprSingle ("," (("$" VarName TypeDeclaration?) | FTScoreVar)
    ":=" ExprSingle)*
```

Listing 2.12: XQuery syntax for the `for` and `let` clauses.

The score variable computation algorithm is not specified in the W3C standard, and it is left to the XQuery Full-Text engine to provide a meaningful relevance computation. The XQuery Full-Text standard only defines the following set of constraints that a score computation algorithm must adhere to:

- the score must be typed as an *xs:double*[9] value;
- the score values must be in the range $[0, 1]$;
- the score must represents the relevance degree of the expression evaluation, and the value must be directly proportional to the relevance estimate.

The score variable is computed once for each element addressed by the Full-Text expression, thus if multiple Full-Text clauses are defined in different parts of a path-expression, only the score associated with the *target element* would be returned. This represents an

[9] The `xs:double` value type corresponds to a Double precision float number as specified in the XQuery 1.0 XDM schema [33].

important behavior of the Full-Text score computation: to allow multiple full-text scores computation for nested fragments the query must be fragmented into multiple `let` or `for` clauses, thus defining different score variables.

An example is presented in Listing 2.13 where the XQuery expression includes two Full-Text clauses: the first related to the `name` node (in line 2), and a second clause related to the `description` node (in line 3). The FLOWR expression defines only one `score $score` variable for the entire path-expression. In such case the returned `$descr` items would be ordered by the relevance score computed for the `description` node and for the node labeled `name`.

```
1   for $descr score $score in
2     //object
3       [./name contains text "Magellan Cloud" using stemming]
4         /description[. contains text "refractor"]
5     order by $score descending
6     return $descr
```

Listing 2.13: XQuery FT with multiple Full-Text expressions and sorting by score.

As previously mentioned, the algorithm for computing the relevance score is not defined in the XQuery Full-Text specification: the algorithm is defined to be *implementation-dependent* and, thus, any of the numerous algorithms could be provided to an XQuery engine: from the standard *Cosine Similarity* approach defined in Information Retrieval by the Vector Space Model to more complex ones as proposed in the literature.

Most of the proposed scoring algorithms adopt a hybrid scoring technique: the computation of the relevance estimate is not entirely based on the textual content of an element, but some details about the underlying structure are also taken into account in such computation. This characteristic requires the scoring algorithm to know the structure of the query, expressed as an XQuery expression, to estimate the relevance.

Such feature, regarding the semantic aspects of the scoring of the XQuery Full-Text extension, poses an issue: the XQuery 1.0 formal semantics, specified in [69], defines the evaluation of axis expressions as functions that, given a sequence of items, return another sequence of items (called in XQuery a *Node-Sets*). For such reason, in any axis evaluation function it is not possible to keep track of which function (or sequence of functions) produced the actual *Node-Set* given as input, thus no axis evaluation function is aware of the complete expression structure that is being evaluated.

The semantics of the scoring functions defined by the XQuery Full-Text language make use of second-order functions [10] where the input argument of the algorithm is represented by the XQuery functions that implement the expression that produces the query result. This allows the scoring algorithm to be aware of the complete path expression that is being evaluated.

[10] Second order functions, or higher-order functions, are a set of functions that accepts functions as input arguments and may return functions as result.

The semantic second-order functions defined for the XQuery Full-Text in [59] are named `fts:scoreSequence(Expr)` and `fts:score(Expr)`: the argument `Expr` represents a XQuery expression. The two functions compute the relevance estimate score of the expression: a sequence of scores is returned by `fts:scoreSequence`, while a single score value is returned by the `fts:score` function.

As an example, if a `for` clause defines a `score` variable as:

```
for $item score $score in Expr
  return ...
```

the expression is semantically evaluated by assuming the following replacements: two new variables named `$i` and `$scoreSeq` are introduced and the the second-order `fts:scoreSequence` function is used.

```
  let $scoreSeq := fts:scoreSequence(Expr)
  for $item at $i in Expr
   let $score := $scoreSeq[$i]
   return ...
```

In the same way the second order function `fts:score` is used when a `let` clause containing a `score` variable is specified. As an example, the expression:

```
let score $score := Expr
```

is evaluated as if was written as:

```
let $score := fts:score(Expr)
```

The score variable, as described from the semantics given above, acts as a user defined variable in the scope of a FLWOR expression; thus it can be specified, for instance, in `where`, `order by` or `return` clauses.

As an example, in Listing 2.14, the XQuery expression extends the user query in Listing 2.13 by defining a score threshold of 0.5 (line 2) and by returning the matched description along with the computed score as an attribute (line 6).

```
1   for $descr score $score in
2     //object[/name contains text "Magellan Cloud" using stemming]/
        description[. contains text "refractor"]
3     order by $score descending
4     where $score > 0.5
5     return
6       <hit score="{$score}">{$descr/text()}<hit>
```

Listing 2.14: XQuery Full-Text with multiple full-text expressions, `where` and `sort-by` clauses.

3 Vagueness in querying the XML structure

Querying XML documents and their tree-like hierarchy with path expressions, like in the XPath and the XQuery languages, allows the specification of structural constraints that are matched against the document structure. As seen in Section 2.2 such constraints specification requires the user to have a complete knowledge of the underlying element nesting and node hierarchy of the XML document.

An inaccurate path expression provided to an XQuery or XPath engine may lead to the common search issue named “too few or too many results” [70] where a query evaluation produces too few or too many results. Other cases of queries that lead to no results at all are the ones that specify non-existing elements in the XML structure, or path expressions where some node relationships have been changed or deleted.

Structure matching in XML documents has been addressed by the Database community and the Information Retrieval one under two different perspectives: the DB community provided exact matching solutions focusing on efficient algorithms, aimed at retrieving relevant sets of fragments. The Information Retrieval community focused on providing a relevance estimate for XML fragments with an approximate constraints evaluation (the proposals in the literature mainly talk about *vague* evaluations).

In the following sections the main algorithms and techniques for structural constraints evaluation are discussed from both the Database and the Information Retrieval point of view. Due to the close relation with the flexible XQuery extension produced by the research work presented in this thesis, more attention has been paid to the approaches that tackle the structure-related constraints evaluation from an IR perspective, thus providing an approximate evaluation of structure-based constraints.

This chapter is structured as follows: the main characteristics of structural constraint evaluation and querying are introduced in Section 3.1, exact structural matching proposals and algorithms are briefly presented in Section 3.2. The Information Retrieval community proposals for querying XML documents with an approximate evaluation of structural constraints are described in Section 3.3.

3.1 Introduction

Since the definition of XML and its main query languages XQuery and XPath, the Database community addressed the issue of querying semi-structured documents as loosely structured databases, while the Information Retrieval community identified such documents as structured texts. The different interpretation of XML documents lead to quite different evaluation and querying proposals by the two communities.

From an Information Retrieval perspective querying structured documents with the specification of structural constraints allows users to express, as stated by Trotman in [71], more precise queries.

The aim of specifying a structural constraint in a user query that has to be run against a semi-structured document collection may be twofold: a first goal is to limit the size of the retrieved units by requiring to retrieve only a particular XML element or a limited set of nodes. As an example, for a collection of books, the user may be only interested to books titles instead of retrieving all the book information. A second purpose of structural constraints is, in contrast to classical keyword-based search, to specify in which context a set of keywords has to be searched: as an example a search of the keyword *1984*, if evaluated on the entire hierarchy of an XML document would produce a set of completely different results than if the search would be evaluated only on a specific context, such as in a `Price`, `Year` or even in a `Title` element. In conclusion, structural constraints allow users to provide more precise queries by both increasing the retrieval precision and by reducing the user effort to localize the exact piece of information that the user was looking for.

Regarding the structure of an XML document, both the Database and Information Retrieval communities model XML documents as ordered rooted labeled trees [33] where XML elements represent the nodes, and the tree edges explicit the relations between them.

A rooted labeled tree is denoted as $T = (V, E, r, \mu)$, where:

- $V = \{v_1, v_2, \cdots, v_n\}$ is the finite set of nodes (corresponding to the finite set of XML elements) of T;
- $E \subset V \times V$ is the set of edges between the tree nodes such that every node can be connected to another node by an edge ($\forall v \in V,\ \exists u \neq v \in V \mid \{v, u\} \in E$), and there are no cycles among edges;
- the element $r \in V$ is a special node called the *root* node;
- $\mu \colon V \to \Sigma$ is the labeling function that maps each node $v \in V$ to its label, where Σ is a finite labels alphabet.

The structural constraints evaluation in XML documents can, thus, be seen as a tree matching process, where both the XML document structure and a user query are handled by using tree graph theory.

In the following sections a state of the art is provided of the main techniques for evaluating queries with structural constraints on XML documents.

The approaches in the literature have been divided into two groups: those techniques aimed at an exact matching, and those defined for an approximate matching.

The first class of approaches is related to Database oriented evaluation strategies, where the query constraints of the nodes hierarchy are evaluated in a strict way. The second class of approaches refers, instead, to the Information Retrieval oriented evaluations, where the structural-based constraints are usually evaluated as hints for the XML fragment selection. It is important to outline that both classes of approaches work on the same syntactic definition of structural constraints that will be called *Standard Structural Constraints* (as defined in the XQuery and XPath languages). What is different is the the way these constraints are evaluated: in the former approach the results of a structural constraint evaluation is a set of fragments, while in the second case the result is a set of weighted fragments that can be ordered.

In the following the concept of *Tree Pattern Query*, or Tree Pattern, is used to indicate a user query involving one or more structural constraints. A Tree Pattern is a "*graphical representation that provides an easy and intuitive way of specifying the interesting part from an input data tree that must appear in query output.*" as stated in Hachicha et al. in [72]. A Tree Pattern (TP) p is formally defined in [72] as a pair $p = (t, F)$ where t is a labeled tree, as previously described, and F is a formula that specifies the constraints on t nodes. The process of matching a tree pattern against a data tree g is defined by a function f that maps nodes of p to nodes in g such that: (1) if two nodes n_1, n_2 in the tree pattern are related through a relation $n_1 \diamondsuit n_2$ (where the $\diamondsuit$ relation represents one of the defined axis of XPath, also listed in Table 2.1), then also their mapped nodes in g must be in the same relationship: $f(n_1) \diamondsuit f(n_2)$, and (2) the formula F of the tree pattern must be satisfied on the mapped nodes in g.

Tree Patterns can be seen as a reformulation of single XPath expressions for fragment selection in user queries [73]; such technique of rewriting XPath expressions to Tree Patterns has been used for query optimization in both XPath and XQuery languages [74].

3.2 Approaches to exact structural matching

Exact structural matching has been addressed by the Database community since the introduction of XML documents. The initial approach adopted by the DB community was to transform XML documents and their element hierarchy into database records, thus allowing querying XML documents by using the SQL [75] language.

Querying XML data with a strongly structured language, such as SQL, over a relational database provides an exact constraint matching by-design: path expressions are evaluated by matching node labels and their structural information, properly encoded in database tuples.

The SQL query engines have been adapted to accept XML documents as input documents to be queried by using the SQL language, some examples include Oracle8i [76, 77], Monet/XQuery [78], STORED [79], SQLGenerator with MySQL [80] or PostgreSQL [81].

Adopting a standard relational-database model for querying XML documents has been the first direction of the Database community to allow querying semi-structured data by leveraging a consolidated set of indexing strategies adopted in the DB field. The different nature of XML documents with respect to database tuples, including the varying structure among *homogeneous* documents and the XML nodes hierarchy, required database management systems to adapt XML documents to relational algebra [82, 83, 84], or to provide different indexing techniques to enhance path expression evaluation over table joins [85, 86, 87].

Different techniques have been proposed for storing and indexing XML documents using Relational Databases: from asking users to define the required XML elements to index, to exploiting the document structure from its DTD [84], and constructing a database ternary tuple for each element in the XML tree; or still to dynamically build a set of tables for each XML element and content such the one proposed in [82]. A further elaboration of the latter technique, proposed in [83], introduced six alternative methods to build a relational schema to perform SQL queries on XML documents by building data tables representing XML edges (axes), nodes, tags/labels and leaf-nodes.

By leveraging standard Relational Database Management Systems, such techniques do not require any modification of the underlying database engine, but they introduce costly table joins to evaluate path expressions. To overcome such complex evaluations, indexes and XML tree summarization techniques have been proposed to efficiently evaluate XPath expressions including values filtering and branching path expressions.

DataGuides [88] are the first proposal that couples the traditional index data structures of the database system with a succinct representation of the node hierarchy of an XML document: such structure summary is dynamically built from the XML document source and used for storing statistical values to enable query optimization over semi-structured data due to its compact representation of the entire document structure. However, handling huge and highly varying document structures with DataGuide may result in having huge DataGuides, thus reducing their efficacy; such issue was taken into account in [87] also for branching queries.

A second set of proposals introduced path-indexes that allow the query engine to perform costly path evaluations using ad-hoc data structures that maintain tag names and node-path relationships grouped for efficient structural matching. Such techniques includes the 1-/2-/T- index structure set [89]; and the $A(k)$ [90] and the $D(k)$ [86] index data structures aiming at reducing the complexity and the dimensions of the fined grained T-index structures.

Other works proposing techniques involving indexes and data structures have been defined to optimize path expressions evaluations, such as:

- string-based indexes representing the XML node hierarchy are divided into *raw paths* and *refined paths* to improve data access and to optimize structure-based queries in [85] by Cooper et al;
- efficient node identification schema coupled with extended DataGuides, proposed by Bremer et al. in [91], allow to include parent/child relationship in the node identification schema, thus avoiding costly structure lookups;
- positional and node identification schema to support XPath location step evaluation by identifying XML document regions by a *preorder* and *postorder* XML tree traversal as proposed by Grust [92] and integrated into the MontetDB/XQuery [78] engine.

Subsequent works in path expressions matching for semi-structured data also included methods for dealing with branching queries that require joins and merging between intermediate results. The concept of *structural-joins* proposed by Alkhalifa et al. [93] emerged as a first solution to provide an efficient evaluation of branching queries (also called twig queries). Enhancements and variations of such approach ended up to different techniques involving node joins and tree traversal optimization such as: StairCase Joins [94], extended Binary trees [95], XML region encoding (XR-tree index) [96], Extended Dewery encoding [97] and others.

Furthermore, not only compressed indexes and efficient data structures have been defined by the database community, but also efficient algorithms for processing XPath and XQuery queries have been elaborated by Chen et al. [74], Gou et al. [98] and Koch [99].

An important contribution has been provided in [100, 9], where Gottlob et al., after evaluating several XPath engines, proposed an efficient query processing algorithm. The work comprised the *Core XPath*: a first formal definition of the semantics of XPath conforming to the W3C standardized language; it included the navigational path processing and logical features of the language; the semantics only omits values manipulation such as string and arithmetical functions computation. The formal definition provided also a first evaluation of the language complexity, and it has been used for subsequent efficient algorithms definition with *top-down* and *bottom-up* path expression evaluation. Gottlob work was inspired by Wadler works in [101] and [102], where the semantics of XPath and XSLT are given. Further research of Gottlob, Koch and Pichler in [103] provided a deeper analysis of the XPath language from a theoretic perspective: both the query and the validation complexity of XPath were addressed and defined.

The above researches in the language theory and algorithms have been coupled by researches aiming at the definition of tree algebras for querying XML documents. Such works include the algebra defined by Fernadez et al. in [104] that anticipates the subsequently defined XQuery language; the *TreeAlgebra for XML* (TAX) defined by Jagadish et al. in [105], and the tree algebra defined by Paparizos et al. in [106].

With the increasing research on efficiently storing and querying XML documents and the complexity introduced by the solutions that maps XML to SQL, more query engines that natively handled XML documents arose since 2002. Some examples of such systems are Timber XML [107] from the university of Michigan, eXist [108] by Meier et al., BaseX [109] developed by the university of Kostanz.

For a complete overview of the techniques proposed for indexing, storing and querying XML documents and semi-structured data a first survey was initially provided by Luk et al.[110], where several indexing paradigms are presented; the work also covers search models and results presentation for the shift from database to XML querying. Krishnamurthy et al. [111] provide a comparative research over the set of engines and approaches leveraging the SQL language for querying XML documents.

Gou et al. [98], instead, analyses both the relational-based and the native XML processing techniques for path expression evaluation with particular interest to branching queries, while Hachicha et al. in [72] focus on Tree Pattern evaluation from a native XML storing and indexing prospective, complementing and updating Gou et al. work.

A third work by Haw et al. [112] focuses on trends of querying semi-structured data: hybrid approaches that leverage both relational database and ad-hoc indexing strategies are taken into account and their advantages and disadvantages described.

3.3 Approximate structural matching

In this section the evaluation of path expressions in an Information Retrieval perspective is considered, and the techniques proposed for defining an approximate structural matching are analyzed. Only the IR approaches defined as *path-based* and *clause-based* by Lalmas in [113, chapter 4.2] are discussed here: such approaches include the XPath and XQuery based proposals that leverage the standard W3C languages for approximate structural constraints evaluation. Furthermore the described proposals, including the NEXI language introduced in Section 2.2.3, represent the state-of-the-art proposals in the IR community for approximate evaluation of structural constraints, and they constitute the contributions more strictly related to the research work reported in this thesis.

From an Information Retrieval perspective, and as stated by in [114], the retrieval process of XML document fragments can exploit the hierarchy of structural information in three main areas: the *Indexing*, *Query Processing* and *Retrieval* steps. The first area covers the indexing process of an XML search engine, i.e. the selection of structural elements to handle. Query evaluation performances depend on the amount of indexed elements. The *Query Processing* area represents the actual structural matching process executed during the path expressions evaluation: the structural matching may be performed in a strict way as described in Section 3.2, or by adopting relaxation techniques further described in this section. The *Retrieval* area is related to provide the final results to the user; techniques for presenting only the user required XML element, or for providing the surrounding context have been proposed for text-oriented XML search engines.

Related to the *Query Processing* step for the evaluation of user defined structural constraints, in this section we focused on the various approaches that have been defined to consider the path expressions not as strict constraints, but as hints for structural retrieval.

In the following Section 3.3.1 an introduction to the approaches proposed by the Information Retrieval community aimed at inquiring XML documents are shortly presented, while Section 3.3.2 deals with the approaches that have been defined to provide an approximate structural matching on XML documents by either defining new query languages or by defining an approximate matching of standard XQuery structural axes.

As it will be outlined, any approximate approach described in Section 3.3.2 allows users to explicitly define which part of a query should be evaluated in an approximate way, and which, instead, should be evaluated exactly as formulated. This aspect, besides not providing a *vague* constraint formulation, also limits the ability to obtain a query evaluation that exactly suits the users requirements.

Section 3.3.3 describes then the approaches aimed at introducing a user defined vagueness in query formulation coupled with an approximate evaluation; such works mainly motivated the research presented in this thesis.

3.3.1 Information Retrieval approaches to query XML documents

The proposals in the literature that have been generated in the Information Retrieval context for querying structured XML documents have been classified as *content-only* CO and *content-and-structure* (CAS) search [10]. The CO/CAS classification has been proposed in the context of the INitiative for the Evaluation of XML Retrieval (INEX) [115], a community driven initiative for evaluating XML search engines; as the Text REtrieval Conference (TREC)[1] is aimed at evaluating standard search engines.

The approaches labeled as CO allow only keyword-based queries without any possibility, for the user, to specify constraints on the document structure: neither to formulate a more detailed query, nor to specify the required result granularity or structure to be retrieved. The query evaluation is performed in an IR style, thus applying stop-word removal, stemming and other classical information retrieval techniques [70], and a ranked list of the retrieved XML elements is produced. Most CO approaches return query results based on the notion of *Lowest Common Ancestor* [2] [118] (LCA): given a rooted tree T and the set of its nodes V, the Lowest Common Ancestor node for a couple of nodes $u, v \in V$ is the node $a \in V$, chosen from all the common ancestors of u and v, that is located farthest from the root element of the tree. CO approaches also adopted LCA variants such as the one presented in XSearch [119] and in Li et al. [120] work.

CAS approaches, instead, were defined to allow the formulation of constraints on both the document content and structure [121]: a first attempt to merge the IR and the DB search paradigms was constituted by CAS approaches that were defined based on the XPath language syntax for the path expression definition.

In [122] an analysis of different evaluation techniques for CAS queries is provided: CAS queries are further classified by the techniques used to match the specified structural constraints when branching queries are submitted: the classification takes into account the difference between a target node (the XML element that the system should return) and the elements that compose one of more selection conditions. The INEX initiative allows systems to deal with the specified structural constraints by two distinct evaluation strategies: *Strict* and *Vague* (the terminology *vague* adopted by the INEX initiative corresponds, to an *approximate* constraints evaluation strategy as previously outlined). Thus, a total of four different evaluation strategies for CAS queries[3] are:

ssCAS strict-strict: both the target element and the elements in the predicate part must be evaluated as a strict requirement;

[1] The Text REtrieval Conference (TREC) documents, documentation and tests are available at `http://trec.nist.gov` website.

[2] Some works named the problem of finding the Lowest Common Ancestor node also as the *Least Common Ancestor* [116] problem, both abbreviated as LCA. A third name *Nearest Common Ancestor* [117] is also used in literature, always referring to the same problem and set of algorithms.

[3] An example query is the following: `books/book[.//paragraph[contains(., "1984")]]/abstract`, where `abstract` is the *target* element, and the predicate `[.//paragraph[contains(., "1984")]]` makes use of the NEXI `contains()` function.

svCAS strict-vague: only the target element must be exactly evaluated, elements in the predicates could be approximately evaluated;

vsCAS vague-strict: opposite to the previous cases, only the target element should be evaluated with a vague matching;

vvCAS vague-vague: both the target element and the elements in the predicates conditions may be evaluated with approximate matching.

The NEXI [28] language represents the first research effort in this direction: its subset of XPath language constructs, in addition to the `contains` constraint on element contents, has represented the language adopted by INEX in 2002; more details about the language have been already presented in Section 2.2.3. These approaches are based, differently from the exact matching provided from DB proposals, on the notion of *structural hint* which considers the query structure as a mere template of the information required by the user; all the fragment *similar* to the template specified in the query are retrieved. Examples of CAS approaches includes TopX [123], TeXQuery [68], FleXPath [8], ELIXIR [65], JuruXML [7], XIRQL [67] and NEXI [28].

Most CAS approaches, like TopX [123] and ELIXIR [65], do not consider the content-related and the structure-related constraints equally important; in fact, they employ a two stage evaluation strategy by which the evaluation of the content predicates is first performed (as done also for Content-Only approaches), and then the obtained results are analyzed with respect to the structural constraints. The analysis of the preliminary results obtained from the CO evaluation, includes a filtering process that removes XML elements from the final result set depending on the structural constraints satisfaction.

In the following section the approaches in the literature that fully take into account the structure requirements expressed in a user query are presented: such works adopt edit distance techniques to compute a similarity between the document nodes hierarchy and the path constraints in the query. Such similarity is handled by using tree matching algorithms and techniques that, by using appropriate transformations, provide approximate structural matching of XML fragments.

3.3.2 Vagueness in structural matching evaluation

A first set of contributions on path expression relaxation are those of Schlieder [124, 6] and Amer-Yahia [5]; Schlieder initially extended the XQL [37] structured query language by an approximate structural matching in the *approXQL* language. The proposed algorithm for tree matching works by first expanding the user query by providing a similar set of queries where node deletion, node renaming and node insertion operations replaced the original query constraints; finally all the transformed queries are simultaneously evaluated. Node renaming operations are executed over a list of labels provided in advance by the user, but without any knowledge of the actual data stored in the XML document. For each query transformation a cost based evaluation is performed; a score is computed by taking into account the similarity between the query and the document to produce a ranked list of XML elements.

JuruXML [7] adopts an extended Vector Space Model [125] to evaluate structural similarity between a document and a user query expressed using a sample document fragment. A fragment score is computed using a double weighting schema: a term and its context (the XML node where the term appears) are combined by an adaptation of the Cosine Similarity [70] formula.

The work of Amer-Yahia et al. in [5] presents a preliminary study of approximation on XML query matching: weighted tree patterns are evaluated against the XML document structure, two values are associated with each node and each edge and used to compute the final scores for query answers. Different techniques of relaxation are introduced: (1) node generalization, (2) leaf node deletion, (3) edge generalization and (4) sub-tree promotion. These relaxation rules allow to exchange a node with its super-type value (or its parent node), removing constraints on leaf nodes or the leaf nodes themselves, altering axes from *child-of* to the more general *descendant* or changing query sub-trees (or branching queries root nodes) context nodes respectively.

FleXPath [8] is the first approach proposing an approximate matching of query constraints by a formalization of the relaxations in the evaluation of the structure of the specified queries; it constitutes the first algebraic framework for spanning the relaxations space. The approach also defines three simple ranking schemes for the retrieved XML fragments: plain keyword-search, structure similarity of the fragment to the query and the third scheme is a simple sum of the two scores to combine structure and content similarity.

The FlexPath approach has been further developed in [126] where the scoring computation is inspired by the classical $tf \cdot idf$ measure proposed in Information Retrieval [70]. The computation takes into account both content and structural scoring, while adopting query relaxation. Two scoring techniques are proposed in [126]: *twig scoring* based on $tf \cdot idf$ and its approximation computation named *path scoring*. Amer-Yahia et al. also compares the two approaches with a *binary scoring* earlier proposed in [127].

Another CAS approach that partially takes into account the structural matching is the one presented by Sauvagnat et al. in [128]; where the score computation of a retrieved XML fragment takes into account the distance of the fragment (as the number of tree edges) from the leaf node that actually contains the matched keyword search. Some penalization strategies have also been defined that allow to assign a lower score to nodes that do not directly contain a matched text, but that inherit content-scores from leaf nodes; the aim of such penalization strategy, and its subsequent refinements, is to handle branching queries and to propagate scores computed at leaf nodes level back to the target node element.

The ApproXML tool [129], proposed by Damiani et al., performs a flexible matching between a document and the standard structural constraints formulated in a query: the evaluation is computed by creating a closure of the document graph by inserting a so called *virtual* edge between document nodes that rely on the same path. During the evaluation of the structural constraints in the user query, the nodes hierarchy is evaluated with an exact approach against the *extended* document tree structure; different weights are then assigned if a document fragment is retrieved by using the virtual edges or by only using the original document edges.

A further work on adopting tree-edit distances to compute structural scores has been proposed by Nierman et al. in [130], where approximate matching and path expressions relaxations (like the ones defined by Amer-Yahia et al. in [5]) are adopted for heterogeneous XML document collections: the approach performs a partitioning of a collection of documents based on their structural similarity computed on the given document DTD. A *minimum edit distance cost* is computed between two trees (representing for example an XML document and the structure-based set of constraints in an user query) by taking into account a set of allowed sequence of tree edit operations. Such cost is then used as the score for a subsequent XML fragment matching, pruning and ranking operations.

Another approach that uses a variant of the tree-edit distance computation is presented by Laitang et al. in [131], where approximate answers are built in advance based on the DTD of a XML document collection. Approximate answers are used to prune the set of XML element candidates and subsequently matched against the the real set of XML document; a final element score is computed by combining the tree-edit distance cost and the content-based score.

Similarly to [130], the VXQL language proposed by Fazzinga et al. in [132] deals with querying heterogeneous XML document collections spread across Peer-to-peer networks: tree edit distances like [5] and [124] are further integrated into the XPath language where node renaming operations are executed by taking into consideration the *semantic distance* between the original and the replaced node label. A further refinement of the approach, by the same authors, is presented in [133] where the XPath axes are evaluated through a set of relaxation and generalization rules.

The approach defined by Buratti et al. in [134] adopts the *path-edit* distances and includes an approximate value-based constraints evaluation, such approach is defined on top of the standard XPath language. A *Path Edit Distance* (PED) algorithm is defined to compute structural scores and to approximately evaluate path expressions; the algorithm includes the path approximations defined in [126] and it implements a node similarity measure like the classical String Edit Distance [135] defined by Levenshtein.

Another technique to provide an approximate evaluation of structural constraints is presented in [136] by Tekli et al.; the approach, disregarding the element and attributes values constraints, defines a structural similarity between two XML elements by taking int account sub-tree similarities, an aspect not handled by most tree-edit or path-edit distance algorithms. The approach also includes the evaluation of semantic similarity of elements: the WordNet ontology is leveraged to compute a *semantic distance* between tag and attribute labels; such distance is then used during the approximate evaluation of structural constraints.

Another contribution that could be labeled as a CAS approach is the one presented by Mazuran et al. in [137]; the proposed approach adopts an approximate structural matching, coupled with the generation of collection summaries (or *gists*) which provide approximate query answers to better support the user in further query specification.

It should be noted that all previous approaches introduce flexibility in the evaluation process of the standard path expressions and axes (such as `child` or `descendant` XQuery axes;) therefore these approaches do not allow users to specify structural constraints that explicitly require the application of an approximate structural matching, which would provide fragment scores distinct from the scores produced by the keyword-based evaluations. This means that the user has no way of distinguishing between structural constraints in the query the evaluation of which has to produce a set of fragments, and flexible structural constraints the evaluation of which has to produce weighted fragments, with structural fragment scores distinct from content related fragment scores usually computed by CAS approaches.

3.3.3 User-based vagueness evaluations

Differently from the approaches described in Section 3.3.1 and 3.3.2 which address the problem of defining an approximate matching of conventional and standard structural constraints (such as the ones defined in the XPath and XQuery languages), only a few contributions in the literature have considered the problem of introducing in the XML query languages new flexible structural constraints.

In 2010 the work performed by Oro et al. called SXPath [138] aimed at allowing users to formulate queries on semi-structured HTML documents by the specification of constraints on *positional* axes. The approach adopts the four cardinal axes and their combination as constraints for retrieving XML elements given their visual position in the HTML document. The proposal, unfortunately, does not tackle an important issue related to visual querying: given the nature of HTML documents the approach does not consider the W3C standard CSS[4] styles for document visual presentation and element positioning with respect to different devices and browsers.

In [139], Zhang et al. propose the definition of a symmetrical evaluation of XPath structural constraints by the introduction of a new *closest* axis. The proposed approach allows the user to express a vague structural constraint by providing a structure invariant query; the work does not, however, provide a score computation for the relaxed path evaluation. Bhowmick et al. in [140] propose a similar approach that closely resembles Zhang's work, the only difference is that the defined `rank-distance` axis is evaluated *on-the-fly* and not pre-computed and stored in an ad-hoc data structure. Furthermore the approaches work in Relational Database management systems, thus rewriting the user queries and their structural constraints from XPath to the SQL language.

[4] Cascading Style-Sheet [17] represents a set of rules for visually present HTML documents: typographic and layout positioning of document elements can be defined.

A similar and extended approach has been proposed by Damiani et al. in [141]: the work presents a preliminary definition of two set of flexible constraints: the first related to the hierarchy of the XML elements and the second related to the document textual contents. While the content-related set of constraints has been partially superseded by the W3C XQuery Full-Text language definition, the structural-based set of constraints has not been further taken into account, neither updated nor formally defined as a complete language extension. On the approach defined in [141] the work presented in this thesis has taken its origin: with the aim of formally and semantically define the structural-based flexible constraints the FleXy language was born. The FleXy language, formally defined during the research work undertaken during my PhD period, has been designed as an extension of the standard XQuery language. The FleXy language is formally described, along with its syntax and semantics, in Chapter 4, while its implementation is presented in Chapter 5.

3.4 Summary

In this Chapter the main issues related to querying XML documents have been presented and discussed: efficiency related algorithms and approximate structural matching techniques have been analyzed as they have been proposed from a Database oriented approach at first, and then from an Information Retrieval point of view.

In Section 3.2, Database community approaches including indexing data structures, efficient tree-pattern-matching and relational-database oriented techniques for querying XML documents have been presented.

Section 3.3, instead, has tackled the issue of querying XML documents from an Information Retrieval point of view, where query evaluation efficiency comes slightly in second order with respect to the ability of retrieving the most relevant document fragments given a user query, handled as a template for the retrieval engine.

The proposed approaches from the Information Retrieval community are mostly focused at providing approximate matching of the standard structural constraints formulated in a user query. Tree-Edit or Path-Edit distances, tree- and sub-tree similarity computation, and XML element label semantic comparison have been proposed in the literature to provide a fragment relevance computation during the XML retrieval evaluation.

Most of the recent IR oriented approaches propose approximate structural constraints evaluation on top of the standard XQuery and XPath query languages, where the exact user query is evaluated by the use of constraint relaxation techniques. Few approaches, described in Section 3.3.3, take into account the possibility to allow users to explicitly formulate a query where vague constraints can be expressed and evaluated. Such vagueness, as outlined in Section 3.3.1 and 3.3.3, would provide users the ability to specify the desired evaluation that has to be performed for structural constraints, and thus, it would avoid the engine to consider the user query as a mere template of a retrieved XML fragment.

One of the most important contributions to vague constraints specification has been envisioned by Damiani et al. in [141]; based on such approach the FleXy language (acronym for ***Fle****xible* ***X****Quer****y***) has been formally defined during the research work described in this thesis. The FleXy language, described in Chapter 4, extends the standard XQuery Full-Text language by adding two new vague structure-based constraints that allow users to express the desired approximation degree that has to be considered during the query evaluation process.

4 FleXy: The XQuery Full-Text extension

In this chapter the main contributions of the research activities reported in this thesis are presented. As initially mentioned in Chapter 3, the work presented in this thesis has been originated from the work by Damiani et al. in [141] (described in Section 3.3.3), where a preliminary definition of flexible constraints was proposed.

The research work reported in this thesis was mainly aimed at formally defining and implementing a formal XQuery extension called FleXy (***Fle****xible* ***X****Quer****y***) that adds flexibility to the XQuery Full-Text language.

By including two new axes constraints, FleXy allows a fine grained query formulation by users. Differently from the approximate approaches defined in Section 3.3.1, FleXy allows users to explicitly express the structural constraints that have to be evaluated in an approximate way (by using the new flexible axes), and those that have to be evaluated in an exact way (by using the standard XQuery expressions). The evaluation of the new flexible axes produces a score that can be combined with the XQuery Full-Text content-based constraints evaluation score; this combination may be directly specified by the user.

The FleXy language has been defined as fully compatible with the standard XQuery Full-Text language; this gives to the new language three important qualities: a first peculiarity is that a query specified by using the standard XQuery language will not return different results from the standard implementation. Second, the FleXy language benefits from the standard XQuery Full-Text constructs and infrastructure for XML element selection and Full-Text search, in particular users can specify full-text and structural score variables as in standard XQuery FLWOR clauses. A third characteristic is that the FleXy language does not require the formulation of complex constraints: the syntax of the new axes has been kept simple and similar, where possible, to the standard XQuery axes.

These characteristics allow the FleXy language to be implemented in any XQuery Full-Text compliant engine: as it will be shown in Chapter 5, a particular attention was paid to the choice of the underlying data structures for an efficient query and axes evaluation.

The FleXy language syntax and semantics have been published in [142] as an intermediate result of the work undertaken in this thesis. Differently from the definitions provided in [142], the work here presented includes further refinements that take into account the comments received during recent conferences participation and demonstrations.

In the following Section 4.1 the motivations and the flexible axes defined in the FleXy language are discussed, while Section 4.2 introduces the two flexible constraints.

Section 4.3 describes the syntax of both axes, and how the structural scores can be accessed. Section 4.4 deals with the definition of the axes semantics, and it discusses the nesting and score inheritance of the introduced scores computation. Finally, in Section 4.5 the evaluation of the axes and the structural score computation are presented.

4.1 Motivations

Querying highly structured databases or document repositories via structured query models (as XQuery and XPath) forces the users to be well aware of the underlying structure, which is not trivial. In the above cases, users could benefit of a query language that allows a direct specification of flexible structural constraints that easily allow to require the relative position of important nodes, independently of an exact knowledge of the underlying structures. Such an issue is also outlined by Yu et al. in [143] and Bhowmick et al. [140] works: both aimed at supporting user queries over complex semi-structured document collections.

To achieve this aim and thus to support users in querying structured document collections without a strong knowledge of their underlying structure, a formal extension of the XQuery Full-Text language is presented. The extension, named FleXy, features the introduction of two new flexible structural axes, specified by the axes `below` and `near`, that allow users to explicitly specify their tolerance to an approximate structural matching, while not forcing them to be aware of all the possible structural variations of the data/document structure.

The FleXy language defines also an ad-hoc approximate matching of the flexible structural constraints, thus allowing both a ranking only based on approximate structure matching, and a ranking based on a combination of content predicates and the new flexible structural predicates (while preserving a ranking based only on content predicates).

The work done in this thesis bases its grounds on a previous research where a flexible extension of the XQuery language was advocated and informally sketched [141, 144]. The work presented in this chapter, as introduced in Section 1, consists of the formal definition of the FleXy syntax and semantics of the XQuery Full-Text extension.

Differently from all the approaches defined in previous works, including the ones presented in Section 3, this extension gives to the user the ability to express flexible structural constraints with an approximate matching, and to obtain a weighted set of fragments as the result of the axes evaluation. For a given query the user can also specify how to combine the scores produced by the structural constraints evaluation and the keyword-based evaluation (as provided by the XQuery Full-Text extension). The combination can be formulated by using the set of standard arithmetic operators defined in the XQuery language.

The FleXy extension allows the users to exploit their, even limited, structural knowledge of the XML document collection structure to formulate successfully queries. The use of flexible axes such as `near` and `below` and their structural relevance scores, allow the user to obtain, in conjunction with the Full-Text features and search functionality:

1. element matching and ranking based on content predicates evaluation only as in the original XQuery Full-Text with the usage of the Full-Text score variable;
2. an element ranking based on the flexible structural constraints evaluation (based on the FleXy axes);
3. an element ranking based on a linear combination of the two above scores, which the user may also specify via the `order-by` FLWOR clause.

Furthermore, the language FleXy is defined as fully-compliant with the XQuery Full-Text extension: the two axes are introduced in the XQuery language among standard axes and path constraints described in Section 2.2.1 and 2.2.2.

4.2 FleXy flexible axes

The main novelty of the FleXy language is the definition of two new axes in the XQuery axis expressions, named `below` and `near`: the constraint `below` is defined as an XPath axis and its evaluation allow to match elements (called *target nodes*) that are direct descendants of a node (called the *context node*). Although the `below` axis evaluation selects the same node set identified by the `descendant` axis evaluation, differently from the `descendant` axis evaluation the `below` constraint evaluation computes a numeric score for each retrieved target-node.

The `near` axis evaluation allows to identify XML nodes (*target nodes*) connected through *any path* to the *context node.* A numeric value can be specified in association with the `near` axis: it defines the maximum number of arcs between the context node and the target node to be taken into account during the axis evaluation: nodes the distance of which is greater than n arcs are filtered out from the possible results.

As mentioned in Section 4.1, a score in the interval $[0, 1]$ is assigned to each XML node identified by the `below` and the `near` axes evaluation: this score is proportional to the closeness of the target node to the context node in the query expression, which represent the closeness of the *real-path* to the *ideal-path* corresponding to the query expression.

It is important to notice that the integration of the `below` and the `near` axes in the XQuery Full-Text syntax allows to specify them in *any* XQuery predicate, as it will be explained in Section 4.3. Furthermore the flexible axes can also be used in conjunction with positional predicates: as the matched elements are returned in decreasing order of their estimated relevance, the positional predicates are referred to the rank of the fragment sequence.

In the following sections some examples and the syntax specification of the `below` and the `near` axes will be provided by using the *unabbreviated* axes nomenclature, where each XQuery axis is explicitly specified without using any abbreviated form. For example, the *descendant* axis applied to the node-label *t* will be written as "`/descendant::`*t*" and not using the abbreviated form "`//`*t*".

4.2.1 The "`below`" axis

The `below` axis is defined as an extension of the XQuery `descendant` axis, and it is aimed at requiring an approximate matching of the query with the descendants Element Nodes of the context-node. The approximate matching of the query fragment containing the `below` axis with a retrieved element associates with the latter a *relevance score*. For a retrieved element, the relevance score is inversely proportional to the distance between the element and the *ideal-path* expressed by the query: such an ideal path is the one where the target node is a direct child of the context node.

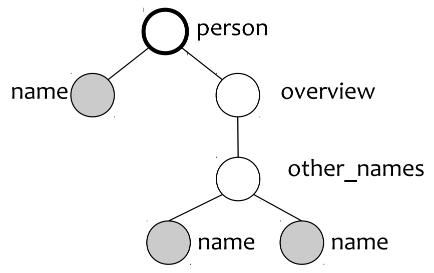
Like any standard XQuery/XPath axis (the standard axes are listed in Table 2.1, the `below` axis can be specified in the form `/below::`*t*, where *t* is a node label; this new axis, due to its descending direction evaluation, is grouped in the set of *forward* axes.

In Fig. 4.1 a simple example of the `below` axis evaluation is shown for the query `person/below::name`; in the figure the node `person` (with a bold border) is the context node, while the nodes with gray filling are the identified target elements.

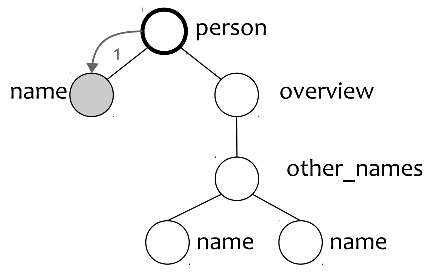
As previously mentioned, the `below` axis evaluation returns the same set of nodes returned by the standard `descendant` axis evaluation, as it can be seen in Fig. 4.1a. In Fig. 4.1b the *ideal path* identified by the `below` axis is shown: the highlighted `name` node, having a direct descendant relationship with the context node, will obtain the highest path relevance score (perfect matching). In Fig. 4.1c the evaluation process performed to match the other `name` nodes is shown. Finally in Fig. 4.1d the complete evaluation process of the `below` axis is shown.

Based on the example in Fig. 4.1 the elements retrieved in decreasing order of relevance estimate, are:

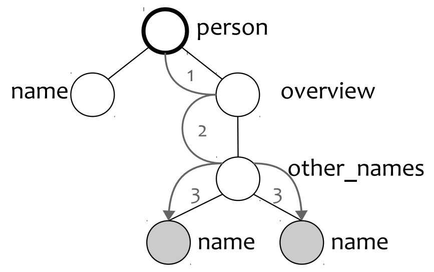
1. the node `person/name` that represents the ideal path (in this path the distance between the target node `name` and the the context node `person` is one, i.e. `name` is a child node of `person`);
2. the two nodes having the path `person/overview/other_names/name`; their lower score is due to their higher distance (3 arcs) from the `person` context node.



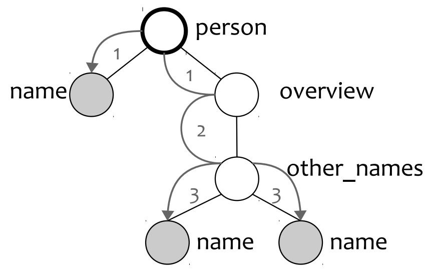
(a) Nodes matched by the `below` axis



(b) Ideal path for the `below` axis evaluation



(c) Deeper nodes matched by the `below` axis evaluation excluding the *ideal path*



(d) The complete set of nodes matched by the `below` axis evaluation

Figure 4.1: Examples of the `below` axis evaluation

A numeric threshold n has been introduced in the `below` axis definition; this allow users to better express how the axis evaluation has to be performed: the threshold limits the maximum distance at which the target nodes must lay. The threshold specification can be provided alongside the axis itself, appending to the axis name the numeric value, thus resulting in multiple axes of the form: `/below`n`::`t, where $n \in \mathbb{N}$ and $\mathbb{N} = \{1, 2, \cdots\}$.

A particular configuration of the `below` axis is provided when the threshold is set to the value 1: in this case the axis evaluation would match only children nodes of the current context node, so that the scores of the retrieved elements would always be evaluated to 1. For the complete axis syntax refer to Section 4.3.1, while the `below` axis relevance score computation will be presented in Section 4.5.

Like other XPath axes, also the `below` axis supports the definition of a relative inverted axis, referred as `below`$^{-1}$. For sake of simplicity the reverted axis is named `above` and it supports both the plain and the limited (with the threshold parameter) evaluation. Also the evaluation of the reverted axis `above::`t, and the one of its threshold variant `above`n`::`t, compute a *path relevance score* for each of the retrieved target nodes.

An important observation is that, although the XQuery standard expressions and functions may allow to formulate complex queries with a behavior similar to the one associated with the `below` axis, the use of explicit flexible constraints is clearly more user-oriented and better complies and integrates with the XQuery Full-Text scoring mechanism.

Furthermore, the definition of a set of XQuery functions aimed at providing the same element matching and the same structural score computation as defined for the `below` axis would introduce a non-trivial overhead to the query evaluation process. In particular such functions would require the evaluation of a set of sub-queries and other computations that would not benefit from a native axis implementation that directly accesses to the engine data structures.

4.2.2 The "`near`" axis

The second flexible axis introduced by the FleXy language extension is the `near` axis, which allows to retrieve target nodes that are "*in the neighborhoods*" of the context node, in all directions including the descendant, siblings and ancestors axes.

A threshold parameter can be associated with the `near` constraint to indicate the maximum distance between the context node and the target node: nodes reachable with more than n arcs from the context node will be excluded from the retrieved elements. The parameter allows users to control the `near` evaluation by avoiding to search in the whole XML graph for matching target nodes.

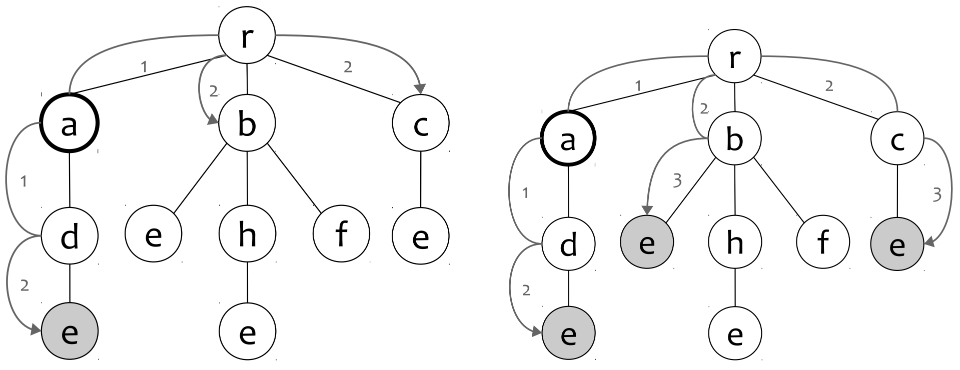
The axis can be specified in the form: `/near`***n***`::t`, where ***t*** is a node label and ***n*** the threshold parameter. Like for the `below` axis, the parameter ***n*** can be specified as a positive natural number. If the threshold parameter is not set, a default value of 1 is assumed.

Like the previously described `below` axis evaluation, also the `near` constraint evaluation computes a relevance score for each matching node. In this case the *ideal paths* for the `near` axis evaluation are considered those node directly connected to the current context node (the immediate father and the immediate children of the context node).

In Fig. 4.2 two examples of queries that specify the `near` axis are shown. In Fig. 4.2a the evaluation of the query `a/near2::e` is shown: the node labeled `a` with bold border is the *context* node, while the `e` nodes with the filled background are the nodes matched by the example query. Note that in this case, only the node labeled `e` with path `r/a/d/e` will be retrieved because all the other nodes labeled `e` have a distance from the context node greater than 2 arcs.

In the same way in Fig. 4.2b the evaluation of the query `a/near3::e` matches the three nodes labeled `e`, while the node having path `r/b/h/e` is not matched due to its distance of 4 arcs from the context node `a`. To summarize the example in Fig. 4.2b the nodes matched by the query `a/near3::e`, in decreasing order of relevance estimate, are:

1. node `r/a/d/e`
2. node `r/b/e` and node `r/c/e`.



(a) Example of the `a/near2::e` query evaluation (b) Evaluation Example of the `a/near3::e` query evaluation

Figure 4.2: Examples of the `near` axis evaluation

The actual score computation function for the `near` and `near`*l* axes evaluation is described in Section4.5.

Differently from the `below` and the `below`*l* axes, the `near` axis specification does not support the definition of a reversed path axis: due to the mixed matching strategy used in the `near` axis where both *upward* and *downward* edge traversal strategies are used, such axis inversion can not be defined.

4.3 The FleXy Syntax

The FleXy language extension has been defined to be fully compliant with the latest XQuery Full-Text language clauses; furthermore as described in Section 2.2.2, the XQuery 1.0 language leverages the XPath 2.0 specification for nodes selection, thus the FleXy extension supports both languages. FleXy introduces in fact the two new axes `below` and `near`, and it includes a new structural score variable to let users define a customized element ranking and sorting, based on the evaluation of the two flexible axes, as it will be explained in Section 4.3.2.

4.3.1 Axes Syntax

Following the XQuery syntax definition provided in [3] and the Core-XPath grammar defined in [9], the Extended Backus–Naur Form (EBNF) representation of the FleXy language extension is specified in Listing 4.1.

```
 AxisStep    ::= (ReverseStep | ForwardStep | NearStep ) PredicateList
 ForwardStep ::= (ForwardAxis NodeTest) | AbbrevForwardStep

```

```
4    NearStep    ::= AxisNear NodeTest
5    AxisNear    ::= "near" intNumber? "::"
6
7    ForwardAxis ::= ("child" "::")
8                  | ("descendant" "::")
9                  | ("attribute" "::")
10                 | ("self" "::")
11                 | ("descendant-or-self" "::")
12                 | ("following-sibling" "::")
13                 | ("following" "::")
14                 | ("namespace" "::")
15                 | AxisBelow
16
17   AxisBelow   ::= "below" intNumber? "::"
```

Listing 4.1: FleXy `below` and `near` axes syntax.

In the previous grammar, the axes threshold values are defined by the grammar production named `intNumber`, and they are evaluated as XQuery atomic values of *xs:integer* type.

In Listing 4.1 the salient modifications to the standard XQuery language are related to the `below` and `near` axes definition; in particular:

line 1 the `AxisStep` is extended with the `NearStep` production, since the behavior of the `near` axis may follow both a forward and a reverse hierarchy traversal, the `NearAxis` is not simply added to any of the `ForwardAxis` or the `ReverseAxis` grammar generators.

line 5 The `near` axis is defined in both forms: simple axis specification with no threshold, or axis specification with the numeric threshold parameter;

lines 15,17 the `below` axis is added to the set of `ForwardAxis` generators, following the `AxisNear` definition; also a new `BelowAxis` grammar generator is added for clarity.

As outlined in Section 4.1 the `below` axis evaluation produces the same *node-set* of the `descendant` axis; however the evaluation of the `below` constraint associates a score with each node in the retrieved *node-set*.

4.3.2 The Structural Score Variable

Defined as an XQuery Full-Text extension, the FleXy language benefits from the score variable availability (cfr. Section 2.3.3), and it further extends such feature by adding a new Score Variable, called *structural score*, the value of which is computed during the `below` and the `near` axes evaluation.

The structural score variable has been added into the FLWOR clause by defining an optional Score Variable identified by the keyword `score-structure`, in conjunction to the XQuery Full-Text `score` variable, in both the `for` and `let` clauses. In Listing 4.2 the extended XQuery `for` clause, is presented (as defined in rule number 35 of [59]): the score variable `StructScoreVar` grammar generator is added to the `FTScoreVar` originally defined in the XQuery Full-Text extension. `Varname` is a variable name definition, `TypeDeclaration` is a variable type declaration; and `ExprSingle` is the actual query for node selection as defined in the XQuery language.

```
ForClause      ::= "for" "$" VarName TypeDeclaration? PositionalVar?
    FTScoreVar? StructScoreVar? "in" ExprSingle ("," "$" VarName
    TypeDeclaration? PositionalVar? FTScoreVar? StructScoreVar? "in"
    ExprSingle)*
FTScoreVar     ::= "score" "$" VarName
StructScoreVar::= "score-structure" "$" VarName
```

Listing 4.2: FleXy structural score variable syntax in `for` clause.

The same score definition has been added to the `let` expression of the XQuery Full-Text extension, originally defined in rule number 38 in [59]. The grammar is shown in Listing 4.3; the same definitions provided in the `for` clause also apply.

```
LetClause      ::= "let" (("$" VarName TypeDeclaration?) | FTScoreVar |
    StructScoreVar) ":=" ExprSingle ("," (("$" VarName TypeDeclaration?)
     | FTScoreVar | StructScoreVar?) ":=" ExprSingle)*
```

Listing 4.3: FleXy structural score variable syntax in `for` clause.

An example of XQuery expression that uses both the Full-Text and the structural scores computed by the `below` axis evaluation is presented in Listing 4.4. By this query the user declares her/his interest in all nodes labeled `name` that contain the text *dicaprio*; such nodes must have a `person` node as ancestor. The resulting `name` nodes are then ranked based on the numeric score computed on the basis of their distance from the context node `person` (as obtained by the `below` axis evaluation), and stored in the `$scoreST` variable.

In particular in line 2 and 3 the Full-Text score and the structural score are defined, respectively; while in line 6 the clause `order by` in the user query specifies how the returned elements must be sorted.

Finally, in line 7 the structure of each retrieved result is defined: it will be built as a single XML element labeled `hit` containing the textual contents of the matched `name` target node; furthermore two attributes of the `hit` element, labeled `sST` and `sFT` will contain the computed structural score and the Full-Text score respectively.

The results are then returned from the FleXy expression evaluation in an XML form, where both structural and Full-Text scores are displayed as well as the `name` node textual contents.

```
for $item
  score $scoreFT
  score-structure $scoreST
  in person/below::name[text() contains text "dicaprio"]

 order by $scoreST
 return <hit sST="{$scoreST}" sFT="{$scoreFT}">{$item/text()}</hit>
```

Listing 4.4: Example of a FleXy expression using Full-Text and Structural scores

The results obtained by the evaluation of the previous query on the XML document fragment[1] in Listing 4.6 are shown in Listing 4.5. It is important to notice that if the structural score would not be taken into account for scoring and sorting the retrieved results, the best ranked results would be instead those in lines 2 and 3 (with a full-text score of 0.62) followed by those in lines 1 and 4 (with a score of 0.45). The `below` axis score computation allows to retrieve as the first query answer the actual actor name, followed by other, and maybe less important, names or nicknames.

```
1  <hit sST="1"    sFT="0.45">Leonardo Wilhelm DiCaprio</hit>
2  <hit sST="0.33" sFT="0.62">Leo DiCaprio</hit>
3  <hit sST="0.33" sFT="0.62">Lenny DiCaprio</hit>
4  <hit sST="0.33" sFT="0.45">Leo W. Dicaprio</hit>
```

Listing 4.5: Results of the evaluation of the FleXy query in Listing 4.4

A linear combination of the two scores can be provided by the user to obtain an overall score, by using the standard XQuery numeric functions. Further details on the `below` axis evaluation and on its scoring function will be given in Section 4.5.

```
<person>
  <name>Leonardo Wilhelm DiCaprio</name>
  <overview>
    <other_names>
      <name>Leo DiCaprio</name>
      <name>Lenny DiCaprio</name>
      <name>Lenny D</name>
      <name>Leo W. Dicaprio</name>
    </other_names>
  </overview>
</person>
```

Listing 4.6: Fragment of an XML document containing the actor details

It is important to notice that the score assigned by the Full-Text evaluation (as described in Section 2.3) is defined as *Implementation Dependent*: the Flexy language has been

[1] The document fragment has been taken from the Internet Movie DataBase (IMBD) collection that composes the INEX Data-Centric evaluation track.

implemented on top of the XQuery engine BaseX[2]. The implementations details will be provided in Section 5.3.

4.4 The FleXy Semantics

In this section the formal definition of the semantics of the FleXy axes `below` and `near` is presented. The semantics of the new axes is defined in compliance with the XQuery Full-Text language by extending the semantics defined by Gottlob et al. in [9]. Furthermore, the score semantics for the path relevance estimation is provided following the XQuery Full-Text definition, thus using second order functions.

Before introducing the FleXy semantics, the Core XPath semantics from Gottlob et al., defined in [9] is shortly presented in Section 4.4.1; the subsequent sections 4.4.2 and 4.4.3 define the `below` and the `near` axes semantics, respectively. Section 4.4.4 concludes the semantic definitions by presenting the second order functions defined in FleXy for the structure score computation.

4.4.1 Core XPath semantics

The semantics of the new flexible axes has been defined by extending the Core XPath semantics defined in by Gottlob et al. in [9]; the Core XPath semantics formally describes the semantics of the XPath language and the manipulation of set of nodes: in particular the Core XPath does not defines arithmetical and string functions. In the following, the Core XPath is shortly introduced: only the main aspects of the semantics are described, in particular the axes evaluation semantics. For a detailed description of the Core XPath semantics, please refer to Gottlob et al. work.

The same conventions adopted in [9] are used here below, in particular Gottlob et al. make use of a particular equivalent notation to design functions and relations. In [9] binary relations are used instead of functions; for sake of simplicity, binary relations share the same name of the related functions; i.e., a function f of one argument ($y = f(x)$ with $x \in \mathit{Dom}$) is equivalently replaced by the binary relation defined as $\{\langle x, f(x)\rangle \mid x \in \mathit{Dom}, f(x) \neq \mathit{null}\}$, where Dom is the set of nodes of an XML document.

The Core XPath semantics describes an XPath axis as a binary relation χ defined as: $\chi \subseteq \mathit{dom} \times \mathit{dom}$. As an example, the *self* axis is described by the binary relation `self` defined as $\{\langle x, x\rangle \mid x \in \mathit{dom}\}$; the other XPath axes, listed in Table 2.1, are defined by a combination of two *primitive* binary relations (defined in [100]) named *firstchild* and *nextsibling* that, given a node x, represent the first child of x, and the next sibling node of x respectively. Finally, the Core XPath describes the generic axis function named

[2] BaseX - The XML Database (http://www.basex.org): An Open Source XQuery engine that implements the Full-Text language extension, and the early XQuery 3.0 language definition. Further details of the BaseX system are provided in Section 5.3

$\chi : 2^{Dom} \rightarrow 2^{Dom}$ defined as: $\chi(X_0) = \{x \mid \exists x_0 \in X_0 : x_0 \chi x\}$, where $X_0 \subseteq Dom$ is a set of nodes. Please notice that the Core XPath Semantics provided in [9] deliberately choose to overload the the relation name χ previously defined.

A function T is defined to complete the semantics of the evaluation of a XPath *location path* expression (described in Section 2.2.1.2); the function $T\colon \Sigma \rightarrow Dom$ (where Σ is the set of node labels and Dom the set of nodes in the XML document) maps each node-label to the set of nodes having that label.

The semantics of the evaluation of an XPath query is defined by the semantic function $\mathcal{S}$. The axis χ evaluation is defined in Equation (4.1) where: N_0 is the set of context nodes, the evaluation of a path expression π given a context node N_0 is denoted as $\mathcal{S}[\![\pi]\!](N_0)$, and the element *root* corresponds to the document root node defined in the XPath DOM.

$$\begin{aligned}\mathcal{S}[\![\chi :: t]\!](N_0) &:= \chi(N_0) \cap T(t)\\ \mathcal{S}[\![/\chi :: t]\!](N_0) &:= \chi(\{root\}) \cap T(t)\\ \mathcal{S}[\![\pi/\chi :: t]\!](N_0) &:= \chi(\mathcal{S}[\![\pi]\!](N_0)) \cap T(t)\end{aligned} \tag{4.1}$$

In the following sections, the semantics of the new axes `below` and `near` is provided by integrating and extending the Core XPath semantics. For the complete Core XPath semantics description please refer to Gottlob et al. work in [9].

4.4.2 The "`below`" axis semantics

As previously outlined the definitions provided in this section are finalized to identify the set of nodes retrieved by the `below` axis evaluation. In this sense, the formal semantics of the axis `below` is the same formal semantics of the `descendant` constraint, as both of them identify the same set of target nodes; in fact both of them allow to match all the descending nodes of the context node $n_0 \in N_0$ with a given label t that identifies the target node. The only and crucial difference between the `below` and the `descendant` constraints is the computation of *path relevance scores* that are produced by the `below` constraint evaluation and not by the `descendant` axis evaluation: for each fragment matching the `below` constraint (and thus returned in the set of retrieved fragments) a score is computed, as it will be described at an operational level in section 4.5.1.

Based on this assumption, the definition of the semantics of the `below` axis is simply inherited by the `descendant` axis semantics as shown in (4.2).

$$\mathcal{S}[\![\texttt{below} :: t]\!](N_0) = \mathcal{S}[\![\texttt{descendant} :: t]\!](N_0) \tag{4.2}$$

As defined in Section 4.2, the `below` axis constraint may be inserted in any query with an unlimited nesting. If more than a single flexible constraint is specified in a query, the relevance score of a retrieved fragment is computed as an aggregation of all the nested `below` axes evaluation. This approach differs from standard XQuery Full-Text score computation in case of multiple full-text constraints: as described in Section 2.3.3 a Full-Text score variable contains only the score computed for the target node, even if multiple Full-Text expressions are provided. The aggregation function applied in case of nested `below` axes is the function *min()*: this supposes that *all* the flexible constraints are satisfied, and that if a nested *ideal-path* is matched during the `below` evaluation, it receives the highest structural score.

Regarding the semantics of the `below` axis when a threshold parameter is specified, the proposed formal definition makes use of a function $Below(N_0, l)$ that returns all the descendant nodes of N_0 that are at most l arcs far from the context node N_0. The corresponding generic semantics of $\texttt{below}\boldsymbol{l}\texttt{::}\boldsymbol{t}$ is shown in Equation (4.3); please notice that, differently from Section 4.2.1, the threshold parameter name has been changed to l to avoid confusion with context nodes items $n \in N_0$.

$$\mathcal{S}[\![\texttt{below}l :: t]\!](N_0) := Below\boldsymbol{l}(N_0) \cap T(t) \tag{4.3}$$

Like all the axes in the Core XPath, the function $Below\boldsymbol{l}(N_0)$ is defined by using the Core XPath *primitive* functions: in particular the evaluation of the `below` axis with a threshold specification is a composition of a finite composition of the `child` relation, where the `below` threshold defines the number of sequential compositions. As an example, the *Below**2*** function is defined as follows:

$$\begin{aligned} Below\boldsymbol{2}(N_0) := \{\langle x, z\rangle \mid x \in N_0 \wedge y, z \in Dom,\\ (x\ \chi_{\texttt{child}}\ y\ \wedge\ y\ \chi_{\texttt{child}}\ z) \vee (\ x\ \chi_{\texttt{child}}\ z)\} \end{aligned} \tag{4.4}$$

where $\chi_{\texttt{child}}$ identifies the `child` axis relation.

4.4.3 The "`near`" axis semantics

The `near` axis, as initially described in Section 4.2.2, allows to match all nodes in the document tree having a maximum distance from the context node of l arcs. Nodes the distance of which is more then l arcs are filtered out from the possible results. Following the CoreXPath semantics previously introduced, and by using the same approach for defining the axis threshold, the `near` axis semantics is defined in Equation (4.5).

$$\mathcal{S}[\![\texttt{near}l :: t]\!](N_0) := Near\boldsymbol{l}(N_0) \cap T(t) \tag{4.5}$$

where $Near\boldsymbol{l}(N_0)$ is the function that returns all nodes with a maximum distance of l from each context node in N_0. The `near` axis may be defined without a threshold specification; in this case the default threshold value of 1 is provided, and the associated semantics is defined by Equation (4.6).

$$\mathcal{S}[\![\texttt{near} :: t]\!](N_0) := Near1(N_0) \cap T(t) \tag{4.6}$$

Similarly to the $Below\boldsymbol{l}(N_0)$, the function $Near\boldsymbol{l}(N_0)$ is defined by a composition of Core XPath function primitives, where the threshold value acts as a composition threshold. As an example, in Equation (4.7) the $Near\boldsymbol{2}$ function is provided:

$$\begin{aligned} Near\boldsymbol{2}(N_0) := \{\langle x, z\rangle \mid\ & x \in N_0 \wedge y, z \in Dom \wedge x \neq y \neq z, \\ & (x\ \chi_{\texttt{below2}}\ z)\ \vee (x\ \chi_{\texttt{parent}}\ z) \\ & (x\ \chi_{\texttt{parent}}\ y\ \wedge\ y\ \chi_{\texttt{child}}\ \ z) \vee (x\ \chi_{\texttt{parent}}\ y\ \wedge\ y\ \chi_{\texttt{parent}}\ z)\} \end{aligned} \tag{4.7}$$

where $\chi_{\texttt{child}}$, $\chi_{\texttt{parent}}$ and $\chi_{\texttt{below2}}$ identify the `child`, `parent` and `below2` axes relations respectively. As shown, the `near2` axis function matches all the elements at a distance of 2 arcs from the x context node by decomposing its evaluation in simpler axes functions. It should be noticed that this function definition imposes that no duplicate nodes are retrieved since the `near` axis involves a combination of both *reverse* and *forward* evaluation directions.

In the same way, the $Near\boldsymbol{1}$ function is defined in Equation (4.8): in this case the nodes matched are only the ones at a distance of one arc from the *context node*, thus following only the `child` and the `parent` axes.

$$Near\boldsymbol{1}(N_0) := \{\langle x, y\rangle \mid x \in N_0 \wedge y \in Dom, (x\ \chi_{\texttt{child}}\ y)\ \vee (x\ \chi_{\texttt{parent}}\ y)\} \tag{4.8}$$

As for the `below` axis, also for the `near` axis evaluation a score is computed that represents the *path relevance* of the matched element to the axis *ideal path* described in Section 4.2.2.

4.4.4 The Score semantics

The score semantics for the structural-score computed by the `below` and the `near` axes evaluation has been defined following the score semantics of the XQuery Full-Text extension, as explained in Section 2.3.3. Also the structural score semantics make use of second order functions, thus allowing to compute and return the structural score similarly to the definition of the full-text scores given in [59].

The two second order functions that have been defined to compute the structural relevance score of a FleXy expression are: `sts:scoreSequence(Expr)` and `sts:score(Expr)`, where the prefix `sts:` has been used to distinguish the newly introduced functions from the Full-Text correspondent functions, and it stands for *StructureScore*.

Both functions compute the relevance score of a considered element to the given query expression `Expr`: the function `sts:score` returns the structural score computed for the single element given in `Expr`. The function `sts:scoreSequence`, instead, accepts as input a sequence of elements: a sequence of scores is then returned that represents the structural score computed for each one of the elements provided as input.

Similarly to the second order functions introduced by the Full-Text extension (described in Section 2.3.3), the `sts:scoreSequence` and the `sts:score` functions are used when a `for` or `let` FLWOR expression is evaluated. Like in the example provided for the `score` variable type in Section 2.3.3, the function `sts:scoreSequence` is introduced in `for` clauses, while the function `sts:score` is applied in `let` clauses.

The following query:

```
for $item score-structure $scoreST in Expr
  return ...
```

is semantically evaluated as if it was rewritten as:

```
1   let $STScoreSeq := sts:scoreSequence(Expr)
2   for $item at $i in Expr
3     let $scoreST := $STScoreSeq[$i]
4     return ...
```

The score variable `$scoreST`, defined as a `score-structure` type, is computed using the `sts:scoreSequence` function; two new variables are used to process the evaluation of the expression: one to store the set of scores (the variable `$STScoreSeq` defined in line 1) and one to identify the index position of the currently evaluated element (the variable `$i` defined in line 2). Finally, each score is acquired during the `for` iteration by accessing the set of scores computed by the `sts:scoreSequence` function and stored in the `$STScoreSeq` variable as shown in line 3.

In the same way the second order function `sts:score` is introduced when a `let` clause defines a `score-structure` variable; the XQuery expression

```
let score-structure $scoreST := Expr
```

is semantically equivalent to the following form, where the second-order function `sts:score` is used:

```
let $scoreST := sts:score(Expr)
```

4.5 The FleXy axes evaluations

In this section the evaluation functions of the new axes `below` and `near` are defined; each function computes the *path relevance score* of a target node, given a context node.

Each score is computed in the interval $[0, 1]$ where the value 1 represents a full satisfaction of the axis constraint evaluation (this value is produced when the target node matches the *ideal-path* definition). Score values less than 1 are assigned to target nodes that do not fully satisfy the axis evaluation; the more a target node is *far* from the context node the lower the score. A score value of zero is associated with target nodes that are not relevant to the flexible axis evaluation; such nodes are then not retrieved and filtered out from the returned node set.

As previously outlined the notion of *ideal-path*, or *path-closeness*, is related to the concept of node distance intended as the number of arcs connecting two nodes following the shortest path. The definitions of the `below` and the `near` axes evaluation functions are based on a count of the of arcs between the context node and the target node to compute the path relevance score; however, it is very important to outline that alternative evaluation functions could be defined and easily integrated in the Flexy definition and implementation.

4.5.1 The "`below`" constraint evaluation function

As previously stated, the `below` axis evaluation produces the same *node-set* result produced by the XPath `descendant` axis evaluation; however, for each retrieved node a score is computed based on the *distance* between the context node and the target node. The path relevance score for the `below` axis evaluation W_{below} with a context node c and a target node t is computed as:

$$W_{below}(c, t) = \frac{1}{|descArcs(c, t)|} \tag{4.9}$$

Where $descArcs(c, t)$ is a function that, given two XML nodes c and t, returns the set of arcs connecting node c with t following the shortest path. Based on the semantics provided in section 4.4.2, the `below` axis evaluation requires that the node t must be a descendant of node c, or, more formally, that $t \in \chi_{\texttt{descendant}}(c)$.

When the axis `below` is used by specifying a threshold value the associated scoring function is defined in Equation (4.10) (it extends the definition provided in Equation (4.9) by introducing the threshold l).

$$W_{below}(c, t, l) = \begin{cases} \frac{1}{|descArcs(c,t)|} & \text{if} \quad |descArcs(c, t)| \leq l \\ 0 & \text{else.} \end{cases} \tag{4.10}$$

The score computed by the `below` axis evaluation is inversely proportional to the distance of the nodes c and t, thus giving to nodes t that are *closer* to the context node c a higher score than the score given to target nodes *far* from the context node.

Other decreasing functions could be defined to compute the `below` axis structural scores, like negative exponential functions providing a smoother decreasing curve for score computation given the distance between the two nodes.

4.5.2 The “`near`" constraint evaluation function

As defined in Section 4.4.3, the `near` axis evaluation allows to retrieve nodes that are *close* to the given context node in every path direction; in the `near` axis evaluation the maximum allowed distance that can occur between the two nodes is taken into account. Furthermore, if no threshold is specified the maximum allowed distance between a context node and a target node is set to the default value of 1.

In Equation (4.11) the function proposed to compute the *path relevance score* based on the `near` axis is defined, where c is a context node, t is the a target node and l is the maximum allowed distance threshold. The function $arcs(c, e)$ returns the set of arcs between the node c and the node t in the shortest path.

$$W_{near}(c,t,l) = \begin{cases} \frac{1}{|arcs(c,t)|} & \text{if} \quad |arcs(c,t)| \leq l \\ 0 & \text{else.} \end{cases} \tag{4.11}$$

Like for the `below` scoring, the score is inversely proportional to the distance of the context node from the target node. The function assigns higher values if the target node is *closer* to the context node, while the relevance score approaches the zero value as the distance increases.

4.5.3 Flexible constraints aggregation

An important issue related to the flexible axes evaluation concerns the aggregation of queries involving multiple flexible constraints and branching in fragment selection. As described in Section 4.3, the flexible constraints evaluation allows to associate with each node involved in the flexible part of the query a relevance score in the range $[0, 1]$, which is used to compute a ranking of the selected fragments.

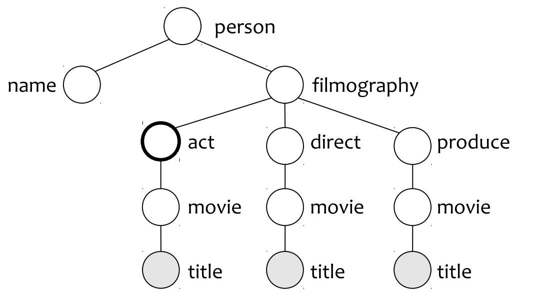
While in flat queries this can be done without any difficulties, a particular observation should be made for branching queries where the selection node that needs to be ranked appears in a different branch than the one/ones that use the flexible constraints, thus obtaining a path relevance score.

Let us consider the following complex query:

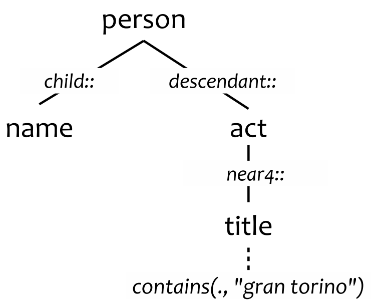
```
for $item score $sft score-structure $sst in
  /person[descendant::act/near4::title[contains(.,"gran torino")]]
    /child::name

order by $sst descending
return
  <hit>{$item/text()}</hit>
```

and the XML document fragment shown in Fig. 4.3a. Let us suppose that the user is interested in finding the names of people involved in the movie entitled "*Gran Torino*." The user interest is mainly in, but not limited to, people who acted in such movie: by using the constraint `near` the user requires also to find people who worked as director, producer, etc. (even if with a lower structural relevance).



(a) Example document fragment for the branching query example.



(b) The tree-representation for the branching query example.

In Fig. 4.3b, the tree representation of the query is shown: the underlined *name* element identifies the target node; the edges between two nodes identify the *axes*-constraints (the label explicit the specified axis, i.e., `child`, `near` and `descendant`). Dotted lines represent filtering functions, in this example the `contains` function. The element `person` is also called *branching point*.

From the above example it may be noticed how the evaluation of the right branch of Fig. 4.3b can produce a set of more than one element for a single person (i.e., *Clint Eastwood* was involved in the movie as the main *actor*, the *director* and the *producer*).

In this case, each retrieved fragment has a score associated, and it is not clear how the final score should be computed and assigned to the branching point to allow a ranking of the elements matched in the left branch of the example.

The proposed solution to address this situation adopts an optimistic aggregation interpretation: the axis evaluation assigns in fact to the branching point element (in the example the `person` element) a score which is the maximum value among those obtained by evaluating the right branches.

Although the choice of applying the `max()` aggregation is quite natural to obtain an optimistic aggregation, it is important to outline that other aggregation schemes could be defined by the user, by means of nested `for` and `let` queries, where the computed scores are aggregated based on the user indications by using either XQuery arithmetic operations or the `fn:min()` and `fn:max()` functions.

5 Implementation

In this chapter the main issues addressed and the main activities undertaken in relation to the implementation of the FleXy language are presented. To the aim of implementing the Flexy language as an extension of the XQuery Full-Text language two main directions have been undertaken: (1) to modify an existing XML query engine with a highly efficient data structure capable of efficiently processing Flexy queries, (2) to implement the Flexy language as a full extension of an existing open source engine that efficiently implements the XQuery Full-Text language.

The first research direction started with an analysis of the syntax and the semantics described in the previous sections to clearly identify the data-structure that could provide an efficient language parsing and evaluation strategy on top of which to implement the FleXy axes constraints and the structural relevance score computation. In Section 5.1 the research activities undertaken to achieve these objectives are reported, by illustrating the data structure, named multi-μPID, designed for providing an efficient flexible axes evaluation as defined by the FleXy language. Such work has represented a first attempt to efficiently integrate the `below` and the `near` axes evaluation on top of an XML query engine. The work undertaken in this first phase has confirmed the feasibility of introducing such flexibility in XML querying, without affecting the overall system performances.

The outcomes of the first research direction, and the consequent need to define an integrated and standard implementation of the Flexy language on the top of an XQuery FullText engine has motivated the definition of a second research direction, which has concerned the implementation of the FleXy constraints on top of an existent, fully featured and XQuery Full-Text compliant engine. The main aim has been to extend an open source engine by including the Flexy constraints, thus defining a new open source engine, highly efficient and available to a wider numeber of users. A first phase of this second implementation strategy has concerned the analysis of the available open source XQuery engines, in order to select the one on which to implement the XQuery Full-Text extensions; this analysis is shortly reported on Section 5.2, where a summary of the considered system features, adaptability and adherence to the XQuery and the XQuery Full-Text languages are reported. The pros and cons of each system are summarized in Section 5.2.6 where the BaseX XML engine, that fully supports the XQuery Full-Text language and implements an efficient node encoding schema, has been selected as the implementation framework for the FleXy language.

Finally in Section 5.3 the software developments performed on top of the BaseX engine are described: details about the implemented functionalities, the integration performed on top of the BaseX query parser and the evaluation algorithms of the FleXy axes are also presented.

5.1 A preliminary FleXy implementation: *multi-minPID*

In this section a first research activity aimed at efficiently implementing the FleXy language constraints and score computation on an XQuery engine is presented. Such work, called *multi-minPID* has been inspired by and has extended a proposal in the literature to integrate XQuery structural constraints with content matching and ranking proposed. Such proposal, defined by Bremer et al. [145], was named XQuery/IR. My contribution has leveraged the XML node identification schema proposed by the same authors, called *minPID* [91].

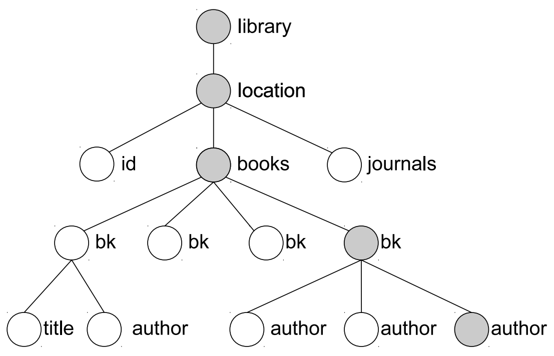
As it will be discussed later, even if providing a good infrastructure for the flexible constraints evaluation, the *multi-minPID* has been replaced by a more feature complete and performing system subsequently implemented during the research performed in my PhD period, and presented in Section 5.3. In particular, the XQuery/IR engine and the preliminary FleXy constraints implementation have been developed when the W3C definition of the Full-Text extension was still in an early stage and no XQuery Full-Text engine was released yet.

In Section 5.1.1 the encoding schema *minPID* is described, while Section 5.1.2 presents the preliminary evaluation of the FleXy constraints on top of the XQuery/IR engine. Section 5.1.3 summarizes the findings of such preliminary work.

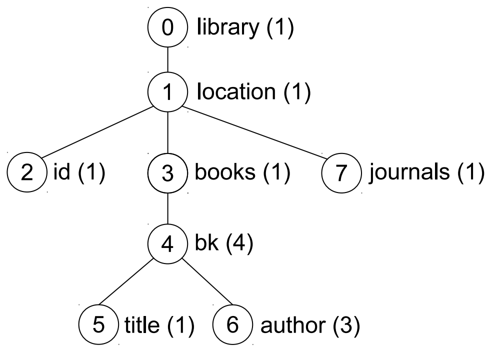
5.1.1 Introduction

The work presented in [145] represents one of the first approaches aimed at integrating in the XQuery language a result ranking based on a keyword-based search performed with Information Retrieval techniques. In particular, a fragment score computation was there implemented by introducing a *rank by* operator in the XQuery language that offered a simple $tf \cdot idf$ element score computation. Such system, named XQuery/IR [146], has been provided by the authors as a prototype, and it has been subsequently extended within my PhD research to provide a preliminary implementation of the FleXy features.

As previously outlined, the XQuery/IR prototype includes an ad-hoc node encoding schema, called minPID, which has been used to index and store both the structure and the content of a single XML document. Such encoding schema allows, as stated in Bremer et al. work, to efficiently evaluate the set of XQuery Path Expressions and queries that include branching. The approach uses the notion of*Minimal Path Identifier*s (minPID), and an efficient encoding schema for such identifiers. A *Minimal Path Identifier* is a way to encode XML paths based on an enriched variant of a DataGuide [88] called XDG, an acronym that stands for eXtended DataGuides, as defined by Bremer et al. in [145]. Similarly to a DataGuide structure, an XDG represents a concise description of an XML document structure, as it is a looser representation than both DTDs and XML Schema. An eXtended DataGuide, like its predecessor DataGuide, can be directly derived from a document, and it enumerates all the rooted label paths (the sequence of labels that connects the document root node to any of the nodes in the document tree) that are present in the node hierarchy of an XML document. For each path in the document a unique identifier is assigned and called *XDG node number* (or `XDGNode#)`; the maximum number of instances of each node label (*sibling fanout*) in the source document hierarchy is also stored.



(a) An example XML document.



(b) The eXtended DataGuide (XDG) for the XML document in Fig. 5.1a.

Figure 5.1: An XML document and its corresponding eXtended DataGuide.

In Fig. 5.1 an example of an XML document is shown with the corresponding eXtended DataGuide graph; each *rooted label path* in the document hierarchy is represented once, and for each node in the XDG the maximum *sibling node fanout* is shown between round brackets. In the example in Fig. 5.1b, the node labeled *bk* has a maximum number of 4 siblings in the entire XML tree, while the number displayed inside each node represents the assigned *XDG node number* based on a pre-left order node visit.

Based on such eXtended DataGuide definition a *Minimal Path Identifier* of a node $n \in D$ is a pair $minPID(n) = (p, s)$, where p is the sequence of XDG nodes identifiers that represent the rooted label path of the node n in the XDG, while s is the sequence of sibling position of each node $p_i \in p$. For example, consider the path expression $library/location/books/bk[4]/author[3]$ (highlighted in Fig. 5.1a), and its corresponding rooted label path $library/location/books/bk/author$; a sibling position is specified for the `bk` (position 4) node and the `author` (position 3) node. Following the XDG shown in Fig. 5.1b, the *minPID* for such path expression is $(< 0, 1, 3, 4, 6 >, < 1, 1, 1, 4, 3 >)$.

The relationship between two nodes $n_1, n_2 \in D$ in an XML document D, can be easily determined based on their minPIDs encoding: descendant and child relationships can be easily determined first by comparing the corresponding *XDG node identifier* of the two nodes, and then by their sibling positions. In the same way the parent and ancestor relationship are evaluated by combining the information stored in the eXtended DataGuide about the rooted label path and the sequence of sibling positions.

5.1.2 Flexy on top of XQuery/IR

Based on the previously described eXtended DataGuide and the *minPID* encoding schema adopted by Bremer et al.'s prototype, the FleXy language extension has been implemented by leveraging such data structures. Furthermore, the original system has been extended to allow the indexing and the querying of multiple document collections, as the original design of the tool was only conceived to handle a unique XML document representing the entire document collection. The definition of *minPID* has been then extended to the following definition in which the minPID of a node $n \in D$ is composed of a tuple $minPID(n) = (d, p, s)$, where d is the document identifier, p is the sequence of `XDGnodes#` from the XDG built on the document D, and s is the sequence of sibling position of each node $p_i \in p$. Document collections with slight hierarchy differences between each document can in this way share a unique XDG across the whole collection, thus reducing the overhead of having an XDG for each single document.

Both the `below` and the `near` axes, alongside their parametric variants, have been implemented in the XQuery/IR tool, where the eXtended DataGuide has been strongly used to identify the set of relevant nodes that have to be retrieved by the flexible axes. In particular, the evaluation of a FleXy axis is performed by first identifying the labels of the *target* and the *context* nodes from the path expression query, then subsequently, for each XDG defined in the system, the following steps are executed:

1. retrieve the set of `XDGNode#` corresponding to the *context* and *target* labels;
2. for each of the matched `XDGNode#`, identify the set of target `XDGNode#` that satisfy the flexible axis by a substring matching between the corresponding *rooted label paths*, and compute the structural score for each target XDG node;
3. finally, retrieve the set of XML nodes having a *minPID* that matches the identified set of `XDGNode#` and that belongs to the current XDG.

The simple iterative process here described performs a costly string comparison between the identified set of *rooted label paths* to identify the correct set of candidate target nodes. Furthermore the implementation provided on top of the XQuery/IR engine inherits the disadvantage of the system structure where a high number of intermediate results have to be computed when branching queries are formulated.

Unfortunately, during the FleXy integration, different issues arose regarding the underlying XQuery/IR data structure implementation, where different bugs have been discovered and, although with an insufficient code documentation, fixed.

Besides being unmaintained and thus no support has been provided for bug fixing from the authors, the XQuery/IR engine also lacked a, either partial, support for the XQuery FLWOR clauses parsing and no query execution component was implemented. For these reasons, besides proving the feasibility of the implementation of the FleXy language constraints, no further development has been provided on top of the XQuery/IR engine.

5.1.3 Summary

Some evaluation tests have been performed with the FleXy implementation previously described, but the timings for computing the `below` and the `near` axis evaluation introduced an high overhead to the entire system. While the implementation showed the feasibility of the FleXy approach and the value added by the structural score computation, the prototype extended by the original Bremen et al. [145], lacked of some of the important features of a complete XQuery engine.

Furthermore, although described as a complete and fully working system, the available prototype released by the XQuery/IR authors only included the index building module and an incomplete set of FLWOR matching functions. In particular the XQuery parsing and evaluation components were not implemented, while the described text index was not extensible to match the upcoming W3C Full-Text features (and its content scoring technique).

For that reason the subsequent FleXy implementation has been proceeded by a deep analysis of the state of the art XQuery engines that both adopts efficient encoding schemes suitable for the FleXy axes evaluation, and that were moving toward the integration, and the at most complete support, of the XQuery/XPath Full-Text extension.

5.2 XQuery engines analysis

In this section the main XQuery engines that have been considered as candidates for the FleXy language integration are shortly introduced, by describing their main characteristics and peculiarities. In particular, the adherence to the standard XQuery, XPath and XQuery/XPath Full-Text languages syntax and semantics have been taken into account to identify which XQuery engine would provide a better environment on top of which to implement the set of FleXy axes and structural score computation.

It is important to outline that the list of engines here presented does not represent a complete and exhaustive analysis of the wide number of available XQuery engines; moreover, for some of the presented systems, some information could not not be fully updated at the moment of reading this manuscript, due to the intensive and continous activity performed by the underlying developer community.

A particular note should be made regarding the functionalities and characteristics of the XQuery engines presented here: some of them could have released updated versions of their core system that, at time of writing, could not have been evaluated, analyzed, nor adopted for the implementation of the Flexy constraints. In particular, the collection and the analysis of XQuery compliant systems has been performed during the spring of 2012, while the implementation took place during the subsequent months.

The engines have been selected as the most relevant representatives of full compliant XQuery systems, and they have been analyzed by taking into account, as the most relevant, the following characteristics:

- Open Source: the systems must be open sourced, to provide a better understanding of their internal evaluation and to perform the integration with the extended FleXy constraints; furthermore a current active development of the system would be beneficial.
- Full compliance with the XQuery and XPath languages syntax and semantics, in particular for the path expressions and the axis-step evaluation;
- the systems must provide and adhere to the W3C Full-Text language extension syntax; a *minimal conformance* as defined by the W3C test suite [62] must be provided, in particular the score computation and the score value access.

In the following sections the XQuery engines MonetDB, eXist-DB, BaseX, Zorba and others are presented and described; Section 5.2.6 provides an overview and a summary of the analyzed set of query systems and other engines that have been encountered during this work. Section 5.2.6 also summarizes the motivations that lead to the adoption of BaseX as the base engine on top of which the FleXy language has been implemented and evaluated.

5.2.1 MonetDB/XQuery

The MonetDB/XQuery[1] is an Open Source XQuery engine developed by combining the relational database engine MonetDB and the PathFinder [147] project; both systems have been implemented in C language. Pathfinder, developed by the university of Kostanz, is a project aimed at providing a *Purely Relational XQuery Processor* on top of relational databases: it performs a compilation process that transforms XQuery expressions into SQL code (or to other variants), thus allowing relational database management systems, where both XML elements and contents have been opportunely encoded and stored as tabular data, to act as XQuery processors. The PathFinder goal is to leverage standard and consolidated database engines to evaluate XQuery expressions without requiring to build a specifically built XQuery engine.

The PathFinder system requires that the set of XML elements that compose an XML document are stored in the underlying database tables by adopting a tree encoding

[1] The MonetDB/XQuery engine is available at `http://www.menetdb.org/XQuery`

schema that efficiently provides the ability to maintain the document order of the elements and to evaluate XQuery node-test and location-step expressions.

The XML-to-tables translation performed by PathFinder uses a variation of the tree node encoding schema proposed in [92] where *Preorder* and *Postorder* node traversal techniques [148] are adopted to compute a tabular representation of XML elements and its hierarchy. Such tabular representation comprises, for each node v the pre-oder visit value of the node in the tree ($pre(v)$), the number of descendant nodes ($size(v)$), the node type as specified by the XQuery DOM such as *element*, *attribute* or *text* node kind ($kind(v)$) and the distance of the node v to the root element node ($level(v)$). The encoding schema used by the PathFinder system, and implemented in the MonetDB/XQuery query engine, is thus called *pre/size/level*.

Given such encoding schema, the XPath and the XQuery axes can be characterized by the four functions $pre()$, $size()$, $kind()$ and $level()$; for example, given a context node v, the set of `descendant` nodes y can be defined as Equation (5.1), while the set of `child` nodes of v as Equation (5.2).

$$\texttt{descendant}(v) := \{x \mid \texttt{pre}(x) > \texttt{pre}(v) \wedge \texttt{pre}(x) < \texttt{pre}(v) + \texttt{size}(v)\} \tag{5.1}$$

$$\begin{aligned}\texttt{descendant}(v) :=& \{x \mid \texttt{pre}(x) > \texttt{pre}(v) \wedge \texttt{pre}(x) < \texttt{pre}(v) + \texttt{size}(v) \\ & \wedge \texttt{level}(x) = \texttt{level}(v) + 1\}\end{aligned} \tag{5.2}$$

Although the encoding schema adopted by the PathFinder system looks promising for the evaluation of the FleXy constraints `below` and `near`, due to both the presence of the *level* attribute for each node and the possibility to access descendant elements in constant time, such encoding would require an expensive set of computations to access `parent` and `ancestors` nodes. A solution has been proposed by the BaseX engine, described in Section 5.2.3, that provides constant time access also to ancestors nodes.

As MonetDB/XQuery leverages the underlying relational database representation of XML documents, the MonetDB/XQuery engine also includes a preliminary implementation of the XQuery Update Facility [35] feature coupled with transactional safe update capabilities, query caches and user-defined functions.

The current development of MonetDB, unfortunately, has stopped the support and the integration of the MonetDB relational database engine with the PathFinder system due to a lack of resources[2]. The latest available MonetDB/XQuery engine has been released

[2] On the MonetDB/XQuery webpage, the following message announces that the development and integration of PathFinder with the current releases of MonetDB have stopped: *March 2011: MonetDB/XQuery project is frozen. Due to lack of development and manpower to port the software to MonetDB version 5, we had to freeze the code base. We do not fix any bugs or problems with MonetDB/XQuery*", no more updates have been published recently.

on March 2011, that integrates the PathFinder system with the MonetDB version 4; although the encoding schema adopted by PathFinder provides good performances in evaluating XQuery expressions, the Full-Text extension was not integrated in MonetDB/XQuery. For that reason the MonetDB/XQuery engine was discarded from the candidates of possible engines where the FleXy constraints could be integrated.

5.2.2 eXist-db Engine

The eXist-db engine[3] is an open source XQuery processor written in Java language that has been extended, since its first definition by Meier [108], to integrate the XQuery language up to the latest 3.0 W3C recommendation.

The eXist-db engine adopts a different XML element encoding schema from the *pre/-size/level* used in MonetDB/XQuery and the *pre/size/dist* of BaseX. A variation of the level-order numbering schema proposed by Lee et al. [149] is adopted in eXist: the original encoding technique is defined for complete k-trees, where spare node identifiers are assigned in case of incomplete or unbalanced trees branches. The Lee et al. constraint of k-ary tree completeness has been partially altered in the eXist encoding schema: Meier's observation is that in typical XML documents the maximum number of children can greatly vary at different hierarchy levels, thus requiring, in a worst case scenario, to insert (and waste) a huge number of spare node identifiers. In the eXist encoding schema, instead, the k-ary completeness is computed at each tree level, thus minimizing the number of virtual node identifiers to add in the tree representation.

Subsequent works carried by community efforts on the open source eXist-db project provided the integration of the engine with different XQuery extended modules and the ability to invoke Java methods within an XQuery expression.

The current development status of the eXist-DB engine, unfortunately, lacks a Full-Text compliance integration: a full-text search functionality is provided by an external module called `ft:` [4] by leveraging the community driven and open source Apache Lucene framework [150]. The Lucene framework provides a set of capabilities for text search engines, in particular features as document and terms indexing, advanced text analysis, and a specially defined query language are available from a Java based set of methods, classes and APIs. The advantages obtained by integrating such framework, such as the usage of a solid search framework as Lucene is, are however affected by the missing coherence of the Lucene query syntax with the one defined by W3C and included in the Full-Text standard.

Another important discrepancy between the eXist and the Full-Text extension resides on how the score of a full text search is computed: the W3C definition requires the scores to be computed in the interval $[0, 1]$ (see Section 2.3.3 for details), but in a Lucene query

[3] The eXist-db query engine is available at `http://exist-db.org` webpage.
[4] The eXist full-text search module named `ft:` is described in the eXist documentation available at `http://exist-db.org/exist/apps/doc/lucene.xml`.

evaluation the relevance score is not bounded in such interval. As also shown in the online demo of the engine[5]: the execution of the query in Listing 5.1 by the eXist engine produces the results (partially shown) in Listing 5.2, where the computed scores outside the $[0, 1]$ interval are shown.

```
  for $m in //SPEECH[ft:query(., "boil bubble")]
    let $score := ft:score($m)
    order by $score descending

  return <m score="{$score}">{$m}</m>
```

Listing 5.1: Full-Text search specified using eXist `ft:` search module.

```
<m score="2.572969">
    <SPEECH>
      <SPEAKER>Second Witch</SPEAKER>
      <LINE>Fillet of a fenny snake,</LINE>
      <LINE>In the cauldron boil and bake;</LINE>
  (...)
</m>
<m score="1.0049098">
  <SPEECH>
    <SPEAKER>ALL</SPEAKER>
    <LINE>Double, double toil and trouble;</LINE>
  (...)
```

Listing 5.2: Results obtained by the eXist evaluation of query in Listing 5.1

Furthermore the Lucene framework works on a fielded document representation, where no hierarchy or structure is taken into account; this requires the eXist-db engine to internally index and process a set of mappings between a processed XML document and the Lucene XML-agnostic document representation. In particular, as examplified in the eXist-db full-text documentation, the adopted indexing procedure requires a manual intervention of the user to configure the set of indexed and ignored XML elements.

From the documentation of eXist-db a note is provided regarding the adoption of the XQuery Full-Text in the engine; such note simply states that the Lucene index will be used to provide the standard W3C extension, but no more updates are given about the development or the estimated implementation plan.

5.2.3 BaseX

As cited on its website, BaseX is an *XML Database engine and XPath/XQuery processor*; it was initially developed by the University of Konstanz, and it has been further

[5] The Full-Text example is taken from the online demo available at: `http://exist-db.org/exist/apps/demo/examples/basic/fulltext.html`.

developed as an open source product with commercial support[6].

The constant and continuous development of the BaseX engine is proved by its adoption and by the implementation of the most recent XQuery and XML related technologies and standards on top of this engine: examples are the the XQuery Update Facility and the XQuery 3.0 language that were introduced, and then consolidated, in BaseX since their initial W3C drafts; furthermore the set of available XQuery modules is continuously updated.

BaseX has been distinguished as being the first XQuery engine to integrate, and fully adhere, to the XQuery/XPath Full-Text language extension, where other engines only provided a customized and reduced variant for full-text searches. BaseX supports the full set of Full-Text options, such as wildcards, stemming and stop words and more than 20 languages. Most notably, the founder and the main developer of BaseX, Christian Grüen, is also one of the few members of the XQuery and XPath Full Text 1.0 Test Suite [62] project.

The first engine prototype has been presented in Grüen et al. work [109], where the first BaseX core was presented, and some comparisons were provided; a subsequent work [151] of the same authors further extended BaseX's data structures to allow the XQuery Full-Text extension evaluation. The main characteristic of BaseX is the adoption of an ad-hoc indexing schema inherited from the *pre/size* proposed by Grust et al. [92] and implemented in the MonetDB/XQuery [78]. The difference resides on the optimization introduced for the *parent* and *child* axis evaluation, where a third value, called *dist*, allows to access parent and child nodes in constant time, thus improving the query evaluation efficiency.

BaseX also represents the first approach to implement the state of the art of XML data storage and handling by integrating both an in-memory query processing, and an efficient and compressed indexing schema. The integration of query optimization and query rewriting techniques, coupled with the index optimization, resulted in an efficient and highly scalable XQuery engine, able to index up to 500Gb of XML data.

Some efficiency evaluations are provided in Grün work [152], where the BaseX engine is compared against other open-source XQuery engines such as MonetDB/XQuery, eXist and Zorba: the provided results show how both the XML indexing and the query evaluation of BaseX outperform the mentioned systems, thus enforcing the efficiency of the BaseX encoding and query evaluation systems.

BaseX has been primarily chosen for being the first system to implement the full XQuery Full-Text extension languages. Other aspects of the BaseX project that motivated this choice rely on the availability of its source code, and the active community behind the BaseX development.

[6] *BaseX, The XML Database.* website is available at `www.basex.org`

5.2.4 Zorba

The query engine ZORBA[7] is a C/C++ based open source system that implements the standard W3C query languages such as XQuery and XPath with the latest standard extensions and language modules; it was initially presented by Bamford et al. in the *XQuery Reloaded* [153] article. The project is supported by the *FLWOR Foundation*[8]: a non-profit organization for promoting the adoption of the XQuery technology and supporting open source projects for commercial products.

An important aspect of the ZORBA XQuery processor engine resides on the possibility offered by the system to query not only XML documents, but also JSON (JavaScript Object Notation) [154] structured data that share a wide set of commonalities with XML documents. ZORBA, alongside XQuery, also integrates the JSONiq [155] query language, a language for JSON documents and inspired by the XQuery syntax and constructs[9].

The ZORBA system is structured differently from other XQuery engines where both indexing and querying facilities are bounded together: ZORBA provides an overlay query engine where different storage and indexing methods can be plugged in and used as the underlying storage engine. This aspect, although simplifying the introduction of the FleXy axis constraints in the ZORBA language parser and interpreter, would definitely require both to provide the implementation and the evaluation of the XQuery flexible extension on top of the different storage engines available for the ZORBA system, varying from in-memory storage (default ZORBA implementation) to disk or DB storage approaches. All the implemented storage systems do not tackle neither the storage of unstructured data with ad-hoc data structures, nor encoding schemas that support efficient node traversal: the ZORBA engine purely acts as an overlay framework to provide XQuery/JSONiq support, but it delegates to the underlying storage platform the complete evaluation of the query constraints.

The ZORBA engine, probably due to the different types of supported storage engines, only partially supports the W3C XQuery Full-Text extension: although the engine obtained a high conformance score in the W3C Full-Text conformance evaluation, the ZORBA engine neither support, nor provides, the Full-Text relevance score computation in any FLWOR expression.

```
for $item score $s
    in doc("books.xml")/bib/book[author contains text "Melton"]
return <hit score="{$score}">{$item}</hit>
```

Listing 5.3: A Full-Text search provided for testing the ZORBA XQuery engine.

[7] The "ZORBA *NoSQL Query Porcessing*" engine documentation, source-code and development blog is located at: `http://www.zorba.io/`.

[8] The FLWOR Foundation homepage is available at `www.flworfound.org`.

[9] More details about the JSONiq query language and its commonalities with XQuery can be found on the JSONiq website: `http://jsoniq.org`.

An example of such missing feature is also shown in the online Zorba demonstration tool[10] , where if the the query in Listing 5.3 is formulated, the system would not execute the query, and only the following error message is reported:

```
Zorba error [zerr:ZXQP0004]: not yet implemented: score
```

The missing conformance to the XQuery Full-Text extension, in particular the unavailability of a relevance score computation and the *ScoreVariable* in any FLWOR clause, further motivated the choice of not to adopt the actual Zorba 2.9 system for the FleXy language implementation.

5.2.5 Other XQuery engines with Full-Text

Other Open-Source XQuery engines exist implementing full-text search capabilities over XML documents; these include MXQuery [156], Nux [157] and others.

As previously described, a query engine candidate for the FleXy language integration must have three characteristics to be taken into account, and then to be further integrated with the flexible language: the engines described here below miss one or same of the requirements. These engines have then not been conisdered as a suitable candidates, given their missing strict adherence with the XQuery and XQuery/XPath Full-Text standards syntax and evaluation. In the following sections the main characteristics and the reasons that lead to excluding such systems from the evaluation and the implementation of the FleXy language are described.

5.2.5.1 MXQuery

The MicroXQuery (MXQuery) [156] engine is a Java-based and open-source[11] engine that implements (most of) the features of the standard XQuery language.

The MXQuery engine has been developed within a collaboration between Siemens and ETH Zurich; the MXQuery system is designed as a *streaming* XQuery engine: in such kind of systems the evaluation of queries is not performed on a DOM document representation, but on XML data that are streamed and parsed serially at application runtime.

Furthermore the engine has been designed by adopting techniques that allow the system to be integrated in small and embedded devices like mobile phones and computers with limited resources as demonstrated in [158], where MXQuery has been integrated on a small integrated circuit board.

[10] The Zorba online demonstration tool is available at `http://zorbawebsite2.my28msec.com/html/demo`; the tool is based on the latest version 2.9.0 of the engine.

[11] MXQuery engine is hosted at `http://www.mxquery.org`, while its source code is available under the SourceForce repository: `http://sourceforge.net/projects/mxquery/`

The XQuery parser applies query normalization using the rules of the Query formal semantics, and query expressions are optimized by using techniques such as query rewriting rules and duplicates elimination. Advanced algorithms, like the query optimizer and cost-based query rewriting available in BaseX, are not implemented in MXQuery. Furthermore, MXQuery has also been used as a reference implementation for the XQuery Update language [35] and the XQuery Scripting [159] extensions in 2009.

Unfortunately the MXQuery development has stopped in latest years apart from some porting of the library to the Android OS. The latest MXQuery release, at time of writing, dates back to March 2009 with release 0.6 that, from its change-log, provides only a *partial* and *experimental* support for the XQuery Full-Text language extension, the main issues[12] reported regarding XQuery Full-Text are: (1) the missing support of the scoring computation in `for` clauses and (2) the availability of English text parsing only. The important issue about the missing score computation, the support for XQuery Full-Text defined as experimental, and the stopped development of the tool, prevented the tool to be taken into account for implementing the FleXy language.

Some discrepancy were found during the evaluation of the MXQuery engine: the latest available release reported online is *0.6.0*, while the results provided in the W3C XQuery Full-Text Test Suite [160] refer to a *0.7.0* release version. The source code of such updated release could not been found on the MXQuery website, neither under the SVN source code tracking provided by SourceForge, nor on the (broken since March 2012) continuous integration system, intended to provide pre-compiled versions of the MXQuery engine, available at `http://sgv-jenkins-01.ethz.ch/job/MXQuery/`.

5.2.5.2 Nux query engine

The Nux [157] query engine, similarly to MXQuery, is a XQuery engine built for streaming querying XML data written in Java. While supporting standard XQuery and XPath languages, it introduces a full-text matching feature by integrating the Apache Lucene [150] search framework in query formulation as an extended set of functions for the XQuery and XPath languages.

The Nux engine leverages the Apache Lucene framework to provide full text search capabilities while querying XML documents similarly to the eXist-db engine described in Section 5.2.2. The same observation made for the eXist engine also apply here for the Nux integration with the Lucene framework: the relevance extimation score computed by the Apache framework does not adhere to the W3C Full-Text definition where the full-text relevance score must be in the interval $[0, 1]$.

Although the technique proposed by Nux to integrate a full-text search seems promising as shown in the example in Listing 5.4, the tool does not support the Full-Text language extension as defined by the W3C. Nux misses the syntax and the features defined by

[12] The list of known issues and bugs of MXQuery are available online at the *Bugs* section of the MXQuery webstite: `http://mxquery.org/?page_id=63`.

the standardized language, for example stop-word removal and the definition of a thesaurus. Furthermore, the constraints evaluation using a function wrapper to the Lucene framework, requires the user to: (1) specify to retrieve only elements having a non zero relevance score, and (2) to compute the full-text score twice to access the score value.

```
declare namespace lucene = "java:nux.xom.pool.FullTextUtil";
declare variable $query := "+salmon~ +fish* manual~";
(: declare variable $query as xs:string external; :)

for $book in /books/book[author="James" and lucene:match(abstract,
   $query) > 0.0]
  let $score := lucene:match($book/abstract, $query)
  order by $score descending
return $book
```

Listing 5.4: Example of full-text search in Nux XQuery engine

The Nux development, as for the MXQuery engine, has stopped since 2006 and the non strict adherence to the Full-Text language definition has prevented the engine to be considered as a candidate tool for the integration with the FleXy language.

5.2.6 XQuery and XQuery Full-Text engines comparison

In this section a summary and a comparison of the main features of the previously described XQuery systems is provided: the comparison here presented is not intended as a complete deep analysis of each system's performance, indexing and querying timings. Other works have envisioned a benchmark tool for XQuery engines, an example is the XQBench system proposed by Fisher [161], where a complete CPU, RAM and Disk I/O performance evaluation and benchmark should be provided for XQuery engines; such tool, unfortunately, has not been finalized into an online tool as initially promised. Other works in that direction have been further carried out by the same group in [162], but they only focus on Database related systems.

The system comparison provided here is, instead, a feature-oriented evaluation of XQuery engines, in particular for the engines described in the previous sections, and concerning the set of features listed in Section 5.2. Such *required* features are the ones taken into account as discriminators for selecting the candidate system on top of which the FleXy language would be implemented.

In Table 5.1 the system characteristics of the engines analyzed are reported. The systems information here presented have been updated at October 2013, while the final decision, as subsequently motivated, of adopting the BaseX engine as the FleXy implementation candidate, has been taken in spring 2012.

Name	Status	Language	XQuery Full-Text
MonetDB/XQuery	Unmaintained	C/C++	None
BaseX	Actively developed	Java	Complete
Zorba	Actively developed	C/C++	Partial, no scores
eXist	Actively developed	Java	Apache Lucene syntax
Nux	Unmaintained	Java	Apache Lucene syntax
MXQuery	Unmaintained	Java	Partial, no scores
Qizx/Open	Unmaintained	Java	Custom FT syntax

Name	License	Latest Release	Homepage
MonetDB/XQuery	Public/Unknown	4.0 (2011-03-01)	www.monetdb.org/XQuery
BaseX	BSD	7.7.2 (2013-10-07)	www.basex.org
Zorba	Apache 2.0	2.9.0 (2013-05-15)	www.zorba.io
eXist	GNU L-GPL	2.1 (2013-07-18)	www.exist-db.org
Nux	Public/Unknown	1.6 (2006-06-18)	acs.lbl.gov/software/nux/
MXQuery	Apache 2.0	0.6.0 (2009-05-04)	www.mxquery.org
Qizx/Open	Mozilla/Public	4.1 (2010-10-12)	www.axyana.com/qizxopen

Table 5.1: XQuery engines summary

As visible from Table 5.1, only few open source XQuery engines strives to provide a standard Full-Text extension implementation, while other systems that provide full text search features rely on the Apache Lucene framework syntax and indexing capabilities.

Regarding the XQuery Full-Text extension comparison, in Table 5.2 a summary of the Full-Text test suite results, as taken from the W3C Test suite page[13] is provided to show how each engine complies with the standard Full-Text syntax and evaluation options. The three columns of Table 5.2 represent: (1) the *Minimal conformance* as the adherence of the engine to the required features of the Full-Text specification, (2) the support of the full set of Full-Text *Expressions* such as scoring variables and the comparison operators, and (3) the implementation of the *Optional* features of the Full-Text extension, such as the `stop-word`, `language` and other operators.

Some notes should be made about the compatibility levels shown in the Test Suite results: only the BaseX and MXQuery engines are reported to obtain a complete conformance with the XQuery Full-Text extension, while other engines only partially adhere to the Full-Text definition. An incoherence between the results reported by the W3C and the MXQuery engine exists: the evaluation performed on the MXQuery is reported for the release 0.7.0; such release, instead, could not be found online and thus the evaluation could not be reproduced. As also described in Section 5.2.5.1. the latest

[13] The XQuery Full-Text Test Suite results are available from the W3C webpage `http://dev.w3.org/2007/xpath-full-text-10-test-suite/PublicPagesStagingArea/ReportedResults/XQFTTSReport.html`; while the Zorba results have been taken from the engine documentation available at the following address: `http://www.zorba.io/documentation/2.9/zorba/conformanceXQFTTS.html`.

available MXQuery release is the version number 0.6.0, and it lacks the relevance score computation (as also described in the list of missing features of such version); thus it could not obtain a complete conformance to the set of Full-Text Expressions.

Name	**Min. Conformance**	**FT Expressions**	**Opt. Features**
BaseX 6.3	Complete	Complete	Complete
MXQuery 0.7	Complete	Complete? (See notes)	Complete
Qizx 4.1	99.1%	Partial (with errors)	98.2%
Zorba 2.5	89.51%	Partial (with errors)	84.8%

Table 5.2: XQuery Full-Text Test Suite Result Summary for the selected XQuery engines

A final remark regards the actively maintained and developed projects: only two of the three active projects, namely the BaseX and the Zorba engines, implement the XQuery Full-Text extension, while eXist-db adopted the Lucene framework to provide full-text search, thus not adhering to the W3C standardized language.

From the above considerations and query engines comparison, the solely, at time of writing, engine that would entirely support the W3C standard Full-Text language is BaseX; it is available under an open source license and it is actively maintained and developed as confirmed by the recent releases and the frequent improvements introduced in the engine.

The BaseX engine also adopts an efficient encoding schema that is proven to provide an efficient and optimized XML node traversal; in particular, for the `descendant` and `ancestor` axes, the *pre/size/dist* schema, as previously described in Section 5.2.3 and further detailed in Section 5.3.2, it supports direct access at a constant time traversal for `parent` and `child` nodes. This aspect is crucial and it represents an important characteristic for an efficient evaluation of the `below` and the `near` axes as defined in the FleXy language: for the above reasons the BaseX engine has been chosen as the main framework for the implementation of the FleXy language.

5.3 FleXy implementation in BaseX

In this section the implementation of the FleXy axes on top of the BaseX query engine are presented: in particular the algorithms defined for processing the FleXy constraints, and those for computing the structure relevance score leveraging the index data structure of BaseX are described. The resulting XQuery engine, including both the Full-Text extension and the FleXy evaluation has been named Flex-BaseX [163].

In Section 5.3.1 the BaseX engine is presented, while Section 5.3.2 provides further details about the encoding and indexing schema adopted by BaseX. Details are provided regarding the efficient query evaluation process that is performed by the engine, such as query rewriting and query optimization.

Section 5.3.3 describes the *Below* and *Near* axes integration on the BaseX engine and the extension provided for the Query syntax interpreter. Finally the algorithms for the `below` and the `near` axes are described in Section 5.3.4 and Section 5.3.6 respectively, by taking into account the algorithms for the evaluation of the parametric variants of the axes. The algorithm that computes the `below` reverse axis named `above` is described after the `below` algorithm, in Section 5.3.5.

5.3.1 BaseX Overview

BaseX, as previously introduced in Section 5.2.3, is Java-based Open Source XQuery engine, supported by community driven development and available for commercial and ad-hoc support.

The BaseX query engine is leveraged by an efficient XML documents storage repository with efficient indexing and representation schema, on top of which all the major features of XML querying languages have been implemented. BaseX supports the execution of queries expressed in standard W3C languages such as XQuery, XPath, XQuery 3.0, XSLT; recently defined standards as XQuery/XPath Full-Text, XQuery Update Facility and EXPath modules are included in the standard BaseX distribution. Furthermore the engine exposes different functionality to interact with common data and document formats such as JSON, binary files, SQL and Databases connection and so on.

The BaseX engine has been designed to be used in a client/server infrastructure by the numerous API exposed by the BaseX server, or as a standalone engine; BaseX also includes a command line interface (CLI) and a full-featured Graphical User Interface (GUI).

The availability of a wide set of supported languages and extensions, coupled by the rich user interface that includes document navigation and visualization tools (such as a document explorer, an XML tree view and a XML map visualization) further extend the set of functionality provided by the engine, thus offering an improved XML querying experience to the BaseX users.

The BaseX engine that has been initially used for the FleXy implementation was the BaseX version 7.2 released on Match 2012; the FleXy implementation has than been updated following the subsequent releases of BaseX source code and index structure changes, until the 7.7 engine version (released on August 2013).

BaseX source code is available under the GitHub repository at `https://github.com/BaseXdb/basex`, while the list of previous releases are can be found at `https://github.com/BaseXdb/basex/releases`. Further details about the FleXy-BaseX implementation are provided in Section 5.3.3.

5.3.2 BaseX Data Structures

The BaseX query engine was initially presented in Grün et al. article [109] entitled "*Pushing XPath Accelerator to its limits*", where the authors presented an improved encoding schema for XML databases that outperforms the state of the art XML storage and querying techniques. The proposed XML storage model was based on a *Pre/Post*-order encoding schema variation as initially proposed by Grust [92] and subsequently extended by Grüen et al. work.

The two techniques are based on tree traversal approaches defined by Knuth [148] named *preorder* and *postorder*; in the *preorder* node traversal the root node is first visited, while a recursive preorder visit is applied to its child nodes, from the leftmost node to the rightmost one. Such node traversal is also addressed as the natural *document order* of the tree nodes, and it corresponds also to the order in which the XML elements are sequentially parsed. In the *Postorder* traversal, instead, a root note is visited only after all of its descendant nodes; also in postorder traversal the children visit is performed from left to right. In both traversal approaches a node *pre/post* value can be assigned by sequentially increasing such value every time a node is visited; both assignments and encoding computation can be performed in linear time by a single tree traversal.

The *pre*- and *post*- order values, and their correlation can be utilized to determine descendant and ancestor relationships between nodes in a tree; based on this observation Grust et al. [92] proposed a first *pre/size/level* encoding schema, that provides an efficient node traversal by identifying node set and node regions. This approach, implemented in MonetDB/XQuery engine, encodes in the the *size* parameter the number of descendant nodes, while *level* represents the number of levels between a node and the root node.

While the presence of the *level* attribute would be beneficial to the computation of the path relevance score as defined by the FleXy language, such *pre/size/level* encoding schema presents a high cost associated with the evaluation of the *parent* axis that requires a set of expensive operations with the *pre* and *level* values. The inclusion of a direct reference to the parent *pre* value in the encoding schema would require, in case of updates of the XML tree structure, a complete renumbering of all the *pre* values.

The BaseX encoding schema, instead, uses a *pre/size/dist* encoding for each XML node, where the *dist* attribute represents the relative distance between the node and its parent *pre* value, thus providing a direct access to the parent node, while minimizing the cost for XML tree updates. The *size* property is used in combination with a sequential storage of nodes to provide a sequential and constant time access to children and descendant elements.

A simplified example of `parent`, `descendant` and `child` axis evaluation as performed by the BaseX *pre/size/dist* encoding schema is presented in Algorithms 5.1, 5.3 and 5.2 respectively. A complete analysis and description of encoding and evaluation schemes based on *pre*- and *post*- order node traversal, including *pre/size/level* and *pre/size/dist*,

Algorithm 5.1 Axis.Parent(*node*: Node): Node

```
parent = node.$pre$ − node.$dist$
if parent > 0 then
  return new Node(parent)
else
  return null
end if
```

Algorithm 5.2 Axis.Child(*node*: Node): NodeSet

```
result := new NodeSet()
child := new Node(node.$pre$ +1)
while (child.$pre$ < node.$pre$ + node.$size$) do
  result.$add$(child)
  child := new Node(child.$pre$ + child.$size$)
end while
return result
```

are beyond the scope of this thesis work and can be looked up in Grust [92] and Grün [109, 152] works.

By leveraging the encoding schema previously described, the BaseX engine implements a set of indexes and data-structures to perform all the evaluations required to efficiently process the provided queries; The BaseX data-structures include:

Name Indexes that store and convert attributes names and tag names from a variable-length representation to fixed-size references;

Path Summary that represents a summary of the internal document structure for query rewriting and constraints optimization;

Values Index to map element attributes to the corresponding values;

Full-Text index that allows to efficiently process the XQuery Full-Text extension and its set of parsing and matching parameters.

Further details about the BaseX data-structures are available in Grün [152], where an analysis of optimization and rewriting techniques on *pre/size/dist* encoding schema, alongside the description of the BaseX low-level data handling are provided.

More details about the XQuery Full-Text implementation on top of BaseX indexes, as well as its complex and articulated processing strategies are available in [151].

Algorithm 5.3 Axis.Descendant(*node*: Node): NodeSet

```
result := new NodeSet()
desc := new Node(node.$pre$ +1)
while (desc.$pre$ < node.$pre$ + node.$size$) do
    result.$add$(desc)
    desc := new Node(desc.$pre$ +1)
end while
return result
```

5.3.3 FleXy Integration

As described in Section 5.3.3, the main characteristic of the BaseX data structures is the usage of *pre/dist/size* triplets to efficiently evaluate XPath and XQuery structural axes such as `descendant`, `parent`, `child` and `ancestor`.

The BaseX engine defines, for path-traversal processes, a set of constructs that facilitate the iterative process of path evaluation and the access to each node elements, attributes and value or textual contents stored in an indexed permanent database structure or stored in the main memory.

In Fig. 5.2 the BaseX class diagram of the path traversal evaluation is presented: the Java class naming scheme reflects the standard XQuery and XPath axis and their expression names. In particular, a *LocationPath* is defined as composed of one or more *AxisStep*, where an *AxisStep* consists of an *Axis*, a *nodeTest* and a set of optional *Expressions* as predicates [152]. As previously described, BaseX performs an iterative processing to perform path-traversals: the set of nodes are obtained from the current context node by the evaluation of the Axis expression; such nodes are then further filtered out by the evaluation of the NodeTest and the optional Expression, if present.

As depicted in the class diagram, each *Node* object stores its *pre/dist/size* values for tree evaluation, as well as the score variable to store the value of the Full-Text relevance evaluation (if performed).

By leveraging the standard Java language constructs, the evaluation of a path expression provides a generic node *Iterator*; in particular, for the *Axis* class, an *Iterator* is specified that returns all nodes that match such axis evaluation, and the method *Next()* allows an iterative evaluation and retrieval of such items.

With the goal of integrating the `below` and the `near` axes evaluation in the BaseX engine, the definition of both axes has been included in the class *Axis*, and their implementation has been provided by adopting the *pre/dist/size* encoding schema.

In Fig. 5.3 the extension of the class diagram in Fig. 5.2 is presented: it includes both the new axes defined by the FleXy language and the variable to store the structural relevance score.

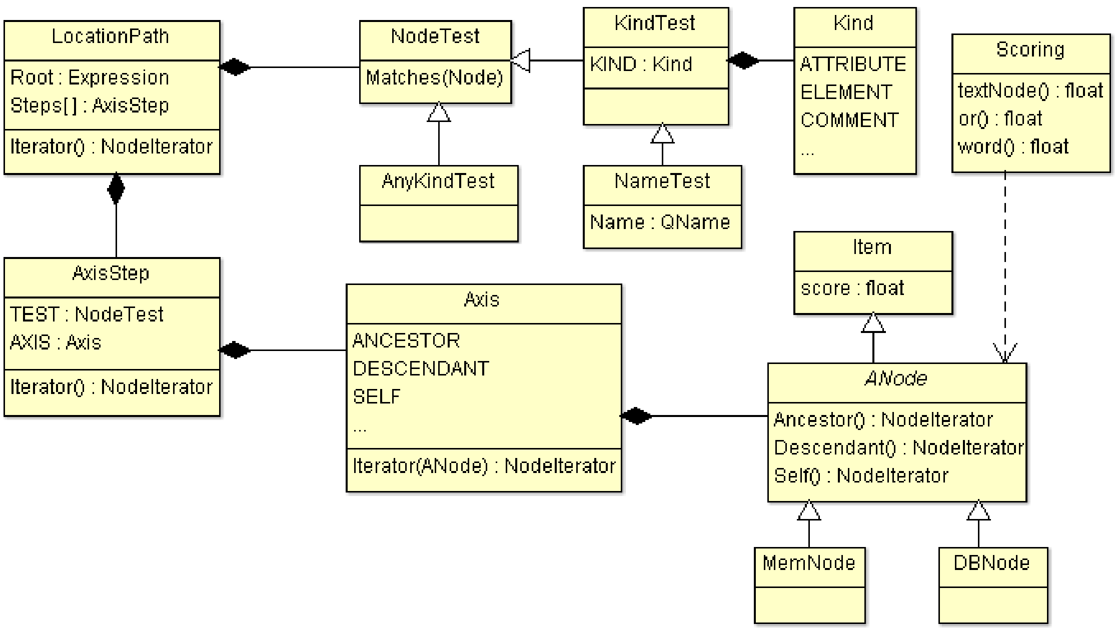


Figure 5.2: The Class Diagram for the path expressions part of the BaseX engine, it also includes the Full-Text extension, as visible from the `Item` class that contains the `score` variable.

The variable named `scoreST`, added to the *Node* class, stores the structural score computed by the `near` and `below` axes evaluation; such score is then used to acquire and to evaluate the ScoreVariable when formulated in a FLWOR clause expression. The structural-score has been added by following the same approach as performed in [151], where the Full-Text extension of XQuery has been integrated into BaseX, and each Node object has been extended with the full-text relevance score.

Regarding the axes that have been integrated in the BaseX code, a specific nomenclature has been adopted:

- the `below` and the `belowLimited` methods compute the *Below* axis matching and its parametric variant respectively;
- the `above` performs the below inverse axis evaluation, while the `aboveLimited` its parametric counterpart;
- the *Near* and its parametric variant axes evaluation have been integrated by the `near` and `nearLimited` methods: the `near` represents a simple invocation of the `nearLimited` function where a default value for the threshold parameter is supplied. Furthermore, a `NearBelowLimited` helper function has been added: it is used during the iterative evaluation of the `near` axis.
- Finally the class `ScoreStructure` contains the score computation algorithms for the three axes, as described in Section 4.5.

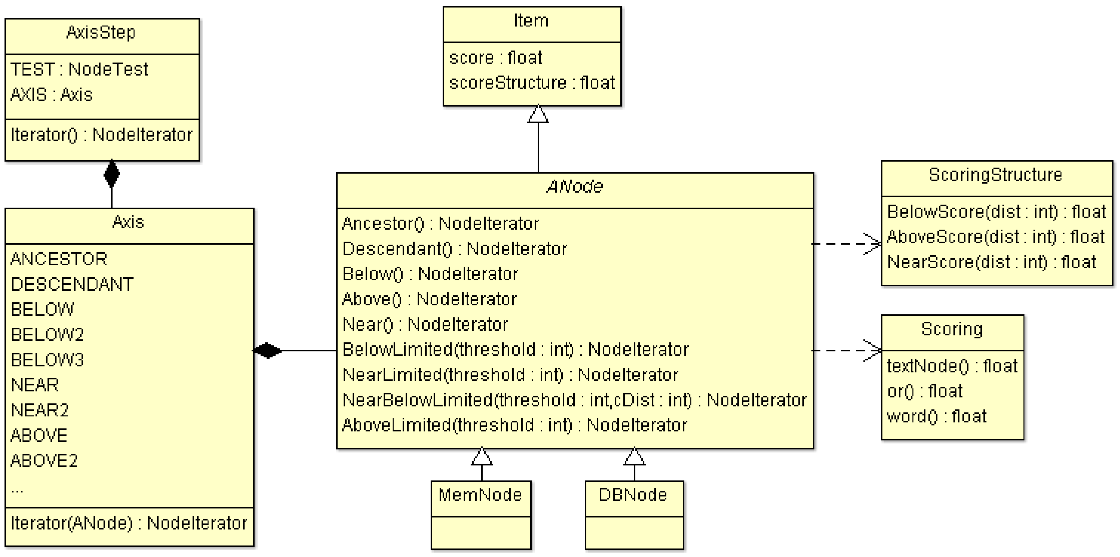

AxisStep
TEST : NodeTest
AXIS : Axis
Iterator() : NodeIterator
Axis
ANCESTOR
DESCENDANT
BELOW
BELOW2
BELOW3
NEAR
NEAR2
ABOVE
ABOVE2
...
Iterator(ANode) : NodeIterator
Item
score : float
scoreStructure : float
ANode
Ancestor() : NodeIterator
Descendant() : NodeIterator
Below() : NodeIterator
Above() : NodeIterator
Near() : NodeIterator
BelowLimited(threshold : int) : NodeIterator
NearLimited(threshold : int) : NodeIterator
NearBelowLimited(threshold : int,cDist : int) : NodeIterator
AboveLimited(threshold : int) : NodeIterator
MemNode
DBNode
ScoringStructure
BelowScore(dist : int) : float
AboveScore(dist : int) : float
NearScore(dist : int) : float
Scoring
textNode() : float
or() : float
word() : float

Figure 5.3: Class Diagram for the extended BaseX engine as implemented with the FleXy language extension: it includes the new axes evaluation, the structural relevance score and the score computation algorithms.

5.3.3.1 Language Interpreter Extension

The BaseX language interpreter, responsible of query checking and query execution planning, allows BaseX to parse, build and evaluate queries expressed using XQuery 1.0, XPath 1.0, XPath 2.0, XQuery/XPath Full-Text and the new constructs of the XQuery 3.0 language; such list of accepted languages only represents the main features of the system, where different XQuery modules, as described in the **XQuery Extensions** part of the Section 2.2.2, have been integrated in the BaseX query engine.

The BaseX language interpreter, thus, plays an important role during the query evaluation: each query expression is parsed and checked for its syntactic correctness, and the corresponding evaluation plan is build. Such BaseX component integrates the set of classes defined in the query engine to represent a generic FLWOR expression; such classes are depicted in Fig. 5.4.

The class diagram presents the main components that have been extended to integrate the FleXy language evaluation: besides the axes classes and the `QueryParser` class, also the `for` and the `let` clauses have been be modified. In particular the `score-structure` variable has been be defined and its value made available during a query evaluation; this allows the complete integration with the full set of FLWOR clauses, from the `for/let` to the `order by` and `return` clauses, including the upcoming `group-by` clause introduced in XQuery 3.0 language.

The most important modifications that have been made to the BaseX language interpreter are summarized in the following. For the complete list of changes performed on

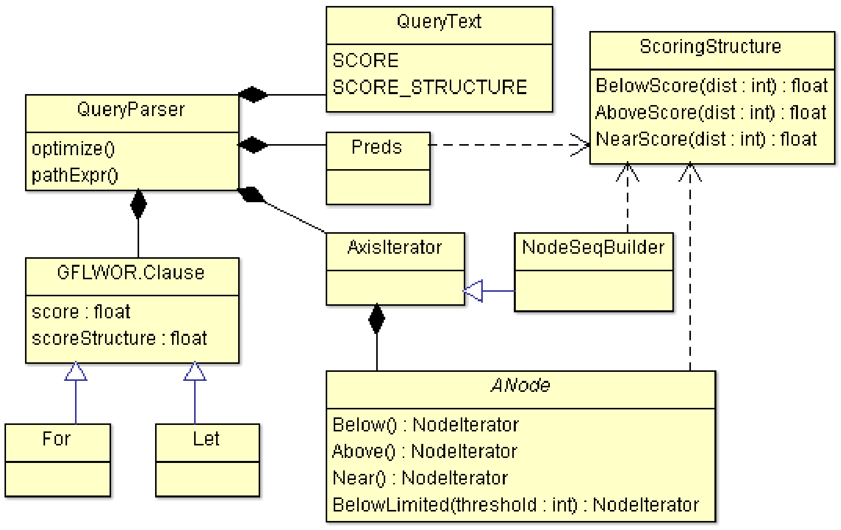


Figure 5.4: The BaseX class diagram for the language interpreter as extended with the FleXy axes evaluation classes.

the BaseX engine source code, please refer to the online repository of the FlexBaseX engine as specified later.

QueryParser.java and **QueryText.java** These two classes are responsible of parsing and checking for the query validity; the **QueryParser** class is also responsible of a first building of the query evaluation tree for the BaseX engine, the two classes have has been extended to allow the score-structure definition in the `let` and the `for` clauses.

GFLWOR.java This class represents the main core of a FLWOR expression (the class name stands for General FLWOR expressions), and it handles the variables definition and their computation during all the evaluation process; it has been extended to handle the `score-structure` processing and computation in the set of FLWOR clauses.

Let.java and **For.java**: the two classes, as their names suggest, represent the `let` and the `for` clauses evaluations: their are responsible for the evaluation of the expressions and the correct variable values binding in the `let` clause.

NodeSequenceBuilder.java and **Preds.java** are the two main classes where the structural scores aggregation is computed for query predicates and for *Node Sets*; the classes implement the aggregation as described in Section 4.5.3.

ScoreStructure.java this class implements the main scoring algorithms for the FleXy axes `below`, `near` and `above`, alongside their parametric variants.

5.3.4 The Below Axis

The `below` axis, as described in Section 4.4, computes the same *Node Set* returned by the `descendant` XQuery axis. For that reason the algorithm for accessing all the *below* nodes of a context node has been taken from the `descendant` axis one, while the algorithm for the parametric variant of the axis has been defined to take into account the threshold parameter specified by the query path expression.

In Algorithm 5.4 the iterative process of matching and scoring XML nodes for the `below` axis is presented: it describes the *NodeIterator.Next()* method invocation for the *NodeIterator* class that performs the actual axis node matching and iteration. The algorithm has been extended from the `descendant` axis by adding the scoring feature needed for the `below` axis evaluation.

Algorithm 5.4 Below.Next() : Node

```
Require: data (the database reference)
Require: contextPre (context node pre value)
Require: pre := contextPre (a cursor for the context node pre value)
1: pre := pre + data.AttributesSize(pre)
2: if pre = contextPre + data.Size(contextPre) then
3:    return null
4: end if
5: aNode := new Node(pre)
6: aNode.scoreStructure := BelowScore(NodeDistance(contextPre, pre))
7: return aNode
```

In Algorithm 5.4 the function `data.AttributesSize()` returns the size of the given element's attributes set, thus allowing to consider only the child and descendant XML elements skipping elements attributes. The function `Data.Size()` returns the element size, thus obtaining the size of the element given as a parameter, in line 2 this allow to iterate only over the actual set of elements contained in the `contextPre` element, thus retrieve only the descendants elements of the context node without accessing other nodes.

The functions `BelowScore()` and the `NodeDistance()` applied to `contextPre` and `pre` at line 6 performs the structural relevance score computation for the `below` axis: where the `contextPre` and `pre` identifiers represents the *parent* node (or *context* node) and a *descendant* node respectively. The computation is performed by first retrieving the distance from the context node and the target node, and then by applying the scoring function described in Section 4.5.1. An observation related to the `NodeDistance()` function should be made: due to the data structure implementation of BaseX, where for each node only the triplet `pre/dist/size` is stored and no information about the depth of a node is provided, the node nesting must be kept for depth computation. In particular that required to implement a stack of nodes *pre* values to exactly identify, for each target node, its depth level, and to compute the distance from the context node.

The `NodeDistance()` function described in the Algorithm 5.4 and the algorithms presented in the following sections, represents a simplified variant of the actual code implemented in the BaseX engine: the `NodeDistance` computation would require, in fact, a second iteration over all the descendant nodes of the context node down to the current target node, thus requiring a double elements traversing procedure. The implemented *distance* computation has been inlined with axis `below` computation: an efficient *dist*-values stack is implemented to keep track of the node hierarchy while traversing the set of descendant nodes. This technique is required given the *pre/dist/size* schema of BaseX and has been adopted for all the following `above` and `near` axis matching and score computation.

Algorithm 5.5 BelowLimited.Next() : Node

```
Require: data (the database reference)
Require: contextPre (context node pre value)
Require: threshold (the threshold parameter)
Require: pre := contextPre (a cursor for the context node pre value)
 1: contextSize := contextPre + data.Size(contextPre)
 2: pre := pre + data.AttributesSize(p)
 3: while nodeDistance(contextPre, pre) $\geq$ threshold and pre $\leq$ contextSize do
 4:   skip node P and its descendants
 5:   pre := pre + data.size(pre)
 6: end while
 7: if pre = contextSize then
 8:   return null
 9: end if
10: aNode := new Node(pre)
11: aNode.scoreStructure := BelowScore(NodeDistance(contextPre, pre), threshold)
12: return aNode
```

Regarding the `belowLimited` axis implementation, the Algorithm 5.5 describes the evaluation performed: the difference between the `Below` and the `BelowLimited` is the threshold parameter called `threshold` and the slightly different `BelowScore()` function invocation, where a second parameter allow the score computation to take into account also the threshold. This feature allows to define fine grained scoring functions for the axis by computing a structural score with a more smoothing factor when the distance between the context node and the target reaches the specified threshold. The currently provided BaseX implementation provides the score computation as defined in Section 4.5.1. In future works a more configurable axis evaluation is envisioned, where the user, other than giving a threshold, may choose the preferred scoring function; providing such feature, unfortunately, would compromise the aimed simplicity of the FleXy language and should be further investigated.

5.3.5 The Above axis

To more efficiently integrate the `below` axis evaluation in the query engine, thus leveraging the query rewriting and optimization features that allow BaseX to efficiently execute queries, the `below` inverse axis evaluation has also been implemented: the scoring function applied to the `above` evaluation is the same described in Equation (4.9).

The algorithm for the `above` axis evaluation is described in Algorithm 5.6: the BaseX *pre/dist/size* encoding scheme allow to directly access the parent node by simply subtracting, from the current node *pre* identifier its distance *dist* from the parent node. As mentioned before, the node *distance* stored by BaseX corresponds to the difference between a parent and child *pre* values, and should not be confused with the number of arcs that separates two tree nodes.

Algorithm 5.6 Above.Next() : Node

```
Require: data (the database reference)
Require: contextPre (context node pre value)
Require: pre := contextPre (a cursor for the context node pre value)
1: pre := pre - data.Dist(pre)
2: if pre = -1 then
3:    return null
4: end if
5: aNode := new Node(pre)
6: aNode.scoreStructure := AboveScore(NodeDistance(pre, contextPre))
7: return aNode
```

Lines 1 and 2 the actual identification of the parent node is performed, the check avoids to retrieve a non-existent node in the case of reaching the root tree node. In line 6, instead, the score computation is performed by the use of the `AboveScore()` function coupled with the previously defined `NodeDistance()` function. The computation performed by the `AboveScore()` function assigns a path relevance score given the distance between the context node and the current target node by applying the function defined in Section 4.5.1. The definition of two different functions, the `BelowScore()` and the `AboveScore()`, that compute the same scoring is due to provide users and future extensions of the FleXy language, an easy way to define and implement different scoring algorithms given the traversed path that connects a context node to a target node. In Algorithm 5.7 the evaluation performed for the axis `above`, with a threshold parameter, is provided: the same observations apply as for the `belowLimited` score and distance computation.

Algorithm 5.7 AboveLimited.Next() : Node

```
Require: data (the database reference)
Require: contextPre (context node pre value)
Require: threshold (the threshold parameter)
Require: pre := contextPre (a cursor for the context node pre value)
1: pre := pre - data.Dist(pre)
2: if pre = -1 or NodeDistance(pre, contextPre) $\geq$ threshold then
3:     return null
4: end if
5: aNode := new Node(pre)
6: aNode.scoreStructure := AboveScore(NodeDistance(pre, contextPre), threshold)
7: return aNode
```

5.3.6 The Near axis

The `near` axis implementation required to identify, for a given *context* node, the *closest* nodes corresponding to *target* nodes. The `near` algorithm has been developed to incrementally evaluate close nodes starting from the *context* node descendants, and proceeding with ancestor (and their descendants) nodes. The evaluation handles the specified maximum allowed distance from the *context* node and the set of *target* nodes by sequentially iterating over the *context*'s ancestors and by repeating the *target* nodes identification and scoring. As visible in Algorithm 5.8, the set of nodes identified as *target* nodes can contain repeated elements: such elements are filtered out in a subsequent BaseX evaluation, where only the best candidates are kept as the resulting *Node Set*.

The idea behind the algorithm that evaluates the `near` axis is to leverage a function similar to the *BelowLimited* axis evaluation to provide an iterative retrieval of the candidate *near* nodes. Given the possibility to provide different scoring functions between the `below` and the `near` axis evaluation, a new algorithm called `NearBelowLimited` is used instead of `BelowLimited`. The new algorithm differs from the `BelowLimited` only by the scoring function used: a `NearScore(distance)` is applied during the `NearBelowLimited` that implements the `near` axis score computation as described in Section 4.5.2.

The process performed in the *Near* matching algorithm for a given context node is composed by an iterative two step evaluation procedure: as a first instance the iterator retrieves all the *descendant* nodes that can be reached by the given threshold, thus using the same *BelowLimited* matching algorithm. The second evaluation step is performed when all the candidate descendants nodes have been visited: a temporary context node is set to the parent of the actual context node and the evaluation is carried on by matching all the descendants node from the newly identified context node. In this case the *NearBelowLimited* is evaluated by decreasing the threshold value, given the change of the current context node. The iterative process is performed until the threshold reaches a zero value, or the temporary context node is does not have a parent, thus the root element node has been reached. In particular the *Near* iterator wraps a second iterator that iterates through the node candidates retrieved by the *NearBelowLimited* iterator evaluation. The `NearBelowLimitedIterator` provides a third parameter, other than the contextNode and the threshold value, that allow to adjust the distance computation by taking into account an additional path traversal performed outside its evaluation.

Algorithm 5.8 Near.Next() : Node

```
Require: data (the database reference)
Require: contextPre (context node pre value)
Require: threshold (the threshold parameter)
Require: tempPre := contextPre (a cursor for the context node pre value)
Require: backSteps The number of backward steps performed, initially 0
Require: iterator (The current node iterator, initially set to the NearBelowLimited
    iterator: NearBelowLimitedIterator(contextPre, threshold))
 1: if not iterator.hasMore() then
 2:    while threshold $\geq$ 1 and not iterator.hasMore() do
 3:       Track of previous pre node value
 4:       prevPre = tempPre
 5:       Move to the parent context-node
 6:       tempPre = tempPre - data.Dist(tempPre)
 7:       Update the correct threshold, to match the context node change
 8:       threshold := threshold $-1$
 9:       backSteps := backSteps $+1$
10:       iterator := NearBelowLimitedIterator(tempPre, threshold, backSteps, prevPre)
11:    end while
12: end if
13: return iterator.Next()
```

In Algorithm 5.8 at line 1 the current node iterator, initialized during the construction of the `Near` axis to a `NearBelowLimitedIterator` iterator, is checked if more nodes are available, if not (thus the current limited descendant axis have been completely traversed), the temporary context node is replaced by its parent (line 6) and the threshold updated to reflect such context change (line 8). Line 4 the `prevPre` variable allow to track the previous child element and skip the node subtree that has already been traversed, this allow to avoid duplicated node retrieval. Finally, in line 9 the distance correction is computed to account for the parent traversal execution; such `backStep` is then summed during the actual distance computation performed in `NearBelowLimitedIterator`.

6 Evaluations and FleXy User-Case

In this section the evaluation of the Flex-BaseX engine are presented, along with a preliminary user-case of such engine applied to Patent Retrieval tasks.

In Section 6.1 the performed evaluations of the `below` and `near` axes are provided; the experiments have been conducted to compare the FleXy axes performances against standard XQuery axes.

Finally in Section 6.2 a preliminary implementation of a Patent Retrieval tool, named PatentLight is described: it uses the FleXy language implemented in the Flex-BaseX engine to provide an easy categorization of patents by leveraging the XML format of patents and their internal hierarchy.

6.1 Evaluations

In this section the evaluations performed to examine the efficiency of the flexible axes described in the previous sections are presented.

As described in Section 5.3, both the `below` and the `near` axes have been implemented on top of the BaseX XML Database system, where the XQuery Full-Text language has been extended to include: the evaluation of the `below` and `near` flexible axes, the computation of the structural relevance score and the aggregation of structural scores when a *branching query* is provided.

In Section 6.1.1 the environment used to perform the evaluations is presented, alongside the tools and the evaluation strategies adopted; while the collections and their structure are described in Section 6.1.2.

In the following sections 6.1.3 and 6.1.4 the performed evaluations and the obtained results, rispectively for the `below` and the `near` axes, are presented as well as the description of the executed queries.

In particular the evaluations have been performed to evaluate two main aspects of the FleXy language as implemented on top of the BaseX query engine: a first evaluation is carried to measure the overhead introduced by the axes and the structural relevance score computation, while the second aspect evaluated is a comparison of the `below` and the `near` axes against their XQuery counterparts.

In particular for the `below` axis the evaluations are executed by comparing the evaluation time of the flexible axis with the `descendant` standard XQuery counterpart. Concerning the `near` axis evaluation, instead, no standard XQuery axis counterpart could be identified due to the innovative nature of the proposed flexible axis. The evaluations have been carried by comparing the `near` axis evaluation against the set of equivalent queries expressed with standard XQuery constraints and merging them with the `union` operator.

Of course, given the novelty of the structural scoring approach, the FleXy axes evaluations are expected to provide reasonably higher evaluation times, for example in comparison to the `descendant` axis evaluation.

Finally Section 6.1.5 concludes the evaluations and summarizes the obtained results.

6.1.1 Environment

The environment where the new flexible axes have been tested is composed of an Intel $i7$ 3GHz with 4GByte of RAM; the Operative System is Ubuntu/Linux Server LTS version 12.04 64Bit with kernel $3.2.0 - 49$, mounting a Saegate 500 GByte S-ATA Hard Disk at 7200 rpm.

The BASEX engine, and the FleXy integration, have been developed using the Java language and, thus, the server is also equipped with the Java Virtual Machine (JVM) version number $1.7.0 - 25$ and using the OpenJDK Runtime Environment version 2.3.10.

The evaluations have been performed by using the Server/Client architecture offered by the BASEX engine, thus avoiding engine start-up and configuration loading overheads. Furthermore the BASEX server has been configured to use the whole amount of available main memory by using the Java Virtual Machine parameter `-Xmx=4G`.

The FLEX-BASEX engine has been initially implemented on top of the 7.2 release of the BASEX engine; subsequently the flexible axes, the structural score computation and the aggregation algorithms have been updated with the latest BASEX releases. The evaluations performed have been executed with the latest BASEX engine code-base, updated at the 7.7 development version[1].

During each evaluation, the set of queries defined for each task have been executed 7 times each, by using the BASEX parameter `RUN` that allows to specify the number of times a given query must be performed. The average execution time is then stored for the subsequent analysis. A note should be made on the reported timings: BaseX splits a query execution in four different stages: named *Parsing*, *Compiling*, *Evaluating* and result *Printing*. The performed evaluations take into account all of these aspects, in particular the most relevant timing, and the compared one, is the *Evaluation* execution

[1] The BaseX 7.7 development version refers to the latest development branch of the query engine released on 2013-07-10, the source code is available from the BaseX online repository at `https://github.com/BaseXdb/basex`.

step, while the *Printing* step has been ignored given its solely dependency on the number of retrieved results. Also the *Parsing* and the *Compiling* step timings have been disregarded given the fact that they rely on the BaseX query parsing and query execution planning; for each query, in fact, the internal query optimization has been checked to not rewrite the user query, but to exactly execute the provided query.

Moreover, due to the nature of the BaseX indexing system that caches both queries and the opened databases, the evaluations have also been performed by unloading the BaseX server system between each run to avoid caching. Such techniques, as shown in the subsequent sections, do not show noticeable differences during the execution of each query.

6.1.2 Data Collections

The set of collections used to evaluate the Flex-BaseX engine have been taken from the INEX DataCentric collection, in particular from the IMBD (Internet Movie DataBase) XML collection. Performance tests have been executed with an increasing number of documents and collection size to verify the overhead introduced by the flexible axis evaluation in comparison with standard XQuery axes constraints.

In Table 6.1 an overview of the different collections built from the INEX IMDB collection is reported: the collection size, the number of documents and the number of XML elements are shown. Such collections have been used to evaluate both the `below` and the `near` axes evaluations and score computations, with and without expressing a desired threshold parameter. The last collection *C9*, unfortunately has shown some difficulties during the evaluation in the BaseX engine. As further shown in the evaluation sections, the collection size and the number of retrieved results during the evaluation of the `below` and `descendant` queries exceeds the 4Gb of memory allocated for the server engine, and thus it requires the system to use an on-disk temporary swap storage. This issue highly influences the execution time of the affected queries resulting in a partial system slowdown that prevents the effectiveness of the comparison.

Name	**Documents**	**Size** (Mb)	**XML Elements**
C1	29,487	137.20	2,232,122
C2	58,974	272.73	4,556,070
C3	117,947	545.89	9,060,745
C4	235,893	1,091.06	18,237,678
C5	471,785	2,179.22	36,529,019
C6	629,046	2,924.00	48,751,225
C7	754,849	3,511.34	58,021,829
C8	943,561	4,359.34	71,606,796
C9	1,887,121	8,710.38	143,287,407

Table 6.1: Summary of the INEX-IMDb collections.

A second collection of documents named *XMark* has been generated by using the XMLGen tool that is the utility, from the XMark project[2] , that builds XML documents for testing purposes. Such set of documents has been used to evaluate and compare the `below` and the `descendant` axis: the documents total sizes are shown in Table 6.2, but despite the smaller document sizes, if compared to the latest collections in the previous table, they show a higher number of elements and the document hierarchy spans to a maximum of 13 depth levels, in contrast to the maximum depth of 5 reached by the elements in the IMDB collection.

Name	**Size** (Mb)	**XML Elements**	**XMark -f**
xmarkC1	116.20	3,355,270	1
xmarkC2	1,030.43	29,730,814	10
xmarkC3	2,016.21	58,177,822	18
xmarkC4	3,024.12	87,256,566	37

Table 6.2: Summary of the XMark document collections and the XMLGen tool `-f` parameter used to generate the document.

Furthermore an ad-hoc document collection, built with a simple but at the same time articulated node hierarchy, has been defined to provide a first evaluation and comparison for the `near` axis evaluation; such collection has been named *CollectionNodeE*, given the root node labeled `NodeE` and the presence of multiple nodes that share such label in different positions of the XML tree. An example document in the collection is presented in Fig. 6.1, where the XML elements (drawn as circles), the textual contents (drawn as rectangles) and their *pre/dist/size* encoding numbers, as assigned by the BaseX engine, are elicited. In particular the collection has been generated from a modified version of the XMLGen tool where the generated document structure has been defined to be like the one shown in Fig. 6.1, in particular each node in the structure has randomly repeated from 1 to 4 times. Such decision has been taken after comparing the IMDB document structure and the one generated in the XMark documents: the node hierarchy of such collections does not contain a set of repeated node label and a node distribution that would avoid the BaseX query optimized strategies: the *CollectionNodeE* has been designed to take into account such features. In summary the collection is composed of 400200 different XML documents, for a total size of 1.62 GBytes; the number of XML elements is $31,980,739$.

6.1.3 The Below evaluation

The first set of evaluations are related to the `below` axis: a set of queries have been defined to allow a correct comparison between the `below` axis and the standard XQuery

[2] The XMark tool is an XML benchmark that provides an XML generator software that built arbitrary sized documents, the tool is available at `www.xml-benchmark.org`.

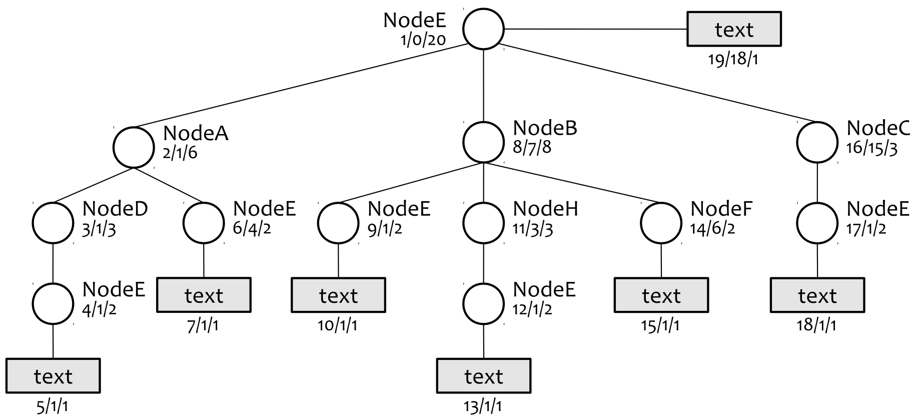


Figure 6.1: *CollectionNodeE*: an example document in the collection.

`descendant` axis. The queries take into account two aspects of the Flex-BaseX engine: query optimization and engine caches. Before actually executing a query, the BaseX engine performs a query optimization and constraints rewriting based on some collection statistics computed at index time as described in [152]. One of the most important rewriting rules implemented in BaseX is the identification, by using a structure summary built at indexing time, of unique paths in the collection, thus if the path expression `nodeA//nodeC` is provided and the `nodeC` only exists with the hierarchy `nodeA/nodeB/nodeC`, then the query is internally rewritten by using only the `child::` axis traversal, thus allowing a faster bottom-up or top-down evaluation for the resulting query `nodeA/child::nodeB/child::nodeC`.

For this reason the evaluation queries have been designed to address elements that would not trigger the query rewriting process, thus requiring the complete execution of the `descendant` axis traversal when specified. To verify that both queries with the `below` and the `descendant` axes are not treated differently in the BaseX due to the internal query rewriting, further checks on the query plan provided by the BaseX engine have been performed. An example of the set of performed queries has been provided in Listing 6.1, the queries in this example match and retrieve the same set of nodes, this to avoid to compute evaluation timings that depend on the number of actually matched items.

```
person/descendant::name
person/below::name
person/below3::name
person/below4::name
person/below5::name
```

Listing 6.1: Set of queries executed for the `below` axis evaluation for the IMDB collection.

The queries are executed on all the available collections and their timings used for the evaluation. As described in Section 6.1.1 all the queries have been executed 7 times for each collection and the average execution time acquired. It is important to notice that the `name` element is located, for each document in the IMDB collection, under three different paths:

- `person/name`;
- `person/overview/alternate_names/name`;
- `person/overview/nicknames/name`.

For this reason the BaseX optimization algorithms are not able to rewrite the provided query `person/descendant::name` to a simpler expression that makes use, as an example, of the `child` axis.

In Table 6.3 the number of retrieved results, and the size of the returned data are shown for each collection given the reference `descendant` query. In this case the retrieved elements are returned by the system without explicitly involving any XQuery FLWOR expression, but the structural score is always computed whenever a FleXy axis is specified. No score re-ranking, nor score display have been added in the set of evaluated queries to the aim of providing a comparable timing results, where no other computations are required with the exception of the solely axis evaluation.

Collection	**Query**	**Results**	**Size** (Mb)
C1	`person/descendant::name`	17907	0.50
C2	`person/descendant::name`	35858	1.01
C3	`person/descendant::name`	71833	1.99
C4	`person/descendant::name`	143473	3.98
C5	`person/descendant::name`	285564	7.91
C6	`person/descendant::name`	381438	10.57
C7	`person/descendant::name`	477722	13.25
C8	`person/descendant::name`	637137	17.67
C9	`person/descendant::name`	1274842	35.07

Table 6.3: Number of results retrieved for each IMDB collection by the `descendant` axis evaluation.

In Table 6.4 the evaluation timings obtained from the execution of the query set are presented: as expected the `below` axis evaluation performance results with higher timings than the counterpart `descendant` axis. given the added costs for the computation of the structural relevance score.

Regarding the threshold variants of the `below` axis, the behaviour shown reflects the lower number of nodes that have to be visited by the `below3`, and the `below4` axes, thus reducing their execution time, while retrieving the same set of results as discussed previously.

Collection	Below	Below3	Below4	Below5	Descendant
C1	0.16	0.10	0.15	0.16	0.12
C2	0.31	0.16	0.28	0.32	0.23
C3	0.60	0.28	0.51	0.62	0.44
C4	1.14	0.52	0.98	1.18	0.86
C5	2.33	1.00	1.93	2.39	1.73
C6	3.16	1.34	2.55	3.19	2.30
C7	3.91	1.61	3.14	3.91	2.78
C8	5.18	2.17	4.18	5.17	3.69
C9	13.77	8.84	10.45	14.11	10.56

Table 6.4: The `below` and the `descendant` axes evaluation timings (in seconds).

The `below5` variant, instead provides equal, or in some cases worst timings than the `below` axis evaluation: this aspect is motivated by the structure of the documents in the IMDB collection, where the maximum tree height, thus the maximum distance of a leaf node from the root node, is exactly 5. The evaluation of `below5`, in fact, has been provided in the set of relevant evaluations exactly to evaluate the overhead introduced by the threshold evaluation of the *BelowLimited* algorithm, in comparison to the `below` axis evaluation where no threshold has to be checked at each node visiting.

In Fig. 6.2 the comparison of the evaluation timings between the `descendant`, and the `below` axes variants are shown, furthermore in Fig. 6.3 the percentage of performances improvement, or decrease, are illustrated as the ratio of the flexible axis timings over the `descendant` axis evaluation timing.

As shown from the graph in Fig. 6.3, the `below3` axis evaluation allows to obtain a reduction of the computation times on average of the 40% if compared to the `descendant` axis evaluation. This fact is motivated by the great number of nodes that are filtered out from the sequential evaluation performed by the BaseX engine. Regarding the `below` and `below5` axes their evaluation requires, on average, a 40% more time than the `descendant` counterpart, this is caused by the node distance and the score computation as performed by both axes, and by the threshold evaluation of the parametric variant of the `below` axis.

The slight performances variations are related to the nodes distributions in the collections: greater improvements are shown for collections having higher number of nodes at deeper levels, while lower improvements (or even decreases) are obtained if all the nodes are in average distributed at the same level of the applied threshold. The case of lower performances when all the nodes are at the same level of the threshold, thus no nodes are filtered out given their distance from a context node, is showed for the `below5` axis evaluation in the previous graph.

Another important behavior that can be observed from the graph in Fig. 6.3 is that the performances measured for the collection *C9* show that all the `below` axis variants tend

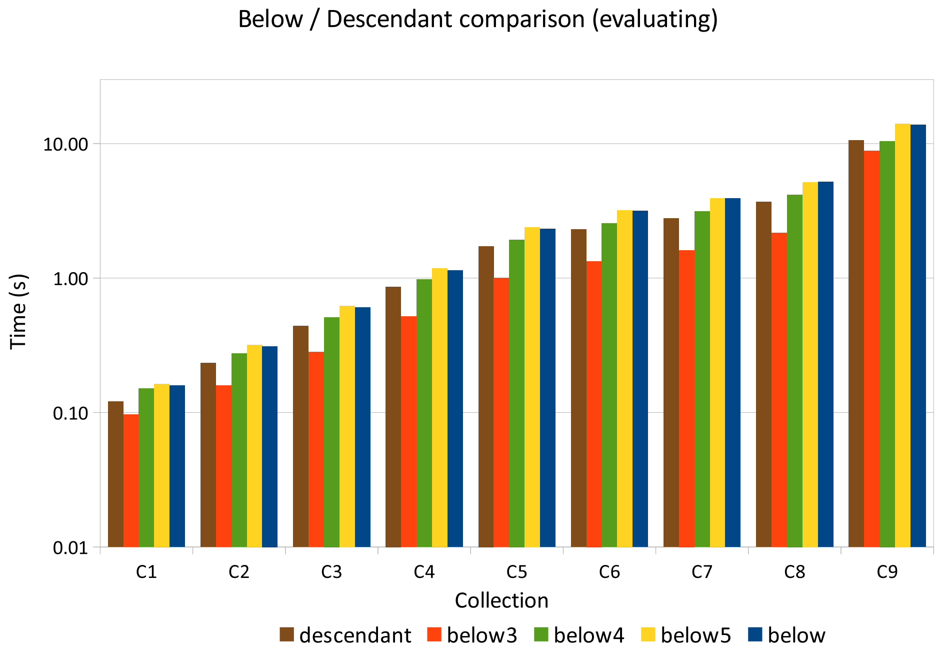


Figure 6.2: Evaluation timings comparison between the `descendant` and the `below` axis set.

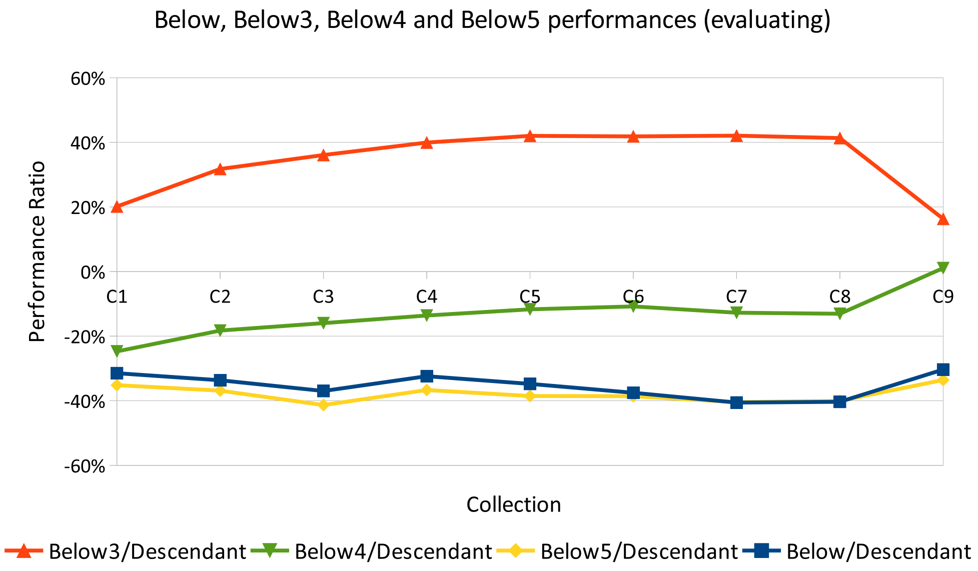


Figure 6.3: Performance ratio for the `below`, `below3`, `below4` and the `below5` axes evaluation as compared to the baseline `descendant` axis evaluation on the IMDB collection.

to perform as the standard `descendant` axis. This could be, at a first glance, addressed as a strange behavior of the system, where linear evaluation timings have been measured for all the previous collections. Further and deep investigations, instead, confirms that this issue is related to the size of the handled collection and the number of visited and matched nodes (in this case 1274842) that consumed the available main RAM memory of the server and forced both the Java Virtual Machine to frequently execute its Garbage Collector routines, and the SYSTEM BASEX system to use a temporary on-disk storage to complete the evaluation. Such finding is also confirmed by the number of nodes that each axis visited during its evaluation: given a linear growth of the number of visited nodes from the previous collections, the evaluations of all the axes performed on the *C9* collection shows, instead, a high fluctuation of the evaluation timings, including the `descendant` axis evaluation. For that reason the evaluations performed on the collection *C9* would not be taken into account, given their incomparable and unstable timing results.

As a subsequent evaluation the *Compiling* and the *Parsing* tasks have been compared between the set of `below` axes and the `descendant` axis. In Fig. 6.4 and Fig. 6.5 both the *Compiling* and *Parsing* timing graphs are shown; their exact timings are shown in Table 6.5 and in Table 6.6 respectively. As the graphs show, no relevant overheads are shown in the two tasks for the set of `below` variant axes in comparison with the `descendant` axis compiling and parsing timings.

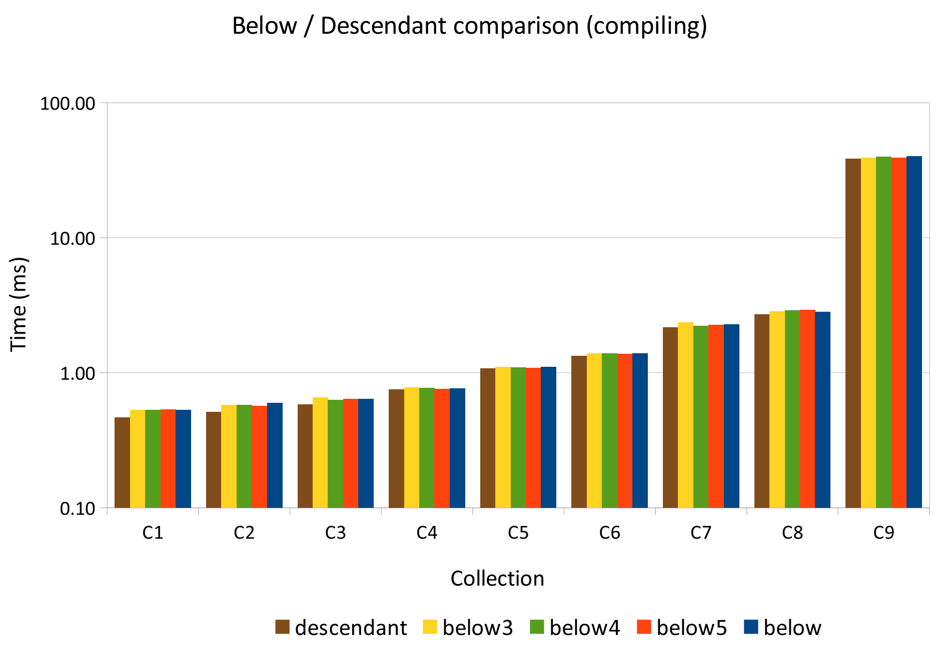


Figure 6.4: Compiling time comparison between the `descendant` axis and the set of `below` axes.

A second set of evaluations has been conducted with the XMark collection reported in

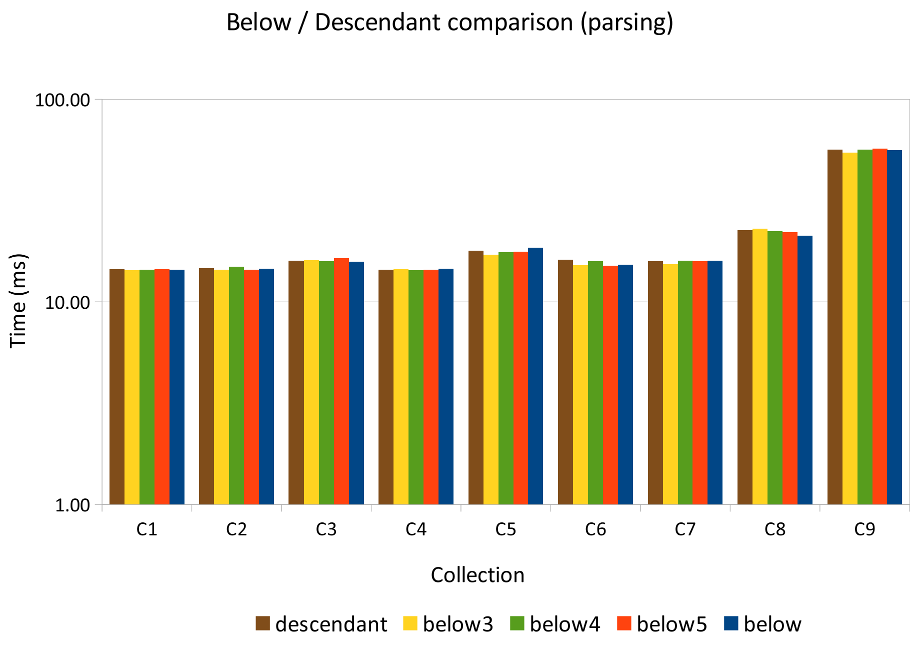


Figure 6.5: Parsing time comparison between the `descendant` axis and the set of `below` axes.

Table 6.2. In this case the higher depth levels of the documents if compared to the IMDB collection, would represent a good benchmark also for the threshold variants of the `below` axis. The set of queries executed by the FLEX-BASEX engine are presented in Listing 6.2, where the `descendant` query (at line 1) represents the baseline timing of the subsequent `below` queries. The set of queries, also in this case, has been designed to avoid the BASEX optimization techniques and, thus, provide a correct comparison between the FleXy axes and the standard `descendant` axis.

The set of matching tags in the XMark collection are found in different paths: a first `name` tag is under the path `site/people/person/name`, a second tag labeled *name* can be found for each item under the hierarchy `site/regions/`*X*`/item/name` where `X` represents one of the XMark defined regions such as *Africa*, *Asia*, *Europe* and so on. A third tag `name` is found under the `site/categories/category/name` path. Given this triple location for the `name` tag, the query optimizer could not effectively rewrite the `descendant`, nor the `below`, query into a simpler query evaluation plan.

```
1  site/descendant::name
2  site/below::name
3  site/below4::name
4  site/below5::name
```

Listing 6.2: Queries executed for the evaluation of the `below` and the `descendant` axes comparison for the XMark collection.

Collection	**Below**	**Below3**	**Below4**	**Below5**	**Descendant**
C1	0,53	0,53	0,53	0,53	0,46
C2	0,60	0,58	0,58	0,57	0,51
C3	0,64	0,66	0,63	0,64	0,58
C4	0,77	0,78	0,77	0,76	0,75
C5	1,10	1,11	1,09	1,08	1,08
C6	1,39	1,38	1,39	1,38	1,33
C7	2,29	2,36	2,22	2,27	2,17
C8	2,82	2,84	2,91	2,91	2,71
C9	40,11	38,99	39,71	38,98	38,58

Table 6.5: The `Below` and `Descendant` compiling timings (ms)

Collection	**Below**	**Below3**	**Below4**	**Below5**	**Descendant**
C1	14,36	14,38	14,31	14,43	14,45
C2	14,52	14,86	14,37	14,36	14,58
C3	15,77	15,79	16,01	16,36	15,91
C4	14,50	14,32	14,48	14,35	14,35
C5	18,42	17,52	17,01	17,63	17,80
C6	15,17	15,83	15,13	15,03	16,09
C7	15,88	15,90	15,29	15,84	15,80
C8	21,16	22,32	22,88	22,02	22,49
C9	55,94	56,26	54,36	56,97	56,28

Table 6.6: The `Below` and `Descendant` parsing timings (ms) for the IMDB Collection.

Collection	**Query**	**Results**	**Size** (Mb)
xmarkC1	`site/descendant::name`	50180	1.44
xmarkC2	`site/descendant::name`	443898	12.79
xmarkC3	`site/descendant::name`	868500	25.02
xmarkC4	`site/descendant::name`	1302750	37.40

Table 6.7: Number of results retrieved for each XMark collection by the `descendant` axis evaluation.

In Fig. 6.6 the evaluation timings are shown for the tests executed against the XMark collections. As expected the `below4` evaluation, given the threshold parameter, is evaluated in less time than the other axes for the first three XMark collections, while its performances decrease in the latest *xmarkC4* collection.

This behavior also affects the `below5` and the `below` axes that increasingly reduce their performances: differently from the IMDB collection where the degradation stabilized around the 40%, in the XMark test the `below` axis required at most the double of the time of the `descendant` axis to finish the evaluation as shown in Fig. 6.6. This aspect is caused by the data structure (more specifically an array of integer values used as a stack) used in the Flex-BaseX engine to maintain the current node deep level while iterating trough the set of candidate nodes: the iterative process implemented in BaseX build a new Below axis iterator for each document in the collection and while the IMDB collection is composed of a great number of medium sized XML documents the XMark test is composed of a single huge XML document and the Below iterator is built once. This impacts on the memory allocation of the integer value stack and the performances and the scalability of the entire evaluation of the flexible axes highly depends on the overhead introduced by such stack data structure: improvements have been observed if the standard and general purpose Java Stack class implementation is substituted with a tailored and ad-hoc stack solution, like the one actually developed for the Flex-BaseX engine.

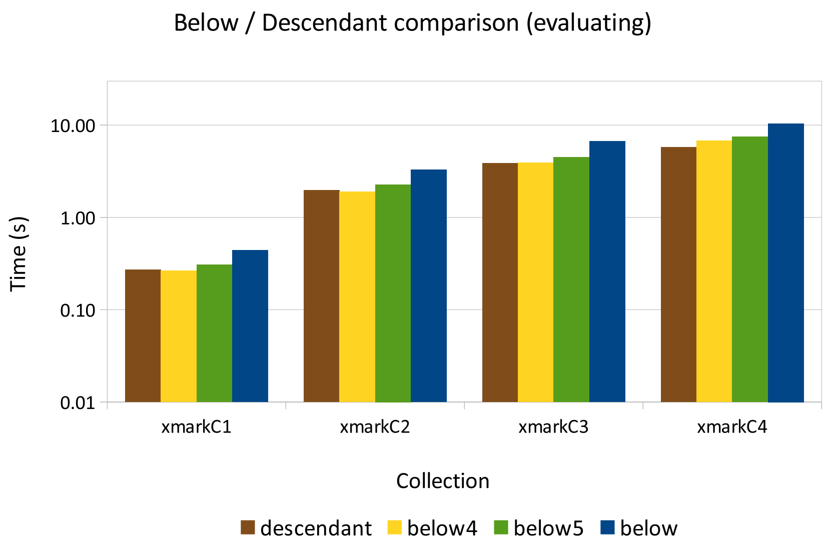


Figure 6.6: Evaluation timings comparison between the `descendant` and the `below` axes set for the XMark collections.

Collection	Below	Below4	Below5	Descendant
xmarkC1	0.44	0.26	0.31	0.27
xmarkC2	3.28	1.89	2.27	1.96
xmarkC3	6.64	3.93	4.51	3.87
xmarkC4	10.41	6.80	7.48	5.75

Table 6.8: XMark evaluation timings for the `below`, `below4`, `below5` and `descendant` axes (timings in seconds)

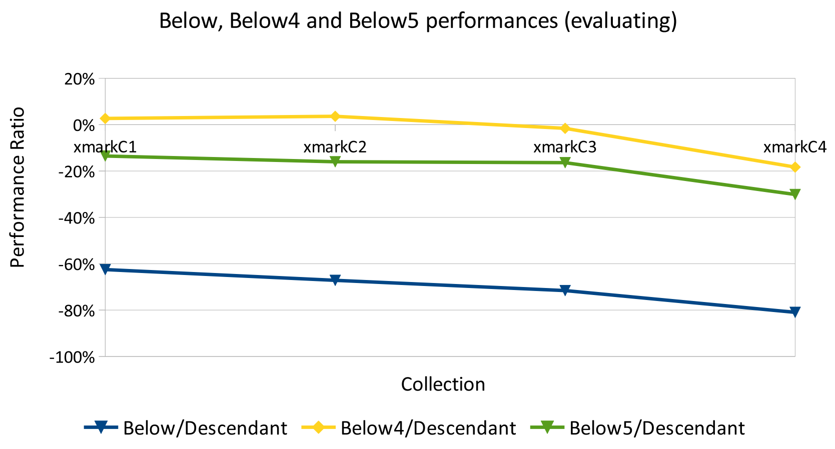


Figure 6.7: Performance comparison between the `descendant` and the `below` axes variants for the XMark collection.

6.1.4 The `near` Evaluation

In this subsection the `near` axis evaluations are presented. It is important to notice that, as previously outlined, no equivalent axes exist that mimic the behavior of the flexible axis defined in the FleXy language. Differently from the `below` axis that has been compared with the `descendant` axis, excluding the structural score computation, the `near` axis could not be directly compared with standard XQuery axes.

A first evaluation could be the comparison of the `near` axis, and its parametric variants, with a set of XQuery expressions that allow to match the same Node-Set matched by the flexible `near` axis. In Listing 6.3 a first example of such constraint rewriting is presented: the path expression `nodeA//nodeB/near::nodeE` that includes the axis `near` constraint, identically evaluated as the `near1` axis, is rewritten into an XQuery expression composed by two distinct sub-expressions: the first (in line 1) allows to match a child node, thus at distance of 1 arc, labeled `nodeE` from the context node `nodeB`, while the second expression in line 3 follows the backward traversal of the document tree matching a parent node labeled `nodeE` from the context node `nodeB`. The two XQuery sub-expressions are joint together by the standard XQuery `union` operator; it allows to produce a single Node-Set as the concatenation of the *Node-Sets* produced by the evaluation of two XQuery expressions; such operator strictly resembles the `union` operator as defined in the SQL language.

```
1   nodeA//nodeB/child::nodeE
2     union
3   nodeA//nodeB/parent::nodeE
```

Listing 6.3: An XQuery expression that mimic the `near1` axis matching evaluation by using the `union` operator.

In Listing 6.4 the `near2` axis evaluation has been rewritten by using only standard XQuery axes such as the `child` and the `parent` axes; in line 1 the set of nodes at a distance 1 are matched (like the `near1` axis evaluation, while lines from 3 to 7 match the target nodes labeled `nodeE` at a distance 2 from the context node `nodeB`. In particular, line 3 matches the descending nodes at level 2, in line 5 the *grandfather* is taken into account, while in line 7 the target node labeled `nodeE` is looked for in the sibling nodes of the context node.

```
1    nodeA//nodeB/child::nodeE union nodeA//nodeB/parent::nodeE
2      union
3    nodeA//nodeB/child::*/child::nodeE
4      union
5    nodeA//nodeB/parent::*/parent::nodeE
6      union
7    nodeA//nodeB/parent::*/child::nodeE
```

Listing 6.4: An XQuery expression that mimics the `near2` axis matching evaluation by using the `union` operator.

Such simple `near` axis rewriting examples, however, introduce an issue related to the definition of an equivalent evaluation of the `near` axis where only standard XQuery axes are involved: not considering the missing computation of a structural relevance score, the use of the `union` operator would produce duplicated results in the final Node-Set. Such issue is more and clearly noticeable when a distance of 3, or greater, is specified as the `neat` threshold parameter: in such case the rewriting production would provide a `parent-child-child` path traversal sequence starting from the context node, that would partially cover the same node set matched by the `child` axis.

A second issue that compromises an effective comparison of the `near` axis, and its variants, against a standard XQuery expression is, other than the node duplication previously otlined, the evaluation process applied when `union` operators are formulated. In particular most XQuery engines, including the BaseX engine on top of which the FleXy constraints have been implemented, would evaluate each distinct sub-query separately and the set of results merged together. This would require the system to evaluate more than one query in contrast to a single XQuery expression provided by the `near` axis formulation; in particular 2 sub-expressions are needed for the `near1` axis, 5 for the equivalent of the `near2` and 8 for the `near3` axis. In Listing 6.5 the query used for the `near2` axis evaluation is presented, while in Listing 6.6 the rewritten XQuery is shown. Both the query expressions return the actual path of the selected item by using the standard function `path()`, thus no different evaluation is provided, but the axis itself.

```
for $i in
  nodeE/nodeA/near2::nodeE
return <hit>{$i/path()}</hit>
```

Listing 6.5: One of the FleXy/XQuery expressions used to evaluate the `near2` axis.

Name	**Parsing** (ms)	**Compiling** (ms)	**Evaluating** (s)
near1	134.34	7.73	2.01
near1-rewritten(2)	139.37	7.75	2.07
near2	133.13	7.65	2.11
near2-rewritten(5)	140.92	7.64	2.62
near3	134.75	7.78	2.21
near3-rewritten (8)	148.56	7.77	3.59

Table 6.9: Parsing, compiling and evaluating timings for the `near` variants and the equivalent rewritten expressions.

```
for $i in
  nodeE/nodeA/child::nodeE
  union
  nodeE/nodeA/parent::nodeE
  union
  nodeE/nodeA/child::*/child::nodeE
  union
  nodeE/nodeA/parent::*/parent::nodeE
  union
  nodeE/nodeA/parent::*/child::nodeE

return <hit>{$i/path()}</hit>
```

Listing 6.6: The *rewritten* query executed as a `near2` axis equivalent.

A first comparison performed between the `near` axis variants and their XQuery equivalent set of expressions confirmed that such evaluation is not fair: the system overhead introduced by the multiple sub-expressions does not play in favor for the axis rewrite approach. In Table 6.9 the Parsing, Compiling and Evaluating timings are shown for each `near` axis variant and their *rewritten* counterpart, where the number of sub-expressions are indicated between rounded brackets. The results confirm that the evaluation of the *rewritten* counterpart of the `near` axis variants requires more time than the `near` axis. Such increment is related to the number of arcs that need to be evaluated, in particular to the number of sub-expressions that are generated in each *rewritten* counterpart. Regarding the Parsing and the Compiling timings, the same trend identified for the Evaluating is shown, in this case such differences, however, are only of a few milliseconds and, thus, they are negligible.

The evaluation have been performed on top of the collection named *CollNodeE*, and previously described in Section 6.1.2; this choice has been motivated by the lower variability in the document structure found in the IMDb collection, such feature and the presence of nodes having the same label and placed in different hierarchy level are required to better evaluate the `near` axis and to avoid the BaseX query optimization and rewriting process.

6.1.5 Conclusions

In this section the axes `below` and `near`, alongside their parametric variants, have been evaluated in comparison, if available, to the standard set of the XQuery language axes. The axis `below` has been compared to the `descendant` axis, while for the `near` axis, for which no standard counterparts exists, an axis rewriting strategy is proposed.

The `below` axis comparison shows how the score computation introduced by the FleXy language, in addition to the required distance computation implemented in the BaseX query engine, produces an overhead, as shown in Fig. 6.3 and in Fig. 6.7. Such overhead, of course is proportional both to the number of nodes that are traversed during the evaluation and the structure of the collection: an overhead of 40% is measured for collections composed of several medium documents, while higher performance loss can be noticed if a single huge XML file with a complex structure is queried.

The score computation overhead can be considered acceptable as a constant comparison ratio is noticeable during the evaluations performed on the IMDB collection. In the worst case scenario, represented by the *xmarkC4* collection, the `below` performance can still be considered acceptable, as a linear increment in the evaluation timings is measured, while the obtained results contains the important information computed regarding their structural relevance, information missing from the results obtained with the `descendant` evaluation.

The `near` axis evaluation, however, presented some issues as no other standard XQuery axis could provide the same result set of the FleXy constraint. A preliminary comparison has been proposed by rewriting the `near` axis, and its parametric variants, by a set of queries involving a combination of the standard `parent` and `child` XQuery axes. As discussed in Section 6.1.4, such comparison has been confirmed to provide unfair results as the rewriting technique would require the XQuery engine to evaluate an increasing number of sub-queries to obtain the same result of the specified `near` axis, thus a correct timing comparison and evaluation of the `near` axis can not be performed.

An important result, however, is shown in the `below3` axis evaluation shown in Fig. 6.3 on the IMDB collection, where the threshold provided by the user allows to effectively reduce the number of traversed nodes and to perform, on average, the evaluation with a 20% less time if compared to the standard `descendant` axis evaluation. The same result is marginally noticeable for the `below4` axis evaluation performed on the XMark collection shown in Fig. 6.7, where a reduced evaluation timing shown up to the *xmarkC3* collection.

Such behaviour, besides providing the vague evaluation of the user constraints and the structural relevance score computation, could be leveraged by the query engine during the query optimization phrase. As shown in Fig. 6.3, when the threshold value provided to the `below` axis exactly matches the maximum distance where a target node can be found in the document node hierarchy, a performance gain in the query evaluation can be measured (in particular in the `below3` axis evaluation). Such aspect may be leveraged by the BaseX query engine, in particular in the query optimization phase where a `descendant` axis could be rewritten into a `below`*n* axis, when the threshold parameter *n* is computed on the path summary already available in the query system. Such rewrite technique involving the `below` axis, of course, should take into account the maximum distance parameter to avoid the distance computation overhead and eventually provide a score-less `below` evaluation to better improve the axis evaluation.

6.2 FleXy use-case: the PatentLigth application

In this section a use case of the FleXy language (as implemented on the Flex-BaseX engine) is presented: by adopting the flexibility introduced by the FleXy `below` and `near` axes evaluation with the associated computation of the structural relevance score, the PatentLight [164] search application has been designed to help non-expert users to easily identify and retrieve patents relevant to their needs. With the PatentLight application, the patent retrieval task has been addressed as a particular XML retrieval task, given the highly structured XML format in which the online patents collections are available.

Section 6.2.1 introduces the patent retrieval task as addressed by PatentLight, and the set of the novel features proposed. Section 6.2.2 describes the related works in the patent retrieval field: the techniques and the currently available patent search tools are described with their characteristics and limitations. Section 6.2.3 describes the PatentLight tool and the flexible approach adopted, while, Section 6.2.4 concludes this Section.

6.2.1 Introduction

Patent Retrieval is a challenging research problem with important open issues; the research contributions related to this task stress in fact the point that expert users spend hours, even days, trying to localize relevant patents from a large set of search results; such behavior is also reported by a survey conducted by Azzopardi et al. in [165]. Moreover, in the last decade patent collections are available on the Internet such as the *United States Patent and Trade Office*[3] (USPTO), the *European Patent Office* [4] (EPO) and the *World Intellectual Property Organization*[5] (WIPO); this way, also non-patent specialists can access patents online. Thus, it is becoming more and more important to develop effective patent search applications for both experts and non-specialists users.

Usual approaches to patent retrieval guide the users in query formulation by giving them the possibility to search patents only based on specific contents: CARROT4Marec[6],

[3] The USPTO patent registry is available at: `www.uspto.gov`.
[4] The European Patent Office is available at `www.epo.org` website.
[5] Available at the following page: `swww.wipo.int/pctdb/en/`.
[6] Available at `www.ir-facility.org/prototypes/carrot4marec/` (accessed on 2012-04-12).

for example, allows to evaluate queries only on specific patent fields such as the English title and the English abstract. A web portal dedicated to patent search named PatentSearcher is presented in [166], where multiple patent fields, such as such as abstract, claims, description and images descriptions, and other features are used to to retrieve and rank patents. The *PatentSearcher* search engine evaluates a keyword based query by giving more importance to a keyword if it is matched in the *Abstract* field, than if it is matched in other sections such as *Description* or *Claims*.

In XML-patents some *tags* are also used to indicate the roles of people involved in the patent. The USPTO patent collection, used as the main patent collection for the PatentLight application evaluation here described, structures the information related to a person by using the following elements: a `last-name` and a `first-name` tags. Their position in the XML document nodes hierarchy defines the person's role in the patent such as Applicant, Examiner, Agent or Assignee. Other details about the patent, such as the claims, the invention title, and the invention description are structured as shown in the example document of Fig. 6.8.

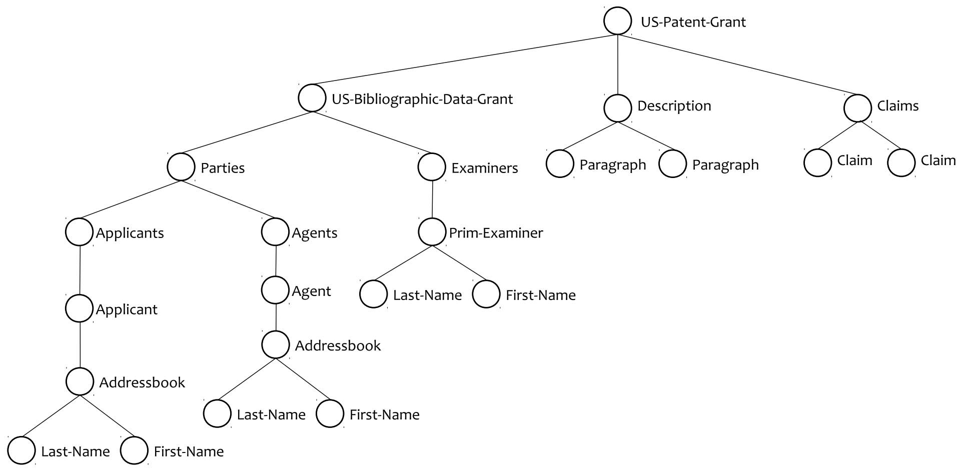


Figure 6.8: A fragment of an USPTO Patent document.

It is important to notice that not only the *tag* labels are used to represent the semantics of their contents, but also their structural position carries important details about an element semantic content; both of such structural and label semantics should be taken into account when querying XML documents. This aspect is indeed leveraged by the PatentLight application to provide its novel search features.

In fact none of the above systems is able to guess the user search intent, and in case of ambiguous queries some useful search results could not be retrieved. Let us suppose that a user formulates a query with the following ambiguous keyword: *"Bell"* with the intent of searching patents where the examiner is named *Bell*[7] Such ambiguous query

[7] The example query could be argued to be too simple, but well represents an ambiguous user need. Such keyword query could be, as well, specified in a more complex query where multiple keywords

has been evaluated with both CARROT4Marec and the *PatentsSearcher* applications; the obtained results do not respect the user search intent: both systems search, in fact, the specified keyword in the predefined fields (i.e., title and description respectively), and do not take into account the examiner field. Thus the retrieved patents deal with the concept *Bell* as an *invention*, and not as a *person*.

With respect to the other existing patent search engines, the patent search application *PatentLight* allows to: (1) categorize search results according to the information on the document structure, (2) rank the retrieved patents related to each category by analyzing information on both the document structure and the document content. The two main characteristics of the PatentLight are here briefly listed:

Patent Categorization The PatentLight application allows to categorize the results of the evaluation of a keyword based query by activating distinct pre-defined queries; each pre-fedined query corresponds to a category. For each category a different set of elements is analyzed and the matching patents are retrieved and ranked accordingly. This is a novel strategy with respect to the other patent applications that analyze only the content of some predefined elements as explained above. By selecting one of these categories the user is guided in locating patents according to a specific search task. In particular the categories identified by the PatentLight application are: *Titles*, *People*, *Descriptions* and *Claims*. For example, patents will be retrieved in the category *Titles* if the keywords specified by the user match the contents of the tag `title`; while the category *People* will contain patents for which the given keywords match the textual contents of tags related to people, such as the `Applicant` or the `Agents` tags.

Patent Ranking Patents contents are internally represented by using the XML format; this means that standard XML query languages, like XQuery or XPath, can be used to search patents. PatentLight is based on the FleXy extension of the XQuery Full-Text language, where the `below` and `near` axes are defined: this characteristic allows to produce a ranked list of patents (and patent fragments) by considering both the content relevance (by leveraging the XQuery Full-Text extension) and the XML element structural position in the XML patent document (by leveraging the FleXy extension).

6.2.2 Related Works

Patent retrieval is a very complex task, and in the literature several applications have been developed to help and support users in the patent search task. A patent is a structured document composed of several sections such as descriptions, claims, title, agents, assignees, examines, classifications, etc. Usually, patents documents are defined in the XML format, and patent applications make use of IR techniques to index patents by considering some tags and their textual content.

are specified, but the same subsequent observations about current patent engines apply.

In the literature some patent applications have been defined in order to search patents only on specific tags content. For example the technique proposed by Mase et al. in [167] analyzes only the `claims` XML element contents. In fact, a frequent and important inquiring is the one aimed at the retrieval of patents that may invalidate an invention. In particular, a keyword-based retrieval method is defined, and the following four steps are applied: (1) query terms extraction from the claim text by applying standard text analysis such as tokens identification and stop words removal, (2) query term-weighting by only using document frequency, *df*, (3) patent retrieval using the `claims` element as the only target, and (4) ranking of the retrieved patents.

In the approach proposed by Xue et al. [168] the problem of an automatic query generation for patent search is studied: a query manipulation technique that takes into account several patent features is proposed and analyzed. A first technique consists at indexing the textual contents of six different patent tags such as title, abstract, a brief summary, descriptions of the figures, detailed text descriptions, and claims. A standard Information Retrieval indexing technique is adopted, while the set of extracted terms are subsequently weighted by considering three different formulas based on term frequency, inverse document frequency and a combination of the two. Another feature takes into account the International Patent Classification (IPC) class assigned to a patent: by analyzing the `class` tag, where patents are categorized into two levels: primary class code and secondary class code. Each feature and their combination have been evaluated by the authors on the USPTO collection. The evaluations have shown better performances in case of a joint use of all the features.

This is an evidence of the fact that a patent application can achieve a good performance when patents search is simultaneously executed on several tags. For example, important patents applications, such as Google Patents, Delphion[8], PatentsSearcher, and CARROT4Marec, search for patents by jointly analyzing the content of title, and abstract tags when a user formulates keyword-based queries.

The PatentLight application proposed within this research activity allows to search, as described in the subsequent Section 6.2.3, for patents by simultaneously considering a greater number of XML tags than the above applications such as inventors, agents, assignees, classifications, claims, etc. The objective is to satisfy all the possible user search topical interests and not to limit patent retrieval by analyzing only the information in the title and in the abstract sections.

6.2.3 The PatentLight tool

The PatentLight application, as previously outlined, has been developed by leveraging, as the underlying patent query engine the Flex-BaseX XQuery engine as described in Section 5. In particular the patent search application has been defined and tested on

[8] The Delphion search engine, available at `www.delphion.com` (accessed 2012-04-12) requires a mandatory subscription.

the USPTO patent collection, freely available online. The USPTO collection is the patent corpus adopted by most patent search applications such as Google Patents and Patents Searcher. The newly defined patent engine, besides allowing users to perform a classic *advanced search* where queries may be formulated through an ad-hoc interface on specific tags such as *inventor*, *country*, *claims* and so on, consists in a basic keyword-based search where, differently from other approaches, the set of retrieved patents are organized according to thematic categories. Such categories are defined to better help users in locating relevant patents and they have been defined by exploiting the structure of the XML-patents documents.

The keyword-based query is automatically re-written into four distinct FleXy queries, one for each of the four thematic categories defined based on the structure of the patents belonging to the USPTO patent collection. The aim of the categorization is to organize the XML-patents into meaningful semantic XML-elements, i.e. such XML-elements have to cover the main information defined into a patent. This way, when a user formulates an ambiguous query, the categorization process allows to easily capture what the user topical search intent is by individuating all the possible interpretations associated with a patent.

The definition of the right set of categories assumes a key aspect during the categorization process. The four thematic categories that have been defined by analyzing the XML structure of the patents and are: *People*, *Titles*, *Descriptions*, and *Claims*. The choice of these categories has been made by considering the main, possibly useful information defined in the patents.

Furthermore, by analyzing other patents collections such as EPO and WIPO we have noticed that the same meaningful information individuated in the USPTO patents are also present in these collections, even if in different tags. This enforces our categories selection, and confirms that our system can be easily adapted to those collections.

Formally, let $\mathbf{E}$ be the set of XML-elements defined in the USPTO-patents, E be a subset of elements $E \subset \mathbf{E}$, and Cat be a set of categories, then one or more elements $e_1, e_2, ..., e_n \in E$ are mapped into a category $c \in Cat$, i.e. $\{e_1, ..., e_m\} \mapsto c$ with $m \neq n$. The subset of elements E has been selected from $\mathbf{E}$ in order to cover important aspects from patents. In Listing 6.7 the considered categories along with the corresponding XML-elements, enclosed in curly brackets, are presented.

```
People:        {Applicants,Agents,Assignees,Examiners}
Titles:        {Invention-title}
Descriptions: {Description}
Claims:        {Claims}
```

Listing 6.7: The set of defined categories and the corresponding XML elements.

For each category a FleXy query is defined in order to search the query terms in pre-established elements as shown in the following Listing 6.8.

```
People:   //Applicants/near::Last-Name[text() contains text "query terms"]
Titles:   //invention-title[text() contains text "query terms"]
Descriptions:  //description/below::p[text() contains text "query terms"]
Claims:   //claims/below::claim-text[text() contains text "query terms"]
```

Listing 6.8: The set of defined categories and the corresponding XML elements.

It can be noticed that the FleXy query related to the category *People* uses the `near` axis constraint, and the considered context node is the tag `Applicants`; this means that the applicant role (corresponding to the inventor role) is assumed to have more importance with respect to the other roles defined in the patent such as *agent*, *examiner*, and *assignee*. This choice has been motivated to be coherent with respect to standard patents search applications (i.e., GOOGLE PATENTS, PATENTS SEARCHER, etc.). In fact, by analyzing the advanced search of these applications (where a user is able to search patents through specific tags), it is possible to search patents by considering the role of `inventor` (or the one of `assignee`). In case of user queries formulated by the standard textual search area, when a user writes a *name* of a person it is supposed that he/she interested in finding inventors of patents. However, it is important to note that by the PATENTLIGHT approach also patents containing a person *name* with a different role will be retrieved given the use of the `near` axis for flexibly evaluating the query.

An example query

In the following an example of the FleXy query generated by the keywords specified by a user are described; let us suppose that the ambiguous keyword-based query *Bell* is formulated by the user, and the search results are categorized as previously outlined.

In Listing 6.9 the complete syntax of the XQuery expression used for retrieving the results of the category *People* is shown, while in Listing 6.10 the XQuery expression to match the category *Descriptions* is provided.

```
for $item score $sft score-structure $sst in
  //Applicants/near::last-name[text() contains text "Bell"
    or ../first-name/text() contains text "Bell"]
 order by $sst descending, $sft descending

return
 <result scoreST="{$ss}" scoreFT="{$sft}">
   <text>{$item/text()}, {$item/parent::*/first-name/text()}</text>
   <path>{$item/fn:path()}</path>
   <patent-file>{data($item/root()/us-patent-grant/@file)}</patent-file>
 </result>
```

Listing 6.9: FleXyXQuery expression for the *People* category matching.

In both query expressions the standard XQuery Full-Text score and the FleXy structural score are computed and used to sort the set of obtained results in each category.

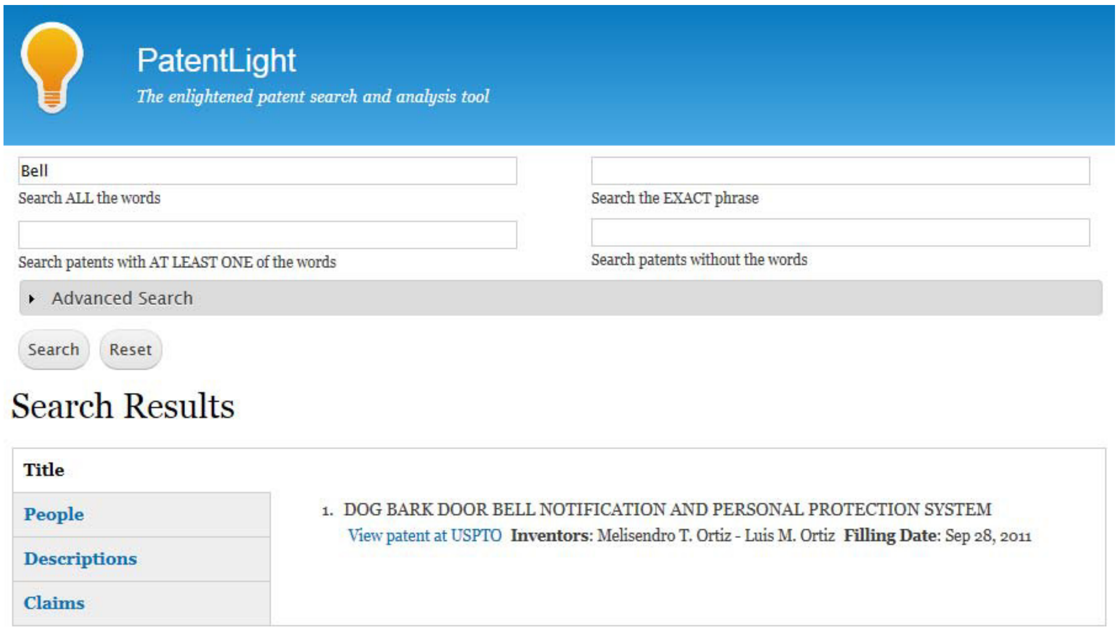
Furthermore, the matched results have been designed to share the same XML structure: an element labeled **`result`** contains three tags, namely **`text`**, **`path`**, and **`patent-file`** that in turn contain the matched text (as the concatenation of the person name and surname for the category *People*, or the matched paragraph text for the category *Descriptions*), the rooted label path of the XML element where the content has been found, and the file-name of the patent document. This allows to provide a uniform result structure for the retrieved set of patents, furthermore it facilitates the `PatentLight` backend/frontend communication and the display of the results.

```
for $item score $sft score-structure $ss in
   description/below::p[text() contains text "Bell"]
 order by $sst descending, $sft descending

return
 <result scoreST="{$ss}" scoreFT="{$sft}">
   <text>{$item/text()}</text>
   <path>{$item/fn:path()}</path>
   <patent-file>{data($item/root()/us-patent-grant/@file)}</patent-file>
 </result>
```

Listing 6.10: FleXyXQuery expression for the *Descriptions* category matching.

In Fig. 6.9 the graphical user interface for the standard keyword-based search of the `PatentLight` application is shown: in the example the user query and its results have also been displayed.



1. DOG BARK DOOR BELL NOTIFICATION AND PERSONAL PROTECTION SYSTEM
View patent at USPTO **Inventors:** Melisendro T. Ortiz - Luis M. Ortiz **Filling Date:** Sep 28, 2011

Figure 6.9: The `PatentLight` application user interface with query result categorization.

6.2.4 Conclusions

In this section the PatentLight application has been described: it represents a first use case of the FleXy language extension applied to the Patent Retrieval search task aimed at helping the user in finding relevant patents, represented in the XML format, in a simple way. In particular the Flex-BaseX engine has been leveraged to provide a preliminary result categorization while querying the structured format of the US patents collection.

In the PatentLight application several strategies that allow to search and explore patents from an innovative point of view with respect to the works proposed in the literature have been studied. According to the information searched by the user, the main advantages of the proposed strategy are: (1) to categorize search results by considering the tags in the XML structure, and (2) to rank the search-results by considering the flexible constraints on both structure and content.

In particular the FleXy axes `below` and `near` have been combined with the XQuery Full-Text language search features to match XML elements of a patent document by a double search criterion: the standard full-text search computes the relevance estimate score for a fragment given the user keyword-based query, while the flexible axes evaluation is leveraged to provide both the results categorization and the result ranking computation by the structural position of each retrieved fragment.

Further analyses and evaluations are planned to be performed on the search and categorization approach defined by the PatentLight tool by taking into account, besides the relevance computation of the results, the user search behavior and the impact of the patent categorization on user search satisfaction.

7 Conclusions

The *eXtensible Markup Language* allows data and textual contents to be structured in a hierarchical form; its simple and powerful labeled tree structure has been adopted by an increasing number of domains: office productivity tools and communication protocols are only few of the fields where XML has been used.

The two main languages defined by the W3C to query XML documents, XPath and XQuery, allow users to inquiry XML document contents and their structure to select a subset of the document elements; the XQuery language further extends the XPath features by providing data manipulation and XML elements restructuring capabilities. Both languages, coming from a data-centric perspective of XML documents, provide an SQL-like evaluation of the query clauses: an exact matching of content-based and structure-based constraints is performed by XPath and XQuery.

Such Database oriented point of view adopted by the two languages has been opposed by the approaches defined by the Information Retrieval community, where XML documents are treated from a document-oriented perspective. Information Retrieval approaches tackled user formulated queries as imprecise requirements, and both content-based and structure-based constraints are conceived as a mere template during the constraints evaluation, where an approximate matching is performed. A joint work of both communities culminated into the definition of an extension of the XPath and XQuery languages named Full-Text and subsequently standardized by the W3C. The XQuery/XPath Full-Text extension defines a rich set of keyword-based search capabilities, the evaluation of which produces a set of weighted set of elements: a score is computed and assigned to each retrieved XML element expressing the relevance of its textual contents to the given user query. The Full-Text extension also embraces the IR full-text matching style, where stop-words removal, thesauri and other operations can be performed during the query evaluation and element retrieval process.

Regarding the document structure, the IR approaches proposed a set of approximate structural matching techniques, where the standard structure-based constraints, as formulated in an XQuery or XPath expression, are evaluated by adopting path relaxation algorithms. The approximate IR approaches, however, do not take into account the user perspective, and they mainly hide the underlying approximate evaluation performed: the user does not have any possibility neither to specify the desired extent of the approximation, nor which constraints should be, instead, strictly evaluated.

The FLEXY language, defined by the research undertaken during the development of this thesis, has been designed to fill this gap: to allow users to effectively express and formulate vague structure-based constraints. FleXy has been formally defined as an extension of the XQuery Full-Text language, thus leveraging the standard XML element selection and manipulation characteristics of XQuery, and the full-text capabilities offered by the Full-Text extension.

The two new axes named `below` and `near` provide a vague specification of structural constraints, the evaluation of which produces, similarly to the Full-Text extension, a set of weighted elements: in this case a structural relevance score is computed by taking into account the *distance* from the user required target element and the actually retrieved one. Thresholds variants of the FleXy constraints have also been defined; they allow the user to specify the extent of the application of the vague structural constraints.

The XPath and XQuery structure-based constraints, and their strict evaluation, force the users to be well aware of the underlying structure of an XML document collection before formulating a successfully query. This aspect is not trivial, in particular for complex XML element structures. The `below` and `near` axes can thus be specified by users to exploit their, even limited, structural knowledge of the structure of the XML document collection to formulate vague queries that can retrieve relevant results even if an inexact structural constraint is specified.

The work performed in this thesis can be summarized by the following contributions:

- the FLEXY language has been formally defined as an extension of the XQuery/XPath Full-Text language: the new axes `below` and `near` (and their parametric variants) have been defined by their syntax and semantics in Chapter 4. The complete FleXy syntax has been described as fully compatible with the XPath and XQuery language, while the formal semantics of the axes has been defined by extending the Core XPath semantics [9] as defined by Gottlob et al. The FleXy structural scoring is further described in detail through the associated algorithms, and the definition of the structural relevance score variable has been provided to integrate it with the Full-Text content relevance score;
- the implementation of the FleXy language including the vague axis `below` and `near`, the scoring algorithms and the complete vague evaluation are described in Chapter 5. A preliminary implementation was provided as a feasibility study of the vague constraints evaluation, in terms of data structures adaptability and computation efficiency. A complete integration of the FleXy language has been subsequently conducted on top of a fully featured XQuery engine after an accurate analysis of different XQuery compliant alternatives. The BASEX engine has been selected as the basis for the FleXy integration due to its efficient data structures, to the availability of its source code and, most importantly, to its complete adherence to the XQuery Full-Text extension;
- the efficiency of the FleXy language, as implemented on top of the BASEX engine, has been evaluated by comparing the timings of the axes `below` and `near` against their non-vague counterparts in XQuery to the aim of measuring the overhead

introduced by the flexible evaluation. The `below` axis has been compared with the standard `descendant` axis; for the `near` axis, which does not have any standard alternative in XQuery, a query rewriting technique has been provided. As expected, the obtained results outlined that the evaluation of the FleXy axes introduces a noticeable overhead: a linear increasing evaluation timing is shown when a single huge document composes the queried collection (test performed with a set of increasing size XMark documents), while a constant performance ratio is measured for collections composed of multiple small/medium XML documents (test performed with the IMDB collection).
  A quite important outcome of the performed evaluations is that a performance gain has being measured with the threshold axis variants, where the number of traversed elements of the collection is significantly reduced with respect to the standard XQuery `descendant` axis evaluation.

- On top of the BaseX engine extended with the FleXy language, a patent search application named PatentLight has been developed to exploit the structure of the patents XML documents and their textual contents. By leveraging the Full-Text extension and the vague structure-based constraints evaluation of the FleXy language, the system helps non-expert users to effectively retrieve relevant patents by offering a result categorization strategy.

In short, we believe that the proposed FleXy language represents the missing link between the Database and Information Retrieval perspectives of XML documents querying. With the ability to express vagueness in structural constraints, besides the possibility of formulating keyword-based search as addressed by the XQuery Full-Text extension, the user can finally formulate queries that explicitly represent her/his needs, in a way tolerant to imprecision, thus having the full control on the desired evaluation process.

Publications

[1] Marrara Stefania, Panzeri Emanuele, and Pasi Gabriella. “A double layer indexing structure for flexible querying of XML documents”. In: *Proceedings of 9th International Conference on Adaptivity, Personalization and Fusion of Heterogeneous Information*. RIAO ’10. Paris, France, France, 2010, pp. 130–131.

[2] Marrara Stefania, Panzeri Emanuele, and Pasi Gabriella. “An Analysis of an Efficient Data Structure for Evaluating Flexible Constraints on XML Documents”. In: *Proceedings of the 9th International Conference on Flexible Query Answering Systems (FQAS)*. Vol. 7022. Lecture Notes in Computer Science. 2011, pp. 294–305.

[3] Stefania Marrara, Emanuele Panzeri, and Gabriella Pasi. “A Flexible XML Query Language for NON Dummies”. In: *Proceedings of the 2nd Italian Information Retrieval (IIR) Workshop*. Vol. 704. CEUR Workshop Proceedings. 2011.

[4] Silvia Calegari, Emanuele Panzeri, and Gabriella Pasi. “PatentLight: A Patent Search Application”. In: *Proceedings of the 4th Information Interaction in Context Symposium (IIIX)*. Nijmegen, The Netherlands, 2012, pp. 242–245.

[5] Emanuele Panzeri and Gabriella Pasi. “An Approach to Define Flexible Structural Constraints in XQuery”. In: *Proceedings of the 8th International Conference on Active Media Technology (AMT)*. Vol. 7669. Lecture Notes in Computer Science. 2012, pp. 307–317.

[6] Emanuele Panzeri and Gabriella Pasi. “A flexible extension of XQuery Full-Text”. In: *Proceedings of the 4th Italian Information Retrieval (IIR) Workshop*. Vol. 964. CEUR Workshop Proceedings. 2013, pp. 29–32.

[7] Emanuele Panzeri and Gabriella Pasi. “Flexible structural constraints in XQuery Full-Text”. In: *Proceedings of the 10th Conference on Open Research Areas in Information Retrieval (OAIR)*. Paris, France, France, 2013, pp. 51–54.

[8] Elio Masciari, Emanuele Panzeri, and Gabriella Pasi. “An approach to define flexible structural constraints in XQuery (Discussion Paper)”. In: *In Proceedings of the 21st Italian National Conference on Advanced Data Base Systems (SEBD)*. 2013.

[9] Emanuele Panzeri and Gabriella Pasi. “Flex-BaseX: An XML engine with a flexible extension of XQuery Full-Text”. In: *Proceedings of the 36th international ACM SIGIR conference on Research and development in Information Retrieval (SIGIR)*. 2013.

[10] Calegari Silvia, Comerio Marco, Maurino Andrea, Panzeri Emanuele, and Pasi Gabriella. “A Semantic and Information Retrieval Based Approach to Service Contract Selection”. In: *Proceedings of the 9th International Conference on Service-Oriented Computing (ICSOC)*. Vol. 7084. Lecture Notes in Computer Science. 2011, pp. 389–403.

Bibliography

[1] W3C. *Extensible Markup Language (XML)*. Recommendation 1.0. World Wide Web Consortium (W3C), Jan. 1996. URL: http://www.w3.org/TR/xml/.

[2] W3C. *XML Path Language (XPath)*. Recommendation 1.0. World Wide Web Consortium (W3C), Nov. 1999. URL: http://www.w3.org/TR/xpath/.

[3] W3C. *XQuery: An XML Query Language*. Recommendation 1.0. World Wide Web Consortium (W3C), Nov. 2007. URL: www.w3.org/TR/xquery.

[4] Ludovic Denoyer and Patrick Gallinari. "The Wikipedia XML Corpus". In: *SIGIR Forum* 40.1 (June 2006), pp. 64–69.

[5] Sihem Amer-Yahia, SungRan Cho, and Divesh Srivastava. "Tree Pattern Relaxation". In: *Proceedings of the 8th International Conference on Extending Database Technology: Advances in Database Technology*. EDBT '02. London, UK, UK, 2002, pp. 496–513.

[6] Torsten Schlieder. "Similarity Search in XML Data using Cost-Based Query Transformations". In: *Proceedings of the Fourth International Workshop on the Web and Databases*. WebDB '01. Santa Barbara, California, USA, 2001, pp. 19–24.

[7] Yosi Mass, Matan Mandelbrod, Einat Amitay, David Carmel, Yoëlle S. Maarek, and Aya Soffer. "JuruXML - an XML Retrieval System at INEX'02". In: *Proceedings of the First Workshop of the INitiative for the Evaluation of XML Retrieval (INEX)*. Schloss Dagstuhl, Germany, 2002, pp. 73–80.

[8] Sihem Amer-Yahia, Laks V. S. Lakshmanan, and Shashank Pandit. "FleXPath: Flexible Structure and Full-text Querying for XML". In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. SIGMOD '04. New York, NY, USA, 2004, pp. 83–94.

[9] Georg Gottlob, Christoph Koch, and Reinhard Pichler. "Efficient Algorithms for Processing XPath Queries". In: *ACM Transactions on Database Systems* 30.2 (June 2005), pp. 444–491.

[10] Gabriella Kazai, Norbert Gövert, Mounia Lalmas, and Norbert Fuhr. "The INEX Evaluation Initiative". In: *Intelligent Search on XML Data*. Vol. 2818. Lecture Notes in Computer Science. 2003, pp. 279–293.

[11] Priscilla Walmesley. *XQuery*. 1st. O'Reilly, 2007, p. 511. ISBN: 978-0-596-00634-1.

[12] Joe Fawcett, Danny Ayers, and Liam R. E. Quin. *Beginning XML*. 5th. Wrox, 2008, p. 864. ISBN: 978-1-1181-6213-2.

[13] John Simpson. *XPath and XPointer.* 1st ed. O'Reilly, 2002, p. 208. ISBN: 978-0-596-00291-6.

[14] W3C. *World Wide Web Consortium webpage.* http://www.w3.org.

[15] W3C. *Extensible Markup Language (XML).* Tech. rep. World Wide Web Consortium (W3C), 2013. URL: `http://www.w3.org/XML`.

[16] ISO. *ISO 8879:1986 Information processing - Text and Office Systems - Standard Generalized Markup Language (SGML).* Recommendation. International Organization for Standardization (ISO), Oct. 1986. URL: `http://www.iso.org/iso/catalogue%5C_detail.htm?csnumber=16387`.

[17] W3C. *Cascading Style Sheets (CSS).* Recommendation 2.1. World Wide Web Consortium (W3C), June 2011. URL: `http://www.w3.org/TR/CSS2/`.

[18] W3C. *XSL Transformations (XSLT).* Recommendation 1.0. World Wide Web Consortium (W3C), Nov. 1999. URL: `http://www.w3.org/TR/xslt/`.

[19] W3C. *XML Document Type Declaration.* Tech. rep. World Wide Web Consortium (W3C), Oct. 2004. URL: `http://www.w3.org/TR/xml/%5C#dt-doctype`.

[20] James Clark and Murata Makoto. *RELAX NG.* Recommendation. The Organization for the Advancement of Structured Information Standards (OASIS), Dec. 2001. URL: `http://relaxng.org/spec-20011203.html`.

[21] ISO. *ISO/IEC 19757-3:2006 Information technology - Document Schema Definition Language (DSDL) - Part 3: Rule-based validation - Schematron.* Tech. rep. International Organization for Standardization (ISO), June 2006. URL: `http://schematron.com`.

[22] W3C. *XML Schema.* Recommendation 1.0. World Wide Web Consortium (W3C), Oct. 2004. URL: `http://www.w3.org/TR/xmlschema-1/`.

[23] W3C. *XHTML The Extensible HyperText Markup Language.* Recommendation 1.0. World Wide Web Consortium (W3C), Aug. 2002. URL: `http://www.w3.org/TR/xhtml1/`.

[24] W3C. *SOAP: Messaging Framework (Second Edition).* Recommendation 1.2. World Wide Web Consortium (W3C), Apr. 2007.

[25] W3C. *Web Services Description Language (WSDL).* Recommendation 1.1. World Wide Web Consortium (W3C), Mar. 2001.

[26] IETF. *RFC 3920: Extensible Messaging and Presence Protocol (XMPP): Core.* Recommendation. Internet Engineering Task Force (IETF), Oct. 2004. URL: `http://www.ietf.org/rfc/rfc3920.txt`.

[27] Mounia Lalmas and Ricardo Baeza-Yates. "Structured Document Retrieval". In: *Encyclopedia of Database Systems.* 2009, pp. 2867–2868.

[28] Andrew Trotman and Börkur Sigurbjörnsson. "Narrowed extended XPath i (NEXI)". In: *Proceedings of the Third international conference on Initiative for the Evaluation of XML Retrieval (INEX 2004).* Lecture Notes in Computer Science. Berlin, Heidelberg, 2005, pp. 16–40.

[29] W3C. *DOM level 1 (Document Object Model)*. Recommendation 1.0. World Wide Web Consortium (W3C), Oct. 1998. URL: `http://www.w3.org/TR/REC-DOM-Level-1/`.

[30] W3C. *XML Pointer Language (XPointer)*. Working Draft. World Wide Web Consortium (W3C), Aug. 2002. URL: `www.w3.org/TR/xptr/`.

[31] W3C. *XForms*. Recommendation 1.1. World Wide Web Consortium (W3C), Oct. 2009. URL: `www.w3.org/TR/xforms/`.

[32] W3C. *XML Path Language (XPath)*. Recommendation 2.0. World Wide Web Consortium (W3C), Nov. 2007. URL: `www.w3.org/TR/xpath20`.

[33] W3C. *DOM level 3 (Document Object Model) Core Specification*. Recommendation 1.0. World Wide Web Consortium (W3C), Apr. 2004. URL: `http://www.w3.org/TR/DOM-Level-3-Core/`.

[34] Cezar Andrei, Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Zacharioudakis Markos. *Extending XQuery with Collections, Indexes, and Integrity Constraints*. Working Draft. http://www.flworfound.org/pubs/xqddf.pdf: Flowr Foundation, 2009.

[35] W3C. *XQuery Update Facility 1.0*. Recommendation. World Wide Web Consortium (W3C), Mar. 2011. URL: `http://www.w3.org/TR/xquery-update-10/`.

[36] Donald D. Chamberlin, Jonathan Robie, and Daniela Florescu. “Quilt: An XML Query Language for Heterogeneous Data Sources”. In: *Selected papers from the Third International Workshop WebDB 2000 on The World Wide Web and Databases*. London, UK, UK, 2001, pp. 1–25.

[37] Jonathan Robie, Joe Lapp, and David Schach. *XML Query Language (XQL)*. Technical Report. World Wide Web Consortium (W3C), 1998. URL: `http://www.w3.org/TandS/QL/QL98/pp/xql.html`.

[38] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. *XMLQL: a query language for XML*. Technical Report. World Wide Web Consortium (W3C), Aug. 1998. URL: `http://www.w3.org/TR/NOTE-xml-ql/`.

[39] Peter Buneman, Mary Fernandez, and Dan Suciu. “UnQL: a query language and algebra for semistructured data based on structural recursion”. In: *The VLDB Journal* 9.1 (Mar. 2000), pp. 76–110.

[40] Angela Bonifati and Stefano Ceri. “Comparative Analysis of Five XML Query Languages”. In: *SIGMOD Rec.* 29.1 (Mar. 2000), pp. 68–79.

[41] W3C. *XML Query Languages: Experiences and Exemplars*. Technical Report. World Wide Web Consortium (W3C), Sept. 1999. URL: `http://www.w3.org/1999/09/ql/docs/xquery.html`.

[42] W3C. *XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)*. Recommendation 2.0. World Wide Web Consortium (W3C), Dec. 2010. URL: `http://www.w3.org/TR/xpath-functions/`.

[43] W3C. *XSL Transformations (XSLT) Version 2.0.* Recommendation 2.0. World Wide Web Consortium (W3C), Jan. 2007. URL: `http://www.w3.org/TR/xslt20/`.

[44] W3C. *XQuery, XPath, and XSLT Functions and Operators Namespace Document.* Technical Report. World Wide Web Consortium (W3C), Dec. 2011. URL: `http://www.w3.org/2005/xpath-functions/`.

[45] Unicode. *The Unicode Standard.* http://www.unicode.org/standard/versions/.

[46] W3C. *Scalable Vector Graphics (SVG).* Recommendation 1.1. World Wide Web Consortium (W3C), Aug. 2011. URL: `http://www.w3.org/TR/wsdl/`.

[47] W3C EXPath. *EXPath Community Group.* http://www.w3.org/community/expath/.

[48] W3C. *XProc: An XML Pipeline Language.* Recommendation 1.0. World Wide Web Consortium (W3C), May 2010. URL: `http://www.w3.org/TR/xproc/`.

[49] W3C EXPath. *File Module.* EXPath Candidate Module. EXPath Community Group, June 2012. URL: `http://expath.org/spec/file`.

[50] W3C EXPath. *Geospatial API Module.* EXPath Candidate Module. EXPath Community Group, Sept. 2010. URL: `http://expath.org/spec/geo`.

[51] W3C EXPath. *Cryptographic Module.* EXPath Candidate Module. EXPath Community Group, Aug. 2011. URL: `http://expath.org/spec/crypto`.

[52] W3C EXPath. *ZIP Module.* EXPath Candidate Module. EXPath Community Group, Oct. 2010. URL: `http://expath.org/spec/zip`.

[53] W3C EXPath. *HTTP Client Module.* EXPath Candidate Module. EXPath Community Group, Jan. 2010. URL: `http://expath.org/spec/http-client`.

[54] W3C EXPath. *SQL Client Module.* EXPath Candidate Module. EXPath Community Group, Sept. 2011. URL: `http://expath.org/spec/sql`.

[55] W3C. *XQuery 3.0: An XML Query Language.* Working Draft 3.0. World Wide Web Consortium (W3C), Dec. 2011. URL: `http://www.w3.org/TR/xquery-30/`.

[56] Andrew Trotman and Richard A. O'Keefe. "he Simplest Query Language That Could Possibly Work". In: *Proceedings of the Second international conference on Initiative for the Evaluation of XML Retrieval (INEX 2003).* Lecture Notes in Computer Science. Berlin, Heidelberg, 2003, pp. 167–174.

[57] Paul Ogilvie. "Retrieval Using Structure for Question Answering". In: *First Twente Data Management Workshop (TDM 2004) on XML Databases and Information Retrieval.* CTIT Workshop Proceedings Series. University of Twente, Enschede, The Netherlands, 2004, pp. 17–25.

[58] Andrew Trotman and Börkur Sigurbjörnsson. "NEXI, Now and Next". In: *Advances in XML Information Retrieval.* Vol. 3493. Lecture Notes in Computer Science. Berlin, Heidelberg, 2005, pp. 41–53.

[59] W3C. *XQuery/XPath FullText.* Recommendation 1.0. World Wide Web Consortium (W3C), Mar. 2011. URL: `www.w3.org/TR/xpath-full-text-10`.

[60] W3C. *XQuery and XPath Full Text requirements.* Working Group Note. World Wide Web Consortium (W3C), Jan. 2011. URL: `http://www.w3.org/TR/xpath-full-text-10-requirements/`.

[61] W3C. *XQuery and XPath Full Text 1.0 Use Cases.* Working Group Note. World Wide Web Consortium (W3C), Jan. 2011. URL: `http://www.w3.org/TR/xpath-full-text-10-use-case/`.

[62] W3C. *XQuery and XPath Full Text 1.0 Test Suite (XQFTTS).* World Wide Web Consortium (W3C), Dec. 2010. URL: `http://dev.w3.org/cvsweb/2007/xpath-full-text-10-test-suite/`.

[63] ISO. *ISO/IEC 13249-2 Information technology - Database languages - SQL Multimedia and Application Packages - Part 2: Full-Text, SQL/MM.* Tech. rep. 2.0. International Organization for Standardization (ISO), 2003.

[64] Daniela Florescu, Donald Kossmann, and Ioana Manolescu. "Integrating Keyword Search into XML Query Processing". In: *Proceedings of the 9th International World Wide Web Conference on Computer Networks.* Amsterdam, The Netherlands, 2000, pp. 119–135.

[65] Taurai Tapiwa Chinenyanga and Nicholas Kushmerick. "Expressive Retrieval from XML Documents". In: *Proceedings of the 24Th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval.* SIGIR '01. New York, NY, USA, 2001, pp. 163–171.

[66] Anja Theobald and Gerhard Weikum. "Adding Relevance to XML". In: *Proceedings of the Third International Workshop on the Web and Databases (WebDB).* Vol. 1997. Lecture Notes in Computer Science. 2000, pp. 105–124.

[67] Norbert Fuhr and Kai Großjohann. "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 (SIGIR).* 2001, pp. 172–180.

[68] Shiem Amer-Yahia, Chavdar Botev, and Jayavel Shanmugasundaram. "Texquery: a full-text search extension to xquery". In: *Proceedings of the 13th international conference on World Wide Web.* WWW '04. New York, NY, USA, 2004, pp. 583–594.

[69] W3C. *XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition).* Recommendation 2.0. World Wide Web Consortium (W3C), Dec. 2010. URL: `http://www.w3.org/TR/xquery-semantics/`.

[70] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval.* 1st ed. Cambridge University Press, 2008.

[71] Andrew Trotman. "Processing Structural Constraints". In: *Encyclopedia of Database Systems.* 2009, pp. 2191–2195.

[72] M. Hachicha and J. Darmont. "A Survey of XML Tree Patterns". In: *IEEE Transactions on Knowledge and Data Engineering* 25.1 (2013), pp. 29–46.

[73] Sergio Flesca, Filippo Furfaro, and Elio Masciari. “On the minimization of XPath queries”. In: *Proceedings of 29th International Conference on Very Large Data Bases*. VLDB ’03. 2003, pp. 153–164.

[74] Zhimin Chen, Laks V. S. Jagadish H. V.and Lakshmanan, and Stelios Paparizos. “From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery”. In: *Proceedings of 29th International Conference on Very Large Data Bases*. VLDB ’03. 2003, pp. 237–248.

[75] Donald D. Chamberlin and Raymond F. Boyce. “SEQUEL: A Structured English Query Language”. In: *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. SIGFIDET ’74. New York, NY, USA, 1974, pp. 249–264.

[76] Sandeepan Banerjee, Vishu Krishnamurthy, Muralidhar Krishnaprasad, and Ravi Murthy. “Oracle8i-the XML enabled data management system”. In: *Proceedings of the 16th International Conference on Data Engineering*. 2000, pp. 561–568.

[77] Zhen Hua Liu, Muralidhar Krishnaprasad, and Vikas Arora. “Native XQuery processing in oracle XMLDB”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. SIGMOD Conference. 2005, pp. 828–833.

[78] Peter Boncz, Torsten Grust, Maurice Van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. “MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine”. In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’06. New York, NY, USA, 2006, pp. 479–490.

[79] Alin Deutsch, Mary Fernandez, and Dan Suciu. “Storing Semistructured Data with STORED”. In: *SIGMOD Rec.* 28.2 (June 1999), pp. 431–442.

[80] Rebecca J. Cathey, Steven M. Beitzel, Eric C. Jensen, David Grossman, and Ophir Frieder. “Using a Relational Database for Scalable XML Search”. In: *The Journal of Supercomputing* 44.2 (May 2008), pp. 146–178.

[81] Sabine Mayer, Torsten Grust, Maurice Van Keulen, and Jens Teubner. “An Injection of Tree Awareness: Adding Staircase Join to PostgreSQL”. In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases*. VLDB ’04. 2004, pp. 1305–1308.

[82] Daniela Florescu and Donald Kossmann. “Storing and Querying XML Data using an RDMBS”. In: *Bulletin of the Technical Committee on Data Engineering* 22.3 (1999), pp. 27–34.

[83] Daniela Florescu and Donald Kossmann. *A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database*. Technical Report RR-3680. Projet RODIN. INRIA, 1999. URL: `http://hal.inria.fr/inria-00072991`.

[84] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. “Relational Databases for Querying XML Documents: Limitations and Opportunities”. In: *Proceedings of the 25th International Conference on Very Large Data Bases*. VLDB ’99. San Francisco, CA, USA, 1999, pp. 302–314.

[85] Brian Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, and Moshe Shadmon. “A Fast Index for Semistructured Data”. In: *Proceedings of the 27th International Conference on Very Large Data Bases*. VLDB ’01. Rome, Italy, 2001, pp. 341–350.

[86] Qun Chen, Andrew Lim, and Kian Win Ong. “D(K)-index: An Adaptive Structural Summary for Graph-structured Data”. In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’03. New York, NY, USA, 2003, pp. 134–144.

[87] Radim Bača, Michal Krátký, and Václav Snášel. “On the Efficient Search of an XML Twig Query in Large DataGuide Trees”. In: *Proceedings of the 2008 International Symposium on Database Engineering &#38; Applications*. IDEAS ’08. New York, NY, USA, 2008, pp. 149–158.

[88] Roy Goldman and Jennifer Widom. “DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases”. In: *Proceedings of the 23rd International Conference on Very Large Data Bases*. VLDB ’97. San Francisco, CA, USA, 1997, pp. 436–445.

[89] Tova Milo and Dan Suciu. “Index Structures for Path Expressions”. In: *Proceedings of the 7th International Conference on Database Theory*. ICDT ’99. London, UK, UK, 1999, pp. 277–295.

[90] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. “Exploiting local similarity for indexing paths in graph-structured data”. In: *Proceedings of the 18th International Conference on Data Engineering*. 2002, pp. 129–140.

[91] Jan-Marco Bremer and Michael Gertz. *An efficient XML node identification and indexing scheme*. Technical Report CSE-2003-04. Department of Computer Science, University of California at Davis, 2003.

[92] Torsten Grust. “Accelerating XPath location steps”. In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’02. 2002, pp. 109–120.

[93] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. “Structural Joins: A Primitive for Efficient XML Query Pattern Matching”. In: *Proceedings of the 18th International Conference on Data Engineering*. ICDE ’02. San Jose, CA, USA, 2002, pp. 141–152.

[94] Torsten Grust, Maurice van Keulen, and Jens Teubner. “Staircase Join: Teach a Relational DBMS to Watch Its (Axis) Steps”. In: *Proceedings of the 29th International Conference on Very Large Data Bases*. Vol. 29. VLDB ’03. Berlin, Germany, 2003, pp. 524–535.

[95] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. “Efficient structural joins on indexed XML documents”. In: *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB ’02. Hong Kong, China, 2002, pp. 263–274.

[96] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng-Chin Ooi. “XR-tree: indexing XML data for efficient structural joins”. In: *Data Engineering, 2003. Proceedings. 19th International Conference on.* 2003, pp. 253–264.

[97] Jiaheng Lu, Tok Wang Ling, Chee-Yong Chan, and Ting Chen. “From Region Encoding to Extended Dewey: On Efficient Processing of XML Twig Pattern Matching”. In: *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB ’05. Trondheim, Norway, 2005, pp. 193–204.

[98] Gang Gou and Rada Chirkova. “Efficiently Querying Large XML Data Repositories: A Survey”. In: *IEEE Transactions on Knowledge and Data Engineering* 19.10 (2007), pp. 1381–1403.

[99] Christoph Koch. “Processing queries on tree-structured data efficiently”. In: *Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. Chicago, Illinois, USA, 2006, pp. 213–224.

[100] Georg Gottlob, Christoph Koch, and Reinhard Pichler. “Efficient algorithms for processing XPath queries”. In: *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB ’02. Hong Kong, China, 2002, pp. 95–106.

[101] Philip Wadler. *Two semantics of XPath*. http://homepages.inf.ed.ac.uk/wadler/papers/xpath-semantics/xpath-semantics.pdf. 2000.

[102] Philip Wadler. “A formal semantics of patterns in XSLT”. In: *Markup Technologies*. 2000, pp. 183–202.

[103] Georg Gottlob, Christoph Koch, Reinhard Pichler, and Luc Segoufin. “The Complexity of XPath Query Evaluation and XML Typing”. In: *Journal of ACM* 52.2 (Mar. 2005), pp. 284–335.

[104] Mary Fernandez, Jerome Simeon, and Philip Wadler. “An Algebra for XML Query”. In: *FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science*. Vol. 1974. Lecture Notes in Computer Science. 2000, pp. 11–45.

[105] H.V. Jagadish, LaksV.S. Lakshmanan, Divesh Srivastava, and Keith Thompson. “TAX: A Tree Algebra for XML”. In: *Database Programming Languages*. Vol. 2397. Lecture Notes in Computer Science. 2002, pp. 149–164.

[106] Stelios Paparizos and H.V. Jagadish. “The Importance of Algebra for XML Query Processing”. In: *Current Trends in Database Technology – EDBT 2006*. Vol. 4254. Lecture Notes in Computer Science. 2006, pp. 126–135.

[107] H.V. Jagadish et al. “TIMBER: A native XML database”. In: *The VLDB Journal* 11.4 (2002), pp. 274–291.

[108] Wolfgang Meier. “eXist: An Open Source Native XML Database”. In: *Web, Web-Services, and Database Systems.* Vol. 2593. Lecture Notes in Computer Science. 2003, pp. 169–183.

[109] Christian Grün, Alexander Holupirek, Marc Kramis, Marc H. Scholl, and Marcel Waldvogel. “Pushing XPath Accelerator to its Limits”. In: *Proceedings of the First International Workshop on Performance and Evaluation of Data Management Systems.* ExpDB ’06. 2006.

[110] Robert W.P. Luk, H.V. Leong, Tharam S. Dillon, Alvin T.S. Chan, W. Bruce Croft, and James Allan. “A survey in indexing and searching XML documents”. In: *Journal of the American Society for Information Science and Technology.* JASIST ’02 53.6 (2002), pp. 415–437.

[111] Rajasekar Krishnamurthy, Raghav Kaushik, and JeffreyF. Naughton. “XML-to-SQL Query Translation Literature: The State of the Art and Open Problems”. In: *Database and XML Technologies.* Vol. 2824. Lecture Notes in Computer Science. 2003, pp. 1–18.

[112] Su-Cheng Haw and Chien-Sing Lee. “Evolution of Structural Path Indexing Techniques in XML Databases: A Survey and Open Discussion”. In: *Proceedings of the 10th International Conference on Advanced Communication Technology.* Vol. 3. ICACT ’08. 2008, pp. 2054–2059.

[113] Mounia Lalmas. *XML retrieval.* San Rafael, California: Morgan & Claypool Publishers, 2009. ISBN: 9781598297874 1598297872 1598297864 9781598297867. URL: `http://www.morganclaypool.com/doi/abs/10.2200/S00203ED1V01Y200907ICR007` (visited on 09/19/2013).

[114] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval: the concepts and technology behind search.* Second. New York: Addison Wesley, 2011. ISBN: 9780321416919-0321416910.

[115] *Proceedings of the First Workshop of the INitiative for the Evaluation of XML Retrieval (INEX).* Schloss Dagstuhl, Germany, 2002.

[116] Michael A. Bender and Martín Farach-Colton. “The LCA Problem Revisited”. In: *LATIN 2000: Theoretical Informatics.* Vol. 1776. Lecture Notes in Computer Science. 2000, pp. 88–94.

[117] Dov Harel and Robert Andre Tarjan. “Fast Algorithms for Finding Nearest Common Ancestors”. In: *SIAM Journal on Computing* 13.2 (1984), pp. 338–355. eprint: `http://epubs.siam.org/doi/pdf/10.1137/0213024`.

[118] Michael A. Bender, Martín Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. “Lowest common ancestors in trees and directed acyclic graphs”. In: *Journal of Algorithms* 57.2 (2005), pp. 75–94.

[119] Sara Cohen, Jonathan Mamou, Yaron Kanza, and Yehoshua Sagiv. “XSEarch: A Semantic Search Engine for XML”. In: *Proceedings of the 29th International Conference on Very Large Data Bases.* Vol. 29. VLDB ’03. Berlin, Germany, 2003, pp. 45–56.

[120] Guoliang Li, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. “Effective Keyword Search for Valuable LCAS over XML Documents”. In: *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management.* CIKM ’07. New York, NY, USA, 2007, pp. 31–40.

[121] Michal Cutler, Yungming Shih, and Weiyi Meng. “Using the structure of HTML documents to improve retrieval”. In: *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems.* USITS’97. Berkeley, CA, USA, 1997, pp. 22–22.

[122] Andrew Trotman and Mounia Lalmas. “The Interpretation of CAS”. In: *Advances in XML Information Retrieval and Evaluation.* Vol. 3977. Lecture Notes in Computer Science. 2006, pp. 58–71.

[123] Martin Theobald, Ralf Schenkel, and Gerhard Weikum. “TopX and XXL at INEX 2005”. In: *Advances in XML Information Retrieval and Evaluation.* Vol. 3977. Lecture Notes in Computer Science. 2006, pp. 282–295.

[124] Torsten Schlieder. “Schema-Driven Evaluation of Approximate Tree-Pattern Queries”. In: *Advances in Database Technology — EDBT 2002.* Vol. 2287. Lecture Notes in Computer Science. 2002, pp. 514–532.

[125] David Carmel, Yoelle S. Maarek, Matan Mandelbrod, Yosi Mass, and Aya Soffer. “Searching XML Documents via XML Fragments”. In: *Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Informaion Retrieval.* SIGIR ’03. New York, NY, USA, 2003, pp. 151–158.

[126] 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 ’05. Trondheim, Norway, 2005, pp. 361–372.

[127] Amelie 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 ’05. Washington, DC, USA, 2005, pp. 162–173.

[128] Karen Sauvagnat, Lobna Hlaoua, and Mohand Boughanem. “XFIRM at INEX 2005: Ad-Hoc and Relevance Feedback Tracks”. In: *Advances in XML Information Retrieval and Evaluation.* Vol. 3977. Lecture Notes in Computer Science. 2006, pp. 88–103.

[129] Ernesto Damiani et al. “The APPROXML Tool Demonstration”. In: *Proceedings of the 8th International Conference on Extending Database Technology: Advances in Database Technology.* EDBT ’02. London, UK, UK, 2002, pp. 753–755.

[130] Andrew Nierman and H. V. Jagadish. “Evaluating Structural Similarity in XML Documents”. In: *Proceedings of the Fifth International Workshop on the Web and Databases.* WebDB ’02. Madison, Wisconsin, USA, 2002, pp. 61–66.

[131] Cyril Laitang, Karen Pinel-Sauvagnat, and Mohand Boughanem. “DTD Based Costs for Tree-Edit Distance in Structured Information Retrieval”. In: *Proceedings of the 35th European Conference on Advances in Information Retrieval (ECIR).* Lecture Notes in Computer Science. 2013, pp. 158–170.

[132] Bettina Fazzinga, Sergio Flesca, and Andrea Pugliese. “Retrieving XML data from heterogeneous sources through vague querying”. In: *ACM Transactions on Internet Technology.* TOIT ’09 9.2 (2009).

[133] Bettina Fazzinga, Sergio Flesca, and Filippo Furfaro. “XPath Query Relaxation through Rewriting Rules”. In: *IEEE Transactions on Knowledge and Data Engineering* 23.10 (2011), pp. 1583–1600.

[134] Giacomo Buratti and Danilo Montesi. “Ranking for Approximated XQuery Full-Text Queries”. In: *Sharing Data, Information and Knowledge.* Vol. 5071. Lecture Notes in Computer Science. 2008, pp. 165–176.

[135] V.I. Levenshtein. “Binary Codes Capable of Correcting Deletions, Insertions and Reversals”. In: *Soviet Physics Doklady* 10 (1966), pp. 707–710.

[136] Joe Tekli and Richard Chbeir. “A novel XML document structure comparison framework based-on sub-tree commonalities and label semantics”. In: *Web Semantics* 11 (Mar. 2012), pp. 14–40.

[137] Mirjana Mazuran, Elisa Quintarelli, and Letizia Tanca. “Data Mining for XML Query-Answering Support”. In: *IEEE Transactions on Knowledge and Data Engineering* 24.8 (2012), pp. 1393–1407.

[138] Ermelinda Oro, Massimo Ruffolo, and Steffen Staab. “SXPath: extending XPath towards spatial querying on web documents”. In: *Proceedings of the Very Large Data Bases Endowment* 4.2 (Nov. 2010), pp. 129–140.

[139] Shuohao Zhang and Curtis Dyreson. “Symmetrically Exploiting XML”. In: *Proceedings of the 15th International Conference on World Wide Web.* WWW ’06. New York, NY, USA, 2006, pp. 103–111.

[140] Sourav S. Bhowmick, Curtis Dyreson, Erwin Leonardi, and Zhifeng Ng. “Towards Non-directional Xpath Evaluation in a RDBMS”. In: *Proceedings of the 18th ACM Conference on Information and Knowledge Management.* CIKM ’09. New York, NY, USA, 2009, pp. 1501–1504.

[141] Ernesto Damiani, Stefania Marrara, and Gabriella Pasi. “A Flexible Extension of XPath to Improve XML Querying”. In: *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval.* SIGIR ’08. New York, NY, USA, 2008, pp. 849–850.

[142] Emanuele Panzeri and Gabriella Pasi. “An Approach to Define Flexible Structural Constraints in XQuery”. In: *Proceedings of the 8th International Conference on Active Media Technology (AMT).* Vol. 7669. Lecture Notes in Computer Science. 2012, pp. 307–317.

[143] Cong Yu and H. V. Jagadish. “Querying complex structured databases”. In: *Proceedings of the 33rd international conference on Very large data bases.* VLDB ’07. Vienna, Austria, 2007, pp. 1010–1021.

[144] Alessandro Campi, Ernesto Damiani, Sam Guinea, Stefania Marrara, Gabriella Pasi, and Paola Spoletini. “A fuzzy extension of the XPath query language”. In: *Journal of Intelligent Information Systems* 33.3 (Dec. 2009), pp. 285–305.

[145] Jan-Marco Bremer and Michael Gertz. “Integrating document and data retrieval based on XML”. In: *The VLDB Journal* 15.1 (Jan. 2006), pp. 53–83.

[146] Jan-Marco Bremer and Michael Gertz. “XQuery/IR: Integrating XML Document and Data Retrieval”. In: *Proceedings of the Fifth International Workshop on the Web and Databases.* WebDB ’02. Madison, Wisconsin, USA, 2002, pp. 1–6.

[147] Stefan Klinger. “Pathfinder - Full Text or Extending a Purely Relational XQuery Compiler with a Scoring Infrastructure for XQuery Full Text”. PhD thesis. 2010.

[148] Donald E. Knuth. *The art of Computer Programming, Volume 1: Foundametal Algorithms.* Addison-Wesley, 1997. ISBN: 0201896834 9780201896831 0201485419 9780201485417 0201038099 9780201038095.

[149] Yong Kyu Lee, Seong-Joon Yoo, Kyoungro Yoon, and P. Bruce Berra. “Index Structures for Structured Documents”. In: *Proceedings of the First ACM International Conference on Digital Libraries.* DL ’96. New York, NY, USA, 1996, pp. 91–99.

[150] Apache Software Foundation. *Apache Lucene: a high-performance, full-featured text search engine library.* http://lucene.apache.org/.

[151] Christian Grün, Sebastian Gath, Alexander Holupirek, and Marc H. Scholl. “XQuery Full Text Implementation in BaseX”. In: *Proceedings of the 6th International XML Database Symposium on Database and XML Technologies (XSym 2009).* Vol. 5679. Lecture Notes in Computer Science. Lyon, France, 2009, pp. 114–128.

[152] Christian Grün. “Storing and querying large XML instances”. PhD thesis. Universität Konstanz, Dec. 2010.

[153] Roger Bamford et al. “XQuery Reloaded”. In: *Proceedings of the 35rd International Conference on Very Large Data Bases.* VLDB ’09 2.2 (2009), pp. 1342–1353.

[154] ECMA International. *JSON (JavaScript Object Notation, Standard ECMA-262.* Technical Report 3rd edition. European Association for Standardizing Information and Communication Systems, 2011. URL: `http://www.json.org`.

[155] Daniela Florescu and Ghislain Fourny. “JSONiq: The History of a Query Language”. In: *IEEE Internet Computing* 17.5 (2013), pp. 86–90.

[156] MXQuery. *MXQuery (MicroXQuery): A lightweight, full-featured XQuery Engine.* http://mxquery.org/.

[157] MXQuery. *MXQuery (MicroXQuery): A lightweight, full-featured XQuery Engine.* http://acs.lbl.gov/software/nux/.

[158] P.M. Fischer and J. Teubner. “MXQuery with Hardware Acceleration”. In: *Data Engineering (ICDE), 2012 IEEE 28th International Conference on.* 2012, pp. 1293–1296.

[159] W3C. *XQuery Scripting Extension 1.0.* Working Draft. World Wide Web Consortium (W3C), Apr. 2010. URL: `http://www.w3.org/TR/xquery-sx-10/`.

[160] W3C. *XQuery and XPath Full Text 1.0 Test Suite Result Summary.* Technical Report. World Wide Web Consortium (W3C), Jan. 2011. URL: `http://dev.w3.org/2007/xpath-full-text-10-test-suite/PublicPagesStagingArea/ReportedResults/XQFTTSReport.html`.

[161] Peter M. Fischer. “XQBench - A XQuery Benchmarking Service”. In: *Proceedings of the international conference XML Prague 2010.* XML PRague 2010. Prague, Czech Republic, 2010, pp. 341–355.

[162] Martin Kaufmann, Peter M. Fischer, Donald Kossmann, and Norman May. “A generic database benchmarking service”. In: *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE).* 2013, pp. 1276–1279.

[163] Emanuele Panzeri and Gabriella Pasi. “Flex-BaseX: An XML engine with a flexible extension of XQuery Full-Text”. In: *Proceedings of the 36th international ACM SIGIR conference on Research and development in Information Retrieval (SIGIR).* 2013.

[164] Silvia Calegari, Emanuele Panzeri, and Gabriella Pasi. “PatentLight: A Patent Search Application”. In: *Proceedings of the 4th Information Interaction in Context Symposium (IIIX).* Nijmegen, The Netherlands, 2012, pp. 242–245.

[165] Hideo Joho, Leif A. Azzopardi, and Wim Vanderbauwhede. “A survey of patent users: an analysis of tasks, behavior, search functionality and system requirements”. In: *Proceedings of the third symposium on Information interaction in context,IIiX ’10.* IIiX ’10. New Brunswick, New Jersey, USA, 2010, pp. 13–24.

[166] Vagelis Hristidis, Eduardo Ruiz, Alejandro Hernández, Fernando Farfán, and Ramakrishna Varadarajan. “Patentssearcher: a novel portal to search and explore patents”. In: *Proceedings of the 3rd international workshop on Patent information retrieval.* PaIR ’10. Toronto, ON, Canada, 2010, pp. 33–38.

[167] Hisao Mase, Tadataka Matsubayashi, Yuichi Ogawa, Makoto Iwayama, and Tadaaki Oshio. “Proposal of two-stage patent retrieval method considering the claim structure”. In: 4.2 (June 2005), pp. 190–206.

[168] Xiaobing Xue and W. Bruce Croft. “Automatic query generation for patent search”. In: *Proceedings of the 18th ACM conference on Information and knowledge management.* CIKM ’09. Hong Kong, China, 2009, pp. 2037–2040.